IRIS Performer™ C Reference Pages

Document Number 007-2783-001

CONTRIBUTORS

Written by Sharon Clay, Michael Garland, Brad Grantham, Don Hatch, Jim Helman, Michael Jones, T. Murali, John Rohlf, Allan Schaffer, Christopher Tanner, and Jenny Zhao
Production by Derrald Vogt
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson, Erik Lindholm, and Kay Maitz

IRIS Performer™ C Reference Pages
Document Number 007-2783-001

NAME

Performer – Overview of IRIS Performer and summary of the C Language Bindings: libpr, libpf, libpfdu, libpfdb, libpfui, and libpfutil.

DESCRIPTION

Welcome to the IRIS Performer application development environment.

IRIS Performer provides a comprehensive programming interface (with ANSI C and C++ bindings) for creating real-time visual simulation and other interactive graphics applications. IRIS Performer 2.0 supports both the IRIS Graphics Library (IRIS GL) and the industry standard OpenGL graphics library; these libraries combine with the IRIX operating system and REACT extensions to form the foundation of a powerful suite of tools and features for creating real-time visual simulation applications on Silicon Graphics systems.

IRIS Performer is an integral part of the Onyx/RealityEngine and Indigo2/Impact visual simulation systems and provides interfaces to the advanced features of RealityEngine class graphics. IRIS Performer is compatible with all SGI graphics platforms and attains maximum performance on each. IRIS Performer provides an extensible basis for creating real-time 3D graphics applications in the fields of visual simulation, entertainment, virtual reality, broadcast video, and computer aided design. IRIS Performer is the flexible, intuitive, toolkit-based solution for developers who want to optimize performance on Silicon Graphics systems.

Take a Test Drive

If you are new to IRIS Performer, the best way to start learning about it is to go for a test drive. The Performer-based sample application **perfly** is installed in the */usr/sbin* directory. To start **perfly**, all that you need to do is type

```
perfly esprit.flt
```

Type "man pfiXformer" for details on how to drive, fly, or tumble; and rerun **perfly** with the command line option "-help" for a full list of features. Type "?" while running **perfly** to print a list of keyboard command sequences to the shell window. The source code for this program is in */usr/share/Performer/src/sample/perfly*.

IRIS Performer Overview

IRIS Performer consists of two main libraries, **libpf** and **libpr**, and four associated libraries, **libpfdu**, **libpfdb**, **libpfui**, and **libpfutil**.

The basis of IRIS Performer is the performance rendering library **libpr**, a low level library providing high speed rendering functions based on **pfGeoSets**, efficient graphics state control using **pfGeoStates**, and other application-neutral functions. Layered above **libpr** is **libpf**, a real-time visual simulation environment providing a high-performance multi-processing database rendering system that takes best

advantage of IRIS symmetric multiprocessing CPU hardware. The database utility library **libpfdu** provides powerful functions for defining both geometric and appearance attributes of three dimensional objects, encourages sharing of state and materials, and generates efficient triangle strips from independent polygonal input. The database library **libpfdb** uses the facilities of **libpfdu**, **libpf**, and **libpr** to import database files in many popular industry standard database formats. These loaders also serve as a guide to developers creating new database importers. **libpfui** contains the user interface, and input management facilities common to many interactive applications. Completing the suite of libraries is **libpfutil**, the IRIS Performer utility library. It provides a collection of important convenience routines implementing such diverse tasks as smoke effects, MultiChannel Option support, graphical user interface tools, input event collection and handling, and various traversal functions.

In addition to these SGI-developed tools, IRIS Performer also includes sample code, databases, games, and movies contributed by the *Friends of Performer*: companies and individuals with services of general interest to the IRIS Performer community.

Program Structure

Most IRIS Performer application programs have a common general structure. The following steps are typically involved in preparing for a real-time simulation:

1. Initialize IRIS Performer with **pfInit**.
2. Specify number of graphics pipelines with **pfMultipipe**, choose the multiprocessing configuration by calling **pfMultiprocess**, and specify the hardware mode with **pfHyperpipe** if needed.
3. Initiate the chosen multiprocessing mode by calling **pfConfig**.
4. Initialize the frame rate with **pfFrameRate** and set the frame-extend policy with **pfPhase**.
5. Create, configure, and open windows with **pfNewPWin**, **pfPWinFBConfigAttrs**, and **pfOpenPWin**, as required.
6. Create and configure display channels with **pfNewChan**, **pfChanTravFunc**, **pfChanFOV**, and **pfChanScene** as required.

Once the application has created a graphical rendering environment as shown above, the remaining task is to iterate through a main simulation loop once per frame.

7. Compute dynamics, update model matrices, etc.
8. Delay until the next frame time: **pfSync**

9. Perform latency critical viewpoint updates.
10. Draw a frame by calling **pfFrame**.

In many applications the viewpoint will be set in step 7 and both step 8 and step 9 are not required. The more general case is shown since it is typical in head-tracked and other cases where low-latency applications with last-minute position input must be used.

The libpr Performance Rendering Library

Libpr consists of many low-level hardware oriented facilities generally required for real-time and other performance-oriented graphics applications. These features include

High-speed rendering functions using the innovative **pfGeoSet**.

Efficient graphics state management and mode control based on the **pfGeoState**.

Display lists suitable for rendering between multiple processes.

An extensive collection of fast linear algebra and math routines.

Intersection computation and detection services.

A colortable mechanism for rapid switching of database appearance.

Asynchronous file I/O system for real-time file operations.

Memory allocation oriented to shared memory and mutual exclusion.

High speed clock functions that hide the complexities of hardware clocks.

GeoSets are collections of drawable geometry which group same-type graphics primitives (e.g. triangles or quads) into one data object. The GeoSet contains no geometry itself, only pointers to data arrays and index arrays. Geometry arrays may be indexed or non-indexed (i.e. stored in order) depending upon application requirements. Because all the primitives in a GeoSet are of the same type and have the same attributes, rendering of most databases is performed at maximum hardware speed. There are many GeoSet rendering methods, one for each combination of geometry and attribute specification. However, in IRIS Performer, all GeoSet rendering is performed through a single render dispatching routine, **pfDrawGSet**.

GeoStates provide graphics state definitions (e.g. texture or material) for GeoSets. When used in conjunction with Performer state management functions, GeoSets can be rendered in a prescribed way without concern for the inherited modes of the graphics pipeline. GeoSets may share GeoStates. Less-used

machine modes are not supported.

State Management and Mode Control. IRIS Performer provides functions that bundle together graphics library state control functions such as lighting, materials, texture, and transparency. They have two purposes: to track state and to allow the creation of display lists that can be rendered later. The application program can set states in three ways: globally, locally (via **GeoState**), and directly. State changes made using direct graphics library calls are not "known" to the IRIS Performer state tracking mechanisms, and thus defeat IRIS Performer state management. However, functions exist to push state, pop state, and get the current state so proper intermixing of direct graphics library and IRIS Performer functions can be achieved.

Display Lists are supported in IRIS Performer. These are not typical graphics library display lists, but rather simple token and data mechanisms that do not cache geometry or state data and are designed to allow efficient multiprocessing. These display lists use IRIS Performer state and rendering commands. They also support function callbacks to allow application programs to perform any required special processing during display list rendering.

Windows for IRIS GL, IRIS GL mixed model (GLX), and OpenGL applications can be configured, created and managed with the **pfWindow** routines.

Math Support is provided by an extensive set of point, segment, vector, plane, matrix, cylinder, sphere and frustum functions.

Intersection and **collision detection** functions are provided to test for the intersection of line segments with cylinders, spheres, boxes, planes, and geometry. Intersection functions for spheres, cylinders, and frusta are also provided.

ColorTables are supported by allowing GeoSet color indexes to refer to common tables of RGBA color information. Color tables are global and may be of any size. Any number of color tables may exist at one time and they can be activated at any time. The active color table may be switched in real-time without performance impact.

Asynchronous File I/O is provided by a simple non-blocking file access method. This is provided to allow applications to retrieve file data during real-time operation.

Memory Allocation is supported with routines to allocate memory from process heap storage, shared memory arenas, and datapool memory. Shared arenas must be used when multiple processes need to access data. The arena is created by the application program. Datapools allow applications to create shared arenas visible to any process where allocations can be locked for easy mutual exclusion on a per allocation basis.

High Speed Clock support is based on a high speed clock access routine that reports elapsed time in seconds as a double precision floating point number to highest machine resolution.

Statistics are maintained by IRIS Performer on the geometry that is drawn, state changes, transformations, and most internal operations. These statistics can used for application tuning and form the basis for IRIS Performer's automatic system load management.

The libpf Visual Simulation Library

libpf is a high level library built on **libpr** that is architected and implemented to meet the specific needs of real-time graphics software. Applications developed with **libpf** are able to provide smooth motion through elaborate scenes at programmable frame rates, all with very little code development. **libpf** provides

Hierarchical scene graph processing and operators.

Transparent multiprocessing for parallel simulation, culling and drawing.

Graphics load measurement and frame rate management.

Level of detail selection with smooth fade and rotational invariance.

Rapid culling to the viewing frustum through hierarchical bounding volumes.

Multiprocessed intersection detection and reporting.

Dynamic coordinate systems for highly interactive graphics.

Multibuffering of changes to the scene graph for simple multiprocessing.

Multiprocessing

libpf provides a pipelined multiprocessing model for implementing visual simulation applications. The application, visibility culling and drawing tasks can all run in separate processes. The simulation process updates the scene, the cull process traverses the scene checking for visibility and generates display lists which are then rendered by the drawing process. **libpf** multibuffering capabilities allow each process to have copies of the scene graph and the user data appropriate to its target frame time.

The simulation, culling, and drawing for a graphics pipeline may be combined into one, two or three processes to allow an application to be tailored to different hardware and expected CPU demand in each process. For example, culling and drawing are normally done by separate processes in order to obtain maximum graphics performance, but if an application is simulation bound, it may wish to combine both cull and draw into a single process.

Statistics are maintained for each IRIS Performer process - application, cull and draw. These statistics can

be displayed in a channel, printed, and queried using the **pfFrameStats** routines.

Graphics Pipes, Windows, and Channels

In addition to the functionality it derives from **libpr**, **libpf** supports multiple channels per window, multiple windows per graphics pipe, grouping of channels to form video walls, and frame synchronization between multiple graphics pipes. **libpf** maintains a graphics stress value for each channel and uses it to attempt to maintain a fixed frame rate by manipulating levels-of-detail (LODs). Like many graphics libraries, **libpf** assumes a coordinate system with +Z up, +X to the right and +Y into the screen.

Database

libpf supports a general database hierarchy which consists of the following node types:

pfNode	General node (base class)
pfScene	Top level node.
pfGroup	Node with multiple children.
pfSCS	Static coordinate system.
pfDCS	Dynamic coordinate system.
pfLayer	Layer or decal node.
pfLOD	Level of detail node.
pfSwitch	Switch node.
pfSequence	Sequential animation node.
pfGeode	Fundamental geometry node.
pfBillboard	Special tracking leaf node.
pfLightPoint	One or more emissive light points.
pfLightSource	Definition of a graphics hardware light.
pfPartition	Special culling acceleration node.
pfText	2D and 3D text geometry.
pfMorph	Geometry morphing node.

Each of these is derived from **pfNode** and any function which requires a **pfNode*** as an argument can accept any of the above types. Similarly **pfSCS**, **pfDCS**, **pfLOD**, **pfSequence** and **pfSwitch** are derived from **pfGroup** and can be used in any function which takes a **pfGroup*** as an argument.

Nodes can be assembled into a directed graph to represent a scene with its modeling hierarchy. Geometry and graphics state information is contained in **pfGeoStates** and **pfGeoSets** which are attached to **pfGeodes**.

Intersection inquiries are made via groups of line segments which can be tested against a subgraph of the scene. Masks and callbacks can be specified to allow evaluation of line-of-sight visibility, collisions, and terrain intersections. **libpf** also provides earth-sky and weather functions for modeling fog, haze and other atmospheric effects.

The libpfdu Database Utility Library

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

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

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

To go along with each pfGeoSet, the builder creates a pfGeoState (IRIS Performer's encapsulated state primitive). The builder generates pfGeoStates that share as many attributes as possible with other pfGeoStates in the scene graph.

Having created these primitives (pfGeoSets and pfGeoStates) the builder will place them in a leaf node (pfGeode), and optionally create a spatial hierarchy by running the new database through a spatial breakup utility function which is also contained in **libpfdu**.

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

The libpfui User Interface Library

The **libpfui** library provides building blocks for writing manipulation components for user interfaces. This library provides both C and C++ interfaces. Provided are separate components for motion control (-**pfiInputCoordXform**), collision detection between the viewer and objects in the scene (**pfiCollide**), and picking of objects in the scene based on current mouse coordinates (**pfiPick**). The **pfiInputCoordXform** utilities update transformation matrices that can be used to drive motion in an application. The actual mapping of user events is orthogonal to these motion models and can be done using the input collection

utilities in **libpfutil**, or directly with custom application code. The **pfiXformer** is a re-implementation of the old **pfuXformer** based on these components and combines several different kinds of motion control in one complex component. The **pfiXformer** also provides mapping of user input events, such as mouse and keyboard, to motion controls which is described in the **pfiXformer** reference page. Examples of how to use these utilities can be found in

```
/usr/share/Performer/src/pguide/libpfui/
```

The libpfutil Utility Library

The **libpfutil** library contains a large number of miscellaneous functions that provide support for the following important tasks.

Processor control enables the user to specify which CPU a particular Performer process runs on and to devote a particular processor to a given process.

Multiprocess rendezvous lets master and slave processes synchronize in a multiprocessing environment.

GLX mixed model routines are provided for compatibility with previous versions of IRIS Performer. Current development should be based on the pfWindow and pfPipeWindow routines that provide a single API for managing IRIS GL, IRIS GL mixed model, and OpenGL windows.

GL and X input handling is handled by an exhaustive set of commands that operate on compressed, space-efficient queues of events.

Cursor control is provided to easily manipulate the cursors associated with each window managed by IRIS Performer.

X fonts are supported so that they can be used to draw text in IRIS Performer windows. The main task of these functions is to simplify the use of X fonts and present a high-level interface to the user.

Graphical User Interfaces (GUIs) are made easily accessible to the user through a set of functions that provide simple means to create a GUI, set up widgets, manipulate them, set user-defined functions to control their behavior and do other common tasks.

Scene graph traversal routines provide for different, highly-customizable traversal mechanisms for the IRIS Performer scene graph.

MultiChannel Option (MCO) is supported on RealityEngine graphics systems by a set of functions that generically initialize channels for using MCO.

Path following mechanisms allow the user to follow a pre-defined path in a walkthrough application.

Functions to create paths are also provided.

Various **draw styles** like haloed lines and wireframe images are supported as a demonstration of the uses of multi-pass rendering.

Other utilities supported are for **timer control** to track time in real-time independently of the frame-rate, managing **hash tables**, a simple **geometric simplification** scheme for generating very simple level-of-detail representations of the scene graph, **texture loading** and **texture animation**, **random number generation**, **flybox control**, **smoke** and **fire simulation** and converting **light point states** into textures.

The libpfdb Database Library

libpfdb is a collection of independent libraries (one for each supported file format) that read or write a particular scene description file format. These loaders are implemented using the IRIX Dynamic Shared Object facility and are demand loaded as needed.

The loaders in **libpfdb** have been developed by Silicon Graphics, by modeling tool vendors, and by Performer customers. Many are provided in source form as part of this IRIS Performer distribution. Use these loaders as templates to write custom loaders for whatever formats you require in your applications. The different kinds of file formats supported by IRIS Performer are listed below

3ds	AutoDesk *3DStudio* binary data
bin	Minor SGI format used by *powerflip*
bpoly	Side Effects Software *PRISMS* binary
byu	Brigham Young University CAD/FEA data
dwb	Coryphaeus Software *Designer's Workbench*
dxf	AutoDesk *AutoCAD* ASCII format
flt11	MultiGen public domain Flight v11 format
flt14	MultiGen *OpenFlight* v14 format
gds	McDonnell-Douglas GDS *things* data
gfo	Minor SGI format (radiosity output)
im	Minor SGI format (IRIS Performer example)
irtp	AAI/Graphicon *Interactive Real-Time PHIGS*
iv	SGI OpenInventor / Silicon Studio Keystone
lsa	Lightscape Technologies radiosity (ASCII)
lsb	Lightscape Technologies radiosity (binary)
m	University of Washington mesh data
medit	Medit Productions *medit* modeling tool
nff	Eric Haines' ray tracing test data format
obj	Wavefront Technologies data format

phd	Minor SGI format (polyhedra)
poly	Side Effects Software *PRISMS* ASCII data
pts	University of Washington point data
ptu	Minor SGI format (IRIS Performer example)
s1k	US ARMY SIMNET databases (Texas Instruments)
sgf	US NAVY standard graphics format
sgo	Minor SGI format
spf	US NAVY simple polygon format
sponge	Sierpinski sponge 3D fractal generator
star	Yale University compact star chart data
stla	3D Structures Stereolithography (ASCII)
stlb	3D Structures Stereolithography (binary)
sv	Format of John Kichury's *i3dm* modeler
tri	University of Minnesota Geometry Center data
unc	University of North Carolina data

Source code for many of these loaders is provided with IRIS Performer. Loader source code is located in and below the directory

```
/usr/share/Performer/src/libpfdb
```

While most loaders do in fact "load" data from files, scene graphs can also be generated procedurally. The *sponge* loader is an example of such automatic generation; it builds a model of the Menger (Sierpinski) sponge, without requiring an input file. To see the sponge run **perfly** specify the number of recursions (0, 1, 2, ...) as the filename. For example

```
perfly 2.sponge
```

Learning More

Once you've seen IRIS Performer in action, you will want to learn more about it. The IRIS Performer Programming Guide and the IRIS Performer Release Notes are the primary sources of information, but the following overview will give you a head start in your learning process.

IRIS Performer Sample Code

The IRIS Performer sample code can be found in

```
/usr/share/Performer/src/pguide - small examples
```

and

```
/usr/share/Performer/src/sample - sample applications
```

and its subdirectories. The "apps" subdirectory contains the various flying demos like **perfly** and the Performer **town** demo. The "pguide" subdirectory has further subdirectories for each IRIS Performer library. Each of these directories has example and sample programs that highlight the features of the corresponding library.

IRIS Performer Documentation

In addition to the reference pages on IRIS Performer, an on-line Programming Guide is also provided. To read this, run *Insight* and click on the *Performer Programming Guide* button.

IRIS Performer World Wide Web Home Page

Silicon Surf, the Silicon Graphics World Wide Web Home Page, contains an archive of IRIS Performer-related technical and promotional material in the *Extreme Tech* section. The information from the IRIS Performer FTP site and mailing list is also accessible via the WWW.

Explore Silicon Surf using the URL

```
http://www.sgi.com/
```

or go directly to the IRIS Performer information with the URL

```
http://www.sgi.com/Technology/Performer.html
```

IRIS Performer INTERNET FTP Site

An archive of IRIS Performer-related material is available via anonymous FTP from Silicon Graphics. The FTP address is

```
ftp://sgigate.sgi.com/pub/Performer
```

Current contents of the IRIS Performer FTP site include

README	Overview file
FAQ	The IRIS Performer FAQ

INFO-PERFORMER	Information about the IRIS Performer mailing list
src/	Sample source code and miscellaneous patches
docs/	IRIS Performer documents including SIGGRAPH '94 paper
selected-topics/	Directory of info, Q&A, etc. from mailing list
monthly-archives/	Raw monthly archives of the mailing list
CortaillodCentre/	Goodies from SGI's Cortaillod Office
RealityCentre/	Goodies from SGI's RealityCentre in the UK

IRIS Performer Electronic Mailing List

The IRIS Performer mailing list is a resource for developers who are using IRIS Performer to maximize the performance of their graphics applications on Silicon Graphics hardware. The *info-performer* list is intended to be an unmoderated, free-form discussion of IRIS Performer with issues both technical and non-technical; and to provide feedback to Silicon Graphics about the product. Much like the comp.sys.sgi.* newsgroups, it is not an official support channel but is monitored by the IRIS Performer development team, so it's an excellent source of early information about upcoming events and product features, as well as a venue for asking questions and having them answered.

To subscribe to the *info-performer* mailing list, send email to

```
info-performer-request@sgi.com
```

Once your request is processed you will receive submission and posting instructions, some guidelines, and a current copy of the Performer Frequently-Asked-Questions (FAQ) list.

The mailing list has become rather large and carries several hundred messages per month. Mailing list archives are available in the Performer FTP area (see above) in

```
ftp://sgigate.sgi.com/pub/Performer/monthly-archives/
```

IRIS Performer Frequently Asked Questions

Silicon Graphics maintains a publicly accessible directory of questions that developers often ask about IRIS Performer, along with answers to those questions. Each question-and-answer pair is provided in a file of its own, named by topic. To obtain any of these files, use anonymous FTP to connect to *sgigate.sgi.com*; then *cd* to the directory

```
/pub/Performer/selected-topics
```

and use *ls* to see a list of available topics. Alternatively, use a World Wide Web browser to look at

```
ftp://sgigate.sgi.com/pub/Performer/selected-topics
```

The Friends of Performer

A number of leading companies in the visual simulation, database modeling, game authoring, and, virtual reality marketplaces produce tools and products that are based on and work with IRIS Performer. Several of these companies have provided samples of their work for your use and enjoyment. These software gifts are in the *friends* component of the IRIS Performer distribution, and are installed in the directory

```
/usr/share/Performer/friends
```

Check out the gifts and the products that these companies offer.

IRIS Performer Application Programming Interface

The IRIS Performer application programming interface (API) has been designed by following a consistent set of naming principles that are outlined below. Following that review is a complete listing of the API grouped by topic for your use as both a quick reference and as an API directory.

Each of the **libpf**, **libpr**, **libpfdu**, **libpfdb**, **libpfui**, and **libpfutil** functions also has a complete reference page description available via the IRIX *man* and *xman* commands. Refer to these reference pages for a thorough discussion of the functions and data types, features and limitations, performance and resource implications, and sample code showing how these functions are used in practice.

IRIS Performer Software Conventions

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

All type, command and token names, associated with **libpf** or **libpr** are preceded by the letters *pf*, denoting the IRIS Performer library. Functions from the other libraries also affix an identifying letter suffix (*d*, *i*, or *u*) to the *pf* prefix for scope resolution purposes.

Library	Prefix	Example
libpf	*pf*	**pfNewGeode**
libpr	*pf*	**pfNewGSet**
libpfdu	*pfd*	**pfdNewGeom**
libpfdb	*pfd*	**pfdLoadFile_medit**
libpfui	*pfi*	**pfiResetXformerPosition**
libpfutil	*pfu*	**pfuDownloadTexList**

Command and type names are mixed-case, while token names are uppercase. For example, **pfTexture** is a type name and **PFTEX_SHARPEN** is a token name. Underscores are not used in function names except in the **libpfdb** libraries, where the underscore serves to separate the common loader name (**pfdLoad**) from the file type extension (**medit** in the example above).

In type names, the part following the **pf** is usually spelled out in full, as is the case with **pfTexture**, but in some cases a shortened form of the word is used. For example, **pfDispList** is the name of the display-list type.

Much of IRIS Performer's interface involves setting parameters and retrieving parameter values. For the sake of brevity, the word **Set** is omitted from function names, so that instead of **pfSetMtlColor**, **pfMtlColor** is the name of the routine used for setting the color of a pfMaterial. **Get**, however, is not omitted from the names of routines that get information, such as **pfGetMtlColor**.

Routine names are constructed by appending a type name to an operation name. The operation name always precedes the type name. In this case, the operation name is unabbreviated and the type name is abbreviated. For example, the name of the routine that applies a pfTexture is **pfApplyTex**.

Compound type names are abbreviated by the first initial of the first word and the entire second word. For example, to draw a display list, which is type pfDispList, use **pfDrawDList**.

Symbolic token names incorporate another abbreviation, usually shorter, of the type name. For example

pfTexture tokens begin with **PFTEX_**.

pfDispList tokens begin with **PFDL_**.

This convention ensures that tokens for a particular type have their own name space.

Other tokens and identifiers follow the conventions of ANSI C and C++ wherein a valid identifier consists of upper and lower-case alphabetic characters, digits, and underscores, and the first character is not a

digit.

LIBPF

Initialization

pfInit initializes all internal IRIS Performer data structures while **pfExit** cleans up before returning control to the application. The other functions provide support for multiprocessed execution. This involves configuring IRIS Performer for multiple processes and threads and multiple and multiplexed (hyper) pipes.

int	**pfInit**(void);
void	**pfExit**(void);
int	**pfMultipipe**(int numPipes);
int	**pfGetMultipipe**(void);
int	**pfHyperpipe**(int numHyperPipes);
int	**pfGetHyperpipe**(pfPipe *pipe);
int	**pfMultiprocess**(int mpMode);
int	**pfGetMultiprocess**(void);
int	**pfMultithread**(int pipe, uint stage, int nprocs);
int	**pfGetMultithread**(int pipe, uint stage);
int	**pfConfig**(void);
pid_t	**pfGetPID**(int pipe, uint stage);
uint	**pfGetStage**(pid_t pid, int *pipe);
void	**pfStageConfigFunc**(int pipe, uint stage, pfStageFuncType configFunc);
pfStageFuncType	**pfGetStageConfigFunc**(int pipe, uint stage);
int	**pfConfigStage**(int pipe, uint stage);

Frame Control

IRIS Performer is designed to run at a fixed frame rate. **pfFrame, pfSync** and associated functions set a frame rate the application should run at, initiate each new frame of IRIS Performer processing and synchronize the application process with the specified frame rate.

pfApp, pfCull, pfDraw and **pfDBase** trigger the default IRIS Performer processing for each stage of the graphics pipeline. User-defined callbacks can be specified for each of these stages using the **pf*Func** functions. Data can be allocated for each stage and also passed down the different stages of the pipeline.

The other functions in this set manipulate IRIS Performer memory (**pfMemory**) and its associated reference counts.

void	**pfAppFrame**(void);

int	**pfSync**(void);
int	**pfFrame**(void);
void	**pfApp**(void);
void	**pfCull**(void);
void	**pfDraw**(void);
void	**pfDrawBin**(int bin);
void	**pfIsectFunc**(pfIsectFuncType func);
pfIsectFuncType	**pfGetIsectFunc**(void);
void*	**pfAllocIsectData**(int bytes);
void*	**pfGetIsectData**(void);
void	**pfPassIsectData**(void);
void	**pfDBase**(void);
void	**pfDBaseFunc**(pfDBaseFuncType func);
pfDBaseFuncType	**pfGetDBaseFunc**(void);
void*	**pfAllocDBaseData**(int bytes);
void*	**pfGetDBaseData**(void);
void	**pfPassDBaseData**(void);
void	**pfPhase**(int phase);
int	**pfGetPhase**(void);
void	**pfVideoRate**(float vrate);
float	**pfGetVideoRate**(void);
float	**pfFrameRate**(float rate);
float	**pfGetFrameRate**(void);
int	**pfFieldRate**(int fields);
int	**pfGetFieldRate**(void);
int	**pfGetFrameCount**(void);
double	**pfGetFrameTimeStamp**(void);
void	**pfFrameTimeStamp**(double t);
int	**pfGetId**(void *mem);
int	**pfAsyncDelete**(void *mem);
int	**pfCopy**(void *dst, void *src);

pfPipe Functions

A **pfPipe** is a software rendering pipeline which renders one or more **pfChannel**s into one or more **pfPipeWindow**s. Typically one **pfPipe** is created for each hardware graphics pipeline.

pfPipe*	**pfGetPipe**(int pipeNum);
int	**pfInitPipe**(pfPipe *pipe, pfPipeFuncType configFunc);

pfPipe C API

These functions create and manipulate **pfPipe**s. Control can be exercised over the hardware screen used by the **pfPipe** and the way a **pfPipe** swaps color buffers at the end of each frame.

pfType*	**pfGetPipeClassType**(void);
void	**pfPipeSwapFunc**(pfPipe* pipe, pfPipeSwapFuncType func);
pfPipeSwapFuncType	**pfGetPipeSwapFunc**(const pfPipe* pipe);
void	**pfGetPipeSize**(const pfPipe* pipe, int *xs, int *ys);
void	**pfPipeScreen**(pfPipe* pipe, int scr);
int	**pfGetPipeScreen**(const pfPipe* pipe);
void	**pfPipeWSConnectionName**(pfPipe* pipe, const char *name);
const char*	**pfGetPipeWSConnectionName**(const pfPipe* pipe);
pfChannel*	**pfGetPipeChan**(const pfPipe* pipe, int i);
int	**pfGetPipeNumChans**(const pfPipe* pipe);
pfPipeWindow*	**pfGetPipePWin**(const pfPipe* pipe, int i);
int	**pfGetPipeNumPWins**(const pfPipe* pipe);
int	**pfGetPipeHyperId**(const pfPipe* pipe);
int	**pfMovePWin**(pfPipe* pipe, int where, pfPipeWindow *pw);
pfBuffer*	**pfGetCurBuffer**(void);

pfBuffer C API

The **pfBuffer** data structure logically encompasses **libpf** objects such as **pfNode**s. Newly created objects are automatically "attached" to the current **pfBuffer** specified by **pfSelectBuffer**. Later, any objects created in *buf* may be merged into the main IRIS Performer processing stream with **pfMergeBuffer**. In conjunction with a forked **DBASE** process (see **pfMultiprocess** and **pfDBaseFunc**), the **pfBuffer** mechanism supports asynchronous parallel creation and deletion of database objects. This is the foundation of a real-time database paging system.

pfBuffer*	**pfNewBuffer**(void);
void	**pfBufferScope**(pfBuffer* buffer, pfObject *obj, int scope);
int	**pfGetBufferScope**(pfBuffer* buffer, pfObject *obj);
void	**pfMergeBuffer**(void);
int	**pfUnrefDelete**(void *mem);
int	**pfDelete**(void *mem);
int	**pfBufferInsert**(void *parent, int index, void *child);
int	**pfBufferRemove**(void *parent, void *child);
int	**pfBufferAdd**(void *parent, void *child);
int	**pfBufferReplace**(void *parent, void *oldChild, void *newChild);
void	**pfSelectBuffer**(pfBuffer* buffer);
void	**pfInitGfx**(void);

pfPipeWindow C API

A **pfPipeWindow** creates a window on the screen managed by a given **pfPipe**. Programs render to a **pfPipeWindow** by attaching a **pfChannel** of that **pfPipe** to the **pfPipeWindow**. Various ways of controlling the behavior of **pfPipeWindow**s are provided including specifying their position and size on the screen, specifying user-specified callbacks to configure them in the **DRAW** process, controlling lists of **pfWindows** that can draw into a singe **pfPipewindow**, and manipulating **pfChannel**s assigned to the **pfPipeWindow**s.

pfPipeWindow*	**pfNewPWin**(pfPipe *p);
pfType*	**pfGetPWinClassType**(void);
void	**pfPWinName**(pfPipeWindow* pwin, const char *name);
const char*	**pfGetPWinName**(pfPipeWindow* pwin);
void	**pfPWinWSConnectionName**(pfPipeWindow* pwin, const char *name);
const char*	**pfGetPWinWSConnectionName**(pfPipeWindow* pwin);
void	**pfPWinMode**(pfPipeWindow* pwin, int mode, int val);
int	**pfGetPWinMode**(pfPipeWindow* pwin, int mode);
void	**pfPWinType**(pfPipeWindow* pwin, uint type);
uint	**pfGetPWinType**(pfPipeWindow* pwin);
pfState*	**pfGetPWinCurState**(pfPipeWindow* pwin);
void	**pfPWinAspect**(pfPipeWindow* pwin, int x, int y);
void	**pfGetPWinAspect**(pfPipeWindow* pwin, int *x, int *y);
void	**pfPWinOriginSize**(pfPipeWindow* pwin, int xo, int yo, int xs, int ys);
void	**pfPWinOrigin**(pfPipeWindow* pwin, int xo, int yo);
void	**pfGetPWinOrigin**(pfPipeWindow* pwin, int *xo, int *yo);
void	**pfPWinSize**(pfPipeWindow* pwin, int xs, int ys);
void	**pfGetPWinSize**(pfPipeWindow* pwin, int *xs, int *ys);
void	**pfPWinFullScreen**(pfPipeWindow* pwin);
void	**pfGetPWinCurOriginSize**(pfPipeWindow* pwin, int *xo, int *yo, int *xs, int *ys);
void	**pfGetPWinCurScreenOriginSize**(pfPipeWindow* pwin, int *xo, int *yo, int *xs, int *ys);
void	**pfPWinOverlayWin**(pfPipeWindow* pwin, pfWindow *ow);
pfWindow*	**pfGetPWinOverlayWin**(pfPipeWindow* pwin);
void	**pfPWinStatsWin**(pfPipeWindow* pwin, pfWindow *sw);
pfWindow*	**pfGetPWinStatsWin**(pfPipeWindow* pwin);
void	**pfPWinScreen**(pfPipeWindow* pwin, int screen);
int	**pfGetPWinScreen**(pfPipeWindow* pwin);
void	**pfPWinShare**(pfPipeWindow* pwin, int mode);
uint	**pfGetPWinShare**(pfPipeWindow* pwin);
void	**pfPWinWSWindow**(pfPipeWindow* pwin, pfWSConnection dsp, pfWSWindow wsw);
Window	**pfGetPWinWSWindow**(pfPipeWindow* pwin);
void	**pfPWinWSDrawable**(pfPipeWindow* pwin, pfWSConnection dsp, pfWSDrawable gxw);
pfWSDrawable	**pfGetPWinWSDrawable**(pfPipeWindow* pwin);
pfWSDrawable	**pfGetPWinCurWSDrawable**(pfPipeWindow* pwin);
void	**pfPWinFBConfigData**(pfPipeWindow* pwin, void *data);
void*	**pfGetPWinFBConfigData**(pfPipeWindow* pwin);
void	**pfPWinFBConfigAttrs**(pfPipeWindow* pwin, int *attr);

int*	**pfGetPWinFBConfigAttrs**(pfPipeWindow* pwin);
void	**pfPWinFBConfig**(pfPipeWindow* pwin, XVisualInfo *vis);
XVisualInfo*	**pfGetPWinFBConfig**(pfPipeWindow* pwin);
void	**pfPWinFBConfigId**(pfPipeWindow* pwin, int vId);
int	**pfGetPWinFBConfigId**(pfPipeWindow* pwin);
void	**pfPWinIndex**(pfPipeWindow* pwin, int index);
int	**pfGetPWinIndex**(pfPipeWindow* pwin);
pfWindow*	**pfGetPWinSelect**(pfPipeWindow* pwin);
void	**pfPWinGLCxt**(pfPipeWindow* pwin, pfGLContext gc);
pfGLContext	**pfGetPWinGLCxt**(pfPipeWindow* pwin);
void	**pfPWinList**(pfPipeWindow* pwin, pfList *wl);
pfList*	**pfGetPWinList**(const pfPipeWindow* pwin);
int	**pfAttachPWinWin**(pfPipeWindow* pwin, pfWindow *w1);
int	**pfDetachPWinWin**(pfPipeWindow* pwin, pfWindow *w1);
int	**pfAttachPWin**(pfPipeWindow* pwin, pfPipeWindow *pw1);
int	**pfDetachPWin**(pfPipeWindow* pwin, pfPipeWindow *pw1);
pfWindow*	**pfSelectPWin**(pfPipeWindow* pwin);
void	**pfSwapPWinBuffers**(pfPipeWindow* pwin);
pfFBConfig	**pfChoosePWinFBConfig**(pfPipeWindow* pwin, int *attr);
int	**pfIsPWinOpen**(pfPipeWindow* pwin);
int	**pfQueryPWin**(pfPipeWindow* pwin, int which, int *dst);
int	**pfMQueryPWin**(pfPipeWindow* pwin, int *which, int *dst);
pfPipe*	**pfGetPWinPipe**(pfPipeWindow* pwin);
int	**pfGetPWinPipeIndex**(const pfPipeWindow* pwin);
void	**pfPWinConfigFunc**(pfPipeWindow* pwin, pfPWinFuncType func);
pfPWinFuncType	**pfGetPWinConfigFunc**(pfPipeWindow* pwin);
int	**pfGetPWinChanIndex**(pfPipeWindow* pwin, pfChannel *chan);
void	**pfConfigPWin**(pfPipeWindow* pwin);
void	**pfOpenPWin**(pfPipeWindow* pwin);
void	**pfClosePWin**(pfPipeWindow* pwin);
void	**pfClosePWinGL**(pfPipeWindow* pwin);
int	**pfRemoveChan**(pfPipeWindow* pwin, pfChannel *chan);
void	**pfAddChan**(pfPipeWindow* pwin, pfChannel *chan);
void	**pfInsertChan**(pfPipeWindow* pwin, int where, pfChannel *chan);
int	**pfMoveChan**(pfPipeWindow* pwin, int where, pfChannel *chan);
pfChannel*	**pfGetChan**(pfPipeWindow* pwin, int which);
int	**pfGetNumChans**(const pfPipeWindow* pwin);
void	**pfNodePickSetup**(pfNode* node);

pfChannel C API

A **pfChannel**'s primary function is to define a viewing frustum which is used both for viewing and for culling. A **pfChannel** can be associated with a **pfPipe** with **pfNewChan**. All aspects of the **pfChannel**'s viewing frustum, field of view (FOV), aspect ratio, view point and viewing direction can be modified. A

custom culling volume for the **pfChannel** can be set (**pfChanCullPtope**).

Different queries can be made about the **pfChannel** (**pfGetChan***) and user-defined traversal functions and mode can be set (**pfChanTrav***). Functions are provided to control IRIS Performer's level-of-detail (LOD) behavior by specifying view position, field-of-view, and viewport pixel size (**pfChanLOD***). **pfChanStress** can be used to specify when the system is at stress so that the LOD behavior is suitably modified.

The **pfScene** and the **pfEarthSky** that the **pfChannel** culls and draws are set using **pfChanScene** and **pfChanESky**, respectively. The **pfChannel**'s **pfGeoState** and **pfGeoStateTable** can also be specified. Screen to world-space ray intersections on a pfChannel's scene can be performed using **pfChanPick** and related functions.

IRIS Performer can also sort the database into "bins" which are rendered in a user-specified order. In addition, geometry within a bin may be sorted by graphics state like texture or by range for front-to-back or back-to-front rendering. Functions are provided to achieve this behavior (**pfChanBinSort** and friends).

pfChannel*	**pfNewChan**(pfPipe *p);
pfType*	**pfGetChanClassType**(void);
int	**pfGetChanFrustType**(const pfChannel* chan);
void	**pfChanAspect**(pfChannel* chan, int which, float xyaspect);
float	**pfGetChanAspect**(pfChannel* chan);
void	**pfGetChanFOV**(const pfChannel* chan, float *fovH, float *fovV);
void	**pfChanNearFar**(pfChannel* chan, float n, float f);
void	**pfGetChanNearFar**(const pfChannel* chan, float *n, float *f);
void	**pfGetChanNear**(const pfChannel* chan, pfVec3 ll, pfVec3 lr, pfVec3 ul, pfVec3 ur);
void	**pfGetChanFar**(const pfChannel* chan, pfVec3 ll, pfVec3 lr, pfVec3 ul, pfVec3 ur);
void	**pfGetChanPtope**(const pfChannel* chan, pfPolytope *dst);
int	**pfGetChanEye**(const pfChannel* chan, pfVec3 eye);
void	**pfMakePerspChan**(pfChannel* chan, float l, float r, float b, float t);
void	**pfMakeOrthoChan**(pfChannel* chan, float l, float r, float b, float t);
void	**pfMakeSimpleChan**(pfChannel* chan, float fov);
void	**pfOrthoXformChan**(pfChannel* chan, pfFrustum *fr, const pfMatrix mat);
int	**pfChanContainsPt**(const pfChannel* chan, const pfVec3 pt);
int	**pfChanContainsSphere**(const pfChannel* chan, const pfSphere *sphere);
int	**pfChanContainsBox**(const pfChannel* chan, const pfBox *box);
int	**pfChanContainsCyl**(const pfChannel* chan, const pfCylinder *cyl);
void	**pfApplyChan**(pfChannel* chan);
pfPipe*	**pfGetChanPipe**(const pfChannel* chan);
pfPipeWindow*	**pfGetChanPWin**(pfChannel* chan);

int	**pfGetChanPWinIndex**(pfChannel* chan);
void	**pfChanFOV**(pfChannel* chan, float fovH, float fovV);
void	**pfChanViewport**(pfChannel* chan, float l, float r, float b, float t);
void	**pfGetChanViewport**(const pfChannel* chan, float *l, float *r, float *b, float *t);
void	**pfGetChanOrigin**(const pfChannel* chan, int *xo, int *yo);
void	**pfGetChanSize**(const pfChannel* chan, int *xs, int *ys);
void	**pfChanShare**(pfChannel* chan, uint mask);
uint	**pfGetChanShare**(const pfChannel* chan);
void	**pfChanAutoAspect**(pfChannel* chan, int which);
int	**pfGetChanAutoAspect**(const pfChannel* chan);
void	**pfGetChanBaseFrust**(const pfChannel* chan, pfFrustum *frust);
void	**pfChanViewOffsets**(pfChannel* chan, pfVec3 xyz, pfVec3 hpr);
void	**pfGetChanViewOffsets**(const pfChannel* chan, pfVec3 xyz, pfVec3 hpr);
void	**pfChanView**(pfChannel* chan, pfVec3 vp, pfVec3 vd);
void	**pfGetChanView**(pfChannel* chan, pfVec3 vp, pfVec3 vd);
void	**pfChanViewMat**(pfChannel* chan, pfMatrix mat);
void	**pfGetChanViewMat**(const pfChannel* chan, pfMatrix mat);
void	**pfGetChanOffsetViewMat**(const pfChannel* chan, pfMatrix mat);
void	**pfChanCullPtope**(pfChannel* chan, const pfPolytope *vol);
void	**pfGetChanCullPtope**(const pfChannel* chan, pfPolytope *vol);
void*	**pfAllocChanData**(pfChannel* chan, int size);
void	**pfChanData**(pfChannel* chan, void *data, size_t size);
void*	**pfGetChanData**(const pfChannel* chan);
size_t	**pfGetChanDataSize**(const pfChannel* chan);
void	**pfChanTravFunc**(pfChannel* chan, int trav, pfChanFuncType func);
pfChanFuncType	**pfGetChanTravFunc**(const pfChannel* chan, int trav);
void	**pfChanTravMode**(pfChannel* chan, int trav, int mode);
int	**pfGetChanTravMode**(const pfChannel* chan, int trav);
void	**pfChanTravMask**(pfChannel* chan, int which, uint mask);
uint	**pfGetChanTravMask**(const pfChannel* chan, int which);
void	**pfChanStressFilter**(pfChannel* chan, float frac, float low, float high, float s, float max);
void	**pfGetChanStressFilter**(const pfChannel* chan, float *frac, float *low, float *high, float *s, float *max);
void	**pfChanStress**(pfChannel* chan, float stress);
float	**pfGetChanStress**(const pfChannel* chan);
float	**pfGetChanLoad**(const pfChannel* chan);
void	**pfChanScene**(pfChannel* chan, pfScene *s);
pfScene*	**pfGetChanScene**(const pfChannel* chan);
void	**pfChanESky**(pfChannel* chan, pfEarthSky *es);
pfEarthSky*	**pfGetChanESky**(const pfChannel* chan);

```
void            pfChanGState(pfChannel* chan, pfGeoState *gstate);
pfGeoState*     pfGetChanGState(const pfChannel* chan);
void            pfChanGStateTable(pfChannel* chan, pfList *list);
pfList*         pfGetChanGStateTable(const pfChannel* chan);
void            pfChanLODAttr(pfChannel* chan, int attr, float val);
float           pfGetChanLODAttr(const pfChannel* chan, int attr);
void            pfChanLODState(pfChannel* chan, const pfLODState *ls);
void            pfGetChanLODState(const pfChannel* chan, pfLODState *ls);
void            pfChanLODStateList(pfChannel* chan, pfList *stateList);
pfList*         pfGetChanLODStateList(const pfChannel* chan);
int             pfChanStatsMode(pfChannel* chan, uint mode, uint val);
pfFrameStats*   pfGetChanFStats(pfChannel* chan);
void            pfChanBinSort(pfChannel* chan, int bin, int sortType, int *sortOrders);
int             pfGetChanBinSort(pfChannel* chan, int bin, int *sortOrders);
void            pfChanBinOrder(pfChannel* chan, int bin, int order);
int             pfGetChanBinOrder(const pfChannel* chan, int bin);
int             pfAttachChan(pfChannel* chan, pfChannel *chan1);
int             pfDetachChan(pfChannel* chan, pfChannel *chan1);
void            pfPassChanData(pfChannel* chan);
int             pfChanPick(pfChannel* chan, int mode, float px, float py, float radius,
                    pfHit **pickList[]);
void            pfClearChan(pfChannel* chan);
void            pfDrawChanStats(pfChannel* chan);
int             pfChanNodeIsectSegs(pfChannel* chan, pfNode *node, pfSegSet *segSet,
                    pfHit **hits[], pfMatrix *ma);
```

pfEarthSky C API

These functions provide a means to clear the frame and Z-buffer, draw a sky, horizon and ground plane, and to implement various weather effects like fog and clouds.

```
pfEarthSky*     pfNewESky(void);
pfType*         pfGetESkyClassType(void);
void            pfESkyMode(pfEarthSky* esky, int mode, int val);
int             pfGetESkyMode(pfEarthSky* esky, int mode);
void            pfESkyAttr(pfEarthSky* esky, int mode, float val);
float           pfGetESkyAttr(pfEarthSky* esky, int mode);
void            pfESkyColor(pfEarthSky* esky, int which, float r, float g, float b, float a);
void            pfGetESkyColor(pfEarthSky* esky, int which, float *r, float *g, float *b, float *a);
void            pfESkyFog(pfEarthSky* esky, int which, pfFog *fog);
pfFog*          pfGetESkyFog(pfEarthSky* esky, int which);
```

pfNode C API

A **pfNode** is an abstract type which cannot be explicitly created. The **pfNode** routines operate on the common aspects of other IRIS Performer node types which are derived from **pfNode**. IRIS Performer provides four major traversals of the scene graph: **ISECT**, **APP**, **CULL**, and **DRAW**. These functions (-**pfNodeTrav***) can be used to set which nodes are traversed, the functions to be invoked during the traversal, when the traversal is initiated and what data is provided to the traversal.

```
pfType*      pfGetNodeClassType(void);
void         pfNodeTravMask(pfNode* node, int which, uint mask, int setMode, int bitOp);
pfNode*      pfFindNode(pfNode* node, const char *name, pfType *type);
int          pfNodeName(pfNode* node, const char *name);
const char*  pfGetNodeName(const pfNode* node);
void         pfNodeTravFuncs(pfNode* node, int which, pfNodeTravFuncType pre,
                 pfNodeTravFuncType post);
void         pfGetNodeTravFuncs(const pfNode* node, int which, pfNodeTravFuncType *pre,
                 pfNodeTravFuncType *post);
void         pfNodeTravData(pfNode* node, int which, void *data);
void*        pfGetNodeTravData(const pfNode* node, int which);
uint         pfGetNodeTravMask(const pfNode* node, int which);
void         pfNodeBufferMode(pfNode* node, int mode, int val);
int          pfGetNodeBufferMode(const pfNode* node, int mode);
pfGroup*     pfGetParent(const pfNode* node, int i);
int          pfGetNumParents(const pfNode* node);
void         pfNodeBSphere(pfNode* node, pfSphere *sph, int mode);
int          pfGetNodeBSphere(pfNode* node, pfSphere *sph);
pfNode*      pfLookupNode(const char *name, pfType* type);
int          pfNodeIsectSegs(pfNode* node, pfSegSet *segSet, pfHit **hits[]);
int          pfFlatten(pfNode* node, int mode);
pfNode*      pfClone(pfNode* node, int mode);
pfNode*      pfBufferClone(pfNode* node, int mode, pfBuffer *buf);
```

pfGroup C API

A **pfGroup** is the internal node type of the IRIS Performer hierarchy and is derived from **pfNode**. The functions allow children to be added to and deleted from a **pfGroup** node and queries to be made about a **pfGroup** node's children.

```
pfGroup*     pfNewGroup(void);
pfType*      pfGetGroupClassType(void);
int          pfAddChild(pfGroup* group, pfNode *child);
int          pfInsertChild(pfGroup* group, int index, pfNode *child);
```

int **pfRemoveChild**(pfGroup* group, pfNode *child);
int **pfReplaceChild**(pfGroup* group, pfNode *oldn, pfNode *newn);
int **pfBufferAddChild**(pfGroup* group, pfNode *child);
int **pfBufferRemoveChild**(pfGroup* group, pfNode *child);
pfNode* **pfGetChild**(const pfGroup* group, int i);
int **pfGetNumChildren**(const pfGroup* group);
int **pfSearchChild**(const pfGroup* group, pfNode *n);

pfScene C API

A **pfScene** is the root of a hierarchical database which may be drawn or intersected with. **pfGeoStates** can be attached to and removed from a **pfScene**.

pfScene* **pfNewScene**(void);
pfType* **pfGetSceneClassType**(void);
void **pfSceneGState**(pfScene* scene, pfGeoState *gs);
pfGeoState* **pfGetSceneGState**(const pfScene* scene);
void **pfSceneGStateIndex**(pfScene* scene, int gs);
int **pfGetSceneGStateIndex**(const pfScene* scene);

pfSCS C API

These functions manipulate the matrix associated with a **pfSCS** node. A **pfSCS** node represents a static coordinate system -- a modeling transform that cannot be changed once created.

pfSCS* **pfNewSCS**(pfMatrix m);
pfType* **pfGetSCSClassType**(void);
void **pfGetSCSMat**(pfSCS* scs, pfMatrix m);
const pfMatrix* **pfGetSCSMatPtr**(pfSCS* scs);

pfDCS C API

These functions manipulate the matrix associated with a **pfDCS** node. A **pfDCS** node represents a dynamic coordinate system -- a modeling transform that can be changed after it is created.

pfDCS* **pfNewDCS**(void);
pfType* **pfGetDCSClassType**(void);
void **pfGetDCSMat**(pfDCS* dcs, pfMatrix m);
const pfMatrix* **pfGetDCSMatPtr**(pfDCS* dcs);
void **pfDCSMatType**(pfDCS* dcs, uint val);
uint **pfGetDCSMatType**(const pfDCS* dcs);
void **pfDCSMat**(pfDCS* dcs, pfMatrix m);
void **pfDCSCoord**(pfDCS* dcs, pfCoord *c);
void **pfDCSRot**(pfDCS* dcs, float h, float p, float r);
void **pfDCSTrans**(pfDCS* dcs, float x, float y, float z);

void **pfDCSScale**(pfDCS* dcs, float s);
void **pfDCSScaleXYZ**(pfDCS* dcs, float xs, float ys, float zs);

pfLODState C API

A **pfLODState** is a definition of how an LOD or group of LODs should respond to range and stress. The functions form an interface to create LOD states, set their attributes and give them names.

pfLODState* **pfNewLODState**(void);
pfType* **pfGetLODStateClassType**(void);
void **pfLODStateAttr**(pfLODState* lodstate, int attr, float val);
float **pfGetLODStateAttr**(pfLODState* lodstate, int attr);
int **pfLODStateName**(pfLODState* lodstate, const char *name);
const char* **pfGetLODStateName**(const pfLODState* lodstate);
pfLODState* **pfFindLODState**(const char *findName);

pfLOD C API

Level-of-detail is a technique for manipulating model complexity based on image quality and rendering speed. IRIS Performer uses range-based LOD and adjusts for field-of-view and viewport pixel size. Each **pfLOD** node has the different levels-of-detail as its children. The **pfGroup** API can be used to manipulate this child list. A particular LOD is picked based on a transition range. These transition ranges can be set by **pfLODRange** and **pfLODTransition** to ensure smooth transitions between different LODs. A given **pfLOD** can also be associated with a **pfLODState**.

pfLOD* **pfNewLOD**(void);
pfType* **pfGetLODClassType**(void);
void **pfLODCenter**(pfLOD* lod, pfVec3 c);
void **pfGetLODCenter**(const pfLOD* lod, pfVec3 c);
void **pfLODRange**(pfLOD* lod, int index, float range);
int **pfGetLODNumRanges**(const pfLOD* lod);
float **pfGetLODRange**(const pfLOD* lod, int index);
void **pfLODTransition**(pfLOD* lod, int index, float delta);
int **pfGetLODNumTransitions**(const pfLOD* lod);
float **pfGetLODTransition**(const pfLOD* lod, int index);
void **pfLODLODState**(pfLOD* lod, pfLODState *ls);
pfLODState* **pfGetLODLODState**(const pfLOD* lod);
void **pfLODLODStateIndex**(pfLOD* lod, int index);
int **pfGetLODLODStateIndex**(const pfLOD* lod);
float **pfEvaluateLOD**(pfLOD* lod, const pfChannel *chan, const pfMatrix *offset);

pfSwitch C API

The functions manipulate **pfSwitch** nodes which are interior nodes in the IRIS Performer node hierarchy that select one, all, or none of their children. The mode of selection is set by **pfSwitchVal**.

```
pfSwitch*   pfNewSwitch(void);
pfType*     pfGetSwitchClassType(void);
int         pfSwitchVal(pfSwitch* switch, int val);
int         pfGetSwitchVal(const pfSwitch* switch);
```

pfMorph C API

A **pfMorph** node manipulates the geometric attributes of **pfGeoSets** and other geometric primitives. Its primary use is for geometric morphing where the colors, normals, texture coordinates and coordinates of geometry are smoothly changed over time to simulate actions such as facial and skeletal animation, ocean waves, morph level-of-detail, and special effects. The attributes of a **pfMorph** node, the method of accessing the source arrays of a **pfMorph** attribute (non-indexed or indexed) and the weights attached to these attributes can be set and queried by these functions.

```
pfMorph*    pfNewMorph(void);
pfType*     pfGetMorphClassType(void);
int         pfMorphAttr(pfMorph* morph, int index, int attr, int nelts, void *dst, int nsrcs, float *alist[],
                ushort *ilist[], int n[]);
int         pfGetMorphNumAttrs(const pfMorph* morph);
int         pfGetMorphSrc(const pfMorph* morph, int index, int src, float **alist, ushort **ilist, int *n);
int         pfGetMorphNumSrcs(const pfMorph* morph, int index);
void*       pfGetMorphDst(const pfMorph* morph, int index);
int         pfMorphWeights(pfMorph* morph, int index, float *weights);
int         pfGetMorphWeights(const pfMorph* morph, int index, float *weights);
void        pfEvaluateMorph(pfMorph* morph);
```

pfSequence C API

A **pfSequence** node is a **pfGroup** node that sequences through a range of its children, drawing each child for a certain length of time. Children are added to a **pfSequence** using normal **pfGroup** API. The length of time to draw each child and the range of children to sequence through are set by these functions.

```
pfSequence* pfNewSeq(void);
pfType*     pfGetSeqClassType(void);
void        pfSeqDuration(pfSequence* seq, float sp, int nRep);
void        pfGetSeqDuration(const pfSequence* seq, float *sp, int *nRep);
void        pfSeqInterval(pfSequence* seq, int imode, int beg, int e);
void        pfGetSeqInterval(const pfSequence* seq, int *imode, int *beg, int *e);
void        pfSeqMode(pfSequence* seq, int m);
int         pfGetSeqMode(const pfSequence* seq);
void        pfSeqTime(pfSequence* seq, int index, double time);
double      pfGetSeqTime(const pfSequence* seq, int index);
int         pfGetSeqFrame(const pfSequence* seq, int *rep);
```

pfLayer C API

A **pfLayer** is a node derived from **pfGroup** that supports proper drawing of coplanar geometry on IRIS platforms so as to prevent distracting artifacts caused by numerical precision when rendering coplanar geometry on Z-buffer based machines. These functions create **pfLayer**s and define the base layer and the other (decal) layers.

```
pfLayer*  pfNewLayer(void);
pfType*   pfGetLayerClassType(void);
void      pfLayerBase(pfLayer* layer, pfNode *n);
pfNode*   pfGetLayerBase(const pfLayer* layer);
void      pfLayerDecal(pfLayer* layer, pfNode *n);
pfNode*   pfGetLayerDecal(const pfLayer* layer);
void      pfLayerMode(pfLayer* layer, int mode);
int       pfGetLayerMode(const pfLayer* layer);
```

pfPartition C API

A **pfPartition** node is a type of **pfGroup** node which organizes the scene graphs of its children into a static data structure which can be more efficient for intersections. **pfBuildPart** constructs a spatial partitioning based on the value of *type*. The other functions update a partition and control the values of its attributes.

```
pfPartition*  pfNewPart(void);
pfType*       pfGetPartClassType(void);
void          pfPartVal(pfPartition* part, int which, float val);
float         pfGetPartVal(pfPartition* part, int which);
void          pfPartAttr(pfPartition* part, int which, void *attr);
void*         pfGetPartAttr(pfPartition* part, int which);
void          pfBuildPart(pfPartition* part);
void          pfUpdatePart(pfPartition* part);
```

pfLightPoint C API

A **pfLightPoint** is a **pfNode** that contains one or more light points. A light point is visible as one or more self-illuminated small points but does not illuminate surrounding objects. These functions form an interface to create light points and control various light point parameters like size, number, shape, direction, color, position and intensity in a fog.

```
pfLightPoint*  pfNewLPoint(int n);
pfType*        pfGetLPointClassType(void);
int            pfGetNumLPoints(const pfLightPoint* lpoint);
void           pfLPointSize(pfLightPoint* lpoint, float s);
float          pfGetLPointSize(const pfLightPoint* lpoint);
```

void	**pfLPointFogScale**(pfLightPoint* lpoint, float onset, float opaque);
void	**pfGetLPointFogScale**(const pfLightPoint* lpoint, float *onset, float *opaque);
void	**pfLPointRot**(pfLightPoint* lpoint, float azim, float elev, float roll);
void	**pfGetLPointRot**(const pfLightPoint* lpoint, float *azim, float *elev, float *roll);
void	**pfLPointShape**(pfLightPoint* lpoint, int dir, float he, float ve, float f);
void	**pfGetLPointShape**(const pfLightPoint* lpoint, int *dir, float *he, float *ve, float *f);
pfGeoSet*	**pfGetLPointGSet**(const pfLightPoint* lpoint);
void	**pfLPointPos**(pfLightPoint* lpoint, int i, pfVec3 p);
void	**pfGetLPointPos**(const pfLightPoint* lpoint, int i, pfVec3 p);
void	**pfLPointColor**(pfLightPoint* lpoint, int i, pfVec4 clr);
void	**pfGetLPointColor**(const pfLightPoint* lpoint, int i, pfVec4 clr);

pfLightSource C API

A **pfLightSource** is a **pfNode** which can illuminate geometry in a **pfScene**. The **pfLightSource** routines create **pfLightSource**s,

pfLightSource*	**pfNewLSource**(void);
pfType*	**pfGetLSourceClassType**(void);
void	**pfLSourceColor**(pfLightSource* lsource, int which, float r, float g, float b);
void	**pfGetLSourceColor**(pfLightSource* lsource, int which, float* r, float* g, float* b);
void	**pfLSourceAmbient**(pfLightSource* lsource, float r, float g, float b);
void	**pfGetLSourceAmbient**(pfLightSource* lsource, float* r, float* g, float* b);
void	**pfLSourcePos**(pfLightSource* lsource, float x, float y, float z, float w);
void	**pfGetLSourcePos**(pfLightSource* lsource, float* x, float* y, float* z, float* w);
void	**pfLSourceAtten**(pfLightSource* lsource, float a0, float a1, float a2);
void	**pfGetLSourceAtten**(pfLightSource* lsource, float* a0, float* a1, float* a2);
void	**pfSpotLSourceDir**(pfLightSource* lsource, float x, float y, float z);
void	**pfGetSpotLSourceDir**(pfLightSource* lsource, float* x, float* y, float* z);
void	**pfSpotLSourceCone**(pfLightSource* lsource, float f1, float f2);
void	**pfGetSpotLSourceCone**(pfLightSource* lsource, float* f1, float* f2);
void	**pfLSourceOn**(pfLightSource* lsource);
void	**pfLSourceOff**(pfLightSource* lsource);
int	**pfIsLSourceOn**(pfLightSource* lsource);
void	**pfLSourceMode**(pfLightSource* lsource, int mode, int val);
int	**pfGetLSourceMode**(const pfLightSource* lsource, int mode);
void	**pfLSourceVal**(pfLightSource* lsource, int mode, float val);
float	**pfGetLSourceVal**(const pfLightSource* lsource, int mode);
void	**pfLSourceAttr**(pfLightSource* lsource, int attr, void *obj);
void*	**pfGetLSourceAttr**(const pfLightSource* lsource, int attr);

pfGeode C API

A **pfGeode** is a leaf node in the IRIS Performer scene graph hierarchy. It is a list of **pfGeoSet**s which it draws and intersects with. Functions are provided to creates **pfGeode** and manipulate the list of **pfGeoStates** attached to them.

pfGeode* **pfNewGeode**(void);
pfType* **pfGetGeodeClassType**(void);
int **pfAddGSet**(pfGeode* geode, pfGeoSet *gset);
int **pfInsertGSet**(pfGeode* geode, int index, pfGeoSet *gset);
int **pfReplaceGSet**(pfGeode* geode, pfGeoSet *oldgs, pfGeoSet *newgs);
int **pfRemoveGSet**(pfGeode* geode, pfGeoSet *gset);
pfGeoSet* **pfGetGSet**(const pfGeode* geode, int i);
int **pfGetNumGSets**(const pfGeode* geode);

pfText C API

A **pfText** node is a list of **pfString**s much as a **pfGeode** is a list of **pfGeoSet**s. The two APIs are also similar - a new **pfText** node can be created and the list of **pfString**s attached to it can be manipulated by addition, insertion, removal or replacement.

pfText* **pfNewText**(void);
pfType* **pfGetTextClassType**(void);
int **pfAddString**(pfText* text, pfString *str);
int **pfInsertString**(pfText* text, int index, pfString *str);
int **pfReplaceString**(pfText* text, pfString *oldgs, pfString *newgs);
int **pfRemoveString**(pfText* text, pfString *str);
pfString* **pfGetString**(const pfText* text, int i);
int **pfGetNumStrings**(const pfText* text);

pfBillboard C API

A **pfBillboard** is a **pfGeode** in which each **pfGeoSet** rotates to follow the eyepoint. Billboards are useful for representing complex objects which are roughly symmetrical about one or more axes. A **pfBillboard** can contain any number of **pfGeoSets** which can be added to and removed from the **pfBillboard** using **pfGeode** API. Further, the position, mode and axis of rotation of a **pfBillboard** can also be manipulated.

pfBillboard* **pfNewBboard**(void);
pfType* **pfGetBboardClassType**(void);
void **pfBboardAxis**(pfBillboard* bboard, const pfVec3 axis);
void **pfGetBboardAxis**(pfBillboard* bboard, pfVec3 axis);
void **pfBboardMode**(pfBillboard* bboard, int mode, int val);
int **pfGetBboardMode**(pfBillboard* bboard, int mode);
void **pfBboardPos**(pfBillboard* bboard, int i, const pfVec3 pos);
void **pfGetBboardPos**(pfBillboard* bboard, int i, pfVec3 pos);

pfPath C API

A **pfPath** is a dynamically-sized array of **pfNode** pointers that defines a specific path or chain of nodes through a scene graph. **pfNewPath** creates a new path.

pfPath* **pfNewPath**(void);
pfType* **pfGetPathClassType**(void);
void **pfCullResult**(int result);
int **pfGetParentCullResult**(void);
int **pfGetCullResult**(void);
int **pfCullPath**(pfPath *path, pfNode *root, int mode);

pfTraverser C API

These functions are provided as a means to obtain information about the behavior of the IRIS Performer traversal routines. They can be used to determine the **pfChannel** or **pfNode** currently being culled or drawn, set the matrix for the current traversal, determine the path from the root of the scene graph to the node currently being traversed and the results of culling the node currently being traversed and the parent of the current node.

pfChannel* **pfGetTravChan**(const pfTraverser* trav);
pfNode* **pfGetTravNode**(const pfTraverser* trav);
void **pfGetTravMat**(const pfTraverser* trav, pfMatrix mat);
int **pfGetTravIndex**(const pfTraverser* trav);
const pfPath* **pfGetTravPath**(const pfTraverser* trav);

pfFrameStats C API

A **pfFrameStats** structure contains a **pfStats** class as well as additional statistics classes and support for tracking frame related tasks. Many of the functions correspond directly to similar functions for the **pfStats** class.

pfFrameStats* **pfNewFStats**(void);
pfType* **pfGetFStatsClassType**(void);
uint **pfFStatsClass**(pfFrameStats *fstats, uint mask, int val);
uint **pfGetFStatsClass**(pfFrameStats *fstats, uint emask);
uint **pfFStatsClassMode**(pfFrameStats *fstats, int class, uint mask, int val);
uint **pfGetFStatsClassMode**(pfFrameStats *fstats, int class);
void **pfFStatsAttr**(pfFrameStats *fstats, int attr, float val);
float **pfGetFStatsAttr**(pfFrameStats *fstats, int attr);
uint **pfGetOpenFStats**(pfFrameStats *fstats, uint emask);
uint **pfOpenFStats**(pfFrameStats *fstats, uint enmask);
uint **pfCloseFStats**(uint enmask);
void **pfResetFStats**(pfFrameStats *fstats);
void **pfClearFStats**(pfFrameStats *fstats, uint which);
void **pfAccumulateFStats**(pfFrameStats *fstats, pfFrameStats* src, uint which);
void **pfAverageFStats**(pfFrameStats *fstats, pfFrameStats* src, uint which, int num);
void **pfFStatsCountGSet**(pfFrameStats *fstats, pfGeoSet *gset);

int **pfQueryFStats**(pfFrameStats *fstats, uint which, void *dst, int size);
int **pfMQueryFStats**(pfFrameStats *fstats, uint *which, void *dst, int size);
void **pfDrawFStats**(pfFrameStats *fstats, pfChannel *chan);
void **pfFStatsCountNode**(pfFrameStats *fstats, int class, uint mode, pfNode * node);

LIBPR

Initialization Routines

These routines initialize and configure Performer to use multiple processors and graphics pipelines. All libpf appliciations must call **pfInit** and **pfConfig** before creating a scene graph or initiating rendering with **pfFrame**. **pfInit** initializes shared memory and the clock. **pfConfig** creates multiple processes based on the requested configuration and sets up internal data structures for frame-accurate propagation of data between the processes.

void **prInit**(void);
void **prExit**(void);

Shared Memory

This is an interface to creating and manipulating a shared memory area to house the data structures shared by the different IRIS Performer processes. **pfInitArenas** creates a shared memory arena that can be used to allocate memory, locks and semaphores from. The other functions free this arena, control the directory where it is created, return handles to the shared memory and the semaphore memory and set the base address and size of these shared memory areas.

int **pfInitArenas**(void);
int **pfFreeArenas**(void);
PF_USPTR_T*
pfGetSemaArena(void);
void **pfSemaArenaSize**(size_t size);
size_t **pfGetSemaArenaSize**(void);
void **pfSemaArenaBase**(void *base);
void* **pfGetSemaArenaBase**(void);
void* **pfGetSharedArena**(void);
void **pfSharedArenaSize**(size_t size);
size_t **pfGetSharedArenaSize**(void);
void **pfSharedArenaBase**(void *base);
void* **pfGetSharedArenaBase**(void);
void **pfTmpDir**(char *dir);
const char * **pfGetTmpDir**(void);

Draw Modes

IRIS Performer supports a large number of drawing modes like shading, transparency, anti-aliasing and coplanar geometry. These functions define these modes and enable and disable them.

void **pfShadeModel**(int model);
int **pfGetShadeModel**(void);
void **pfTransparency**(int type);
int **pfGetTransparency**(void);
void **pfAlphaFunc**(float ref, int func);
void **pfGetAlphaFunc**(float* ref, int* func);
void **pfAntialias**(int type);
int **pfGetAntialias**(void);
void **pfDecal**(int mode);
int **pfGetDecal**(void);
void **pfCullFace**(int cull);
int **pfGetCullFace**(void);
void **pfEnable**(int target);
void **pfDisable**(int target);
int **pfGetEnable**(int target);
void **pfClear**(int which, const pfVec4 col);
void **pfClear**(int which, const pfVec4 *col);
void **pfGLOverride**(int mode, float val);
float **pfGetGLOverride**(int mode);

GL Matrix Stack

These functions operate on the graphics library matrix stack. Various standard operations on matrices are supported.

void **pfScale**(float x, float y, float z);
void **pfTranslate**(float x, float y, float z);
void **pfRotate**(int axis, float degrees);
void **pfPushMatrix**(void);
void **pfPushIdentMatrix**(void);
void **pfPopMatrix**(void);
void **pfLoadMatrix**(const pfMatrix m);
void **pfMultMatrix**(const pfMatrix m);

Notification

These functions provide a general purpose error message and notification handling facility for applications using IRIS Performer. User-defined functions can be used as notifiers.

void **pfNotifyHandler**(pfNotifyFuncType handler);
pfNotifyFuncType **pfGetNotifyHandler**(void);
void **pfDefaultNotifyHandler**(pfNotifyData *notice);
void **pfNotifyLevel**(int severity);

int **pfGetNotifyLevel**(void);
void **pfNotify**(int severity, int error, char *format,

Clock Routines

These routines provide a simple and consistent interface to the high resolution hardware-specific timers available on most SGI platforms.

double **pfGetTime**(void);
pid_t **pfInitClock**(double time);
void **pfWrapClock**(void);
void **pfClockName**(char *name);
const char* **pfGetClockName**(void);
void **pfClockMode**(int mode);
int **pfGetClockMode**(void);

File Paths

These functions can be used to specify a UNIX-style file path to search for files in and to find files in such a path.

void **pfFilePath**(const char* path);
const char* **pfGetFilePath**(void);
int **pfFindFile**(const char* file, char path[PF_MAXSTRING], int amode);

Video Clock Routines

These functions provide an interface to the video retrace clock attached to each graphics pipeline. Once a video clock is initialised, its current value can be determined and it can be used to synchronize a process with a time barrier.

int **pfStartVClock**(void);
void **pfStopVClock**(void);
void **pfInitVClock**(int ticks);
void **pfVClockOffset**(int offset);
int **pfGetVClockOffset**(void);
int **pfGetVClock**(void);
int **pfVClockSync**(int rate, int offset);

pfWindow Routines

IRIS Performer provides a system-independent window paradigm. The **prInitGfx** function may be called to initialize the graphics subsystem and acquire the graphics attributes Performer requires. Use **pfGetCurWin** to gain access to the current window.

void **prInitGfx**(void);

pfWindow * **pfGetCurWin**(void);

Window System Routines

The **pfWSConnection** data structure encapsulates the workstation-independent frame-buffer (window) facility in IRIS Performer. These functions serve to define specific windowing attributes necessary for the application, to open and close windows, and to manipulate the window parameters.

void	**pfCloseWSConnection**(pfWSConnection dsp);
pfFBConfig	**pfChooseFBConfig**(pfWSConnection dsp, int screen, int *attr);
pfFBConfig	**pfChooseFBConfigData**(void **dst, pfWSConnection dsp, int screen, int *attr, void *arena);
void	**pfSelectWSConnection**(pfWSConnection);
pfWSConnection	**pfOpenWSConnection**(const char *str, int shared);
pfWSConnection	**pfOpenScreen**(int screen, int shared);
pfWSConnection	**pfGetCurWSConnection**(void);
const char*	**pfGetWSConnectionName**(pfWSConnection);
void	**pfGetScreenSize**(int screen, int *x, int *y);

Query Features

Use the QueryFeature routines to determine the presence, absence, or limitations of features in the underlying graphics implementation, like the availability of attenuation in the lighting model or the availability of multiple graphics pipes.

int	**pfQueryFeature**(int which, int *dst);
int	**pfMQueryFeature**(int *which, int *dst);
void	**pfFeature**(int which, int val);

Query System

Use the QuerySys routines to determine the capacity and limitations of the underlying graphics implementation, like the size of texture memory or the number of stencil planes available.

int	**pfQuerySys**(int which, int *dst);
int	**pfMQuerySys**(int *which, int *dst);

pfObject C API

A **pfObject** is the abstract data type from which the major IRIS Performer data structures are derived. Although **pfObject**s cannot be created directly, most IRIS Performer data structures are derived from them and thus inherit the functionality of the **pfObject** routines and those for **pfMemory**.

pfType*	**pfGetObjectClassType**(void);
void	**pfCopyFunc**(pfCopyFuncType func);
pfCopyFuncType	**pfGetCopyFunc**(void);

void	**pfDeleteFunc**(pfDeleteFuncType func);
pfMergeFuncType	**pfGetMergeFunc**(void);
void	**pfMergeFunc**(pfMergeFuncType func);
pfDeleteFuncType	**pfGetDeleteFunc**(void);
void	**pfPrintFunc**(pfPrintFuncType func);
pfPrintFuncType	**pfGetPrintFunc**(void);
int	**pfGetGLHandle**(const pfObject *obj);
void	**pfUserData**(pfObject* obj, void* data);
void*	**pfGetUserData**(pfObject* obj);

pfType C API

All IRIS Performer data types that derive from **pfObject**/**pfMemory** have an associated **pfType**. The **pfType** can be used to determine the class ancestory of both built-in and add-on data types.

pfType*	**pfNewType**(pfType *parent, char *name);
pfType*	**pfGetTypeParent**(pfType* type);
int	**pfIsDerivedFrom**(pfType* type, pfType *ancestor);
void	**pfMaxTypes**(int n);
pfFog*	**pfGetCurFog**(void);

pfFog C API

pfFog is used to simulate atmospheric phenomena such as fog and haze and for depthcueing. The fog color is blended with the color that is computed for rendered geometry based on the geometry's range from the eyepoint. IRIS Performer provides functions for defining fog color, ranges, and other attributes.

pfFog*	**pfNewFog**(void *arena);
pfType*	**pfGetFogClassType**(void);
void	**pfFogType**(pfFog* fog, int type);
int	**pfGetFogType**(const pfFog* fog);
void	**pfFogRange**(pfFog* fog, float onset, float opaque);
void	**pfGetFogRange**(const pfFog* fog, float* onset, float* opaque);
void	**pfFogOffsets**(pfFog* fog, float onset, float opaque);
void	**pfGetFogOffsets**(const pfFog* fog, float *onset, float *opaque);
void	**pfFogRamp**(pfFog* fog, int points, float* range, float* density, float bias);
void	**pfGetFogRamp**(const pfFog* fog, int* points, float* range, float* density, float* bias);
void	**pfFogColor**(pfFog* fog, float r, float g, float b);
void	**pfGetFogColor**(const pfFog* fog, float* r, float* g, float* b);
float	**pfGetFogDensity**(const pfFog* fog, float range);
void	**pfApplyFog**(pfFog* fog);
pfColortable*	**pfGetCurCtab**(void);

pfColortable C API

A **pfColortable** is a 'color indexing' mechanism used by **pfGeoSets**. **pfGeoSets** can be drawn with the colors defined in the current globally active **pfColortable** rather than by using the **pfGeoset**'s own local color list. This facility can be used for instant large-scale color manipulation of geometry in a scene.

pfColortable* **pfNewCtab**(int size, void *arena);
pfType* **pfGetCtabClassType**(void);
int **pfGetCtabSize**(const pfColortable* ctab);
int **pfCtabColor**(pfColortable* ctab, int index, pfVec4 acolor);
int **pfGetCtabColor**(const pfColortable* ctab, int index, pfVec4 acolor);
pfVec4* **pfGetCtabColors**(const pfColortable* ctab);
void **pfApplyCtab**(pfColortable* ctab);

pfDataPool C API

A **pfDataPool** is similar to a shared memory malloc arena but adds the ability to lock/unlock **pfDataPool** memory for multiprocessing applications. The **pfDataPool** functions allow related or unrelated processes to share data and provide a means for locking data blocks to eliminate data collision.

pfDataPool* **pfCreateDPool**(uint size, char* name);
pfDataPool* **pfAttachDPool**(char* name);
pfType* **pfGetDPoolClassType**(void);
const char* **pfGetDPoolName**(pfDataPool* dpool);
void **pfDPoolAttachAddr**(void *addr);
void* **pfGetDPoolAttachAddr**(void);
int **pfGetDPoolSize**(pfDataPool* dpool);
volatile void* **pfDPoolAlloc**(pfDataPool* dpool, uint size, int id);
volatile void* **pfDPoolFind**(pfDataPool* dpool, int id);
int **pfDPoolFree**(pfDataPool* dpool, void* dpmem);
int **pfReleaseDPool**(pfDataPool* dpool);
int **pfDPoolLock**(void* dpmem);
int **pfDPoolSpinLock**(void* dpmem, int spins, int block);
void **pfDPoolUnlock**(void* dpmem);
int **pfDPoolTest**(void* dpmem);
pfDispList* **pfGetCurDList**(void);
void **pfDrawGLObj**(GLOBJECT obj);

pfDispList C API

A **pfDispList** is a display list that once open, captures certain libpr commands, such as **pfTransparency**, **pfApplyTex**, or **pfDrawGSet**. After it is closed, it may be executed through Performer to perform the recorded commands. **pfDispList**s are designed for multiprocessing, where one process builds a display list of the visible scene and another process draws it.

pfDispList* **pfNewDList**(int type, int size, void *arena);
pfType* **pfGetDListClassType**(void);
int **pfGetDListSize**(const pfDispList* dlist);
int **pfGetDListType**(const pfDispList* dlist);

int **pfDrawDList**(pfDispList* dlist);
void **pfOpenDList**(pfDispList* dlist);
void **pfCloseDList**(void);
void **pfResetDList**(pfDispList* dlist);
void **pfAddDListCmd**(int cmd);
void **pfDListCallback**(pfDListFuncType callback, int bytes, void* data);

pfFont C API

The **pfFont** facility provides the capability to load fonts for 3-D rendering with the string drawing routines from **pfString** and **pfText**. IRIS Performer uses this facility to provide wireframe, flat, extruded, and textured-quad fonts in three dimensions.

pfFont* **pfNewFont**(void *arena);
pfType* **pfGetFontClassType**(void);
void **pfFontCharGSet**(pfFont* font, int ascii, pfGeoSet *gset);
pfGeoSet* **pfGetFontCharGSet**(pfFont* font, int ascii);
void **pfFontCharSpacing**(pfFont* font, int ascii, pfVec3 spacing);
const pfVec3* **pfGetFontCharSpacing**(pfFont* font, int ascii);
void **pfFontAttr**(pfFont* font, int which, void *attr);
void* **pfGetFontAttr**(pfFont* font, int which);
void **pfFontVal**(pfFont* font, int which, float val);
float **pfGetFontVal**(pfFont* font, int which);
void **pfFontMode**(pfFont* font, int mode, int val);
int **pfGetFontMode**(pfFont* font, int mode);

pfGeoSet C API

The **pfGeoSet** (short for "Geometry Set") is a fundamental IRIS Performer data structure. Each **pfGeoSet** is a collection of geometry with one primitive type, such as points, lines, triangles, and homogeneous attribute bindings, such as "untextured with colors per vertex and normals per primitive," so that each **pfGeoSet** may be presented to the graphics subsystem with as little overhead as possible, using an optimized draw routine, one for each type of **pfGeoSet**.

pfGeoSet* **pfNewGSet**(void *arena);
pfType* **pfGetGSetClassType**(void);
void **pfGSetNumPrims**(pfGeoSet* gset, int n);
int **pfGetGSetNumPrims**(const pfGeoSet* gset);
void **pfGSetPrimType**(pfGeoSet* gset, int type);
int **pfGetGSetPrimType**(const pfGeoSet* gset);
void **pfGSetPrimLengths**(pfGeoSet* gset, int *lengths);
int* **pfGetGSetPrimLengths**(const pfGeoSet* gset);
void **pfGSetAttr**(pfGeoSet* gset, int attr, int bind, void* alist, ushort* ilist);

int	**pfGetGSetAttrBind**(const pfGeoSet* gset, int attr);
void	**pfGetGSetAttrLists**(const pfGeoSet* gset, int attr, void** alist, ushort** ilist);
int	**pfGetGSetAttrRange**(const pfGeoSet* gset, int attr, int *min, int *max);
void	**pfGSetDrawMode**(pfGeoSet* gset, int mode, int val);
int	**pfGetGSetDrawMode**(const pfGeoSet* gset, int mode);
void	**pfGSetGState**(pfGeoSet* gset, pfGeoState *gstate);
pfGeoState*	**pfGetGSetGState**(const pfGeoSet* gset);
void	**pfGSetGStateIndex**(pfGeoSet* gset, int id);
int	**pfGetGSetGStateIndex**(const pfGeoSet* gset);
void	**pfGSetHlight**(pfGeoSet* gset, pfHighlight *hlight);
pfHighlight*	**pfGetGSetHlight**(const pfGeoSet* gset);
void	**pfGSetLineWidth**(pfGeoSet* gset, float width);
float	**pfGetGSetLineWidth**(const pfGeoSet* gset);
void	**pfGSetPntSize**(pfGeoSet* gset, float s);
float	**pfGetGSetPntSize**(const pfGeoSet* gset);
void	**pfGSetIsectMask**(pfGeoSet* gset, uint mask, int setMode, int bitOp);
uint	**pfGetGSetIsectMask**(const pfGeoSet* gset);
void	**pfGSetDrawBin**(pfGeoSet* gset, short bin);
int	**pfGetGSetDrawBin**(const pfGeoSet* gset);
void	**pfGSetBBox**(pfGeoSet* gset, pfBox* box, int mode);
int	**pfGetGSetBBox**(pfGeoSet* gset, pfBox* box);
void	**pfDrawGSet**(pfGeoSet* gset);
int	**pfQueryGSet**(const pfGeoSet* gset, uint which, void *dst);
int	**pfMQueryGSet**(const pfGeoSet* gset, uint *which, void *dst);
int	**pfGSetIsectSegs**(pfGeoSet* gset, pfSegSet *segSet, pfHit **hits[]);
void	**pfDrawHlightedGSet**(pfGeoSet* gset);
void	**pfGSetPassFilter**(uint mask);
uint	**pfGetGSetPassFilter**(void);

pfHit C API

These routines support the testing of intersections of line segments with geometry in **pfGeoSets**.

pfType*	**pfGetHitClassType**(void);
int	**pfQueryHit**(const pfHit* hit, uint which, void *dst);
int	**pfMQueryHit**(const pfHit* hit, uint *which, void *dst);
pfGeoState*	**pfGetCurGState**(void);
pfGeoState*	**pfGetCurIndexedGState**(int index);
pfList*	**pfGetCurGStateTable**(void);

pfGeoState C API

pfGeoState is an encapsulation of libpr graphics modes and attributes, and is normally bound to **pfGeoSet**s. The **pfGeoState** represents a complete graphics state, allowing IRIS Performer to draw **pfGeoSets** in an arbitrary order and evaluate state changes in a lazy fashion to reduce overhead caused by changing graphics state.

```
pfGeoState*     pfNewGState(void *arena);
pfType*         pfGetGStateClassType(void);
void            pfGStateMode(pfGeoState* gstate, int attr, int a);
int             pfGetGStateMode(const pfGeoState* gstate, int attr);
int             pfGetGStateCurMode(const pfGeoState* gstate, int attr);
int             pfGetGStateCombinedMode(const pfGeoState* gstate, int attr,
                    const pfGeoState *combState);
void            pfGStateVal(pfGeoState* gstate, int attr, float a);
float           pfGetGStateVal(const pfGeoState* gstate, int attr);
float           pfGetGStateCurVal(const pfGeoState* gstate, int attr);
float           pfGetGStateCombinedVal(const pfGeoState* gstate, int attr,
                    const pfGeoState *combState);
void            pfGStateInherit(pfGeoState* gstate, uint mask);
uint            pfGetGStateInherit(const pfGeoState* gstate);
void            pfGStateAttr(pfGeoState* gstate, int attr, void* a);
void*           pfGetGStateAttr(const pfGeoState* gstate, int attr);
void*           pfGetGStateCurAttr(const pfGeoState* gstate, int attr);
void*           pfGetGStateCombinedAttr(const pfGeoState* gstate, int attr,
                    const pfGeoState *combState);
void            pfLoadGState(pfGeoState* gstate);
void            pfApplyGState(pfGeoState* gstate);
void            pfMakeBasicGState(pfGeoState* gstate);
void            pfApplyGStateTable(pfList *gstab);
pfHighlight *   pfGetCurHlight(void);
```

pfHighlight C API

IRIS Performer supports a mechanism for highlighting individual objects in a scene with a variety of special drawing styles that are activated by applying a **pfHighlight** state structure. Highlighting makes use of outlining of lines and polygons and of filling polygons with patterned or textured overlays.

```
pfHighlight*    pfNewHlight(void *arena);
pfType*         pfGetHlightClassType(void);
void            pfHlightMode(pfHighlight* hlight, uint mode);
uint            pfGetHlightMode(const pfHighlight* hlight);
pfGeoState*     pfGetHlightGState(const pfHighlight* hlight);
void            pfHlightGState(pfHighlight* hlight, pfGeoState *gstate);
void            pfHlightGStateIndex(pfHighlight* hlight, int id);
int             pfGetHlightGStateIndex(const pfHighlight* hlight);
void            pfHlightColor(pfHighlight* hlight, uint which, float r, float g, float b);
void            pfGetHlightColor(const pfHighlight* hlight, uint which, float *r, float *g, float *b);
```

void	**pfHlightAlpha**(pfHighlight* hlight, float a);
float	**pfGetHlightAlpha**(const pfHighlight* hlight);
void	**pfHlightNormalLength**(pfHighlight* hlight, float len, float bboxScale);
void	**pfGetHlightNormalLength**(const pfHighlight* hlight, float *len, float *bboxScale);
void	**pfHlightLineWidth**(pfHighlight* hlight, float width);
float	**pfGetHlightLineWidth**(const pfHighlight* hlight);
void	**pfHlightPntSize**(pfHighlight* hlight, float size);
float	**pfGetHlightPntSize**(const pfHighlight* hlight);
void	**pfHlightLinePat**(pfHighlight* hlight, int which, ushort pat);
ushort	**pfGetHlightLinePat**(const pfHighlight* hlight, int which);
void	**pfHlightFillPat**(pfHighlight* hlight, int which, uint *fillPat);
void	**pfGetHlightFillPat**(const pfHighlight* hlight, int which, uint *pat);
void	**pfHlightTex**(pfHighlight* hlight, pfTexture *tex);
pfTexture*	**pfGetHlightTex**(const pfHighlight* hlight);
void	**pfHlightTEnv**(pfHighlight* hlight, pfTexEnv *tev);
pfTexEnv*	**pfGetHlightTEnv**(const pfHighlight* hlight);
void	**pfHlightTGen**(pfHighlight* hlight, pfTexGen *tgen);
pfTexGen*	**pfGetHlightTGen**(const pfHighlight* hlight);
void	**pfApplyHlight**(pfHighlight* hlight);
int	**pfGetCurLights**(pfLight *lights[PF_MAX_LIGHTS]);

pfLight C API

A **pfLight** is a light source that illuminates scene geometry, generating realistic shading effects. A **pfLight** cannot itself be seen but attributes such as color, spotlight direction, and position can be set to provide illuminative effects on scene geometry.

pfLight*	**pfNewLight**(void *arena);
pfType*	**pfGetLightClassType**(void);
void	**pfLightColor**(pfLight* light, int which, float r, float g, float b);
void	**pfGetLightColor**(const pfLight* light, int which, float* r, float* g, float* b);
void	**pfLightAmbient**(pfLight* light, float r, float g, float b);
void	**pfGetLightAmbient**(const pfLight* light, float* r, float* g, float* b);
void	**pfLightPos**(pfLight* light, float x, float y, float z, float w);
void	**pfGetLightPos**(const pfLight* light, float* x, float* y, float* z, float* w);
void	**pfLightAtten**(pfLight* light, float a0, float a1, float a2);
void	**pfGetLightAtten**(const pfLight* light, float* a0, float* a1, float* a2);
void	**pfSpotLightDir**(pfLight* light, float x, float y, float z);
void	**pfGetSpotLightDir**(const pfLight* light, float* x, float* y, float* z);
void	**pfSpotLightCone**(pfLight* light, float f1, float f2);
void	**pfGetSpotLightCone**(const pfLight* light, float* f1, float* f2);

```
void            pfLightOn(pfLight* light);
void            pfLightOff(pfLight* light);
int             pfIsLightOn(pfLight* light);
pfLightModel*   pfGetCurLModel(void);
```

pfLightModel C API

A **pfLightModel** defines characteristics of the hardware lighting model used to illuminate geometry, such as attenuation, local vs. global lighting model, and ambient energy.

```
pfLightModel*   pfNewLModel(void *arena);
pfType*         pfGetLModelClassType(void);
void            pfLModelLocal(pfLightModel* lmodel, int l);
int             pfGetLModelLocal(const pfLightModel* lmodel);
void            pfLModelTwoSide(pfLightModel* lmodel, int t);
int             pfGetLModelTwoSide(const pfLightModel* lmodel);
void            pfLModelAmbient(pfLightModel* lmodel, float r, float g, float b);
void            pfGetLModelAmbient(const pfLightModel* lmodel, float* r, float* g, float* b);
void            pfLModelAtten(pfLightModel* lmodel, float a0, float a1, float a2);
void            pfGetLModelAtten(const pfLightModel* lmodel, float* a0, float* a1, float* a2);
void            pfApplyLModel(pfLightModel* lmodel);
pfLPointState*  pfGetCurLPState(void);
```

pfLPointState C API

A **pfLPointState** is a **libpr** data structure which, in conjunction with a **pfGeoSet** of type **PFGS_POINTS**, supports a sophisticated light point primitive type. Examples of light points are stars, beacons, strobes, and taxiway lights. Light points are different from light sources in that a **pfLight** is not itself visible but illuminates scene geometry, whereas a light point is visible as a self-illuminated small point that does not illuminate surrounding objects.

```
pfLPointState*  pfNewLPState(void *arena);
pfType*         pfGetLPStateClassType(void);
void            pfLPStateMode(pfLPointState* lpstate, int mode, int val);
int             pfGetLPStateMode(const pfLPointState* lpstate, int mode);
void            pfLPStateVal(pfLPointState* lpstate, int attr, float val);
float           pfGetLPStateVal(const pfLPointState* lpstate, int attr);
void            pfLPStateShape(pfLPointState* lpstate, float horiz, float vert, float roll, float falloff,
                    float ambient);
void            pfGetLPStateShape(const pfLPointState* lpstate, float *horiz, float *vert, float *roll,
                    float *falloff, float *ambient);
void            pfLPStateBackColor(pfLPointState* lpstate, float r, float g, float b, float a);
void            pfGetLPStateBackColor(pfLPointState* lpstate, float *r, float *g, float *b, float *a);
```

```
void        pfApplyLPState(pfLPointState* lpstate);
void        pfMakeLPStateRangeTex(pfLPointState* lpstate, pfTexture *tex, int size, pfFog* fog);
void        pfMakeLPStateShapeTex(pfLPointState* lpstate, pfTexture *tex, int size);
pfMaterial* pfGetCurMtl(int side);
```

pfMaterial C API

In conjunction with other lighting parameters, a **pfMaterial** defines the appearance of illuminated geometry. A **pfMaterial** defines the reflectance characteristics of surfaces such as diffuse color and shininess.

```
pfMaterial* pfNewMtl(void *arena);
pfType*     pfGetMtlClassType(void);
void        pfMtlSide(pfMaterial* mtl, int side);
int         pfGetMtlSide(pfMaterial* mtl);
void        pfMtlAlpha(pfMaterial* mtl, float alpha);
float       pfGetMtlAlpha(pfMaterial* mtl);
void        pfMtlShininess(pfMaterial* mtl, float shininess);
float       pfGetMtlShininess(pfMaterial* mtl);
void        pfMtlColor(pfMaterial* mtl, int acolor, float r, float g, float b);
void        pfGetMtlColor(pfMaterial* mtl, int acolor, float* r, float* g, float* b);
void        pfMtlColorMode(pfMaterial* mtl, int side, int mode);
int         pfGetMtlColorMode(pfMaterial* mtl, int side);
void        pfApplyMtl(pfMaterial* mtl);
pfSprite*   pfGetCurSprite(void);
```

pfSprite C API

pfSprite is an intelligent transformation and is logically grouped with other libpr transformation primitives like **pfMultMatrix**. **pfSprite** rotates geometry orthogonal to the viewer, so the viewer only sees the "front" of the model. As a result, complexity is saved in the model by omitting the "back" geometry. A further performance enhancement is to incorporate visual complexity in a texture map rather than in geometry. Thus, on machines with fast texture mapping, sprites can present very complex images with very little geometry. Classic examples of textured sprites use a single quadrilateral that when rotated about a vertical axis simulate trees and when rotated about a point simulate clouds or puffs of smoke.

```
pfSprite*   pfNewSprite(void *arena);
pfType*     pfGetSpriteClassType(void);
void        pfSpriteMode(pfSprite* sprite, int which, int val);
int         pfGetSpriteMode(const pfSprite* sprite, int which);
void        pfSpriteAxis(pfSprite* sprite, float x, float y, float z);
void        pfGetSpriteAxis(pfSprite* sprite, float *x, float *y, float *z);
void        pfBeginSprite(pfSprite* sprite);
```

void **pfEndSprite**(void);
void **pfPositionSprite**(float x, float y, float z);
void **pfInitState**(usptr_t* arena);
pfState* **pfGetCurState**(void);
void **pfPushState**(void);
void **pfPopState**(void);
void **pfGetState**(pfGeoState *gstate);
void **pfFlushState**(void);
void **pfBasicState**(void);
void **pfOverride**(uint mask, int val);
uint **pfGetOverride**(void);
void **pfModelMat**(pfMatrix mat);
void **pfGetModelMat**(pfMatrix mat);
void **pfViewMat**(pfMatrix mat);
void **pfGetViewMat**(pfMatrix mat);
void **pfTexMat**(pfMatrix mat);
void **pfGetTexMat**(pfMatrix mat);
void **pfInvModelMat**(pfMatrix mat);
void **pfGetInvModelMat**(pfMatrix mat);
void **pfNearPixDist**(float pd);
float **pfGetNearPixDist**(void);

pfState C API

IRIS Performer manages a subset of the graphics library state for convenience and improved performance, and thus provides its own API for manipulating graphics state such as transparency, antialiasing, or fog. Attributes not set within a **pfGeoState** are inherited from the **pfState**.

pfState* **pfNewState**(void);
pfType* **pfGetStateClassType**(void);
void **pfSelectState**(pfState* state);
void **pfLoadState**(pfState* state);
void **pfAttachState**(pfState* state, pfState *state1);

pfString C API

pfString provides a **pfGeoSet** like facility for encapsulating geometry to display a string in 3-D with attributes such as color, arbitrary transformation matrix, and font (see **pfFont**).

pfString* **pfNewString**(void *arena);
pfType* **pfGetStringClassType**(void);
int **pfGetStringStringLength**(const pfString* string);
void **pfStringMode**(pfString* string, int mode, int val);

int	**pfGetStringMode**(const pfString* string, int mode);
void	**pfStringFont**(pfString* string, pfFont* fnt);
pfFont*	**pfGetStringFont**(const pfString* string);
void	**pfStringString**(pfString* string, const char* cstr);
const char*	**pfGetStringString**(const pfString* string);
const pfGeoSet*	**pfGetStringCharGSet**(const pfString* string, int index);
const pfVec3*	**pfGetStringCharPos**(const pfString* string, int index);
void	**pfStringSpacingScale**(pfString* string, float sx, float sy, float sz);
void	**pfGetStringSpacingScale**(const pfString* string, float *sx, float *sy, float *sz);
void	**pfStringGState**(pfString* string, pfGeoState *gs);
const pfGeoState*	**pfGetStringGState**(const pfString* string);
void	**pfStringColor**(pfString* string, float r, float g, float b, float a);
void	**pfGetStringColor**(const pfString* string, float *r, float *g, float *b, float *a);
void	**pfStringBBox**(pfString* string, const pfBox* newbox);
const pfBox*	**pfGetStringBBox**(const pfString* string);
void	**pfStringMat**(pfString* string, const pfMatrix mat);
void	**pfGetStringMat**(const pfString* string, pfMatrix mat);
void	**pfStringIsectMask**(pfString* string, uint mask, int setMode, int bitOp);
uint	**pfGetStringIsectMask**(const pfString* string);
void	**pfDrawString**(pfString* string);
void	**pfFlattenString**(pfString* string);
int	**pfStringIsectSegs**(pfString* string, pfSegSet *segSet, pfHit **hits[]);
pfTexture*	**pfGetCurTex**(void);

pfTexture C API

pfTexture encapsulates texturing data and attributes such as the texture image itself, the texture data format and the filters for proximity and distance.

pfTexture*	**pfNewTex**(void *arena);
pfType*	**pfGetTexClassType**(void);
void	**pfTexName**(pfTexture* tex, const char *name);
const char*	**pfGetTexName**(const pfTexture* tex);
void	**pfTexImage**(pfTexture* tex, uint* image, int comp, int sx, int sy, int sz);
void	**pfGetTexImage**(const pfTexture* tex, uint** image, int* comp, int* sx, int* sy, int* sz);
void	**pfTexLoadImage**(pfTexture* tex, uint* image);
uint*	**pfGetTexLoadImage**(const pfTexture* tex);
void	**pfTexBorderColor**(pfTexture* tex, pfVec4 clr);
void	**pfGetTexBorderColor**(pfTexture* tex, pfVec4 *clr);
void	**pfTexBorderType**(pfTexture* tex, int type);
int	**pfGetTexBorderType**(pfTexture* tex);

```
void        pfTexFormat(pfTexture* tex, int format, int type);
int         pfGetTexFormat(const pfTexture* tex, int format);
void        pfTexFilter(pfTexture* tex, int filt, int type);
int         pfGetTexFilter(const pfTexture* tex, int filt);
void        pfTexRepeat(pfTexture* tex, int wrap, int type);
int         pfGetTexRepeat(const pfTexture* tex, int wrap);
void        pfTexSpline(pfTexture* tex, int type, pfVec2 *pts, float clamp);
void        pfGetTexSpline(const pfTexture* tex, int type, pfVec2 *pts, float *clamp);
void        pfTexDetail(pfTexture* tex, int l, pfTexture *detail);
void        pfGetTexDetail(const pfTexture* tex, int *l, pfTexture **detail);
pfTexture*  pfGetTexDetailTex(const pfTexture* tex);
void        pfDetailTexTile(pfTexture* tex, int j, int k, int m, int n, int scram);
void        pfGetDetailTexTile(const pfTexture* tex, int *j, int *k, int *m, int *n, int *scram);
void        pfTexList(pfTexture* tex, pfList *list);
pfList*     pfGetTexList(const pfTexture* tex);
void        pfTexFrame(pfTexture* tex, float frame);
float       pfGetTexFrame(const pfTexture* tex);
void        pfTexLoadMode(pfTexture* tex, int mode, int val);
int         pfGetTexLoadMode(const pfTexture* tex, int mode);
void        pfTexLevel(pfTexture* tex, int level, pfTexture* ltex);
pfTexture*  pfGetTexLevel(pfTexture* tex, int level);
void        pfTexLoadOrigin(pfTexture* tex, int which, int xo, int yo);
void        pfGetTexLoadOrigin(pfTexture* tex, int which, int *xo, int *yo);
void        pfTexLoadSize(pfTexture* tex, int xs, int ys);
void        pfGetTexLoadSize(const pfTexture* tex, int *xs, int *ys);
void        pfApplyTex(pfTexture* tex);
void        pfFormatTex(pfTexture* tex);
void        pfLoadTex(pfTexture* tex);
void        pfLoadTexLevel(pfTexture* tex, int level);
void        pfSubloadTex(pfTexture* tex, int source, uint *image, int xsrc, int ysrc, int xdst, int ydst,
                int xsize, int ysize);
void        pfSubloadTexLevel(pfTexture* tex, int source, uint *image, int xsrc, int ysrc, int xdst,
                int ydst, int xsize, int ysize, int level);
int         pfLoadTexFile(pfTexture* tex, char* fname);
void        pfFreeTexImage(pfTexture* tex);
void        pfIdleTex(pfTexture* tex);
int         pfIsTexLoaded(const pfTexture* tex);
int         pfIsTexFormatted(const pfTexture* tex);
pfTexEnv*   pfGetCurTEnv(void);
```

pfTexEnv C API

pfTexEnv encapsulates the texture environment and how the texture should interact with the colors of the geometry to which it is bound, i.e. how graphics coordinates are transformed into texture coordinates.

```
pfTexEnv*  pfNewTEnv(void *arena);
pfType*    pfGetTEnvClassType(void);
void       pfTEnvMode(pfTexEnv* tenv, int mode);
int        pfGetTEnvMode(const pfTexEnv* tenv);
void       pfTEnvComponent(pfTexEnv* tenv, int comp);
int        pfGetTEnvComponent(const pfTexEnv* tenv);
void       pfTEnvBlendColor(pfTexEnv* tenv, float r, float g, float b, float a);
void       pfGetTEnvBlendColor(pfTexEnv* tenv, float* r, float* g, float* b, float* a);
void       pfApplyTEnv(pfTexEnv* tenv);
pfTexGen*  pfGetCurTGen(void);
```

pfTexGen C API

The **pfTexGen** capability is used to automatically generate texture coordinates for geometry, typically for special effects like projected texture, reflection mapping, and lightpoints (see **pfLPointState**).

```
pfTexGen*  pfNewTGen(void *arena);
pfType*    pfGetTGenClassType(void);
void       pfTGenMode(pfTexGen* tgen, int texCoord, int mode);
int        pfGetTGenMode(const pfTexGen* tgen, int texCoord);
void       pfTGenPlane(pfTexGen* tgen, int texCoord, float x, float y, float z, float d);
void       pfGetTGenPlane(pfTexGen* tgen, int texCoord, float* x, float* y, float* z, float* d);
void       pfApplyTGen(pfTexGen* tgen);
```

pfCycleMemory C API

The **pfCycleMemory** data type is the low-level memory object used by **pfCycleBuffers** to provide the illusion of a single block of memory that can have a different value for each process that references it at one instant in time. For example, a **pfGeoSet** might have vertex position, normal, color, or texture arrays that are being morphed in process A, culled in process B, drawn in process C, and intersected with in process D, all with different values due to temporal reasons. Refer to the **pfCycleBuffer** overview for a description of how the two features work in concert.

```
pfType*        pfGetCMemClassType(void);
pfCycleBuffer* pfGetCMemCBuffer(pfCycleMemory* cmem);
int            pfGetCMemFrame(const pfCycleMemory* cmem);
```

pfCycleBuffer C API

pfCycleBuffer supports efficient management of dynamically modified data in a multi-stage multiprocessed pipeline. A **pfCycleBuffer** logically contains multiple **pfCycleMemory**s. Each process has a global index which selects the currently active **pfCycleMemory** in each **pfCycleBuffer**. This index can be advanced once a frame by **pfCurCBufferIndex** so that the buffers "cycle". By advancing the index appropriately in each pipeline stage, dynamic data can be frame-accurately propagated down the

pipeline.

```
pfCycleBuffer*  pfNewCBuffer(size_t nbytes, void *arena);
pfType*         pfGetCBufferClassType(void);
pfCycleMemory*
                pfGetCBufferCMem(const pfCycleBuffer* cBuf, int index);
void*           pfGetCurCBufferData(const pfCycleBuffer* cBuf);
void            pfCBufferChanged(pfCycleBuffer* cBuf);
void            pfInitCBuffer(pfCycleBuffer* cBuf, void *data);
int             pfCBufferConfig(int numBuffers);
int             pfGetCBufferConfig(void);
int             pfCBufferFrame(void);
int             pfGetCBufferFrameCount(void);
int             pfGetCurCBufferIndex(void);
void            pfCurCBufferIndex(int index);
pfCycleBuffer*  pfGetCBuffer(void *data);
```

pfMemory C API

A **pfMemory** is the data type from which the major IRIS Performer types are derived and also provides the primary mechanism for allocating memory used by **pfMalloc**.

```
pfType*      pfGetMemoryClassType(void);
void*        pfMalloc(size_t nbytes, void *arena);
void*        pfCalloc(size_t numelem, size_t elsize, void *arena);
char*        pfStrdup(const char *str, void *arena);
void*        pfRealloc(void *data, size_t nbytes);
size_t       pfGetSize(void *data);
void*        pfGetArena(void *data);
void         pfFree(void *data);
void*        pfGetData(const void *data);
pfMemory*    pfGetMemory(const void *data);
const char*  pfGetTypeName(const void *data);
pfType*      pfGetType(const void *data);
int          pfIsOfType(const void *data, pfType *type);
int          pfIsExactType(const void *data, pfType *type);
int          pfRef(void* mem);
int          pfUnref(void* mem);
ushort       pfGetRef(const void* mem);
int          pfCompare(const void* mem1, const void* mem2);
int          pfPrint(const void* mem, uint travMode, uint verbose, FILE* file);
```

```
int         prDelete(void* mem);
int         prUnrefDelete(void* mem);
int         prCopy(void* dst, const void* src);
pfFile*     pfOpenFile(char* fname, int oflag,
```

pfFile C API

pfFile provides a non-blocking, multiprocessing mechanism for file I/O with a similar interface to the standard UNIX file I/O functions. The difference is that these routines return immediately without blocking while the physical file-system access operation completes and also that instead of an integer file descriptor, a **pfFile** handle is used.

```
pfFile*   pfCreateFile(char* fname, mode_t mode);
pfType*
          pfGetFileClassType(void);
int       pfGetFileStatus(const pfFile* file, int attr);
int       pfReadFile(pfFile* file, char* buf, int nbyte);
int       pfWriteFile(pfFile* file, char* buf, int nbyte);
off_t     pfSeekFile(pfFile* file, off_t off, int whence);
int       pfCloseFile(pfFile* file);
```

pfList C API

A **pfList** is a dynamically-sized array of arbitrary, but homogeneously-sized, elements. IRIS Performer provides the facility to create, manipulate, and search a **pfList**.

```
pfList*   pfNewList(int eltSize, int listLength, void *arena);
pfType*
          pfGetListClassType(void);
int       pfGetListEltSize(const pfList* list);
void**    pfGetListArray(const pfList* list);
void      pfListArrayLen(pfList* list, int alen);
int       pfGetListArrayLen(const pfList* list);
void      pfNum(pfList* list, int newNum);
int       pfGetNum(const pfList* list);
void      pfSet(pfList* list, int index, void *elt);
void*     pfGet(const pfList* list, int index);
void      pfResetList(pfList* list);
void      pfCombineLists(pfList* lists, const pfList *a, const pfList *b);
void      pfAdd(pfList* lists, void *elt);
void      pfInsert(pfList* lists, int index, void *elt);
int       pfSearch(const pfList* lists, void *elt);
int       pfRemove(pfList* lists, void *elt);
```

void **pfRemoveIndex**(pfList* lists, int index);
int **pfMove**(pfList* lists, int index, void *elt);
int **pfFastRemove**(pfList* lists, void *elt);
void **pfFastRemoveIndex**(pfList* lists, int index);
int **pfReplace**(pfList* lists, void *oldElt, void *newElt);

pfWindow C API

These functions provide a single API for creating and managing windows that works across the IRIS GL, IRIS GLX Mixed Mode, and OpenGL-X environments. Window system independent types have been provided to match the X Window System types to provide complete portability between the IRIS GL and OpenGL-X windowing environments.

pfWindow* **pfNewWin**(void *arena);
pfType* **pfGetWinClassType**(void);
void **pfWinName**(pfWindow* win, const char *name);
const char* **pfGetWinName**(const pfWindow* win);
void **pfWinMode**(pfWindow* win, int mode, int val);
int **pfGetWinMode**(const pfWindow* win, int mode);
void **pfWinType**(pfWindow* win, uint type);
uint **pfGetWinType**(const pfWindow* win);
pfState* **pfGetWinCurState**(const pfWindow* win);
void **pfWinAspect**(pfWindow* win, int x, int y);
void **pfGetWinAspect**(const pfWindow* win, int *x, int *y);
void **pfWinOriginSize**(pfWindow* win, int xo, int yo, int xs, int ys);
void **pfWinOrigin**(pfWindow* win, int xo, int yo);
void **pfGetWinOrigin**(const pfWindow* win, int *xo, int *yo);
void **pfWinSize**(pfWindow* win, int xs, int ys);
void **pfGetWinSize**(const pfWindow* win, int *xs, int *ys);
void **pfWinFullScreen**(pfWindow* win);
void **pfGetWinCurOriginSize**(pfWindow* win, int *xo, int *yo, int *xs, int *ys);
void **pfGetWinCurScreenOriginSize**(pfWindow* win, int *xo, int *yo, int *xs, int *ys);
void **pfWinOverlayWin**(pfWindow* win, pfWindow *ow);
pfWindow* **pfGetWinOverlayWin**(const pfWindow* win);
void **pfWinStatsWin**(pfWindow* win, pfWindow *ow);
pfWindow* **pfGetWinStatsWin**(const pfWindow* win);
void **pfWinScreen**(pfWindow* win, int s);
int **pfGetWinScreen**(const pfWindow* win);
void **pfWinShare**(pfWindow* win, uint mode);
uint **pfGetWinShare**(const pfWindow* win);
void **pfWinWSWindow**(pfWindow* win, pfWSConnection dsp, pfWSWindow wsWin);

pfWSWindow	**pfGetWinWSWindow**(const pfWindow* win);
void	**pfWinWSDrawable**(pfWindow* win, pfWSConnection dsp, pfWSDrawable wsWin);
pfWSDrawable	**pfGetWinWSDrawable**(const pfWindow* win);
pfWSDrawable	**pfGetWinCurWSDrawable**(const pfWindow* win);
void	**pfWinWSConnectionName**(pfWindow* win, const char *name);
const char*	**pfGetWinWSConnectionName**(const pfWindow* win);
void	**pfWinFBConfigData**(pfWindow* win, void *data);
void*	**pfGetWinFBConfigData**(pfWindow* win);
void	**pfWinFBConfigAttrs**(pfWindow* win, int *attr);
int*	**pfGetWinFBConfigAttrs**(const pfWindow* win);
void	**pfWinFBConfig**(pfWindow* win, pfFBConfig vInfo);
pfFBConfig	**pfGetWinFBConfig**(const pfWindow* win);
void	**pfWinFBConfigId**(pfWindow* win, int vId);
int	**pfGetWinFBConfigId**(const pfWindow* win);
void	**pfWinIndex**(pfWindow* win, int index);
int	**pfGetWinIndex**(const pfWindow* win);
pfWindow*	**pfGetWinSelect**(pfWindow* win);
void	**pfWinGLCxt**(pfWindow* win, pfGLContext gCxt);
pfGLContext	**pfGetWinGLCxt**(const pfWindow* win);
void	**pfWinList**(pfWindow* win, pfList *wl);
pfList*	**pfGetWinList**(const pfWindow* win);
void	**pfOpenWin**(pfWindow* win);
void	**pfCloseWin**(pfWindow* win);
void	**pfCloseWinGL**(pfWindow* win);
int	**pfAttachWin**(pfWindow* win, pfWindow *w1);
int	**pfDetachWin**(pfWindow* win, pfWindow *w1);
pfWindow*	**pfSelectWin**(pfWindow* win);
void	**pfSwapWinBuffers**(pfWindow* win);
pfFBConfig	**pfChooseWinFBConfig**(pfWindow* win, int *attr);
int	**pfIsWinOpen**(const pfWindow* win);
int	**pfQueryWin**(pfWindow* win, int which, int *dst);
int	**pfMQueryWin**(pfWindow* win, int *which, int *dst);
pfWindow*	**pfOpenNewNoPortWin**(const char *name, int screen);
pfStats*	**pfGetCurStats**(void);

pfStats C API

These functions are used to collect, manipulate, print, and query statistics on state operations, geometry, and graphics and system operations. IRIS Performer has the ability to keep many types of statistics. Some statistics can be expensive to gather and might possibly influence other statistics. To alleviate this problem, statistics are divided into different classes based on the tasks that they monitor. The specific statistics classes of interest may be selected with **pfStatsClass**.

pfStats* **pfNewStats**(void *arena);
pfType* **pfGetStatsClassType**(void);
uint **pfStatsClassMode**(pfStats* stats, int class, uint mask, int val);
uint **pfGetStatsClassMode**(pfStats* stats, int class);
void **pfStatsAttr**(pfStats* stats, int attr, float val);
float **pfGetStatsAttr**(pfStats* stats, int attr);
uint **pfStatsClass**(pfStats* stats, uint enmask, int val);
uint **pfGetStatsClass**(pfStats* stats, uint enmask);
uint **pfGetOpenStats**(pfStats* stats, uint enmask);
uint **pfOpenStats**(pfStats* stats, uint enmask);
uint **pfCloseStats**(uint enmask);
void **pfResetStats**(pfStats* stats);
void **pfClearStats**(pfStats* stats, uint which);
void **pfAccumulateStats**(pfStats* stats, pfStats* src, uint which);
void **pfAverageStats**(pfStats* stats, pfStats* src, uint which, int num);
void **pfCopyStats**(pfStats* stats, const pfStats *src, uint which);
void **pfStatsCountGSet**(pfStats* stats, pfGeoSet * gset);
int **pfQueryStats**(pfStats* stats, uint which, void *dst, int size);
int **pfMQueryStats**(pfStats* stats, uint * which, void *dst, int size);
void **pfStatsHwAttr**(int attr, float val);
float **pfGetStatsHwAttr**(int attr);
void **pfEnableStatsHw**(uint which);
void **pfDisableStatsHw**(uint which);
uint **pfGetStatsHwEnable**(uint which);
void **pfFPConfig**(int which, float val);
float **pfGetFPConfig**(int which);
void **pfSinCos**(float arg, float* s, float* c);
float **pfTan**(float arg);
float **pfArcTan2**(float y, float x);
float **pfArcSin**(float arg);
float **pfArcCos**(float arg);
float **pfSqrt**(float arg);

pfVec2 C API

Math functions for 2-component vectors. Most of these routines have macro equivalents which are described in the **pfVec2** man page.

void **pfSetVec2**(pfVec2 vec2, float x, float y);
void **pfCopyVec2**(pfVec2 vec2, const pfVec2 v);
int **pfEqualVec2**(const pfVec2 vec2, const pfVec2 v);

int **pfAlmostEqualVec2**(const pfVec2 vec2, const pfVec2 v, float tol);
void **pfNegateVec2**(pfVec2 vec2, const pfVec2 v);
float **pfDotVec2**(const pfVec2 vec2, const pfVec2 v);
void **pfAddVec2**(pfVec2 vec2, const pfVec2 v1, const pfVec2 v2);
void **pfSubVec2**(pfVec2 vec2, const pfVec2 v1, const pfVec2 v2);
void **pfScaleVec2**(pfVec2 vec2, float s, const pfVec2 v);
void **pfAddScaledVec2**(pfVec2 vec2, const pfVec2 v1, float s, const pfVec2 v2);
void **pfCombineVec2**(pfVec2 vec2, float a, const pfVec2 v1, float b, const pfVec2 v2);
float **pfSqrDistancePt2**(const pfVec2 vec2, const pfVec2 v);
float **pfNormalizeVec2**(pfVec2 vec2);
float **pfLengthVec2**(const pfVec2 vec2);
float **pfDistancePt2**(const pfVec2 vec2, const pfVec2 v);

pfVec3 C API

Math functions for 3-component vectors. Most of these routines have macro equivalents which are described in the **pfVec3** man page.

void **pfSetVec3**(pfVec3 vec3, float x, float y, float z);
void **pfCopyVec3**(pfVec3 vec3, const pfVec3 v);
int **pfEqualVec3**(const pfVec3 vec3, const pfVec3 v);
int **pfAlmostEqualVec3**(const pfVec3 vec3, const pfVec3 v, float tol);
void **pfNegateVec3**(pfVec3 vec3, const pfVec3 v);
float **pfDotVec3**(const pfVec3 vec3, const pfVec3 v);
void **pfAddVec3**(pfVec3 vec3, const pfVec3 v1, const pfVec3 v2);
void **pfSubVec3**(pfVec3 vec3, const pfVec3 v1, const pfVec3 v2);
void **pfScaleVec3**(pfVec3 vec3, float s, const pfVec3 v);
void **pfAddScaledVec3**(pfVec3 vec3, const pfVec3 v1, float s, const pfVec3 v2);
void **pfCombineVec3**(pfVec3 vec3, float a, const pfVec3 v1, float b, const pfVec3 v2);
float **pfSqrDistancePt3**(const pfVec3 vec3, const pfVec3 v);
float **pfNormalizeVec3**(pfVec3 vec3);
float **pfLengthVec3**(const pfVec3 vec3);
float **pfDistancePt3**(const pfVec3 vec3, const pfVec3 v);
void **pfCrossVec3**(pfVec3 vec3, const pfVec3 v1, const pfVec3 v2);
void **pfXformVec3**(pfVec3 vec3, const pfVec3 v, const pfMatrix m);
void **pfXformPt3**(pfVec3 vec3, const pfVec3 v, const pfMatrix m);
void **pfFullXformPt3**(pfVec3 vec3, const pfVec3 v, const pfMatrix m);

pfVec4 C API

Math functions for 4-component vectors. Most of these routines have macro equivalents which are described in the **pfVec4** man page.

void **pfSetVec4**(pfVec4 vec4, float x, float y, float z, float w);
void **pfCopyVec4**(pfVec4 vec4, const pfVec4 v);
int **pfEqualVec4**(const pfVec4 vec4, const pfVec4 v);
int **pfAlmostEqualVec4**(const pfVec4 vec4, const pfVec4 v, float tol);
void **pfNegateVec4**(pfVec4 vec4, const pfVec4 v);
float **pfDotVec4**(const pfVec4 vec4, const pfVec4 v);
void **pfAddVec4**(pfVec4 vec4, const pfVec4 v1, const pfVec4 v2);
void **pfSubVec4**(pfVec4 vec4, const pfVec4 v1, const pfVec4 v2);
void **pfScaleVec4**(pfVec4 vec4, float s, const pfVec4 v);
void **pfAddScaledVec4**(pfVec4 vec4, const pfVec4 v1, float s, const pfVec4 v2);
void **pfCombineVec4**(pfVec4 vec4, float a, const pfVec4 v1, float b, const pfVec4 v2);
float **pfSqrDistancePt4**(const pfVec4 vec4, const pfVec4 v);
float **pfNormalizeVec4**(pfVec4 vec4);
float **pfLengthVec4**(const pfVec4 vec4);
float **pfDistancePt4**(const pfVec4 vec4, const pfVec4 v);
void **pfXformVec4**(pfVec4 vec4, const pfVec4 v, const pfMatrix m);

pfMatrix C API

The pfMatrix data type represents a complete 4x4 real matrix. These routines create transformation matrices based on multiplying a row vector by a matrix on the right, i.e. the vector v transformed by m is v * m. Many actions will go considerably faster if the last column is (0,0,0,1).

Some of these routines have macro equivalents which are described in the **pfMatrix** man page.

void **pfSetMat**(pfMatrix mat, float *m);
int **pfGetMatType**(const pfMatrix mat);
void **pfSetMatRowVec3**(pfMatrix mat, int r, const pfVec3 v);
void **pfSetMatRow**(pfMatrix mat, int r, float x, float y, float z, float w);
void **pfGetMatRowVec3**(pfMatrix mat, int r, pfVec3 dst);
void **pfGetMatRow**(pfMatrix mat, int r, float *x, float *y, float *z, float *w);
void **pfSetMatColVec3**(pfMatrix mat, int c, const pfVec3 v);
void **pfSetMatCol**(pfMatrix mat, int c, float x, float y, float z, float w);
void **pfGetMatColVec3**(pfMatrix mat, int c, pfVec3 dst);
void **pfGetMatCol**(pfMatrix mat, int c, float *x, float *y, float *z, float *w);
void **pfGetOrthoMatCoord**(pfMatrix mat, pfCoord* dst);
void **pfMakeIdentMat**(pfMatrix mat);
void **pfMakeEulerMat**(pfMatrix mat, float hdeg, float pdeg, float rdeg);
void **pfMakeRotMat**(pfMatrix mat, float degrees, float x, float y, float z);
void **pfMakeTransMat**(pfMatrix mat, float x, float y, float z);
void **pfMakeScaleMat**(pfMatrix mat, float x, float y, float z);

void **pfMakeVecRotVecMat**(pfMatrix mat, const pfVec3 v1, const pfVec3 v2);
void **pfMakeCoordMat**(pfMatrix mat, const pfCoord* c);
void **pfGetOrthoMatQuat**(pfMatrix mat, pfQuat dst);
void **pfMakeQuatMat**(pfMatrix mat, const pfQuat q);
void **pfCopyMat**(pfMatrix mat, const pfMatrix v);
int **pfEqualMat**(const pfMatrix mat, const pfMatrix m);
int **pfAlmostEqualMat**(const pfMatrix mat, const pfMatrix m2, float tol);
void **pfTransposeMat**(pfMatrix mat, pfMatrix m);
void **pfMultMat**(pfMatrix mat, const pfMatrix m1, const pfMatrix m2);
void **pfAddMat**(pfMatrix mat, const pfMatrix m1, const pfMatrix m2);
void **pfSubMat**(pfMatrix mat, const pfMatrix m1, const pfMatrix m2);
void **pfScaleMat**(pfMatrix mat, float s, const pfMatrix m);
void **pfPostMultMat**(pfMatrix mat, const pfMatrix m);
void **pfPreMultMat**(pfMatrix mat, const pfMatrix m);
int **pfInvertFullMat**(pfMatrix mat, pfMatrix m);
void **pfInvertAffMat**(pfMatrix mat, const pfMatrix m);
void **pfInvertOrthoMat**(pfMatrix mat, const pfMatrix m);
void **pfInvertOrthoNMat**(pfMatrix mat, pfMatrix m);
void **pfInvertIdentMat**(pfMatrix mat, const pfMatrix m);
void **pfPreTransMat**(pfMatrix mat, float x, float y, float z, pfMatrix m);
void **pfPostTransMat**(pfMatrix mat, const pfMatrix m, float x, float y, float z);
void **pfPreRotMat**(pfMatrix mat, float degrees, float x, float y, float z, pfMatrix m);
void **pfPostRotMat**(pfMatrix mat, const pfMatrix m, float degrees, float x, float y, float z);
void **pfPreScaleMat**(pfMatrix mat, float xs, float ys, float zs, pfMatrix m);
void **pfPostScaleMat**(pfMatrix mat, const pfMatrix m, float xs, float ys, float zs);

pfQuat C API

pfQuat represents a quaternion as the four floating point values (*x*, *y*, *z*, *w*) of a pfVec4. Some of these routines have macro equivalents which are described in the **pfMatrix** man page.

void **pfGetQuatRot**(pfQuat quat, float *angle, float *x, float *y, float *z);
void **pfMakeRotQuat**(pfQuat quat, float angle, float x, float y, float z);
void **pfConjQuat**(pfQuat quat, const pfQuat v);
float **pfLengthQuat**(const pfQuat quat);
void **pfMultQuat**(pfQuat quat, const pfQuat q1, const pfQuat q2);
void **pfDivQuat**(pfQuat quat, const pfQuat q1, const pfQuat q2);
void **pfInvertQuat**(pfQuat quat, const pfQuat q1);
void **pfExpQuat**(pfQuat quat, const pfQuat q);
void **pfLogQuat**(pfQuat quat, const pfQuat q);
void **pfSlerpQuat**(pfQuat quat, float t, const pfQuat q1, const pfQuat q2);

void **pfSquadQuat**(pfQuat quat, float t, const pfQuat q1, const pfQuat q2, const pfQuat a, const pfQuat b);
void **pfQuatMeanTangent**(pfQuat quat, const pfQuat q1, const pfQuat q2, const pfQuat q3);

pfMatStack C API

These routines allow the creation and manipulation of a stack of 4x4 matrices.

pfMatStack* **pfNewMStack**(int size, void *arena);
pfType* **pfGetMStackClassType**(void);
void **pfGetMStack**(const pfMatStack* mst, pfMatrix m);
pfMatrix* **pfGetMStackTop**(const pfMatStack* mst);
int **pfGetMStackDepth**(const pfMatStack* mst);
void **pfResetMStack**(pfMatStack* mst);
int **pfPushMStack**(pfMatStack* mst);
int **pfPopMStack**(pfMatStack* mst);
void **pfLoadMStack**(pfMatStack* mst, const pfMatrix m);
void **pfPreMultMStack**(pfMatStack* mst, const pfMatrix m);
void **pfPostMultMStack**(pfMatStack* mst, const pfMatrix m);
void **pfPreTransMStack**(pfMatStack* mst, float x, float y, float z);
void **pfPostTransMStack**(pfMatStack* mst, float x, float y, float z);
void **pfPreRotMStack**(pfMatStack* mst, float degrees, float x, float y, float z);
void **pfPostRotMStack**(pfMatStack* mst, float degrees, float x, float y, float z);
void **pfPreScaleMStack**(pfMatStack* mst, float xs, float ys, float zs);
void **pfPostScaleMStack**(pfMatStack* mst, float xs, float ys, float zs);

pfSeg C API

A **pfSeg** represents a line segment starting at *pos*, extending for a length *length* in the direction *dir*. The routines assume that *dir* is of unit length, otherwise the results are undefined. pfSeg is a public struct whose data members *pos*, *dir* and *length* may be operated on directly.

void **pfMakePtsSeg**(pfSeg* seg, const pfVec3 p1, const pfVec3 p2);
void **pfMakePolarSeg**(pfSeg* seg, const pfVec3 pos, float azi, float elev, float len);
void **pfClipSeg**(pfSeg* seg, const pfSeg *seg, float d1, float d2);
int **pfClosestPtsOnSeg**(const pfSeg* seg, const pfSeg *seg, pfVec3 dst1, pfVec3 dst2);

pfPlane C API

A **pfPlane** represents an infinite 2D plane as a normal and a distance offset from the origin in the normal direction. A point on the plane satisfies the equation *normal* dot (x, y, z) = offset. pfPlane is a public struct whose data members *normal* and *offset* may be operated on directly.

void **pfMakePtsPlane**(pfPlane* plane, const pfVec3 p1, const pfVec3 p2, const pfVec3 p3);
void **pfMakeNormPtPlane**(pfPlane* plane, const pfVec3 norm, const pfVec3 pos);

void **pfDisplacePlane**(pfPlane* plane, float d);
int **pfHalfSpaceContainsBox**(const pfPlane* plane, const pfBox *box);
int **pfHalfSpaceContainsSphere**(const pfPlane* plane, const pfSphere *sph);
int **pfHalfSpaceContainsCyl**(const pfPlane* plane, const pfCylinder *cyl);
int **pfHalfSpaceContainsPt**(const pfPlane* plane, const pfVec3 pt);
void **pfOrthoXformPlane**(pfPlane* plane, const pfPlane *pln, const pfMatrix m);
void **pfClosestPtOnPlane**(const pfPlane* plane, const pfVec3 pt, pfVec3 dst);
int **pfPlaneIsectSeg**(const pfPlane* plane, const pfSeg *seg, float *d);
int **pfHalfSpaceIsectSeg**(const pfPlane* plane, const pfSeg *seg, float *d1, float *d2);

pfSphere C API

pfSpheres are typically used as bounding volumes in a scene graph. These routines allow bounding spheres to be created and manipulated.

void **pfMakeEmptySphere**(pfSphere* sphere);
int **pfSphereContainsPt**(const pfSphere* sphere, const pfVec3 pt);
int **pfSphereContainsSphere**(const pfSphere* sphere, const pfSphere *sph);
int **pfSphereContainsCyl**(const pfSphere* sphere, const pfCylinder *cyl);
void **pfSphereAroundPts**(pfSphere* sphere, const pfVec3* pts, int npt);
void **pfSphereAroundSpheres**(pfSphere* sphere, const pfSphere **sphs, int nsph);
void **pfSphereAroundBoxes**(pfSphere* sphere, const pfBox **boxes, int nbox);
void **pfSphereAroundCyls**(pfSphere* sphere, const pfCylinder **cyls, int ncyl);
void **pfSphereExtendByPt**(pfSphere* sphere, const pfVec3 pt);
void **pfSphereExtendBySphere**(pfSphere* sphere, const pfSphere *sph);
void **pfSphereExtendByCyl**(pfSphere* sphere, const pfCylinder *cyl);
void **pfOrthoXformSphere**(pfSphere* sphere, const pfSphere *sph, const pfMatrix m);
int **pfSphereIsectSeg**(const pfSphere* sphere, const pfSeg *seg, float *d1, float *d2);

pfCylinder C API

A pfCylinder represents a cylinder of finite length. The routines listed here provide means of creating and extending cylinders for use as bounding geometry around groups of line segments. The cylinder is defined by its *center*, *radius*, *axis* and *halfLength*. The routines assume *axis* is a vector of unit length, otherwise results are undefined. pfCylinder is a public struct whose data members *center*, *radius*, *axis* and *halfLength* may be operated on directly.

void **pfMakeEmptyCyl**(pfCylinder* cyl);
int **pfCylContainsPt**(const pfCylinder* cyl, const pfVec3 pt);
void **pfOrthoXformCyl**(pfCylinder* cyl, const pfCylinder *cyl, const pfMatrix m);
void **pfCylAroundPts**(pfCylinder* cyl, const pfVec3 *pts, int npt);
void **pfCylAroundSegs**(pfCylinder* cyl, const pfSeg **segs, int nseg);
void **pfCylAroundSpheres**(pfCylinder* cyl, const pfSphere **sphs, int nsph);

void **pfCylAroundBoxes**(pfCylinder* cyl, const pfBox **boxes, int nbox);
void **pfCylExtendBySphere**(pfCylinder* cyl, const pfSphere *sph);
void **pfCylExtendByCyl**(pfCylinder* cyl, const pfCylinder *cyl);
void **pfCylExtendByBox**(pfCylinder* cyl, const pfVec3 pt);
int **pfCylIsectSeg**(const pfCylinder* cyl, const pfSeg *seg, float *d1, float *d2);

pfBox C API

A pfBox is an axis-aligned box which can be used for intersection tests and for maintaining bounding information about geometry. A box represents the axis-aligned hexahedral volume: (x, y, z) where min[0] <= x <= max[0], min[1] <= y <= max[1] and min[2] <= z <= max[2]. pfBox is a public struct whose data members *min* and *max* may be operated on directly.

void **pfMakeEmptyBox**(pfBox* box);
int **pfBoxContainsPt**(const pfBox* box, const pfVec3 pt);
int **pfBoxContainsBox**(pfBox* box, const pfBox *inbox);
void **pfXformBox**(pfBox* box, const pfBox *box, const pfMatrix xform);
void **pfBoxAroundPts**(pfBox* box, const pfVec3 *pts, int npt);
void **pfBoxAroundSpheres**(pfBox* box, const pfSphere **sphs, int nsph);
void **pfBoxAroundBoxes**(pfBox* box, const pfBox **boxes, int nbox);
void **pfBoxAroundCyls**(pfBox* box, const pfCylinder **cyls, int ncyl);
void **pfBoxExtendByPt**(pfBox* box, const pfVec3 pt);
void **pfBoxExtendByBox**(pfBox* box, const pfBox *box);
int **pfBoxIsectSeg**(const pfBox* box, const pfSeg *seg, float *d1, float *d2);

pfPolytope C API

A pfPolytope is a set of half spaces whose intersection defines a convex, possibly semi-infinite, volume which may be used for culling and other intersection testing where a tighter bound than a pfBox, pfSphere, or pfCylinder is of benefit.

pfPolytope* **pfNewPtope**(void *arena);
pfType* **pfGetPtopeClassType**(void);
int **pfGetPtopeNumFacets**(pfPolytope* ptp);
int **pfPtopeFacet**(pfPolytope* ptp, int i, const pfPlane *p);
int **pfGetPtopeFacet**(pfPolytope* ptp, int i, pfPlane *p);
int **pfRemovePtopeFacet**(pfPolytope* ptp, int i);
void **pfOrthoXformPtope**(pfPolytope* ptp, const pfPolytope *src, const pfMatrix mat);
int **pfPtopeContainsPt**(const pfPolytope* ptp, const pfVec3 pt);
int **pfPtopeContainsSphere**(const pfPolytope* ptp, const pfSphere *sphere);
int **pfPtopeContainsBox**(const pfPolytope* ptp, const pfBox *box);
int **pfPtopeContainsCyl**(const pfPolytope* ptp, const pfCylinder *cyl);
int **pfPtopeContainsPtope**(const pfPolytope* ptp, const pfPolytope *ptope);

pfFrustum C API

A pfFrustum represents a viewing and or culling volume bounded by left, right, top, bottom, near and far planes.

pfFrustum* **pfNewFrust**(void *arena);
pfType* **pfGetFrustClassType**(void);
int **pfGetFrustType**(const pfFrustum* fr);
void **pfFrustAspect**(pfFrustum* fr, int which, float widthHeightRatio);
float **pfGetFrustAspect**(const pfFrustum* fr);
void **pfGetFrustFOV**(const pfFrustum* fr, float* fovh, float* fovv);
void **pfFrustNearFar**(pfFrustum* fr, float nearDist, float farDist);
void **pfGetFrustNearFar**(const pfFrustum* fr, float* nearDist, float* farDist);
void **pfGetFrustNear**(const pfFrustum* fr, pfVec3 ll, pfVec3 lr, pfVec3 ul, pfVec3 ur);
void **pfGetFrustFar**(const pfFrustum* fr, pfVec3 ll, pfVec3 lr, pfVec3 ul, pfVec3 ur);
void **pfGetFrustPtope**(const pfFrustum* fr, pfPolytope *dst);
void **pfGetFrustGLProjMat**(const pfFrustum* fr, pfMatrix mat);
int **pfGetFrustEye**(const pfFrustum* fr, pfVec3 eye);
void **pfMakePerspFrust**(pfFrustum* fr, float left, float right, float bot, float top);
void **pfMakeOrthoFrust**(pfFrustum* fr, float left, float right, float bot, float top);
void **pfMakeSimpleFrust**(pfFrustum* fr, float fov);
void **pfOrthoXformFrust**(pfFrustum* fr, const pfFrustum* fr2, const pfMatrix mat);
int **pfFrustContainsPt**(const pfFrustum* fr, const pfVec3 pt);
int **pfFrustContainsSphere**(const pfFrustum* fr, const pfSphere *sphere);
int **pfFrustContainsBox**(const pfFrustum* fr, const pfBox *box);
int **pfFrustContainsCyl**(const pfFrustum* fr, const pfCylinder *cyl);
void **pfApplyFrust**(const pfFrustum* fr);

Triangle Intersection

This routine returns the intersection of a triangle with a line segment and is the basis for Performer's performing intersection testing and picking against geometry contained in pfGeoSets.

int **pfTriIsectSeg**(const pfVec3 pt1, const pfVec3 pt2, const pfVec3 pt3, const pfSeg* seg, float* d);

LIBPFDU

Database Conversions

IRIS Performer provides an extensive array of converters which load file-based geometry formats into a **pfScene** hierarchical scene graph. These functions also provide the capability to set attributes which modify the behavior of individual loaders.

pfNode* **pfdLoadFile**(const char *file);

int **pfdStoreFile**(pfNode *root, const char *file);
pfNode* **pfdConvertFrom**(void *root, const char *ext);
void* **pfdConvertTo**(pfNode* root, const char *ext);
int **pfdInitConverter**(const char *ext);
int **pfdExitConverter**(const char *ext);
FILE* **pfdOpenFile**(const char *file);
void **pfdAddExtAlias**(const char *ext, const char *alias);
void **pfdConverterMode**(const char *ext, int mode, int value);
int **pfdGetConverterMode**(const char *ext, int mode);
void **pfdConverterAttr**(const char *ext, int which, void *attr);
void* **pfdGetConverterAttr**(const char *ext, int which);
void **pfdConverterVal**(const char *ext, int which, float val);
float **pfdGetConverterVal**(const char *ext, int which);
void **pfdPrintSceneGraphStats**(pfNode *node, double elapsedTime);

Generate pfGeoSets

These routines are provided to conveniently construct **pfGeoSets** for various geometric objects. The resulting objects are always positioned and sized in canonical ways. The user can then apply a transformation to these **pfGeoSets** to achieve the desired shape and position.

pfGeoSet * **pfdNewCube**(void *arena);
pfGeoSet * **pfdNewSphere**(int ntris, void *arena);
pfGeoSet * **pfdNewCylinder**(int ntris, void *arena);
pfGeoSet * **pfdNewCone**(int ntris, void *arena);
pfGeoSet * **pfdNewPipe**(float botRadius, float topRadius, int ntris, void *arena);
pfGeoSet * **pfdNewPyramid**(void *arena);
pfGeoSet * **pfdNewArrow**(int ntris, void *arena);
pfGeoSet * **pfdNewDoubleArrow**(int ntris, void *arena);
pfGeoSet * **pfdNewCircle**(int ntris, void *arena);
pfGeoSet * **pfdNewRing**(int ntris, void *arena);
void **pfdXformGSet**(pfGeoSet *gset, pfMatrix mat);
void **pfdGSetColor**(pfGeoSet *gset, float r, float g, float b, float a);

Mesh Triangles

Forming independent triangles into triangle strips (or meshes) can significantly improve rendering performance on IRIS systems. Strips reduce the amount of work required by the CPU, bus, and graphics subsystem. IRIS Performer provides this utility facility for converting independent triangles into strips.

pfGeoSet* **pfdMeshGSet**(pfGeoSet *gset);
void **pfdMesherMode**(int mode, int val);
int **pfdGetMesherMode**(int mode);

void **pfdShowStrips**(pfGeoSet *gset);

Optimize Scene Graphs

pfdCleanTree and **pfdStaticize** optimize the scene graph. **pfdCleanTree** removes pfGroups with one or fewer child and pfSCSes with identity transformations. **pfdStaticize** conditionally converts pfDCSes to pfSCSes, usually in preparation for **pfFlatten**.

pfNode* **pfdCleanTree**(pfNode *node, pfuTravFuncType doitfunc);
void **pfdReplaceNode**(pfNode *oldn, pfNode *newn);
void **pfdInsertGroup**(pfNode *oldn, pfGroup *grp);
void **pfdRemoveGroup**(pfGroup *oldn);
pfNode* **pfdFreezeTransforms**(pfNode *node, pfuTravFuncType doitfunc);

Breakup Scene Graphs

pfdBreakup is provided as a utility to break unstructured scene geometry into a spacially subdivided scene hierarchy. Spacially subdivided geometry is more easily culled and less time is spent drawing geometry which does not contribute to the final image.

pfNode* **pfdBreakup**(pfGeode *geode, float geodeSize, int stripLength, int geodeChild);

Generate Hierarchies

For performance reasons, it is desirable that the geometry in a scene be organized into a spatial hierarchy. However, it is often easiest to model geometry using logical, rather than spatial, divisions. **pfdTravGetGSets** and **pfdSpatialize** can be used to partition an already constructed scene.

pfList * **pfdTravGetGSets**(pfNode *node);
pfGroup*
pfdSpatialize(pfGroup *group, float maxGeodeSize, int maxGeoSets);

Share pfGeoStates

It is obviously desirable to share state between database objects in IRIS Performer whenever possible. The notion of pervasive state sharing underpins the entire **pfGeoState** mechanism. Common data such as texture, materials, and lighting models are often duplicated in many different objects throughout a database. This collection of functions provides the means necessary to easily achieve sharing among these objects by automatically producing a non-redundant set of states.

pfdShare* **pfdNewShare**(void);
int **pfdCleanShare**(pfdShare *share);
void **pfdDelShare**(pfdShare *share, int deepDelete);
void **pfdPrintShare**(pfdShare *share);
int **pfdCountShare**(pfdShare *share);
pfList* **pfdGetSharedList**(pfdShare *share, pfType* type);

pfObject* **pfdNewSharedObject**(pfdShare *share, pfObject *object);
pfObject* **pfdFindSharedObject**(pfdShare *share, pfObject *object);
int **pfdAddSharedObject**(pfdShare *share, pfObject *object);
void **pfdMakeShared**(pfNode *node);
void **pfdMakeSharedScene**(pfScene *scene);
int **pfdCleanShare**(pfdShare *share);
int **pfdRemoveSharedObject**(pfdShare *share, pfObject *object);
pfList* **pfdGetNodeGStateList**(pfNode *node);

Combine pfLayers

When multiple sibling layer nodes have been created, efficiency will be improved by combining them together. **pfdCombineLayers** provides for exactly this kind of optimization.

void **pfdCombineLayers**(pfNode *node);

Combine pfBillboards

The performance of pfBillboard nodes is enhanced when they contain several pfGeoSets each as opposed to a scene graph with a large number of single pfGeoSet pfBillboards. The **pfdCombineBillboards()** traversal creates this efficient situation by traversing a scene graph and combining the pfGeoSets of sibling pfBillboard nodes into a single pfBillboard node.

void **pfdCombineBillboards**(pfNode *node, int sizeLimit);

The Geometry Builder

It is seldom the case that database models are expressed directly in internal Performer structures (-**pfGeoSets**). Instead, models are generally described in geometric constructs defined by the modeller. The Performer *GeoBuilder* is meant to simplify the task of translating model geometry into Performer geometry structures. The *GeoBuilder* can also create many kinds of polygon mesh (e.g. triangle-strips) pfGeoSets, which can significantly improve performance.

pfdGeom* **pfdNewGeom**(int numV);
void **pfdResizeGeom**(pfdGeom *geom, int numV);
void **pfdDelGeom**(pfdGeom *geom);
int **pfdReverseGeom**(pfdGeom *geom);
pfdGeoBuilder* **pfdNewGeoBldr**(void);
void **pfdDelGeoBldr**(pfdGeoBuilder* bldr);
void **pfdGeoBldrMode**(pfdGeoBuilder* bldr, int mode, int val);
int **pfdGetGeoBldrMode**(pfdGeoBuilder* bldr, int mode);
int **pfdTriangulatePoly**(pfdGeom *pgon, pfdPrim *triList);
void **pfdAddGeom**(pfdGeoBuilder *bldr, pfdGeom *Geom, int num);
void **pfdAddLineStrips**(pfdGeoBuilder *bldr, pfdGeom *lineStrips, int num);
void **pfdAddLines**(pfdGeoBuilder *bldr, pfdGeom *lines);

void	**pfdAddPoints**(pfdGeoBuilder *bldr, pfdGeom *points);
void	**pfdAddPoly**(pfdGeoBuilder *bldr, pfdGeom *poly);
void	**pfdAddIndexedLineStrips**(pfdGeoBuilder *bldr, pfdGeom *lines, int num);
void	**pfdAddIndexedLines**(pfdGeoBuilder *bldr, pfdGeom *lines);
void	**pfdAddIndexedPoints**(pfdGeoBuilder *bldr, pfdGeom *points);
void	**pfdAddIndexedPoly**(pfdGeoBuilder *bldr, pfdGeom *poly);
void	**pfdAddIndexedTri**(pfdGeoBuilder *bldr, pfdPrim *tri);
void	**pfdAddLine**(pfdGeoBuilder *bldr, pfdPrim *line);
void	**pfdAddPoint**(pfdGeoBuilder *bldr, pfdPrim *Point);
void	**pfdAddTri**(pfdGeoBuilder *bldr, pfdPrim *tri);
int	**pfdGetNumTris**(pfdGeoBuilder *bldr);
const pfList*	**pfdBuildGSets**(pfdGeoBuilder *bldr);
void	**pfdPrintGSet**(pfGeoSet *gset);

The Scene Builder

The Performer *Builder* is meant to manage most of the details of constructing efficient runtime structures from input models. It provides a simple and convenient interface for bringing scene data into the application without the need for considering how best to structure that data for efficient rendering in Performer. The Builder provides a comprehensive interface between model input code (such as database file parsers) and the internal mechanisms of scene representation in Performer. In addition to handling input geometry, as the GeoBuilder does, the Builder also manages the associated graphics state.

void	**pfdInitBldr**(void);
void	**pfdExitBldr**(void);
pfdBuilder *	**pfdNewBldr**(void);
void	**pfdDelBldr**(pfdBuilder *bldr);
void	**pfdSelectBldr**(pfdBuilder *bldr);
pfdBuilder *	**pfdGetCurBldr**(void);
void	**pfdBldrDeleteNode**(pfNode *node);
void	**pfdBldrMode**(int mode, int val);
int	**pfdGetBldrMode**(int mode);
void	**pfdBldrAttr**(int which, void *attr);
void *	**pfdGetBldrAttr**(int which);
pfObject *	**pfdGetTemplateObject**(pfType *type);
void	**pfdResetObject**(pfObject *obj);
void	**pfdResetAllTemplateObjects**(void);
void	**pfdMakeDefaultObject**(pfObject *obj);
void	**pfdResetBldrGeometry**(void);
void	**pfdResetBldrShare**(void);
void	**pfdCleanBldrShare**(void);

void **pfdCaptureDefaultBldrState**(void);
void **pfdResetBldrState**(void);
void **pfdPushBldrState**(void);
void **pfdPopBldrState**(void);
void **pfdSaveBldrState**(void *name);
void **pfdLoadBldrState**(void *name);
void **pfdBldrGState**(const pfGeoState *gstate);
const pfGeoState * **pfdGetBldrGState**(void);
void **pfdBldrStateVal**(int which, float val);
float **pfdGetBldrStateVal**(int which);
void **pfdBldrStateMode**(int mode, int val);
int **pfdGetBldrStateMode**(int mode);
void **pfdBldrStateAttr**(int which, const void *attr);
const void * **pfdGetBldrStateAttr**(int attr);
void **pfdBldrStateInherit**(uint mask);
uint **pfdGetBldrStateInherit**(void);
void **pfdSelectBldrName**(void *name);
void * **pfdGetCurBldrName**(void);
void **pfdAddBldrGeom**(pfdGeom *p, int n);
void **pfdAddIndexedBldrGeom**(pfdGeom *p, int n);
pfNode * **pfdBuild**(void);
pfNode * **pfdBuildNode**(void *name);
void **pfdDefaultGState**(pfGeoState *def);
const pfGeoState* **pfdGetDefaultGState**(void);
void pfdMakeSceneGState(pfGeoState *sceneGState,
void pfdOptimizeGStateList(pfList *gstateList,

Haeberli Font Extensions

This is Paul Haeberli's cool font extension header file - Performer uses Paul's font library to load fonts into pfFont structures.

pfFont* **pfdLoadFont**(const char *ftype, const char *name, int style);
pfFont* **pfdLoadFont_type1**(const char *name, int style);

Texture Callbacks

These routines are now obsolete in that Performer now supports the notion of texture coordinate generation in pfGeoStates via the pfTexGen pfObject. However, these routines are still a good example of how to implement functionality in the draw process through callbacks. Similarly this set of routines also fits into the builder state extension mechanism - see the pfdBuilder man pages.

int **pfdPreDrawTexgenExt**(pfTraverser *trav, void *data);

int **pfdPostDrawTexgenExt**(pfTraverser *trav, void *data);
int **pfdPreDrawReflMap**(pfTraverser *trav, void *data);
int **pfdPostDrawReflMap**(pfTraverser *trav, void *data);
int **pfdPreDrawContourMap**(pfTraverser *trav, void *data);
int **pfdPostDrawContourMap**(pfTraverser *trav, void *data);
int **pfdPreDrawLinearMap**(pfTraverser *trav, void *data);
int **pfdPostDrawLinearMap**(pfTraverser *trav, void *data);
void **pfdTexgenParams**(const float *newParamsX, const float *newParamsY);

Function Extensors

pfdExtensors provide a framework for extending application functionality. They allow generalized callbacks to be attached to the model database. These callbacks can be called from any Performer traversal. The following functions are used to manipulate and install extensors.

int **pfdAddState**(void *name, long dataSize, void (*initialize)(void *data), void (*deletor)(void *data), int (*compare)(void *data1, void *data2), long (*copy)(void *dst, void *src), int token);
void **pfdStateCallback**(int stateToken, int whichCBack, pfNodeTravFuncType callback);
pfNodeTravFuncType **pfdGetStateCallback**(int stateToken, int which);
int **pfdGetStateToken**(void *name);
int **pfdGetUniqueStateToken**(void);
pfdExtensor* **pfdNewExtensor**(int which);
pfdExtensorType* **pfdNewExtensorType**(int token);
int **pfdCompareExtensor**(void *a, void *b);
int **pfdCompareExtraStates**(void *lista, void *listb);
void **pfdCopyExtraStates**(pfList *dst, pfList *src);
pfdExtensor* **pfdGetExtensor**(int token);
pfdExtensorType* **pfdGetExtensorType**(int token);
void * **pfdUniqifyData**(pfList *dataList, const void *data, long dataSize, void *(*newData)(long), int (*compare)(void *, void *), long (*copy)(void *, void *), int *compareResult);

LIBPFUI

void **pfiInit**(void);

pfiMotionCoord

pfType* **pfiGetMotionCoordClassType**(void);
pfiMotionCoord * **pfiNewMotionCoord**(void *arena);

pfiInputCoord

pfType* **pfiGetInputCoordClassType**(void);
pfiInputCoord * **pfiNewInputCoord**(void *arena);
void **pfiInputCoordVec**(pfiInputCoord *ic, float *vec);
void **pfiGetInputCoordVec**(pfiInputCoord *ic, float *vec);

pfiInputXform

Building user interfaces requires managing user input events. These functions provide a window system independent means of handling event streams.

pfiInput * **pfiNewInput**(void *arena);
void **pfiInputName**(pfiInput *in, const char *name);
const char * **pfiIsIXGetName**(pfiInput *in);
void **pfiInputFocus**(pfiInput *in, int focus);
int **pfiGetInputFocus**(pfiInput *in);
void **pfiInputEventMask**(pfiInput *in, int emask);
int **pfiGetInputEventMask**(pfiInput *in);
void **pfiInputEventStreamCollector**(pfiInput *in,
pfiEventStreamHandlerType func, void *data);
void **pfiGetInputEventStreamCollector**(pfiInput *in,
pfiEventStreamHandlerType *func, void **data);
void **pfiInputEventStreamProcessor**(pfiInput *in,
pfiEventStreamHandlerType func, void *data);
void **pfiGetInputEventStreamProcessor**(pfiInput *in,
pfiEventStreamHandlerType *func, void **data);
void **pfiInputEventHandler**(pfiInput *in, pfuEventHandlerFuncType func,
void *data);
void **pfiGetInputEventHandler**(pfiInput *in, pfuEventHandlerFuncType *func,
void **data);
void **pfiResetInput**(pfiInput *in);
void **pfiCollectInputEvents**(pfiInput *in);
void **pfiProcessInputEvents**(pfiInput *in);
int **pfiHaveFastMouseClick**(pfuMouse *mouse, int button, float msecs);
pfiInputXform * **pfiNewIXform**(void *arena);
void **pfiIXformFocus**(pfiInputXform *in, int focus);
int **pfiIsIXformInMotion**(pfiInputXform *ix);
void **pfiStopIXform**(pfiInputXform *ix);
void **pfiResetIXform**(pfiInputXform *ix);
void **pfiUpdateIXform**(pfiInputXform *ix);
void **pfiIXformMode**(pfiInputXform *ix, int mode, int val);
int **pfiGetIXformMode**(pfiInputXform *ix, int mode);

void	**pfiResetIXformPosition**(pfiInputXform *ix);
void	**pfiIXformMat**(pfiInputXform *ix, pfMatrix mat);
void	**pfiGetIXformMat**(pfiInputXform *ix, pfMatrix mat);
void	**pfiIXformInput**(pfiInputXform *ix, pfiInput *in);
pfiInput*	**pfiGetIXformInput**(pfiInputXform *ix);
void	**pfiIXformInputCoordPtr**(pfiInputXform *ix, pfiInputCoord *xcoord);
pfiInputCoord*	**pfiGetIXformInputCoordPtr**(pfiInputXform *ix);
void	**pfiIXformMotionCoord**(pfiInputXform *ix, pfiMotionCoord *xcoord);
void	**pfiGetIXformMotionCoord**(pfiInputXform *ix, pfiMotionCoord *xcoord);
void	**pfiIXformResetCoord**(pfiInputXform *ix, pfCoord *resetPos);
void	**pfiGetIXformResetCoord**(pfiInputXform *ix, pfCoord *resetPos);
void	**pfiIXformCoord**(pfiInputXform *ix, pfCoord *coord);
void	**pfiGetIXformCoord**(pfiInputXform *ix, pfCoord *coord);
void	**pfiIXformStartMotion**(pfiInputXform *xf, float startSpeed, float startAccel);
void	**pfiGetIXformStartMotion**(pfiInputXform *xf, float *startSpeed, float *startAccel);
void	**pfiIXformMotionLimits**(pfiInputXform *xf, float maxSpeed, float angularVel, float maxAccel);
void	**pfiGetIXformMotionLimits**(pfiInputXform *xf, float *maxSpeed, float *angularVel, float *maxAccel);
void	**pfiIXformDBLimits**(pfiInputXform *xf, pfBox *dbLimits);
void	**pfiGetIXformDBLimits**(pfiInputXform *xf, pfBox *dbLimits);
void	**pfiIXformBSphere**(pfiInputXform *xf, pfSphere *sphere);
void	**pfiGetIXformBSphere**(pfiInputXform *xf, pfSphere *sphere);
void	**pfiIXformUpudateFunc**(pfiInputXform *ix, pfiInputXformUpdateFuncType func, void *data);
void	**pfiGetIXformUpudateFunc**(pfiInputXform *ix, pfiInputXformUpdateFuncType *func, void **data);
void	**pfiIXformMotionFuncs**(pfiInputXform *ix, pfiInputXformFuncType start, pfiInputXformFuncType stop, void *data);
void	**pfiGetIXformMotionFuncs**(pfiInputXform *ix, pfiInputXformFuncType *start, pfiInputXformFuncType *stop, void **data);
pfiInputXformTrackball *	**pfiNewIXformTrackball**(void *arena);
void	**pfiIXformTrackballMode**(pfiInputXformTrackball *tb, int mode, int val);
int	**pfiGetIXformTrackballMode**(pfiInputXformTrackball *tb, int mode);
pfiInputXformTrackball *	**pfiCreate2DIXformTrackball**(void *arena);
int	**pfiUpdate2DIXformTrackball**(pfiInputXform *tb, pfiInputCoord *icoord, void *data);
pfType *	**pfiGetIXformTravelClassType**(void);
pfType *	**pfiGetIXformDriveClassType**(void);

pfType *	**pfiGetIXformFlyClassType**(void);
pfType *	**pfiGetIXformTrackballClassType**(void);
pfiInputXformDrive *	**pfiNewIXformDrive**(void *arena);
void	**pfiIXformDriveMode**(pfiInputXformDrive *drive, int mode, int val);
int	**pfiGetIXformDriveMode**(pfiInputXformDrive *drive, int mode);
void	**pfiIXformDriveHeight**(pfiInputXformDrive* drive, float height);
float	**pfiGetIXformDriveHeight**(pfiInputXformDrive* drive);
pfiInputXformDrive *	**pfiCreate2DIXformDrive**(void *arena);
int	**pfiUpdate2DIXformDrive**(pfiInputXform *drive, pfiInputCoord *icoord, void *data);
pfiInputXformFly *	**pfiNewIXFly**(void *arena);
void	**pfiIXformFlyMode**(pfiInputXformFly *fly, int mode, int val);
int	**pfiGetIXformFlyMode**(pfiInputXformFly *fly, int mode);
pfiInputXformFly *	**pfiCreate2DIXformFly**(void *arnea);
int	**pfiUpdate2DIXformFly**(pfiInputXform *fly, pfiInputCoord *icoord, void *data);

pfiCollide

For realistic motion through a scene, an application must detect collisions between the viewer and the scene. These functions provide that functionality. Typical uses of these utilities are to prevent movement through walls and to maintain a constant "driving" distance above the ground.

pfType *	**pfiGetCollideClassType**(void);
pfiCollide *	**pfiNewCollide**(void *arena);
void	**pfiEnableCollide**(pfiCollide *collide);
void	**pfiDisableCollide**(pfiCollide *collide);
int	**pfiGetCollideEnable**(pfiCollide *collide);
void	**pfiCollideMode**(pfiCollide *collide, int mode, int val);
int	**pfiGetCollideMode**(pfiCollide *collide, int mode);
void	**pfiCollideStatus**(pfiCollide *collide, int status);
int	**pfiGetCollideStatus**(pfiCollide *collide);
void	**pfiCollideDist**(pfiCollide *collide, float dist);
float	**pfiGetCollideDist**(pfiCollide *collide);
void	**pfiCollideHeightAboveGrnd**(pfiCollide *collide, float dist);
float	**pfiGetCollideHeightAboveGrnd**(pfiCollide *collide);
void	**pfiCollideGroundNode**(pfiCollide *collide, pfNode* ground);
pfNode *	**pfiGetCollideGroundNode**(pfiCollide *collide);
void	**pfiCollideObjNode**(pfiCollide *collide, pfNode* db);
pfNode *	**pfiGetCollideObjNode**(pfiCollide *collide);
void	**pfiGetCollideMotionCoord**(pfiCollide *collide, pfiMotionCoord* xcoord);

void **pfiCollideFunc**(pfiCollide *collide, pfiCollideFuncType func, void *data);
void **pfiGetCollisionFunc**(pfiCollide *collide, pfiCollideFuncType *func, void **data);
int **pfiUpdateCollide**(pfiCollide *collide);

pfiPick

The *pfiPick* utility facilitates user interaction and manipulation of a scene. It provides a means to translate mouse locations on the screen into the coordinate space of the world being viewed. Having done this, it can also determine what objects are being pointed to by the mouse.

pfType * **pfiGetPickClassType**(void);
pfiPick * **pfiNewPick**(void *arena);
void **pfiPickMode**(pfiPick *pick, int mode, int val);
int **pfiGetPickMode**(pfiPick *pick, int mode);
void **pfiPickHitFunc**(pfiPick *pick, pfiPickFuncType func, void *data);
void **pfiGetPicktHitFunc**(pfiPick *pick, pfiPickFuncType *func, void **data);
void **pfiAddPickChan**(pfiPick *pick, pfChannel *chan);
void **pfiInsertPickChan**(pfiPick *pick, int index, pfChannel *chan);
void **pfiRemovePickChan**(pfiPick *pick, pfChannel *chan);
int **pfiGetPickNumHits**(pfiPick *pick);
pfNode * **pfiGetPickNode**(pfiPick *pick);
pfGeoSet * **pfiGetPickGSet**(pfiPick *pick);
void **pfiSetupPickChans**(pfiPick *pick);
int **pfiDoPick**(pfiPick *pick, int x, int y);
void **pfiResetPick**(pfiPick *pick);

pfiXformer

pfiXformer objects provide a simple means for user-controlled motion in a scene. The **pfiXformer** updates a transformation matrix based on a selected motion model and user input. This transformation matrix can be used by the application for whatever purposes it desires. In particular, the matrix can be used to update the viewpoint defined for a **pfChannel** or the transformation of a **pfDCS** node.

pfType* **pfiGetXformerClassType**(void);
pfiXformer * **pfiNewXformer**(void* arena);
void **pfiXformerModel**(pfiXformer* xf, int index, pfiInputXform* model);
void **pfiSelectXformerModel**(pfiXformer* xf, int which);
pfiInputXform* **pfiGetXformerCurModel**(pfiXformer* xf);
int **pfiGetXformerCurModelIndex**(pfiXformer* xf);
int **pfiRemoveXformerModel**(pfiXformer* xf, int index);
int **pfiRemoveXformerModelIndex**(pfiXformer* xf, pfiInputXform* model);
void **pfiStopXformer**(pfiXformer* xf);
void **pfiResetXformer**(pfiXformer* xf);

void	**pfiResetXformerPosition**(pfiXformer* xf);
void	**pfiCenterXformer**(pfiXformer* xf);
void	**pfiXformerAutoInput**(pfiXformer* xf, pfChannel* chan, pfuMouse* mouse, pfuEventStream* events);
void	**pfiXformerMat**(pfiXformer* xf, pfMatrix mat);
void	**pfiGetXformerMat**(pfiXformer* xf, pfMatrix mat);
void	**pfiXformerModelMat**(pfiXformer* xf, pfMatrix mat);
void	**pfiGetXformerModelMat**(pfiXformer* xf, pfMatrix mat);
void	**pfiXformerCoord**(pfiXformer* xf, pfCoord *coord);
void	**pfiGetXformerCoord**(pfiXformer* xf, pfCoord *coord);
void	**pfiXformerResetCoord**(pfiXformer* xf, pfCoord *resetPos);
void	**pfiGetXformerResetCoord**(pfiXformer* xf, pfCoord *resetPos);
void	**pfiXformerNode**(pfiXformer* xf, pfNode *node);
pfNode *	**pfiGetXformerNode**(pfiXformer* xf);
void	**pfiXformerAutoPosition**(pfiXformer* xf, pfChannel *chan, pfDCS *dcs);
void	**pfiGetXformerAutoPosition**(pfiXformer* xf, pfChannel **chan, pfDCS **dcs);
void	**pfiXformerLimits**(pfiXformer* xf, float maxSpeed, float angularVel, float maxAccel, pfBox* dbLimits);
void	**pfiGetXformerLimits**(pfiXformer* xf, float *maxSpeed, float *angularVel, float *maxAccel, pfBox* dbLimits);
void	**pfiEnableXformerCollision**(pfiXformer* xf);
void	**pfiDisableXformerCollision**(pfiXformer* xf);
int	**pfiGetXformerCollisionEnable**(pfiXformer* xf);
void	**pfiXformerCollision**(pfiXformer* xf, int mode, float val, pfNode* node);
int	**pfiGetXformerCollisionStatus**(pfiXformer* xf);
void	**pfiUpdateXformer**(pfiXformer* xf);
int	**pfiCollideXformer**(pfiXformer* xf);
pfType*	**pfiGetTDFXformerClassType**(void);
pfiTDFXformer *	**pfiNewTDFXformer**(void* arena);
pfiXformer *	**pfiCreateTDFXformer**(pfiInputXformTrackball *tb, pfiInputXformDrive *drive, pfiInputXformFly *fly, void *arena);
void	**pfiTDFXformerStartMotion**(pfiTDFXformer* xf, float startSpeed, float startAccel, float accelMult);
void	**pfiGetTDFXformerStartMotion**(pfiTDFXformer* xf, float *startSpeed, float *startAccel, float *accelMult);
void	**pfiTDFXformerFastClickTime**(pfiTDFXformer* xf, float time);
float	**pfiGetTDFXformerFastClickTime**(pfiXformer* xf);
void	**pfiTDFXformerTrackball**(pfiTDFXformer *xf, pfiInputXformTrackball *tb);
pfiInputXformTrackball *	**pfiGetTDFXformerTrackball**(pfiTDFXformer *xf);
void	**pfiTDFXformerDrive**(pfiTDFXformer *xf, pfiInputXformDrive *tb);

pfiInputXformFly * **pfiGetTDFXformerFly**(pfiTDFXformer *xf);
void **pfiTDFXformerFly**(pfiTDFXformer *xf, pfiInputXformFly *tb);
pfiInputXformDrive * **pfiGetTDFXformerDrive**(pfiTDFXformer *xf);
int **pfiProcessTDFXformerMouseEvents**(pfiInput *, pfuEventStream *, void *data);
void **pfiProcessTDFXformerMouse**(pfiTDFXformer *xf, pfuMouse *mouse, pfChannel *inputChan);
void **pfiProcessTDFTrackballMouse**(pfiTDFXformer *xf, pfiInputXformTrackball *trackball, pfuMouse *mouse);
void **pfiProcessTDFTravelMouse**(pfiTDFXformer *xf, pfiInputXformTravel *tr, pfuMouse *mouse);

LIBPFUTIL

libpfutil Management

Before using any **libpfutil** utilities, the library must be initialized. These functions provide for proper initialization and control of **libpfutil**.

void **pfuInitUtil**(void);
pfDataPool* **pfuGetUtilDPool**(void);
void **pfuExitUtil**(void);
void **pfuDPoolSize**(long size);
long **pfuGetDPoolSize**(void);
volatile void* **pfuFindUtilDPData**(int id);

Processor Control

In certain circumstances, users may wish to control which CPU a particular IRIS Performer subprocess runs on. They might even wish to exclusively devote a particular processor to a given subprocess. These functions provide control of the scheduling of IRIS Performer subprocesses on a machine's processors.

int **pfuFreeCPUs**(void);
int **pfuRunProcOn**(int cpu);
int **pfuLockDownProc**(int cpu);
int **pfuLockDownApp**(void);
int **pfuLockDownCull**(pfPipe *);
int **pfuLockDownDraw**(pfPipe *);
int **pfuPrioritizeProcs**(int onOff);

Multiprocess Rendezvous

These rendezvous functions provide the functionality necessary for synchronizing master and slave processes in a multiprocessing environment.

void **pfuInitRendezvous**(pfuRendezvous *rvous, int numSlaves);
void **pfuMasterRendezvous**(pfuRendezvous *rvous);
void **pfuSlaveRendezvous**(pfuRendezvous *rvous, int id);

GLX Mixed Mode

The libpfutil GLX routines are now provided for compatibility with previous versions of Performer. New development should be done based on the pfWindow and pfPipeWindow API that provides a single API for managing IrisGL, Mixed Mode, and OpenGL windows.

pfuXDisplay * **pfuOpenXDisplay**(int screen);
pfuGLXWindow * **pfuGLXWinopen**(pfPipe *p, pfPipeWindow *pw, const char *name);
void **pfuGetGLXWin**(pfPipe *pipe, pfuGLXWindow *glxWin);
const char * **pfuGetGLXDisplayString**(pfPipe *pipe);
void **pfuGLMapcolors**(pfVec3 *clrs, int start, int num);
int **pfuGLXAllocColormap**(pfuXDisplay *dsp, pfuXWindow w);
void **pfuGLXMapcolors**(pfuXDisplay *dsp, pfuXWindow w, pfVec3 *clrs, int loc, int num);
void **pfuMapWinColors**(pfWindow *w, pfVec3 *clrs, int start, int num);
void **pfuMapPWinColors**(pfPipeWindow *pwin, pfVec3 *clrs, int start, int num);
void **pfuPrintWinFBConfig**(pfWindow *win, FILE *file);
void **pfuPrintPWinFBConfig**(pfPipeWindow *pwin, FILE *file);
pfFBConfig **pfuChooseFBConfig**(Display *dsp, int screen, int *constraints, void *arena);

Input Handling

These functions provide an interface for managing X and GL event streams.

pfuEventQueue * **pfuNewEventQ**(pfDataPool *dp, int id);
void **pfuResetEventStream**(pfuEventStream *es);
void **pfuResetEventQ**(pfuEventQueue *eq);
void **pfuAppendEventQ**(pfuEventQueue *eq0, pfuEventQueue *eq1);
void **pfuAppendEventQStream**(pfuEventQueue *eq, pfuEventStream *es);
void **pfuEventQStream**(pfuEventQueue *eq, pfuEventStream *es);
pfuEventStream * **pfuGetEventQStream**(pfuEventQueue *eq);
void **pfuGetEventQEvents**(pfuEventStream *events, pfuEventQueue *eq);
void **pfuIncEventQFrame**(pfuEventQueue *eq);
void **pfuEventQFrame**(pfuEventQueue *eq, int val);
int **pfuGetEventQFrame**(pfuEventQueue *eq);
void **pfuIncEventStreamFrame**(pfuEventStream *es);
void **pfuEventStreamFrame**(pfuEventStream *es, int val);
int **pfuGetEventStreamFrame**(pfuEventStream *es);
void **pfuInitInput**(pfPipeWindow *pw, int mode);

void **pfuExitInput**(void);
int **pfuMapMouseToChan**(pfuMouse *mouse, pfChannel *chan);
int **pfuMouseInChan**(pfuMouse *mouse, pfChannel *chan);
void **pfuCollectInput**(void);
void **pfuCollectGLEventStream**(pfuEventStream *events, pfuMouse *mouse,
int handlerMask, pfuEventHandlerFuncType handlerFunc);
void **pfuCollectXEventStream**(pfWSConnection dsp, pfuEventStream *events,
pfuMouse *mouse, int handlerMask,
pfuEventHandlerFuncType handlerFunc);
void **pfuGetMouse**(pfuMouse *mouse);
void **pfuGetEvents**(pfuEventStream *events);
void **pfuInputHandler**(pfuEventHandlerFuncType userFunc, uint mask);
void pfuMouseButtonClick(pfuMouse *mouse,
void pfuMouseButtonRelease(pfuMouse *mouse,
double **pfuMapXTime**(double xtime);

Cursor Control

Each window managed by Performer, both **pfWindows** and **pfPipeWindows**, can have an associated cursor. These functions can be used to manage the various cursors desired by an application.

Cursor **pfuGetInvisibleCursor**(void);
void **pfuLoadPWinCursor**(pfPipeWindow *w, Cursor c);
void **pfuLoadWinCursor**(pfWindow *w, Cursor c);
Cursor **pfuCreateDftCursor**(int index);
void **pfuCursor**(Cursor c, int index);
Cursor **pfuGetCursor**(int index);
void **pfuInitGUICursors**(void);
void **pfuGUICursor**(int target, Cursor c);
Cursor **pfuGetGUICursor**(int target);
void **pfuGUICursorSel**(Cursor sel);
Cursor **pfuGetGUICursorSel**(void);
void **pfuUpdateGUICursor**(void);

OpenGL X Fonts

It is convenient to be able to draw text in Performer windows. When programming with OpenGL, an application must use X fonts for this purpose. These functions simplify the use of X fonts for this purpose by hiding much of the low-level font management.

void **pfuLoadXFont**(char *fontName, pfuXFont *fnt);
void **pfuMakeXFontBitmaps**(pfuXFont *fnt);
void **pfuMakeRasterXFont**(char *fontName, pfuXFont *font);

void **pfuSetXFont**(pfuXFont *);
void **pfuGetCurXFont**(pfuXFont *);
int **pfuGetXFontWidth**(pfuXFont *, const char *);
int **pfuGetXFontHeight**(pfuXFont *);
void **pfuCharPos**(float x, float y, float z);
void **pfuDrawString**(const char *s);
void **pfuDrawStringPos**(const char *s, float x, float y, float z);

Simple GUI

Many applications require a simple user interface. Their needs are often far more restricted than the functionality provided by user interface libraries such as Motif. For those cases in which a simple and efficient user interface is required, these functions can be used to provide one.

void **pfuInitGUI**(pfPipeWindow *pw);
void **pfuExitGUI**(void);
void **pfuEnableGUI**(int en);
void **pfuUpdateGUI**(pfuMouse *mouse);
void **pfuRedrawGUI**(void);
void **pfuGUIViewport**(float l, float r, float b, float t);
void **pfuGetGUIViewport**(float *l, float *r, float *b, float *t);
int **pfuInGUI**(int x, int y);
void **pfuFitWidgets**(int val);
void **pfuGetGUIScale**(float *x, float *y);
void **pfuGetGUITranslation**(float *x, float *y);
void **pfuGUIHlight**(pfHighlight *hlight);
pfHighlight * **pfuGetGUIHlight**(void);
pfuPanel* **pfuNewPanel**(void);
void **pfuEnablePanel**(pfuPanel *p);
void **pfuDisablePanel**(pfuPanel *p);
void **pfuGetPanelOriginSize**(pfuPanel *p, float *xo, float *yo, float *xs, float *ys);
pfuWidget * **pfuNewWidget**(pfuPanel *p, int type, int id);
int **pfuGetWidgetType**(pfuWidget *w);
void **pfuEnableWidget**(pfuWidget *w);
void **pfuDisableWidget**(pfuWidget *w);
int **pfuGetWidgetId**(pfuWidget *w);
void **pfuWidgetDim**(pfuWidget *w, int xo, int yo, int xs, int ys);
void **pfuGetWidgetDim**(pfuWidget *w, int *xo, int *yo, int *xs, int *ys);
void **pfuWidgetLabel**(pfuWidget *w, const char *label);
int **pfuGetWidgetLabelWidth**(pfuWidget *w);
const char * **pfuGetWidgetLabel**(pfuWidget *w);

void	**pfuWidgetRange**(pfuWidget *w, int mode, float min, float max, float val);
void	**pfuWidgetValue**(pfuWidget *w, float val);
float	**pfuGetWidgetValue**(pfuWidget *w);
void	**pfuWidgetDefaultValue**(pfuWidget *w, float val);
void	**pfuWidgetDrawFunc**(pfuWidget *w, pfuWidgetDrawFuncType func);
void	**pfuWidgetSelectFunc**(pfuWidget *w, pfuWidgetSelectFuncType func);
void	**pfuWidgetActionFunc**(pfuWidget *w, pfuWidgetActionFuncType func);
pfuWidgetActionFuncType	**pfuGetWidgetActionFunc**(pfuWidget *w);
pfuWidgetSelectFuncType	**pfuGetWidgetSelectFunc**(pfuWidget *w);
pfuWidgetDrawFuncType	**pfuGetWidgetDrawFunc**(pfuWidget *w);
void	**pfuWidgetSelections**(pfuWidget *w, pfuGUIString *attrList, int *valList, void (**funcList)(pfuWidget *w), int numSelections);
void	**pfuWidgetSelection**(pfuWidget *w, int index);
int	**pfuGetWidgetSelection**(pfuWidget *w);
void	**pfuWidgetDefaultSelection**(pfuWidget *w, int index);
void	**pfuWidgetDefaultOnOff**(pfuWidget * w, int on);
void	**pfuWidgetOnOff**(pfuWidget *w, int on);
int	**pfuIsWidgetOn**(pfuWidget *w);
void	**pfuResetGUI**(void);
void	**pfuResetPanel**(pfuPanel *p);
void	**pfuResetWidget**(pfuWidget *w);
void	**pfuDrawTree**(pfChannel *chan, pfNode *node, pfVec3 panXYScale);
void	**pfuDrawMessage**(pfChannel *chan, const char *msg, int rel, int just, float x, float y, int size, int cmode);
void	**pfuDrawMessageCI**(pfChannel *chan, const char *msg, int rel, int just, float x, float y, int size, int textClr, int shadowClr);
void	**pfuDrawMessageRGB**(pfChannel *chan, const char *msg, int rel, int just, float x, float y, int size, pfVec4 textClr, pfVec4 shadowClr);

Scene Graph Traversal

Traversals are widely applicable to many tasks required in Performer applications. These functions provide a customizable, recursive traversal of an IRIS Performer scene graph.

int	**pfuTravCountNumVerts**(pfNode *node);
int	**pfuTraverse**(pfNode *node, pfuTraverser *trav);
void	**pfuInitTraverser**(pfuTraverser *trav);
void	**pfuTravCalcBBox**(pfNode *node, pfBox *box);
void	**pfuTravCountDB**(pfNode *node, pfFrameStats *fstats);
void	**pfuTravGLProf**(pfNode *node, int mode);
void	**pfuTravNodeAttrBind**(pfNode *node, uint attr, uint bind);

void **pfuTravNodeHlight**(pfNode *node, pfHighlight *hl);
void **pfuTravPrintNodes**(pfNode *node, const char *fname);
int **pfuCalcDepth**(pfNode *node);
void **pfuTravCachedCull**(pfNode* node, int numChans);

MultiChannel Option

These functions serve as a generic way of initializing channels when using the Multi-Channel Option (MCO) available on RealityEngine graphics systems.

void **pfuTileChans**(pfChannel **chn, int nChans, int ntilesx, int ntilesy);
void **pfuConfigMCO**(pfChannel **chn, int nChans);
int **pfuGetMCOChannels**(pfPipe *p);
void **pfuTileChan**(pfChannel **chn, int thisChan, int nChans, float l, float r, float b, float t);

MultiPipe Statistics

pfuManageMPipeStats provides a simple mechanism for acquiring frame timing statistics over a period of time and saving them to a disk file.

int **pfuManageMPipeStats**(int nFrames, int nSampledPipes);

Path Following

Automated path following can greatly simplify the construction of interactive walkthrough applications. These functions provide the means for creating and using automated paths.

pfuPath * **pfuNewPath**(void);
pfuPath * **pfuSharePath**(pfuPath *path);
pfuPath * **pfuCopyPath**(pfuPath *path);
pfuPath * **pfuClosePath**(pfuPath *path);
int **pfuFollowPath**(pfuPath *path, float seconds, pfVec3 where, pfVec3 orient);
int **pfuPrintPath**(pfuPath *path);
int **pfuAddPath**(pfuPath *path, pfVec3 first, pfVec3 final);
int **pfuAddArc**(pfuPath *path, pfVec3 center, float radius, pfVec2 angles);
int **pfuAddFillet**(pfuPath *path, float radius);
int **pfuAddSpeed**(pfuPath *path, float desired, float rate);
int **pfuAddDelay**(pfuPath *path, float delay);
int **pfuAddFile**(pfuPath *path, char *name);

Collision Detection

This is the old utility for collision detection. These functions are provided to ease the transition of existing Performer-based applications to the new API. They should not be used in developing new software and are likely to be removed in a future release. Refer to the reference pages for more information.

```
void        pfuCollisionChan(pfChannel *chan);
pfChannel*  pfuGetCollisionChan(void);
void        pfuCollideSetup(pfNode *node, int mode, int mask);
int         pfuCollideGrnd(pfCoord *coord, pfNode *node, pfVec3 zpr);
int         pfuCollideGrndObj(pfCoord *coord, pfNode *grndNode, pfVec3 zpr, pfSeg *seg,
                pfNode *objNode, pfVec3 hitPos, pfVec3 hitNorm);
int         pfuCollideObj(pfSeg *seg, pfNode *objNode, pfVec3 hitPos, pfVec3 hitNorm);
```

Timer Control

Tracking the passage of time is essential for interactive applications. Performer provides **pfuTimer** objects, which are both real-time and independent of frame rate.

```
pfuTimer*  pfuNewTimer(void *arena, int size);
void       pfuInitTimer(pfuTimer *timer, double start, double delta, void (*func)(pfuTimer*),
               void *data);
void       pfuStartTimer(pfuTimer *timer);
void       pfuStopTimer(pfuTimer *timer);
void       pfuEvalTimers(void);
int        pfuEvalTimer(pfuTimer *timer);
int        pfuActiveTimer(pfuTimer * timer);
```

Hash Tables

Hash tables are an ubiquitous data structure. They are used internally by Performer, and many Performer applications will find them very useful. These functions provide a simple hash table facility to all Performer-based systems.

```
pfuHashTable*  pfuNewHTable(int numb, int eltsize, void* arena);
void           pfuDelHTable(pfuHashTable* ht);
void           pfuResetHTable(pfuHashTable* ht);
pfuHashElt*    pfuEnterHash(pfuHashTable* ht, pfuHashElt* elt);
int            pfuRemoveHash(pfuHashTable* ht, pfuHashElt* elt);
int            pfuFindHash(pfuHashTable* ht, pfuHashElt* elt);
int            pfuHashGSetVerts(pfGeoSet *gset);
int            pfuCalcHashSize(int size);
```

Geometric Simplification

These functions can be used to automatically generate very simple level-of-detail representations of a subgraph from the bounding boxes of the geometric objects contained in that subgraph.

```
pfLOD*     pfuBoxLOD(pfGroup *grp, int flat, pfVec4* clr);
pfGeoSet*  pfuMakeBoxGSet(pfBox *box, pfVec4 clr, int flat);
```

Texture Loading

These functions assist in the sharing and downloading of textures, both of which are important for performance. Sharing of texture data reduces memory requirements and can subsequently increase efficiency. For consistent frame rates, it is also very important to download textures into the graphics pipeline's physical texture memory before beginning simulation.

pfTexture* **pfuNewSharedTex**(const char *filename, void *arena);
pfList* **pfuGetSharedTexList**(void);
pfList * **pfuMakeTexList**(pfNode *node);
pfList * **pfuMakeSceneTexList**(pfScene *node);
void **pfuDownloadTexList**(pfList *list, int style);
int **pfuGetTexSize**(pfTexture *tex);

Texture Animation

It may be necessary to animate textures to achieve specific visual effects. These functions allow the application to setup sequences of textures which define an animation.

void **pfuNewTexList**(pfTexture *tex);
pfList * **pfuLoadTexListFiles**(pfList *movieTexList, char nameList[][PF_MAXSTRING], int len);
pfList * **pfuLoadTexListFmt**(pfList *movieTexList, const char *fmtStr, int start, int end);
pfSequence * **pfuNewProjector**(pfTexture *handle);
int **pfuProjectorPreDrawCB**(pfTraverser *trav, void *travData);

Random Numbers

Generating good random numbers is very important for many simulation tasks. These functions provide a portable interface to the system random number generator which is somewhat more convenient than **random**.

void **pfuRandomize**(int seed);
long **pfuRandomLong**(void);
float **pfuRandomFloat**(void);
void **pfuRandomColor**(pfVec4 rgba, float minColor, float maxColor);

Flybox Control

These routines provide a simple interface to the BG Systems flybox but do not provide a flight model based on the flybox.

int **pfuOpenFlybox**(char *p);
int **pfuReadFlybox**(int *dioval, float *inbuf);
int **pfuGetFlybox**(float *analog, int *but);
int **pfuGetFlyboxActive**(void);
int **pfuInitFlybox**(void);

Smoke Simulation

These functions simulate the appearance of smoke and fire. They are included both as a utility in simulations as well as a demonstration of how to model such phenomena.

void **pfuInitSmokes**(void);
pfuSmoke * **pfuNewSmoke**(void);
void **pfuSmokeType**(pfuSmoke *smoke, int type);
void **pfuSmokeOrigin**(pfuSmoke* smoke, pfVec3 origin, float radius);
void **pfuSmokeDir**(pfuSmoke* smoke, pfVec3 dir);
void **pfuSmokeVelocity**(pfuSmoke* smoke, float turbulence, float speed);
void **pfuGetSmokeVelocity**(pfuSmoke* smoke, float *turbulence, float *speed);
void **pfuSmokeMode**(pfuSmoke* smoke, int mode);
void **pfuDrawSmokes**(pfVec3 eye);
void **pfuSmokeTex**(pfuSmoke* smoke, pfTexture* tex);
void **pfuSmokeDuration**(pfuSmoke* smoke, float dur);
void **pfuSmokeDensity**(pfuSmoke* smoke, float dens, float diss, float expansion);
void **pfuGetSmokeDensity**(pfuSmoke* smoke, float *dens, float *diss, float *expansion);
void **pfuSmokeColor**(pfuSmoke* smoke, pfVec3 bgn, pfVec3 end);

LightPointState Utilities

These functions can derive a texture image from a **pfLightPoint** specification.

void **pfuMakeLPStateShapeTex**(pfLPointState *lps, pfTexture *tex, int size);
void **pfuMakeLPStateRangeTex**(pfLPointState *lps, pfTexture *tex, int size, pfFog *fog);

Draw Styles

These functions demonstrate how to use multi-pass rendering to achieve various special drawing effects. Hidden line elimination and haloed lines are two examples of effects which can be created using these functions.

void **pfuPreDrawStyle**(int style, pfVec4 scribeColor);
void **pfuPostDrawStyle**(int style);
void **pfuCalcNormalizedChanXY**(float* px, float* py, pfChannel* chan, int xpos, int ypos);
int **pfuSaveImage**(char* name, int xorg, int yorg, int xsize, int ysize, int saveAlpha);

libpf

libpf is a high-level library for real-time graphics and visual simulation.

This library provides a scene graph structure and database traversals including view culling, rendering and collision detection in a multiprocessed environment.

NAME

pfNewBboard, pfGetBboardClassType, pfBboardPos, pfGetBboardPos, pfBboardMode, pfGetBboard-Mode, pfBboardAxis, pfGetBboardAxis – Create and update automatic rotation billboard nodes.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

pfBillboard * **pfNewBboard**(void);

pfType * **pfGetBboardClassType**(void);

void **pfBboardPos**(pfBillboard* bill, int i, const pfVec3 xyzOrigin);

void **pfGetBboardPos**(pfBillboard* bill, int i, pfVec3 xyzOrigin);

void **pfBboardMode**(pfBillboard* bill, int mode, int val);

int **pfGetBboardMode**(pfBillboard* bill, int mode);

void **pfBboardAxis**(pfBillboard* bill, const pfVec3 axis);

void **pfGetBboardAxis**(pfBillboard* bill, pfVec3 axis);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfBillboard** is derived from the parent class **pfGeode**, so each of these member functions of class **pfGeode** are also directly usable with objects of class **pfBillboard**. Casting an object of class **pfBillboard** to an object of class **pfGeode** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfGeode**.

int **pfAddGSet**(pfGeode* geode, pfGeoSet* gset);
int **pfRemoveGSet**(pfGeode* geode, pfGeoSet* gset);
int **pfInsertGSet**(pfGeode* geode, int index, pfGeoSet* gset);
int **pfReplaceGSet**(pfGeode* geode, pfGeoSet* old, pfGeoSet* new);
pfGeoSet * **pfGetGSet**(const pfGeode* geode, int index);
int **pfGetNumGSets**(const pfGeode* geode);

Since the class **pfGeode** is itself derived from the parent class **pfNode**, objects of class **pfBillboard** can also be used with these functions designed for objects of class **pfNode**.

pfGroup * **pfGetParent**(const pfNode *node, int i);
int **pfGetNumParents**(const pfNode *node);
void **pfNodeBSphere**(pfNode *node, pfSphere *bsph, int mode);
int **pfGetNodeBSphere**(pfNode *node, pfSphere *bsph);
pfNode* **pfClone**(pfNode *node, int mode);
pfNode* **pfBufferClone**(pfNode *node, int mode, pfBuffer *buf);

```
int             pfFlatten(pfNode *node, int mode);
int             pfNodeName(pfNode *node, const char *name);
const char *    pfGetNodeName(const pfNode *node);
pfNode*         pfFindNode(pfNode *node, const char *pathName, pfType *type);
pfNode*         pfLookupNode(const char *name, pfType* type);
int             pfNodeIsectSegs(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void            pfNodeTravMask(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint            pfGetNodeTravMask(const pfNode *node, int which);
void            pfNodeTravFuncs(pfNode* node, int which, pfNodeTravFuncType pre,
                    pfNodeTravFuncType post);
void            pfGetNodeTravFuncs(const pfNode* node, int which, pfNodeTravFuncType *pre,
                    pfNodeTravFuncType *post);
void            pfNodeTravData(pfNode *node, int which, void *data);
void *          pfGetNodeTravData(const pfNode *node, int which);
```

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfBillboard** can also be used with these functions designed for objects of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
void*   pfGetUserData(pfObject *obj);
```

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfBillboard** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *        pfGetType(const void *ptr);
int             pfIsOfType(const void *ptr, pfType *type);
int             pfIsExactType(const void *ptr, pfType *type);
const char *    pfGetTypeName(const void *ptr);
int             pfRef(void *ptr);
int             pfUnref(void *ptr);
int             pfUnrefDelete(void *ptr);
int             pfGetRef(const void *ptr);
int             pfCopy(void *dst, void *src);
int             pfDelete(void *ptr);
int             pfCompare(const void *ptr1, const void *ptr2);
void            pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *          pfGetArena(void *ptr);
```

DESCRIPTION

A pfBillboard is a pfGeode in which each pfGeoSet rotates to follow the eyepoint. Billboards are useful for complex objects which are roughly symmetrical about one or more axes. The billboard tracks the viewer by rotating about an axis or a point to present the same image to the viewer using far fewer polygons than a solid model. A classic example is a textured billboard of a single quadrilateral

representing a tree.

A pfBillboard can contain any number of pfGeoSets. pfGeoSets are added to and removed from the pfBillboard using the **pfAddGSet** and **pfRemoveGSet** routines used with pfGeodes. Each pfGeoSet rotates independently to follow the viewer. By convention, the pfGeoSet is rotated about the +Z axis so that the +Y axis points towards the eye point.

pfNewBboard creates and returns a handle to a pfBillboard. Like other pfNodes, pfBillboards are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetBboardClassType returns the **pfType*** for the class **pfBillboard**. The **pfType*** returned by **pfGetBboardClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfBillboard**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfBboardPos specifies the position *xyzOrigin* for the pfGeoSet with index *i*. **pfGetBboardPos** copies the position of the pfGeoSet with index *i* into *xyzOrigin*.

Billboards can either rotate about an axis or a point.

Axial billboards rotate about the axis specified by **pfBboardAxis**. The rotation is about the origin (0,0,0) of the pfGeoSet. In all cases, the geometry is modeled in the XZ plane, with +Y forward. When rendered, the billboard is rotated so that the -Y axis points back to the eye point. The +Z axis is the pfGeoSet's axis of rotation. An axial rotate billboard is specified by setting the **PFBB_ROT** *mode* of the billboard to the *value* **PFBB_AXIAL_ROT** using **pfBboardMode**. The axis of rotation (*x*, *y*, *z*) is specified using **pfBboardAxis**. **pfGetBboardAxis** returns the axis of the pfBillboard.

Point rotate billboards are useful for spherical objects or special effects such as smoke. They come in two varieties depending on how the remaining rotational degree of freedom is determined (rotating the -Y axis towards the eye, still leaves an arbitrary rotation about the pfGeoSet's Y axis).

> If the **PFBB_ROT** mode on the billboard is set to **PFBB_POINT_ROT_EYE**, the billboard is rotated so that the +Z axis of the pfGeoSet stays upright on the screen.
>
> If the **PFBB_ROT** mode on the billboard is set to **PFBB_POINT_ROT_WORLD**, the billboard is rotated so that the angle between the +Z axis of the pfGeoSet and axis specified with **pfBboardAxis** is minimized.

Both **PFBB_AXIAL_ROT** and **PFBB_POINT_ROT_WORLD** billboards may "spin" about the Y axis of the pfGeoSet when viewed along the rotation or alignment axis.

After the first **pfSync**, the number of pfGeoSets, the number and length of the primitives, and planarity of

the vertices should not be changed.

Some database formats may place a transformation above each billboard for positioning it. As with a pfGeode containing a small amount of geometry, having many billboards with transformation matrices above them can be expensive.

Since billboards always rotate towards the eyepoint, billboards in adjacent channels with the same eyepoint have the same orientation. Channels with different eyepoints will have different billboard orientations.

BUGS

Intersection traversals test only the pfBillboard's bounding volume, not its individual pfGeoSets.

pfFlatten only transforms the position of a billboard, not the axis and applies only a uniform scale to the billboard geometry.

SEE ALSO

pfAddGSet, pfRemoveGSet, pfChanTravMode, pfFlatten, pfLookupNode, pfNodeTravFuncs, pfScene, pfTransparency, pfDelete

NAME

pfNewBuffer, pfSelectBuffer, pfMergeBuffer, pfBufferScope, pfGetBufferScope, pfBufferAdd, pfBufferRemove, pfBufferInsert, pfBufferReplace, pfAsyncDelete, pfGetCurBuffer – Create, select, and merge a pfBuffer.

FUNCTION SPECIFICATION

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

pfBuffer *  pfNewBuffer(void);
void        pfSelectBuffer(pfBuffer* buf);
int         pfMergeBuffer(void);
void        pfBufferScope(pfBuffer *buf, pfObject *obj, int scope);
int         pfGetBufferScope(pfBuffer *buf, pfObject *obj);
int         pfBufferAdd(void *parent, void *child);
int         pfBufferRemove(void *parent, void *child);
int         pfBufferInsert(void *parent, int index, void *child);
int         pfBufferReplace(void *parent, void *oldChild, void *newChild);
int         pfAsyncDelete(void *mem);
pfBuffer*   pfGetCurBuffer(void);
```

PARAMETERS

buf identifies a pfBuffer

obj identifies a pfObject

DESCRIPTION

A pfBuffer is a data structure that logically encompasses **libpf** objects such as pfNodes. Newly created objects are automatically "attached" to the current pfBuffer specified by **pfSelectBuffer**. Later, any objects created in *buf* may be merged into the main IRIS Performer processing stream with **pfMergeBuffer**. In conjunction with a forked **DBASE** process (see **pfMultiprocess** and **pfDBaseFunc**), the pfBuffer mechanism supports asynchronous parallel creation and deletion of database objects. This is the foundation of a real-time database paging system.

pfNewBuffer creates and returns a handle to a pfBuffer.

pfSelectBuffer makes *buf* the current pfBuffer. Once *buf* is current, all subsequently created **libpf** objects will be automatically associated with *buf* and these objects may only be accessed through IRIS Performer routines when *buf* is the current pfBuffer (except for **pfBufferAddChild** and **pfBufferRemoveChild**, see the pfGroup man page). A given pfBuffer should only be current in a single process at any given time. In this way, a pfBuffer restricts access to a given object to a single process, avoiding hard-to-find errors due to multiprocessed data collisions. **pfGetCurBuffer** returns the current pfBuffer.

Only **libpf** objects are subject to pfBuffer access restrictions. **libpf** objects include pfNodes such as pfGroup, pfGeode and pfUpdatables such as pfLODState, pfChannel, pfEarthSky. **libpr** objects such as pfGeoSets, pfGeoStates, and pfMaterials have no pfBuffer restrictions so they may be accessed by any process at any time although care must be taken by the application to avoid multiprocessed collisions on these data structures.

pfMergeBuffer merges the current pfBuffer with the main IRIS Performer pfBuffer. This main pfBuffer is created by **pfConfig** and will resist deletion and merging and should only be made current in the **APP** process (however, it is legal to select a different buffer in the **APP** process). If called in a process other than the **APP**, **pfMergeBuffer** will block until the **APP** calls **pfSync**, at which time the **APP** will merge the current pfBuffer into the main pfBuffer and then allow the process that requested the merge to continue. If called in the **APP**, **pfMergeBuffer** will immediately execute the merge. After **pfMergeBuffer** returns, any objects that were created in the current pfBuffer may only be accessed in the **APP** process when the **APP** pfBuffer has been selected as the current pfBuffer. In other words, the merged pfBuffer has been "reset" and its objects now "exist" only in the **APP** pfBuffer. The addresses of **libpf** objects are not changed by **pfMergeBuffer**.

Any number of pfBuffers may be used and merged (**pfMergeBuffer**) by any number of processes for multithreaded database manipulation, subject to the following restrictions:

1. A given pfBuffer should be current (via **pfSelectBuffer**) in only a single process at any given time.
2. Each process which selects a pfBuffer must be **fork**ed, not **sproc**ed.

Specifically, pfBuffer usage is not restricted to the **DBASE** process (see **pfConfig**).

pfBufferAddChild and **pfBufferRemoveChild** provide access to nodes that do not exist in the current pfBuffer. Either, none, or both of *group* and *node* may exist outside the current pfBuffer. **pfBufferAddChild** and **pfBufferRemoveChild** act just like their non-buffered counterparts **pfAddChild** and **pfRemoveChild** except that the addition or removal request is not carried out immediately but is recorded by the current pfBuffer. The request is delayed until the first **pfMergeBuffer** when both *group* and *node* are found in the main IRIS Performer pfBuffer. The list of **pfBufferAddChild** and **pfBufferRemoveChild** requests is traversed in **pfMergeBuffer** after all nodes have been merged. **pfBufferAddChild** and **pfBufferRemoveChild** return **TRUE** if the request was recorded and **FALSE** otherwise.

In addition to the pfGroup-specific **pfBufferAddChild** and **pfBufferRemoveChild** routines, a pfBuffer allows generic list management for pfGroup, pfGeode, pfText, and pfPipeWindow objects. These functions, **pfBufferAdd**, **pfBufferRemove**, **pfBufferInsert**, **pfBufferReplace** can be used to manage a pfGroup's list of pfNodes, a pfGeode's list of pfGeoSets, a pfText's list of pfStrings, or a pfPipeWindow's list of pfChannels respectively. These routines infer the proper action to take from the argument types. For example, the following code fragment is equivalent to **pfBufferAddChild(group, geode)**:

```
pfGroup *group;
pfGeode *geode;

pfBufferAdd(group, geode);
```

pfBufferAdd, **pfBufferRemove**, **pfBufferInsert**, **pfBufferReplace** all act similarly in that they do not have effect until **pfMergeBuffer** is called and all parties have been merged into the main IRIS Performer buffer. They return -1 if the argument types are not consistent (e.g., pfBufferRemove(group, geoset)), 0 if the request is immediately processed (this happens when all parties already have scope in the current pfBuffer), and 1 if the request is buffered until the next **pfMergeBuffer**.

pfBufferScope sets the scope of *obj* with respect to pfBuffer *buf*. If *scope* is TRUE, then *obj* is "added" to *buf* so that when *buf* is made current (**pfSelectBuffer**) in a process, *obj* may be accessed through IRIS Performer routines in that same process. When *scope* is FALSE, *obj* is "removed" from *buf*. **pfBufferScope**'s primary purpose is to move objects between pfBuffers, particularly from the main **APP** pfBuffer into an application pfBuffer typically used for asynchronous database manipulations. In this case the object's scope would be set to FALSE in the old pfBuffer and TRUE in the new pfBuffer. It is undefined when an object has scope in multiple pfBuffers since this violates the multiprocessing data exclusion requirement of IRIS Performer. **pfGetBufferScope** returns TRUE or FALSE indicating the scope of *obj* in pfBuffer *buf*.

When using pfBuffers for database paging, it is sometimes desirable to retain certain, common database models ("library models") in memory. Examples are trees, houses, and other "culture" which are instanced on paged terrain patches. One instancing mechanism is to create the library models in one pfBuffer and later use **pfBufferAddChild** to attach the models to scene graphs created in another pfBuffer. This is classic instancing which uses transformations (pfSCS) to properly position the models. However, this mechanism suffers from 2 performance problems:

1. **pfMergeBuffer** will adversely impact the **APP** process, proportional to the number of **pfBufferAddChild** and **pfBufferRemoveChild** requests.
2. Transformations in the scene graph reduce IRIS Performer's ability to sort the database (see **pfChanBinSort**) and matrix operations have some cost in the graphics pipeline.

An alternative to classic instancing is "flattening" which creates a clone of the instanced subtree and then applies the transformation to all geometry in the cloned subtree. This method eliminates the performance problems listed above but does increase memory usage.

pfNode* **pfBufferClone**(pfNode *node, int mode, pfBuffer *buf)

is a version of **pfClone** which clones *node* and its subtree, which resides in *buf*, into the current pfBuffer. *mode* is the same argument as that passed to **pfClone** (it is currently ignored). Once cloned, a subtree may be flattened with **pfFlatten**:

Example 1: Instancing with pfBufferAddChild

```
libraryBuffer = pfNewBuffer();
pfSelectBuffer(libraryBuffer);

loadLibraryObjects();

pagingBuffer = pfNewBuffer();
pfSelectBuffer(pagingBuffer);

while (!done)
{
    pfNode     *newStuff;
    pfSCS      *treeLocation;

    /* Load new terrain tile or whatever */
    newStuff = loadStuff();

    /* Create pfSCS which is location of tree */
    treeLocation = pfNewSCS(treeMatrix);

    /* Add library model of a tree to treeLocation */
    pfBufferAddChild(treeLocation, libraryTree);

    /* Add instanced tree to newly loaded stuff */
    pfAddChild(newStuff, treeLocation);
}
```

Example 2: Instancing with pfBufferClone and pfFlatten

```
libraryBuffer = pfNewBuffer();
pfSelectBuffer(libraryBuffer);

loadLibraryObjects();

pagingBuffer = pfNewBuffer();
pfSelectBuffer(pagingBuffer);

while (!done)
{
```

```
    pfNode     *newStuff;
    pfSCS      *treeLocation;

    /* Load new terrain tile or whatever */
    newStuff = loadStuff();

    /* Create pfSCS which is location of tree */
    treeLocation = pfNewSCS(treeMatrix);

    /* Clone tree model from library into current, paging buffer */
    newTree = pfBufferClone(libraryTree, 0, libraryBuffer);

    /* Transform cloned tree */
    pfAddChild(treeLocation, newTree);
    pfFlatten(treeLocation);

    /* Get rid of unneeded treeLocation */
    pfRemoveChild(treeLocation, newTree);
    pfDelete(treeLocation);

    /* Add cloned, flattened tree to newly loaded stuff */
    pfAddChild(newStuff, newTree);
}
```

pfAsyncDelete is a special version of **pfDelete** which is useful for asynchronous database deletion. Instead of having immediate effect, **pfAsyncDelete** simply registers a deletion request at the time of invocation. These deletion requests are then processed in the **DBASE** trigger routine, **pfDBase** (**pfDBase** is automatically called if you have not registered a **DBASE** callback with **pfDBaseFunc**). Thus, if the **DBASE** processing stage is configured as its own process via **pfMultiprocess**, then the deletion will be carried out asynchronously without affecting (slowing down) the main processing pipelines.

pfAsyncDelete may be called from any process and returns -1 if *mem* is NULL or not derived from pfMemory and returns TRUE otherwise. Note that unlike **pfDelete pfAsyncDelete** does not check *mem*'s reference count and return TRUE or FALSE indicating whether *mem* was successfully deleted or not. Instead, the reference count check is delayed until the next call to **pfDBase**. At this time there is no way to query the success of an **pfAsyncDelete** request.

Note that **pfDBase** should only be called from within the database callback function (**pfDBaseFunc**) in the **DBASE** process just like **pfCull** and **pfDraw** should only be called in the pfChannel **CULL** and **DRAW** callbacks respectively (**pfChanTravFunc**).

Example 2: How to use a pfBuffer

```
/* Must create these in shared memory */
static pfGroup **Tiles;
static int     *TileStatus;

/*
 * Load new tiles and delete old ones.
 */
void
pageDBase(void *data)
{
    static pfBuffer    *buf = NULL;
    pfGroup            *root;

    if (buf == NULL)
    {
        buf = pfNewBuffer();
        pfSelectBuffer(buf);
    }

    /* Asynchronously delete unneeded tiles and update their status */
    for (allUnneededTiles)
    {
        /*
         * Scene does not have scope in 'buf' so use pfBufferRemoveChild
         * Tiles[i] is not really removed until pfMergeBuffer
         */
        pfBufferRemoveChild(Scene, Tiles[i]);

        /* Delete Tiles[i] at pfDBase time if Tiles[i] only has Scene as
           a parent.
         */
        pfAsyncDelete(Tiles[i]);

        /* Update tile status */
        TileStatus[i] = TILE_DELETED;
    }

    /*
     * Synchronously load needed tiles and update their status.
     */
    LoadNeededDatabaseTiles(Tiles, TileStatus);
```

```
    for (allLoadedTiles)
    {
        /*
         * Scene does not have scope in 'buf' so use pfBufferAddChild
         * loadedTile[i] is not really added until pfMergeBuffer
         */
        pfBufferAddChild(Scene, loadedTile[i]);
    }

    /*
     * Merge newly loaded tiles into main pfBuffer then carry out
     * all pfBufferAdd/RemoveChild requests.
     */
    pfMergeBuffer();

    /*
     * Carry out pfAsyncDelete requests. Call *after* pfMergeBuffer()
     * so that all pfBufferRemoveChild requests have been processed
     * and child reference counts have been properly decremented.
     */
    pfDBase();
}

    :

pfInit();
Tiles = pfMalloc(sizeof(pfGroup*) * NUM_TILES, pfGetSharedArena());
TileStatus = pfMalloc(sizeof(int) * NUM_TILES, pfGetSharedArena());
pfMultiprocess(PFMP_APP_CULL_DRAW | PFMP_FORK_DBASE);
pfConfig();
    :
pfDBaseFunc(pageDBase);

while(!done)
{
    pfSync();

    /* Remove and request deletion of unneeded tiles */
    UpdateTileStatus(Tiles, TileStatus);

    pfFrame();
}
```

NOTES

pfGetCurBuffer will return the **APP** pfBuffer immediately after **pfConfig** returns.

SEE ALSO

pfBufferAddChild, pfBufferRemoveChild, pfConfig, pfDBaseFunc, pfDelete, pfFrame, pfMultiprocess, pfGroup

NAME

pfNewChan, pfGetChanClassType, pfGetChanPipe, pfChanViewport, pfGetChanViewport, pfGetChanOrigin, pfGetChanSize, pfChanLODState, pfGetChanLODState, pfChanLODStateList, pfGetChanLODStateList, pfGetChanPWinIndex, pfGetChanPWin, pfChanTravFunc, pfGetChanTravFunc, pfAllocChanData, pfChanData, pfGetChanData, pfGetChanDataSize, pfPassChanData, pfClearChan, pfAttachChan, pfDetachChan, pfChanShare, pfGetChanShare, pfChanFOV, pfGetChanFOV, pfChanNearFar, pfGetChanNearFar, pfChanAutoAspect, pfGetChanAutoAspect, pfGetChanBaseFrust, pfGetChanPtope, pfMakePerspChan, pfMakeOrthoChan, pfMakeSimpleChan, pfGetChanFrustType, pfChanAspect, pfGetChanAspect, pfOrthoXformChan, pfGetChanNear, pfGetChanFar, pfGetChanEye, pfApplyChan, pfChanContainsPt, pfChanContainsSphere, pfChanContainsCyl, pfChanContainsBox, pfChanCullPtope, pfGetChanCullPtope, pfChanPick, pfChanNodeIsectSegs, pfChanScene, pfGetChanScene, pfChanESky, pfGetChanESky, pfChanGState, pfGetChanGState, pfChanGStateTable, pfGetChanGStateTable, pfChanStressFilter, pfGetChanStressFilter, pfChanStress, pfGetChanStress, pfGetChanLoad, pfChanTravMode, pfGetChanTravMode, pfChanTravMask, pfGetChanTravMask, pfChanBinSort, pfGetChanBinSort, pfChanBinOrder, pfGetChanBinOrder, pfChanView, pfGetChanView, pfChanViewMat, pfGetChanViewMat, pfChanViewOffsets, pfGetChanViewOffsets, pfGetChanOffsetViewMat, pfGetChanFStats, pfChanStatsMode, pfDrawChanStats, pfChanLODAttr, pfGetChanLODAttr, pfApp, pfCull, pfDraw, pfDrawBin, pfNodePickSetup – Set and get pfChannel definition parameters.

FUNCTION SPECIFICATION

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

pfChannel *       pfNewChan(pfPipe *pipe);
pfType *          pfGetChanClassType(void);
pfPipe *          pfGetChanPipe(const pfChannel *chan);
void              pfChanViewport(pfChannel* chan, float l, float r, float b, float t);
void              pfGetChanViewport(const pfChannel* chan, float* l, float* r, float* b, float* t);
void              pfGetChanOrigin(const pfChannel* chan, int *xo, int *yo);
void              pfGetChanSize(const pfChannel* chan, int *xs, int *ys);
void              pfChanLODState(pfChannel* chan, const pfLODState *ls);
void              pfGetChanLODState(const pfChannel* chan, pfLODState *ls);
void              pfChanLODStateList(const pfChannel* chan, pfList *lsList);
pfList*           pfGetChanLODStateList(const pfChannel* chan);
int               pfGetChanPWinIndex(pfChannel *chan);
pfPipeWindow *    pfGetChanPWin(pfChannel *chan);
```

void	**pfChanTravFunc**(pfChannel* chan, int trav, pfChanFuncType func);
pfChanFuncType	**pfGetChanTravFunc**(pfChannel* chan, int trav);
void *	**pfAllocChanData**(pfChannel* chan, int size);
void	**pfChanData**(pfChannel *chan, void *data, size_t size);
void *	**pfGetChanData**(pfChannel* chan);
size_t	**pfGetChanDataSize**(const pfChannel *chan);
void	**pfPassChanData**(pfChannel* chan);
void	**pfClearChan**(pfChannel* chan);
int	**pfAttachChan**(pfChannel* chan0, pfChannel* chan1);
int	**pfDetachChan**(pfChannel* chan0, pfChannel* chan1);
void	**pfChanShare**(pfChannel* chan, uint mask);
uint	**pfGetChanShare**(pfChannel* chan);
void	**pfChanFOV**(pfChannel* chan, float horiz, float vert);
void	**pfGetChanFOV**(const pfChannel* chan, float* horiz, float* vert);
void	**pfChanNearFar**(pfChannel* chan, float near, float far);
void	**pfGetChanNearFar**(const pfChannel* chan, float* near, float* far);
void	**pfChanAutoAspect**(pfChannel* chan, int which);
int	**pfGetChanAutoAspect**(const pfChannel* chan);
void	**pfGetChanBaseFrust**(const pfChannel* chan, pfFrustum *frust);
void	**pfGetChanPtope**(const pfChannel *chan, pfPolytope *ptope);
void	**pfMakePerspChan**(pfChannel* chan, float left, float right, float bottom, float top);
void	**pfMakeOrthoChan**(pfChannel* chan, float left, float right, float bottom, float top);
void	**pfMakeSimpleChan**(pfChannel* chan, float fov);
int	**pfGetChanFrustType**(const pfChannel* chan);
void	**pfChanAspect**(pfChannel* chan, int which, float widthHeightRatio);
float	**pfGetChanAspect**(const pfChannel* chan);
void	**pfOrthoXformChan**(pfChannel* dst, pfChannel* src, const pfMatrix mat);
void	**pfGetChanNear**(const pfChannel* chan, pfVec3 ll, pfVec3 lr, pfVec3 ul, pfVec3 ur);
void	**pfGetChanFar**(const pfChannel* chan, pfVec3 ll, pfVec3 lr, pfVec3 ul, pfVec3 ur);

int **pfGetChanEye**(const pfChannel* chan, pfVec3 eye);

void **pfApplyChan**(pfChannel *chan);

int **pfChanContainsPt**(const pfVec3 pt, pfChannel* chan);

int **pfChanContainsSphere**(const pfChannel* chan, const pfSphere* sph);

int **pfChanContainsCyl**(const pfChannel* chan, const pfCylinder* cyl);

int **pfChanContainsBox**(const pfChannel* chan, const pfBox* box);

void **pfChanCullPtope**(pfChannel *chan, const pfPolytope *ptope);

void **pfGetChanCullPtope**(const pfChannel *chan, pfPolytope *ptope);

int **pfChanPick**(pfChannel *chan, int mode, float px, float py, float radius, pfHit **picklist[]);

int **pfChanNodeIsectSegs**(pfChannel *chan, pfNode *node, pfSegSet *segSet, pfHit **hits[], pfMatrix *mat);

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

pfScene * **pfGetChanScene**(const pfChannel* chan);

void **pfChanESky**(pfChannel* chan, pfEarthSky *sky);

pfEarthSky * **pfGetChanESky**(const pfChannel* chan);

void **pfChanGState**(pfChannel *chan, pfGeoState *gstate);

pfGeoState* **pfGetChanGState**(const pfChannel *chan);

void **pfChanGStateTable**(pfChannel *chan, pfList *gstable);

pfList* **pfGetChanGStateTable**(const pfChannel *chan);

void **pfChanStressFilter**(pfChannel *chan, float frac, float low, float high, float scale, float max);

void **pfGetChanStressFilter**(const pfChannel *chan, float *frac, float *low, float *high, float *scale, float *max);

void **pfChanStress**(pfChannel *chan, float stress);

float **pfGetChanStress**(const pfChannel *chan);

float **pfGetChanLoad**(const pfChannel *chan);

void **pfChanTravMode**(pfChannel *chan, int trav, int mode);

int **pfGetChanTravMode**(const pfChannel *chan, int trav);

void **pfChanTravMask**(pfChannel* chan, int trav, uint mask);

uint	**pfGetChanTravMask**(const pfChannel* chan, int trav);
void	**pfChanBinSort**(pfChannel *chan, int bin, int sortType, int *sortOrders);
int	**pfGetChanBinSort**(const pfChannel *chan, int bin, int *sortOrders);
void	**pfChanBinOrder**(pfChannel *chan, int bin, int order);
int	**pfGetChanBinOrder**(const pfChannel *chan, int bin);
void	**pfChanView**(pfChannel* chan, pfVec3 xyz, pfVec3 hpr);
void	**pfGetChanView**(pfChannel* chan, pfVec3 xyz, pfVec3 hpr);
void	**pfChanViewMat**(pfChannel* chan, pfMatrix mat);
void	**pfGetChanViewMat**(const pfChannel* chan, pfMatrix mat);
void	**pfChanViewOffsets**(pfChannel* chan, pfVec3 xyz, pfVec3 hpr);
void	**pfGetChanViewOffsets**(const pfChannel* chan, pfVec3 xyz, pfVec3 hpr);
void	**pfGetChanOffsetViewMat**(const pfChannel *chan, pfMatrix mat);
pfFrameStats *	**pfGetChanFStats**(pfChannel* chan);
int	**pfChanStatsMode**(pfChannel* chan, uint mode, uint val);
void	**pfDrawChanStats**(pfChannel* chan);
void	**pfChanLODAttr**(pfChannel* chan, int attr, float val);
float	**pfGetChanLODAttr**(pfChannel* chan, int attr);
void	**pfApp**(void);
void	**pfCull**(void);
void	**pfDraw**(void);
void	**pfDrawBin**(int bin);
void	**pfNodePickSetup**(pfNode *node);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfChannel** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfChannel**. Casting an object of class **pfChannel** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void	**pfUserData**(pfObject *obj, void *data);
void*	**pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfChannel** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *      pfGetType(const void *ptr);
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

PARAMETERS

chan identifies a pfChannel.

node identifies a pfNode.

trav is a symbolic token identifying a traversal:

PFTRAV_CULL

PFTRAV_DRAW

DESCRIPTION

A pfChannel is essentially a view onto a scene. **pfNewChan** creates a new pfChannel on the pfPipe identified by *pipe*. The new pfChannel will be rendered by the *pipe* into a pfPipeWindow window associated with *pipe* (See **pfConfigPWin**). **pfNewChan** creates and returns a handle to a pfChannel. pfChannels are always allocated from shared memory.

pfGetChanClassType returns the **pfType*** for the class **pfChannel**. The **pfType*** returned by **pfGetChanClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfChannel**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

PIPE WINDOWS, PIPES, AND CHANNELS

pfGetChanPipe returns the parent pfPipe of *chan*. **pfGetChanPWin** returns the pfPipeWindow of *chan*.

Multiple pfChannels may be rendered by a single pfPipe into a single pfPipeWindow. It is recommended that multiple pfChannels rather than multiple pfPipes be used to render multiple views on a single

hardware pipeline. If necessary, multiple pfPipeWindows can be rendered by a single pfPipe on a single hardware pipeline. The handle returned by **pfNewChan** should be used to identify the pfChannel in IRIS Performer routines.

Upon creation, pfChannels are automatically assigned to the first pfPipeWindow of its parent pfPipe. **pfGetChanPWin** will return the pfPipeWindow of *chan*.

Channels of a pfPipeWindow are drawn in the order in which they are assigned to the pfPipeWindow. **pfGetChanPWinIndex** can be used to get the position of a channel in its pfPipeWindow list. A return value of (-1) indicates that the channel is not assigned to a pfPipeWindow. Channels can be re-ordered in their pfPipeWindow, or moved to other pfPipeWindows via list style API on pfPipeWindows. See the **pfAddChan**, **pfInsertChan**, **pfMoveChan**, and **pfRemoveChan** man pages for more information.

All active pfChannels are culled and drawn by **pfFrame**. A pfChannel is by default active but can be selectively turned on and off by **PFDRAW_ON** and **PFDRAW_OFF** arguments to **pfChanTravMode**. Multiple pfChannels on a pfPipe will be drawn only if they are assigned to a pfPipeWindow and will be drawn in the order they were assigned to that pfPipeWindow.

pfChanViewport specifies the fractional viewport used by *chan*. *l*, *r*, *b*, *t* specify the left, right, bottom, and top extents of a viewport in the range 0.0 to 1.0. The fractional viewport is relative to the parent pfPipe's graphics window. Channel viewports on a single pfPipe may overlap. Viewport extents are clamped to the range 0.0 to 1.0.

pfGetChanViewport copies the fractional viewport of *chan* into *l*, *r*, *b*, *t*.

pfGetChanOrigin copies the window coordinates of the origin of *chan*'s viewport into *xo* and *yo*.

pfGetChanSize copies the X and Y pixel sizes of *chan*'s viewport into *xs* and *ys*.

APPLICATION-DEFINED CALLBACKS AND DATA

Although IRIS Performer normally handles all culling and drawing, invocation of user written and registered extension functions (*callback functions*) is supported to allow custom culling and drawing by the application. Furthermore, IRIS Performer manages callback data such that when configured for multiprocessing, data contention and synchronization issues are handled transparently.

pfChanTravFunc sets the application, cull or draw-process callback functions for *chan*. The *trav* argument specifies which traversal is to be set and is one of: **PFTRAV_APP**, **PFTRAV_CULL** or **PFTRAV_DRAW**. User-data that is passed to these functions is allocated on a per-channel basis by **pfAllocChanData**. **pfAllocChanData** returns a pointer to a word-aligned buffer of shared memory of *size* bytes. Alternately, applications can provide passthrough data with **pfChanData**. *data* is a memory block of *size* bytes which should be allocated from a shared malloc arena visible to all IRIS Performer processes when multiprocessing (see **pfMultiprocess**).

pfGetChanDataSize returns the size of *chan*'s passthrough data block. **pfGetChanData** returns a pointer to a buffer that was set by **pfChanData** or allocated by **pfAllocChanData** or **NULL** if no buffer has been allocated or set. **pfChanTravFunc** returns the app, cull or draw callback functions for *chan* or **NULL** if the callback has not been set.

In order to propagate user data downstream to the cull and draw callbacks, **pfPassChanData** should be called whenever the user data is changed to indicate that the data should be "passed through" the IRIS Performer rendering pipeline. The next call to **pfFrame** will copy the channel buffer into internal IRIS Performer memory so that the application will then be free to modify data in the buffer without fear of corruption.

In the cull phase of the rendering pipeline, IRIS Performer invokes the cull callback with a pointer to the pfChannel being culled and a pointer to the pfChannel's data buffer. The cull callback may modify data in the buffer. The potentially modified buffer is then copied and passed to the user's draw callback. Modifications to the data buffer are not visible upstream. For example, changes made by the cull or draw process are not seen by the application process.

When IRIS Performer is configured for multiprocessing (see **pfMultiprocess**), it is important to realize that the cull and draw callbacks may be invoked from different processes and thus may run in parallel with each other as well as with the main application process. IRIS Performer provides both shared arenas (see **pfGetSemaArena** and **pfGetSharedArena**) and channel data (**pfAllocChanData**) for interprocess communication.

With user callbacks, it is possible to extend or even completely replace IRIS Performer actions with custom traversal, culling and drawing. **pfApp**, **pfCull** and **pfDraw** trigger the default IRIS Performer processing. This default processing is invoked automatically in the absence of any user callbacks specified by **pfChanTravFunc** , otherwise the user callback usually invokes them directly.

pfApp carries out the application traversal for the channel and should only be invoked in the application callback specified by **pfChanTravFunc**. The application callback is invoked once for each channel group that is sharing **PFCHAN_APPFUNC**.

pfCull should only be called in the cull callback and causes IRIS Performer to cull the current channel and generate an IRIS Performer display list (see **pfDispList**) suitable for rendering if the **PFMP_CULL_DL_DRAW** multiprocessing mode is enabled (see **pfMultiprocess**). Then, in the draw callback only, **pfDraw** will traverse the pfDispList and send rendering commands to the graphics hardware, thus drawing the scene.

If the **PFMP_CULL_DL_DRAW** multiprocessing mode is not set then all display-listable operations will be applied directly to the graphics pipeline rather than accumulated in a pfDispList for subsequent drawing. In essence, the draw process does the work of both **pfCull** and **pfDraw** without the intermediate step of building a pfDispList. This mode avoids the overhead of building and traversing a pfDispList but consequently is not suitable for multipass renderings which require multiple invocations of **pfDraw**.

When the draw callback is invoked, the graphics context will already have been properly configured for drawing the pfChannel. Specifically, the viewport, perspective and viewing matrices are set to the correct values. In addition, graphics library light sources corresponding to the active pfLightSources in the scene will be enabled so that geometry rendered in the draw callback will be properly lit. User modifications of this initial state are not reset by **pfDraw**.

If a draw callback is specified, IRIS Performer will not automatically clear the viewport, leaving control of this to the application. **pfClearChan** called from the draw callback will clear the channel viewport. If *chan* has a pfEarthSky (see **pfChanESky**), then the pfEarthSky will be drawn. Otherwise, the viewport will be cleared to black and the z-buffer cleared to its maximum value.

By default, **pfFrame** causes **pfCull** and **pfDraw** to be invoked for each active pfChannel. It is legal for the draw callback to call **pfDraw** more than once for multipass renderings.

Example 1: Set up channel callbacks and passthrough data

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

void cullFunc(pfChannel *chan, void *data);
void drawFunc(pfChannel *chan, void *data);

int
main()
{
    PassData    *pd;

    /* Initialize IRIS Performer */
    pfInit();
    pfConfig();

    /* Create and initialize pfChannel 'chan' */
    chan = pfNewChan(pfGetPipe(0));
     :

    /* Setup channel passthrough data */
    pd = (PassData*)pfAllocChanData(chan, sizeof(PassData));

    /* Bind cull and draw callback functions to channel */
    pfChanTravFunc(chan, PFTRAV_CULL, cullFunc);
    pfChanTravFunc(chan, PFTRAV_DRAW, drawFunc);
```

```
    pd->val = 0;
    pfPassChanData(chan);
    pfFrame();
     :
}

void
cullFunc(pfChannel *chan, void *data)
{
    PassData    *pd = (PassData*)data;
    pd->val++;
    pfCull();
}

void
drawFunc(pfChannel *chan, void *data)
{
    PassData    *pd = (PassData*)data;
    fprintf(stderr, "%ld\n", pd->val);
    pfClearChan(chan);
    pfDraw();
}
```

SHARING ATTRIBUTES THROUGH CHANNEL GROUPS

IRIS Performer supports the notion of a 'channel group' which is a collection of pfChannels that share certain attributes. A channel group is created by attaching a pfChannel to another with **pfAttachChan**. If *chan0* or *chan1* are themselves members of a channel group, then all channels that are grouped with either *chan0* or *chan1* are combined into a single channel group. All attached channels acquire the share mask and shared attributes of the channel group. A channel is removed from a channel group by **pfDetachChan**.

The attributes shared by the members of a channel group are specified by the *mask* argument to **pfChanShare**. By definition, all channels in a group have the same share mask. A pfChannel that is attached to a channel group inherits the share mask of the group. *mask* is a bitwise OR of the following tokens which enumerate the attributes that can be shared:

PFCHAN_FOV
Horizontal and vertical fields of view are shared.

PFCHAN_VIEW
The view position and orientation are shared.

PFCHAN_VIEW_OFFSETS
The XYZ and HPR offsets from the view direction are shared.

PFCHAN_NEARFAR
The near and far clip planes are shared.

PFCHAN_SCENE
All channels display the same scene.

PFCHAN_EARTHSKY
All channels display the same earth-sky model.

PFCHAN_STRESS
All channels use the same stress filter parameters.

PFCHAN_LOD
All channels use the same LOD modifiers.

PFCHAN_SWAPBUFFERS
All channels swap buffers at the same time, even when the channels are on multiple pfPipes.

PFCHAN_SWAPBUFFERS_HW
All channels swap buffers at the same time. The **GANGDRAW** feature of the **mswapbuffers** function is used to synchronize buffer swapping through hardware interlocking. This feature can synchronize graphics pipelines across multiple machines.

PFCHAN_STATS_DRAWMODE
All channels draw the same statistics graph.

PFCHAN_APPFUNC
The application callback is invoked once for all channels sharing **PFCHAN_APPFUNC**.

PFCHAN_CULLFUNC
All channels invoke the same channel cull callback.

PFCHAN_DRAWFUNC
All channels invoke the same channel draw callback.

PFCHAN_VIEWPORT
All channels use the same viewport specification.

pfGetChanShare returns the share mask of *chan*. The default attributes cause channels within a share group to share all attributes except **PFCHAN_VIEW_OFFSETS**, **PFCHAN_VIEWPORT** and **PFCHAN_SWAPBUFFERS_HW**.

Channel groups are useful for multichannel simulations where many of the viewing parameters are the same across pfChannels. For example, a 3-channel simulation consisting of left, middle, and right views

typically shares the near and far clipping planes. With a channel group, the clipping planes need only be set on a single pfChannel, say the middle one, and all other pfChannels in the group will acquire the same settings.

Example 1: Set up a single pipe, 3-channel simulation

```
left   = pfNewChan(pfGetPipe(0));
middle = pfNewChan(pfGetPipe(0));
right  = pfNewChan(pfGetPipe(0));

/* Form channel group with middle as the "master" */
pfAttachChan(middle, left);
pfAttachChan(middle, right);

/* Set FOV of all channels */
pfMakeSimpleFrust(middle, 45.0f);
pfChanAutoAspect(middle, PFFRUST_CALC_VERT);

/* Set clipping planes of all channels */
pfChanNearFar(middle, 1.0f, 2000.0f);

pfSetVec3(hprOffsets, 0.0f, 0.0f, 0.0f);
pfSetVec3(xyzOffsets, 0.0f, 0.0f, 0.0f);

/*
 * Set up viewport and viewing offsets.
 * Note that these are not shared by default.
 */
pfChanViewport(left, 0.0f, 1.0f/3.0f, 0.0f, 1.0f);
hprOffsets[PF_H] = 45.0f;
pfChanViewOffsets(left, xyzOffsets, hprOffsets);

pfChanViewport(middle, 1.0f/3.0f, 2.0f/3.0f, 0.0f, 1.0f);
hprOffsets[PF_H] = 0.0f;
pfChanViewOffsets(middle, xyzOffsets, hprOffsets);

pfChanViewport(right, 2.0f/3.0f, 1.0f, 0.0f, 1.0f);
hprOffsets[PF_H] = -45.0f;
pfChanViewOffsets(right, xyzOffsets, hprOffsets);
```

VIEWING FRUSTUM

Many pfChannel frustum routines are borrowed from pfFrustum. These routines have the identical function as the pfFrustum routines but operate on the pfChannel's internal viewing frustum. The routine correspondence is listed in the following table.

pfChannel routine	pfFrustum routine
pfMakeSimpleChan	pfMakeSimpleFrust
pfMakePerspChan	pfMakePerspFrust
pfMakeOrthoChan	pfMakeOrthoFrust
pfChanNearFar	pfFrustNearFar
pfGetChanNearFar	pfGetFrustNearFar
pfGetChanFOV	pfGetFrustFOV
pfChanAspect	pfFrustAspect
pfGetChanAspect	pfGetFrustAspect
pfGetChanFrustType	pfGetFrustType
pfOrthoXformChan	pfOrthoXformFrust
pfGetChanNear	pfGetFrustNear
pfGetChanFar	pfGetFrustFar
pfGetChanEye	pfGetFrustEye
pfApplyChan	pfApplyFrust
pfChanContainsPt	pfFrustContainsPt
pfChanContainsSphere	pfFrustContainsSphere
pfChanContainsBox	pfFrustContainsBox
pfChanContainsCyl	pfFrustContainsCyl

The reader is referred to the pfFrustum man page for details on the function descriptions.

In addition to the pfFrustum routines, IRIS Performer provides the **pfChanFOV** and **pfChanAutoAspect** convenience routines.

The *horiz* and *vert* arguments to **pfChanFOV** specify total horizontal and vertical fields of view (FOV) in degrees. If either angle is <= 0.0 or >= 180.0, IRIS Performer will automatically compute that field of view based on the other specified field of view and the aspect ratio of the pfChannel viewport. If both angles are defaulted in this way, IRIS Performer will use its default of *horiz*=45.0 with *vert* matched to the aspect ratio of the pfChannel. Note that the aspect ratio of a pfChannel is defined by its fractional viewport as well as the pixel size of its physical display window.

pfChanFOV constructs a on-axis frustum, one where the line from the eyepoint passing through the center of the image is perpendicular to the projection plane. **pfMakeSimpleChan** also creates an on-axis frustum but both horizontal and vertical fields of view are specified with *fov*.

pfGetChanFOV copies the total horizontal and vertical fields of view into *horiz* and *vert* respectively. If an angle is matched to the aspect ratio of the pfChannel, then the computed angle is returned.

The *which* argument to **pfChanAutoAspect** specifies which FOV extent to automatically match to the aspect ratio of *chan*'s viewport. *which* is a symbolic token and is one of:

PFFRUST_CALC_NONE
Do not automatically modify field of view.

PFFRUST_CALC_HORIZ
Automatically modify horizontal FOV to match channel aspect.

PFFRUST_CALC_VERT
Automatically modify vertical FOV to match channel aspect.

Automatic aspect ratio matching is useful for situations where the initial size of the display window is not known or where the display window may change size during runtime. Aspect ratio matching guarantees that the image will not be distorted in either horizontal or vertical dimensions. **pfMakePerspChan** and **pfMakeOrthoChan** disable automatic aspect ratio matching since it is assumed that the viewing frustum aspect ratio is completely specified by these commands.

pfChanNearFar specifies the near and far clip distances of the viewing frustum. *near* and *far* are the positive, world-coordinate distances along the viewing ray from the eye point to the near and far clipping planes which are parallel to the viewing plane. **pfGetChanNearFar** copies the near and far clipping distances into *near* and *far*. The default values are 1.0 for the near plane and 1000.0 for the far plane.

pfGetChanBaseFrust copies the base viewing frustum of *chan* into *frust*. The base viewing frustum has its eyepoint at the origin and its viewing direction as the +Y axis. The base frustum of a pfChannel is transformed into world coordinates by the viewing transformation (see **pfChanView**). **pfOrthoXformChan** transforms the base frustum of *src* by *mat* and copies the result into the base frustum of the *dst* pfChannel. **pfGetChanPtope** copies the transformed base frustum into *dst*.

Example 1: Two equivalent ways of defining a typical viewing channel.

This method is the easiest and most common.

```
/* Set up a simple viewing frustum */
chan = pfNewChan(pipe0);

/*
 * Set horizontal FOV to 45 degrees and automatically match
 * vertical FOV to channel viewport.
 */
pfChanFOV(chan, 45.0f, -1.0f);
```

Here's how to do the same thing using the basic primitives.

```
/* Set up a simple viewing frustum */
chan = pfNewChan(pipe0);

/*
 * Set horizontal FOV to 45 degrees and automatically match
 * vertical FOV to channel viewport.
 */
pfMakeSimpleChan(chan, 45.0f);
pfChanAutoAspect(chan, PFFRUST_CALC_VERT);
```

Example 2: Set up a 4 channel, 4 pipe video wall with total horizontal and vertical FOVs of 90 degrees.

```
/*
 * ul == upper left  ur == upper right
 * ll == lower left  lr == lower right
 */
llChan = pfNewChan(pfGetPipe(0));
lrChan = pfNewChan(pfGetPipe(1));
urChan = pfNewChan(pfGetPipe(2));
ulChan = pfNewChan(pfGetPipe(3));

/* Form channel group with urChan as the "master" */
pfAttachChan(urChan, llChan);
pfAttachChan(urChan, lrChan);
pfAttachChan(urChan, ulChan);

/*
 * Share viewport but not field of view
 * in addition to the default shared attributes.
 */
share = pfGetChanShare(urChan);
pfChanShare(urChan, (share & ~PFCHAN_FOV) | PFCHAN_VIEWPORT );

/*
 * Set up off-axis viewing frusta which "tile" video wall.
 * pfChannel viewport aspect ratio must be 1:1 or image will
 * be distorted.
 */
pfMakePerspChan(llChan, -1.0f, 0.0f, -1.0f, 0.0f);
pfMakePerspChan(lrChan,  0.0f, 1.0f, -1.0f, 0.0f);
pfMakePerspChan(urChan,  0.0f, 1.0f,  0.0f, 1.0f);
pfMakePerspChan(ulChan, -1.0f, 0.0f,  0.0f, 1.0f);
```

```
pfChanNearFar(urChan, 1.0f, 2000.0f);
```

Example 3: Set up a single pipe, 3-channel simulation.

```
left   = pfNewChan(pfGetPipe(0));
middle = pfNewChan(pfGetPipe(0));
right  = pfNewChan(pfGetPipe(0));

/* Form channel group with middle as the "master" */
pfAttachChan(middle, left);
pfAttachChan(middle, right);

/* Set FOV of all channels */
pfMakeSimpleChan(middle, 45.0f);
pfChanAutoAspect(middle, PFFRUST_CALC_VERT);

/* Set clipping planes of all channels */
pfChanNearFar(middle, 1.0f, 2000.0f);

hprOffsets[PF_P] = 0.0f;
hprOffsets[PF_R] = 0.0f;
pfSetVec3(xyzOffsets, 0.0f, 0.0f, 0.0f);

/*
 * Set up viewport and viewing offsets.
 * Note that these are not shared by default.
 */
pfChanViewport(left, 0.0f, 1.0f/3.0f, 0.0f, 1.0f);
hprOffsets[PF_H] = 45.0f;
pfChanViewOffsets(left, hprOffsets, xyzOffsets);

pfChanViewport(middle, 1.0f/3.0f, 2.0f/3.0f, 0.0f, 1.0f);
hprOffsets[PF_H] = 0.0f;
pfChanViewOffsets(middle, hprOffsets, xyzOffsets);

pfChanViewport(right, 2.0f/3.0f, 1.0f, 0.0f, 1.0f);
hprOffsets[PF_H] = -45.0f;
pfChanViewOffsets(right, hprOffsets, xyzOffsets);
```

Example 4: Custom culling to pfChannel viewing frustum.

```
/*
 * User-supplied cull callback (see pfChanTravFunc)
 */
extern void
myCullFunc(pfChannel *chan, void *data)
{
    pfBox *boundingBox = (pfBox*)data;

    if (pfChanContainsBox(chan, boundingBox))
        drawGSetsWithinBoundingBox();
}
```

pfGetChanAutoAspect returns the aspect ratio matching mode of *chan*.

A pfChannel normally uses its viewing frustum for culling its pfScene (**pfChanScene**). However, a custom culling volume may be specified by **pfChanCullPtope**. If non-NULL, *ptope* identifies a pfPolytope which is used for scene culling. A copy of *ptope*, internal to *chan*, is transformed by *chan*'s viewing matrix before culling. If *ptope* is NULL, *chan* will use its view frustum for culling. A pfPolytope is a set of half spaces whose intersection defines a convex volume. Culling performance will be proportional to the number of facets in *ptope*. **pfGetChanCullPtope** copies the culling polytope of *chan* into *ptope*.

PICKING

pfChanPick is used for screen to world-space ray intersections on a pfChannel's scene. This operation is often referred to as picking. Intersections will only occur with parts of the database that are within the viewing frustum, and that are enabled for picking intersections. The return value of **pfChanPick** is the number of successful intersections with the channel scene according to *mode*.

picklist is a user-supplied pointer. Upon return, the address of an array of pointers to pfHit objects is stored there. The pfHit objects come from an internally maintained pool and are reused on subsequent calls. Hence, the contents are only valid until the next invocation of **pfChanPick** in the current process. They should not be deleted by the application.

The contents of the pfHit object are queried using **pfQueryHit** and **pfMQueryHit**. See the man pages for **pfHit** and **pfNodeIsectSegs** for a description of the queries.

mode specifies the behavior of the traversal and type of information that will be returned from the picking process.

mode is a bitwise OR of tokens. In addition to those tokens that can be specified to **pfNodeIsectSegs** in the *mode* field of the pfSegSet, the following values are also allowed:

PFPK_M_NEAREST
Return the picking intersection closest to the viewpoint.

PFPK_M_ALL
Return all picking intersections.

PFTRAV_LOD_CUR
When traversing pfLODs, select the child to traverse based on range in the specified channel.

When **PFPK_M_ALL** is set, *picklist* will contain all of the successful picking intersections in order of increasing distance from the viewer eyepoint. See the **pfNodeIsectSegs** manual page for information on the **PFIS_** intersection tokens.

px, *py* identify a 2-dimensional point in normalized channel screen coordinates in the range 0.0 to 1.0 (with the lower left corner being (0.0, 0.0)), that corresponds to the channel location to be used for picking. This 2-dimensional point is used to create a ray from the viewer eyepoint through the near clipping plane to intersect with the channel scene.

radius is the radius of the picking region in normalized channel coordinates used for the picking of lines. This argument is provided for coarse picking, and possibly for eventual picking of lines and points which is currently not implemented. If *radius* is non-zero, then the *mode* argument must not specify the **PFTRAV_IS_PRIM** mode.

pfNodePickSetup enables the entire database tree under *node* for picking intersections and should be called with a pointer to the pfChannel's scene graph. This effectively calls **pfNodeTravMask** with **PFIS_SET_PICK**. Selective picking can be done by calling **pfNodeTravMask**, setting the traversal to **PFTRAV_ISECT** and including **PFIS_SET_PICK** in the intersection mask for nodes that are to be enabled for picking intersections. The picking traversal will not continue past any node that has not been enabled for picking intersections. See the **pfNodeTravMask** manual page for more information on intersection setup.

pfChanNodeIsectSegs is identical to **pfNodeIsectSegs** except a pfChannel is provided for evaluating pfLODs during the intersection traversal. In addition, *mat* specifies an initial transform, allowing intersection traversals to begin at non-root nodes. All line segments in *segSet* will be transformed by *mat*. *mat* may be NULL if no initial transform is needed.

EARTH AND SKY

pfChanScene and **pfChanESky** set the pfScene and pfEarthSky that *chan* will cull and draw. **pfChanScene** increments the reference count of *scene* so that *scene* must first be removed from *chan* by **pfChanScene**(chan, NULL) before *scene* can be deleted with **pfDelete**.

pfGetChanScene and **pfGetChanESky** return the current pfScene and pfEarthSky for *chan*.

Example 1: Setting a pfChannel's pfScene.

```
void
cullFunc(pfChannel *chan, void *data)
{
    pfCull();
}

void
drawFunc(pfChannel *chan, void *data)
{
    pfClearChan(chan);
    pfDraw();
}

/* somewhere in application setup phase */
 :
/* set channel's scene */
pfChanScene(chan, scene);

/* bind cull and draw process callbacks */
pfChanTravFunc(chan, PFTRAV_CULL, cullFunc);
pfChanTravFunc(chan, PFTRAV_DRAW, drawFunc);
```

GEOSTATES

pfChanGState sets *chan*'s pfGeoState to *gstate*. If non-NULL, *gstate* is loaded before *chan*'s **DRAW** callback is invoked. Specifically, *gstate* is loaded with **pfLoadGState** so that the state encapsulated by *gstate* becomes the global state that may be inherited by other pfGeoStates within the scene graph. The pfGeoState state inheritance mechanism is described in detail in the pfGeoState man page. Note that the channel pfGeoState is loaded before any scene pfGeoState so that state elements in the scene pfGeoState override those in the channel's pfGeoState. **pfGetChanGState** returns the pfGeoState of *chan*.

pfChanGStateTable sets *chan*'s pfGeoState table to *gstable*. If non-NULL, *gstable* is made the global pfGeoState table with **pfApplyGStateTable** before *chan*'s **DRAW** callback is invoked. Any indexed pfGeoStates, either referenced by a pfScene (**pfSceneGStateIndex**) or by scene pfGeoSets (**pfGSetGStateIndex**) will be accessed through *gstable*. Indexed pfGeoStates are useful for efficiently managing a single database with multiple appearances, e.g., a normal vs. an infrared view of a scene would utilize 2 pfGeoState tables, each referencing a different set of pfGeoStates.

STRESS PROCESSING AND LEVEL-OF-DETAIL

IRIS Performer attempts to maintain the fixed frame rate set with **pfFrameRate** by manipulating levels-of-detail (LODs) to reduce graphics load when rendering time approaches a frame period. At the end of each frame, IRIS Performer computes a load metric for each pfChannel based on the length of time it took to render the pfChannel. Load is simply the actual rendering time divided by the desired frame interval.

pfChanLODState specifies a global pfLODState to be used for this channel.

pfChanLODStateList specifies a pfList of pfLODStates to be indexed into by pfLODs that have specified indexes via **pfLODLODStateIndex**. (See **pfLOD** and **pfLODState**).

If stress processing is enabled, IRIS Performer uses the load metric and a user-defined stress filter to compute a stress value which multiplies effective LOD ranges (see **pfLOD**) for the next frame. Stress > 1.0 'pushes out' LOD ranges so that coarser models are drawn and graphics load is reduced. Stress == 1.0 means the system is not in stress and LODs are not modified.

pfChanStressFilter sets the stress filter used by *chan*. *frac* is the fraction of a frame period *chan* is expected to take to render. *frac* should be 1.0 if only a single pfChannel is drawn on a pfPipe and should be > 0.0 and < 1.0 for multichannel simulations. *frac* allows the application to apportion rendering time amongst multiple channels so that a channel drawing a complex scene may be allocated more time than a channel drawing a simple one. **pfGetChanStressFilter** returns the stress filter parameters for *chan*.

low and *high* define a hysteresis band for system load. When load is >= *low* and <= *high*, stress is held constant. When load is < *low* or > *high*, IRIS Performer will reduce or increase stress respectively until load stabilizes within the hysteresis band. *low* should be <= *high* and they both should be positive. Stress is computed using the following algorithm:

```
/* increase stress when above high load level */
if (load > high)
    S[i] = minimum(S[i-1] + scale*load, max);
else
/* decrease stress when below low load level */
if (load < low)
    S[i] = maximum(S[i-1] - scale*load, 0.0f);
else
/* stress unchanged when between low and high load levels */
    S[i] = S[i-1];
```

where S[i] == stress for frame i and load = time[i] * frameRate / *frac*. By default, *scale* = 0.0 and *max* = 1.0 so that stress is disabled. Stress is clamped to the range [1.0, *max*].

pfChannels in a channel group may share a stress filter (**PFCHAN_STRESS**), and LOD behavior (-

PFCHAN_LOD) (see **pfAttachChan**). It is useful for pfChannels which draw into adjacent displays to share LOD behavior. In this case, the LOD multiplier used by all pfChannels in the channel group is the maximum of each individual pfChannel. This ensures that LOD's which straddle displays will always be drawn at the same LOD on each display.

pfGetChanLoad will return the last computed load for *chan*. The load value is defined as time * frameRate / *frac*.

The application may choose to not use the default IRIS Performer stress filter by calling **pfChanStress** to explicitly set the stress value. Stress values set by **pfChanStress** will override the default stress values computed by the stress filter shown above.

pfGetChanStress returns the last computed stress value for *chan*. The individual stress value is returned regardless of pfChannel attribute sharing (**pfChanShare**).

CUSTOMIZING SCENE GRAPH TRAVERSAL

A pfChannel directs two important traversals: cull and draw. In the cull traversal, the pfChannel defines the viewing frustum that the database is culled to and also defines other parameters that modify level-of-detail behavior. When drawing, the pfChannel defines the parameters of the "camera" which views the scene. In both cases, a pfChannel traverses a pfScene which is attached to the pfChannel via **pfChanScene**. A pfScene is a hierarchy of pfNodes that defines the visual database.

pfChanTravMode sets the traversal mode of *chan*. *trav* specifies a traversal type and is either **PFTRAV_CULL** or **PFTRAV_DRAW**, for the culling and drawing traversal respectively. *mode* specifies the corresponding traversal mode. The culling mode is a bitwise OR of:

PFCULL_VIEW
When set, **PFCULL_VIEW** enables culling to the viewing frustum. If not set, the entire database will be rendered every frame. For best drawing performance it is recommended that **PFCULL_VIEW** be set. Unless **PFCULL_GSET** is also set, IRIS Performer culls the database only down to the pfGeode level.

PFCULL_SORT
When **PFCULL_SORT** is set, IRIS Performer sorts the database into "bins" which are rendered in a user-specified order. In addition, geometry within a bin may be sorted by graphics state like texture or by range for front-to-back or back-to-front rendering. Unless the cull stage of the IRIS Performer pipeline becomes the bottleneck or **PFMP_CULLoDRAW** mode is used, **PFCULL_SORT** should be set for optimal drawing performance. Further sorting details are described below.

PFCULL_GSET
When **PFCULL_GSET** is set, IRIS Performer culls individual pfGeoSets within pfGeodes. At the expense of some extra culling time, this can provide a significantly tighter cull both because of the finer granularity and because pfGeoSet culling uses bounding boxes rather

than bounding spheres. However, when traversing portions of the scene graph under a transformation (pfSCS or pfDCS), IRIS Performer reverts back to a cull which stops at the pfGeode level.

PFCULL_IGNORE_LSOURCES
When **PFCULL_IGNORE_LSOURCES** is not set, IRIS Performer will traverse all paths in the scene hierarchy which end at a pfLightSource node before proceeding with the normal cull traversal (see **pfLightSource**). This is required for pfLightSources to illuminate the scene and will ensure that graphics hardware lighting is properly configured before the user's draw callback is invoked (see **pfChanTravFunc**). If it is set, any pfLightSources in the pfScene will be ignored.

The pfLightSource cull traversal obeys all traversal rules such as node callbacks, traversal masks, transformations (pfSCS and pfDCS nodes), and selectors (pfSwitch and pfLOD).

For drawing, *mode* is either **PFDRAW_OFF** or **PFDRAW_ON**. **PFDRAW_OFF** essentially turns off *chan*. No culling or drawing traversal will take place. Drawing is enabled by default. **pfGetChanTravMode** returns the mode corresponding to *trav* or -1 if *trav* is an illegal or unknown traversal type.

The **PFTRAV_MULTIPASS** traversal mode is only active when the pfChannel's scene has one or more pfLightSources which use projected texture-type lighting. See the pfLightSource man page for more details.

By default, culling to the viewing frustum, culling to pfGeoSet bounding boxes, pfLightSource culling, and sorting is enabled: (**PFCULL_VIEW** | **PFCULL_GSET** | **PFCULL_SORT**) For convenience, this default bitmask is provided by the **PFCULL_ALL** token.

pfChanTravMask sets *chan*'s drawing mask and is used in conjunction with **pfNodeTravMask** for selective culling and drawing of scene graphs on a per-pfChannel basis. During the traversal, the bitwise AND of the traversal mask and the node mask is computed. If the result is non-zero, the node is culled or drawn as usual. If off (zero), the behavior is as follows depending on *trav*:

PFTRAV_CULL
Node is not culled and is considered to be entirely within the viewing frustum. The cull traversal traverses the node and its children without any view culling.

PFTRAV_DRAW
Node is completely ignored. Both cull and draw traversals skip the node and its children.

Node traversal masks are set by **pfNodeTravMask**. The default pfNode and pfChannel masks are 0xffffffff so that a pfChannel culls and draws all pfNodes.

pfGetChanTravMask returns the drawing traversal mask for the specified pfChannel. *trav* is either **PFTRAV_CULL** or **PFTRAV_DRAW**.

As mentioned above, pfChannels can sort the database for improved image quality and improved rendering performance. Database sorting consists of two steps:

1. Partition database into "bins" which are rendered in a particular order.
2. Sort database within each bin by:

2a. Graphics state, in which case there is no particular rendering order or,

2b. Range from the eyepoint in which case the database is rendered either front-to-back or back-to-front.

During the cull traversal, pfGeoSets are placed into the appropriate bin according to their bin identifier that was set by **pfGSetDrawBin**. If the bin identifier is >= 0, the cull traversal will place that pfGeoSet into the bin with that identifier. If the bin identifier is < 0, then the cull traversal will decide in which default bin the pfGeoSet belongs.

IRIS Performer provides 2 default bins: **PFSORT_OPAQUE_BIN** and **PFSORT_TRANSP_BIN** for opaque and transparent geometry respectively. Transparent geometry is that which uses **PFTR_BLEND_ALPHA** type of **pfTransparency**. **PFTR_MS_ALPHA**-type transparency is considered to be opaque for purposes of binning.

Each draw bin has a rendering order set by **pfChanBinOrder**. If *order* is < 0, then *bin* is not ordered at all - pfGeoSets which belong to *bin* are not stored in the bin but are rendered immediately. If *order* is >=0, it defines the order in which the bin is rendered, 0 == first, 1 == second etc. The **PFSORT_OPAQUE_BIN** bin has a default rendering order of 0 and the **PFSORT_TRANSP_BIN** bin has a default rendering order of 1 so that transparent surfaces are rendered after opaque surfaces. It is legal to change the rendering order of the default bins and for different bins to have the same rendering order although the relative order of these bins is undefined.

Normally, **pfDraw** renders all bins in the appropriate order. Individual bins may be rendered with **pfDrawBin** when called in the pfChannel's draw callback (see **pfChanTravFunc**).

pfChanBinSort defines how pfGeoSets are sorted with a bin. *sortType* is a symbolic token which identifies the sorting method for *bin*:

PFSORT_NO_SORT
Do not sort the bin. *sortOrders* is ignored.

PFSORT_FRONT_TO_BACK
Sort the pfGeoSets in the bin in increasing range from the eyepoint. Range is computed as the distance from the pfChannel eyepoint to the center of the pfGeoSet's bounding box. *sortOrders* is ignored.

PFSORT_BACK_TO_FRONT
Sort the pfGeoSets in the bin in decreasing range from the eyepoint. Range is computed as the distance from the pfChannel eyepoint to the center of the pfGeoSet's bounding box. *sortOrders* is ignored.

PFSORT_BY_STATE
Sort the pfGeoSets in the bin by graphics state. The pfGeoSets in *bin* are first sorted by pfGeoState. Then if *sortOrders* is not NULL, the pfGeoSets will be further sorted by the ordered list of **PFSTATE_*** elements in *sortOrders*. In this case, *sortOrders* should consist of a **PFSORT_STATE_BGN** token followed by 0 or more **PFSTATE_*** tokens followed by a **PFSORT_STATE_END** token followed by a **PFSORT_END** token to end the list. The **PFSTATE_*** tokens define a sorting hierarchy. The elements in *sortOrders* are copied into the pfChannel data structure, so in this case it is acceptable to pass static or automatic data not allocated through **pfMalloc**.

Example 1: Sorting configuration example

```
int                 sortOrders[PFSORT_MAX_KEYS], i = 0;

sortOrders[i++] = PFSORT_STATE_BGN;
sortOrders[i++] = PFSTATE_FOG;
sortOrders[i++] = PFSTATE_MATERIAL;
sortOrders[i++] = PFSTATE_TEXTURE;
sortOrders[i++] = PFSORT_STATE_END;
sortOrders[i++] = PFSORT_END;

pfChanBinSort(chan, PFSORT_OPAQUE_BIN, PFSORT_BY_STATE, sortOrders);
pfChanBinSort(chan, PFSORT_TRANSP_BIN, PFSORT_BACK_TO_FRONT, NULL);
```

The default sorting order for the **PFSORT_OPAQUE_BIN** bin is by pfGeoState only and the default sorting order for the **PFSORT_TRANSP_BIN** bin is **PFSORT_BACK_TO_FRONT**.

Sorting by state is limited to the scope of a transformation (pfDCS or pfSCS) or a node with draw callbacks, i.e. - pfGeoSets affected by different transformations or draw callbacks are not sorted together. However, range sorting spans both transformation and draw callback boundaries. Thus a range-sorted scene graph with many transformations and expensive draw callbacks may suffer reduced performance due to an increased number of transformation and draw callback changes.

VIEWPOINT AND CAMERA SPECIFICATION

pfChanView specifies both the origin and direction of view for a pfChannel. *xyz* specifies the x,y,z position of the viewpoint in world coordinates and *hpr* specifies the Euler angles (heading, pitch, and roll) in degrees of the viewing direction relative to the nominal view (as defined below). The order of application

of these angles is ROTy(roll) * ROTx(pitch) * ROTz(heading) where ROTa(angle) is a rotation matrix about world axis *a* of *angle* degrees. In all cases a positive rotation is counterclockwise by the right hand rule. The nominal viewing coordinate system is +Y = forward, +Z = up, +X = right. For example, a roll of 90 degrees and a heading of -90 degrees would align the view direction with the +X world axis and the up direction with the -Y world axis.

pfChanViewMat provides another means of specifying view point and direction. *mat* is a 4x4 homogeneous matrix which defines the view coordinate system such that the upper 3x3 submatrix defines the coordinate system axes and the bottom vector defines the coordinate system origin. IRIS Performer defines the view direction to be along the positive Y axis and the up direction to be the positive Z direction, e.g., the second row of *mat* defines the viewing direction and the third row defines the up direction in world coordinates. *mat* must be orthonormal or results are undefined.

The actual viewing direction used for culling and drawing is modified by the offsets specified by **pfChanViewOffsets**. The argument *xyz* defines a translation from the nominal eyepoint. The Euler angles given in *hpr* define an additional rotation of the viewing direction from that specified by **pfChanView** and **pfChanViewMat**. Although this has similar functionality to **pfChanView**, it is specifically useful for applications which render the same scene into adjacent displays using multiple pfChannels. Two examples where one would use **pfChanViewOffsets** as well as **pfChanView** are offset-eye stereo image viewing applications, and for video wall applications.

Example 1: Set up a single pipe, 3-channel simulation using pfChanViewOffsets.

```
left   = pfNewChan(pfGetPipe(0));
middle = pfNewChan(pfGetPipe(0));
right  = pfNewChan(pfGetPipe(0));

/* Form channel group with middle as the "master" */
pfAttachChan(middle, left);
pfAttachChan(middle, right);

/* Set FOV of all channels */
pfMakeSimpleChan(middle, 45.0f, 45.0f);
pfChanAutoAspect(middle, PFFRUST_CALC_VERT);

/* Set clipping planes of all channels */
pfChanNearFar(middle, 1.0f, 2000.0f);

hprOffsets[PF_P] = 0.0f;
hprOffsets[PF_R] = 0.0f;
pfSetVec3(xyzOffsets, 0.0f, 0.0f, 0.0f);

/*
```

```
 * Set up viewport and viewing offsets.
 * Note that these are not shared by default.
 */
pfChanViewport(left, 0.0f, 1.0f/3.0f, 0.0f, 1.0f);
hprOffsets[PF_H] = 45.0f;
pfChanViewOffsets(left, hprOffsets, xyzOffsets);

pfChanViewport(middle, 1.0f/3.0f, 2.0f/3.0f, 0.0f, 1.0f);
hprOffsets[PF_H] = 0.0f;
pfChanViewOffsets(middle, hprOffsets, xyzOffsets);

pfChanViewport(right, 2.0f/3.0f, 1.0f, 0.0f, 1.0f);
hprOffsets[PF_H] = -45.0f;
pfChanViewOffsets(right, hprOffsets, xyzOffsets);
```

Both translation and rotational offsets are encoded in the graphics library's ModelView matrix. This ensures that fogging is consistent across multiple, adjacent pfChannels. However, proper lighting requires a lighting model which specifies a local viewer. Otherwise, geometry which spans multiple pfChannels will be lit differently on each pfChannel.

Example 2: Local viewer lighting model

```
pfLightModel         *lm;

lm = pfNewLModel(arena);
pfLModelLocal(lm, 1);
pfApplyLModel(lm);
```

pfGetChanView copies the view point/direction into *xyz* and *hpr*.

pfGetChanViewMat copies the viewing matrix (without viewing offsets) into *mat*.

pfGetChanViewOffsets copies the view positional and rotational offsets into the indicated arrays (*xyz* and *hpr*).

pfGetChanOffsetViewMat copies the combined nominal and offset viewing matrices into *mat*. This combined viewing matrix is that used for culling and for configuring the graphics library with the appropriate transformation. It is defined as **offset** * **nominal** where **offset** is specified by **pfChanViewOffsets** and **nominal** is specified by either **pfChanViewMat** or **pfChanView**.

DRAWING FRAME STATISTICS

IRIS Performer keeps track of times spent, and operations done, in the application, cull, and draw stages of the rendering pipeline and accumulates the data in a pfFrameStats structure. **pfGetChanFStats** is used to get this pfFrameStats structure from the indicated channel. **pfChanStatsMode** selects which of the enabled statistics classes should be displayed in that channel by **pfDrawChanStats** or **pfDrawFStats**.

pfDrawChanStats or **pfDrawFStats** must be called during each frame that a statistics display is desired and may be called from any of IRIS Performer's application, cull, or draw processes. This manual page give some pointers on how to interpret the statistics to help in tuning your database. Refer to the IRIS Performer Programming Guide for more detailed information.

pfChanStatsMode takes a pointer to a pfChannel, *chan*, and *mode*, which is currently just **PFCSTATS_DRAW**, and the corresponding value for *val*, which is a statistics class enabling bitmask. The statistics classes displayed by **pfDrawChanStats** or **pfDrawFStats** are those statistics classes that have been enabled by **pfChanStatsMode** for display, and are also enabled for collection. **pfDrawChanStats** displays the contents of the enabled statistics classes of the pfFrameStats structure for channel *chan*, according to the current channel stats draw mode (specified with **pfChanStatsMode**).

At the top of the display is the actual frame rate being achieved and the frame rate set by **pfFrameRate** and the phase set by **pfPhase**. If statistics collection of process frame times has been disabled, then the actual frame rate will not be known and "???" will be shown. When the graphics statistics class is enabled for collection, the average number of pfGeoSets and triangles being displayed is also shown on the top of the statistics display. See the **pfStatsClass** manual page for more information on enabling statistics classes.

For the Process Frame Times Statistics class, **pfDrawChanStats** displays the amount of time, on average, spent by each process on a single frame, as well as the number of frames that missed the goal, or extended beyond the time for the specified goal frame rate. When the **PFFSTATS_PFTIMES_HIST** mode is enabled (on by default), a timing diagram of previous frames is displayed.

Red vertical lines indicate video retrace intervals and green ones indicate frame boundaries. Horizontal bars indicate the time taken by pipeline stages. The three different stages: APP, CULL, AND draw are separated vertically and stages belonging to the same frame are the same color. Each stage of each frame is labeled with the name of the stage and its offset from the current frame. For example, the current application stage is labeled **app0** and **draw-3** is the draw stage of three frames back. Stages that are in the same process are connected by thin vertical lines while stages that are a single process by themselves are not.

> The bar for the application stage is split into a total of five pieces: time spent cleaning the scene graph from changes made by the user (drawn at raised level), time spent waiting for the next frame boundary when the phase is **PFPHASE_LOCK** or **PFPHASE_FLOAT** (drawn with thin, pale, dashed line), the critical time spent between **pfSync** and **pfFrame**, the time spent inside **pfFrame** possibly cleaning the scene graph again and updating and setting off tasks in forked cull and intersection processes (drawn in thin elevated line), and the time spent after **pfFrame** in the

user's application code.

The cull bar is divided into two pieces: first the time spent getting updates from the application process (slightly raised), and the time spent culling the scene graph.

The draw timing bar is divided into four pieces: the lowest piece represents the time actually spent in **pfDraw()** rendering the scene; the darkened parts before and after the **pfDraw()** line represent time spent in the user's channel draw callback routine; the final part displays the time drawing channel statistics.

The draw timing bar is somewhat inaccurate because the time stamps are taken from the host and do not reflect when the graphics pipeline actually finished rendering. Therefore, time for graphics work done in one part of the draw might be counted in a following part when the graphics pipeline FIFO filled up and caused the host to wait. This means that some **pfDraw()** time could be counted in the following user callback time, or in the time to draw the statistics. This statistics class is enabled by default.

When fill statistics are enabled, the main channel will be painted in colors ranging from blue to pink that indicate per-pixel depth-complexity. The brightest (pinkest) areas are those pixels that have been written many times. The statistics displayed, in green, include average total depth complexity (total number of pixel writes), as well as the average, minimum, and maximum number of times a given pixel is written.

When the Graphics Statistics class is enabled for collection and display, detailed statistics on numbers of primitives, attributes, state changes, and matrix transformations are all displayed. These statistics show what is being drawn by the graphics pipeline. When the **PFSTATS_GFX_TSTRIP_LENGTHS** mode is enabled, a histogram of triangle strip lengths showing the percentage of triangles in the scene in strips of given lengths is also displayed. For the strip length statistics, quads are counted as strips of length two and independent triangles are counted as strips of length one. For graphics performance, it is good to have much of the database as possible in triangle strips, and making those triangle strips as long as possible. On a system with RealityEngine graphics, pay special attention to the numbers for texture loads and number of bytes loaded. If these numbers are non-zero, then it means that hardware texture memory is being overflowed and swapped regularly and this will degrade graphics performance.

The CPU statistics display will show some of the statistics seen in **osview(1)**. Graphics context switches occur when there are multiple active graphics windows on the same screen. An application needing high fixed frame rates should not be encurring graphics context switches. Another useful indicator of graphics overload is the **fifonowait** and **fifowait** numbers. An excessive number of times seen waiting on the graphics FIFO could indicate a graphics bottleneck and fill statistics should be examined. If there are an excessive number of process context switches, then it might help performance to restrict the draw process to a single processor and then isolate that processor. IRIS Performer will not do this automatically; however, there are utilities in the IRIS Performer utility library, **libpfutil** (see **pfuLockCPU**), that enable you to do this. These utilities are demonstrated in the IRIS Performer Perfly sample application. These utilities use the IRIX REACT extensions via **sysmp(2)**.

When the Database Statistics class is enabled for collection and display, the number of displayed and evaluated nodes for each node type is shown. When the cull statistics are displayed, a table showing the total number of nodes and pfGeoSets traversed by the cull process, the number of node bounding sphere and pfGeoSet bounding boxes tested, and the total number of nodes, and pfGeoSets, (of those traversed) that were trivially rejected as being outside the viewing frustum, the number that were fully inside the viewing frustum, and the number that intersected the viewing frustum. The database and culling statistics together can show the efficiency of the database hierarchy. If many of the nodes in the database are being traversed by the cull process when only a small percentage are actually visible, then this indicates that the database hierarchy is not spatially coherent. If there are many pfGeoSets in each pfGeode, and many pfGeoSets are being rejected by the cull, then adding more database hierarchy above current nodes may actually speed up the culling traversal because cull tests on nodes would be able to accept or reject large pieces of the database without traversing lower nodes. If the number of pfLOD nodes evaluated is much more then the number that are actually drawn, then adding LOD hierarchy might help to reduce the total number of LOD range calculations, which are fairly expensive.

If there are few nodes in the database relative to the number of pfGeoSets and the cull is taking a small amount of time but the draw is taking longer than desired, then adding more nodes and using a database hierarchy that is spatially coherent should improve the accuracy of the cull and speed up the draw traversal. If there are only a few pfGeoSets per pfGeode and the cull is taking longer than the draw in multiprocess mode, or is taking a significant amount of time in a process shared with the draw, then it might benefit to not cull down to the pfGeoSet level. Refer to the **pfChanTravMode** reference page for information on setting cull traversal modes.

Graphics load is displayed in the lower portion of the statistics window. The load hysteresis band (see **pfChanStress**) is drawn in white and the previous 3 seconds of graphics load is drawn in red. Load is not scaled and ranges from 0.0 to 1.0 within the lower portion of the statistics window.

If stress is active, the display shows a graph of the previous 3 seconds of stress which is drawn in white. Stress is drawn into the upper portion and is scaled to fit.

The **pfDrawChanStats** display is very useful for debugging and profiling a particular application and also for visualizing the behavior of differing multiprocessing modes and pfPipe phases.

NOTES

pfDrawChanStats and **pfDrawFStats** do not actually draw the diagram but set a flag so that the diagram is drawn just before IRIS Performer swaps image buffers.

Drawing the timing diagram does take a small amount of time in the draw process, so it will perturb the frame rate and timing data to some degree.

IRIS Performer level-of-detail behavior is primarily dependent on pfChannel viewing parameters such as view position, field-of-view, and viewport pixel size. IRIS Performer assumes that LODs are modeled for a canonical FOV of 45 degrees and a viewport size of 1024 pixels. IRIS Performer computes an internal

scale value for pfChannels whose FOV or viewport size differ from these defaults. This scale value is used to modify LOD ranges so that correct LOD behavior is maintained. If your LODs were not modeled with the above defaults you may use **PFLOD_SCALE** (see below) to adjust the LOD ranges.

Other LOD modification parameters are set with **pfChanLODAttr**. *attr* is a symbolic token that specifies which LOD parameter to set and is one of the following:

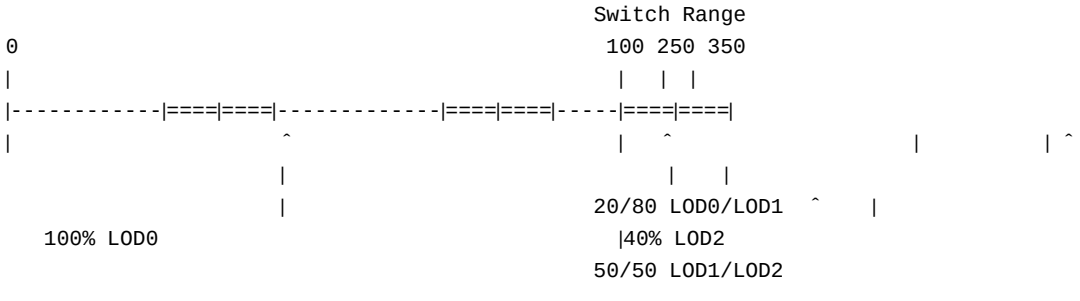
PFLOD_SCALE
val multiplies the range computed between *chan*'s eyepoint and all pfLOD's drawn by *chan*. This is used to globally increase or decrease level of detail on a per-pfChannel basis. The default LOD scale is 1.0. See the **pfLODState** and **pfLOD** man page for more details.

PFLOD_FADE
val specifies the global fade scale used to fade between levels of detail. Fade is enabled when *val* > 0, and is disabled when *val* <= 0. Fade is disabled by default. Note that when computing the actual "fade" or transition distances, this scale is multiplied by individual fade distance values that are specified via **pfLODTransition**. Default pfLOD transition ranges are 1.0. See the **pfLODState** and **pfLOD** man page for more details.

PFLOD_STRESS_PIX_LIMIT
System stress (**pfChanStress**) will not affect LOD's whose projected pixel size exceeds *val* pixels. This feature is disabled by default.

PFLOD_FRUST_SCALE
The range multiplier based on *chan*'s viewport and FOV is multipled by *val*. Typically, this feature is enabled with a value of 1.0 and disabled with a value of 0.0.

LOD fade is useful for avoiding distracting LOD switches. When within the fade range, LODs are drawn semi-transparent so that adjacent LODs smoothly blend together. Fade determines the transparency of an two independent levels of detail. Here is an example for a pfLOD with 3 levels-of-detail and fade range of 30 database units:



Fade transparency is complementary so that fading the same LOD child with (fade) and (1.0 - fade) will generate a fully opaque image. As an example, a fade of 0.7 will cover 70% of the screen area while a fade of (1.0 - fade) = (1.0 - 0.7) = 0.3 will cover the remaining 30% of the screen area.

IRIS Performer ensures that LODs whose switch range is <= 0.0 do not fade in and also clamps the user-specified fade range to half the distance between LOD switches. For example, if a pfLOD is specified with switch ranges 0.0, 100.0, 400.0 and the fade range is 80.0, the result will be:

Example 2: Fade clamping

```
   Range                          LOD(s) drawn
----------                                  ----------------------------------
  0 ->  50             100% LOD0
 50 -> 100             100% -> 50% LOD0 +   0% ->  50% LOD1
100 -> 180              50% ->  0% LOD0 +  50% -> 100% LOD1
180 -> 320                                      100% LOD1
320 -> 400                                 100% ->  50% LOD1
400 -> 480                                  50% ->   0% LOD1
```

Use fade with discretion since it increases rendering time because two LODs instead of one are drawn when range is within the fade interval.

pfGetChanLODAttr returns the value of the LOD modification parameter specified by *attr*.

IRIS Performer computes a stress value based on graphics load (**pfChanStress**) to modify LODs. Specifically, when the system approaches overload, simpler LODs are drawn in order to reduce graphics load. However, in some situations image fidelity considerations make it undesirable to draw low levels-of-detail of objects which are close to the viewer and thus occupy considerable screen space. **PFLOD_STRESS_PIX_LIMIT** limits the effects of stress to LODs whose projected pixel size is less than *val*. Projected pixel size is based on the bounding volume of the LOD and is approximate. When *val* < 0.0, the stress pixel limit is disabled.

PFLOD_SCALE is a global scale that is useful for debugging and for adapting LODs modeled at one FOV and viewport size to the canonical FOV and viewport size used by IRIS Performer. A *val* of 0.0 will cause only the highest LODs are displayed, since the effective distance will be uniformly scaled to 0.0.

All pfChannels on a pfPipe are rendered into a single graphics window so that they can share hardware resources such as textures. Additionally, each channel is rendered in succession rather than in parallel to avoid costly graphics context switching.

For best performance, channel buffers allocated by **pfAllocChanData** should be as small as possible and **pfPassChanData** should be called only when necessary to reduce copying overhead.

When configured as a process separate from the draw, the cull callback should not invoke IRIS GL or OpenGL graphics calls since only the draw process is attached to a graphics context. However, the display listable libpr commands invoked in the cull callback will be correctly added to the current IRIS Performer libpr display list being built for later processing by the draw process.

Callbacks should not modify the IRIS Performer database but may use **pfGet** routines to inquire information as desired.

Draw callbacks should not attempt to perform framebuffer swapping operations directly since IRIS Performer must control this to handle frame and channel synchronization. If user control of buffer swapping is required, register a **pfPipeSwapFunc** callback to cause the named user written function to be used by IRIS Performer for swapping buffers.

Sorting back-to-front is required for accurate rendering of **PFTR_BLEND_ALPHA** surfaces. The ordering mechanism described above provides range sorting on a per-pfGeoSet, not a per-triangle basis so some anomalies may be apparent when rendering transparent surfaces. These anomalies may be reduced by rejecting back-facing polygons (see **pfCullFace** and **PFSTATE_CULLFACE**).

The IRIS Performer world coordinate system is +X = East, +Y = North, +Z = Up and viewing coordinate system is +X = Right, +Y = Forward, +Z = Up. Note that this is not the same as the IRIS GL or OpenGL default coordinate system which uses +X = Right, +Y = Up, +Z = Out of the screen. IRIS Performer internally manages the transformation required to go from a 'Z-up' world to a 'Y-up' world.

Fade-based level of detail transition is supported only on RealityEngine systems and then only when multisampling is enabled.

BUGS

Intersections, and thus picking, with lines and points is not yet implemented.

SEE ALSO

pfConfigPWin, pfAddChan, pfInsertChan, pfMoveChan, pfRemoveChan, pfPipeSwapFunc, pfNodeIsectSegs, pfLoadGState, pfNodeBSphere, pfNodeTravMask, pfStatsClass, pfStatsClassMode, pfConfig, pfCullFace, pfDispList, pfEarthSky, pfESkyFog, pfObject, pfFrame, pfFrameRate, pfFrustum, pfGetSemaArena, pfLightSource, pfLOD, pfMultipipe, pfMultiprocess, pfPolytope, pfPhase, pfScene, pfGetSemaArena, pfTransparency, pfuLockCPU

NAME

pfMultipipe, pfGetMultipipe, pfMultithread, pfGetMultithread, pfMultiprocess, pfGetMultiprocess, pfConfig, pfGetPID, pfGetPipe, pfInitPipe, pfGetStage, pfStageConfigFunc, pfGetStageConfigFunc, pfConfigStage, pfHyperpipe, pfGetHyperpipe, pfGetPipeHyperId – Configure process and pipeline models, get pfPipe handle and process ID.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

int	**pfMultipipe**(int num);
int	**pfGetMultipipe**(void);
int	**pfMultithread**(int pipe, uint stage, int nprocs);
int	**pfGetMultithread**(int pipe, uint stage);
int	**pfMultiprocess**(int mode);
int	**pfGetMultiprocess**(void);
int	**pfConfig**(void);
pid_t	**pfGetPID**(int pipe, uint stage);
pfPipe *	**pfGetPipe**(int pipe);
int	**pfInitPipe**(pfPipe *pipe, pfPipeFuncType configFunc);
uint	**pfGetStage**(pid_t pid, int *pipe);
void	**pfStageConfigFunc**(int pipe, uint stageMask, pfStageFuncType configFunc);
pfStageFuncType	**pfGetStageConfigFunc**(int pipe, uint stageMask);
void	**pfConfigStage**(int pipe, uint stageMask);
void	**pfHyperpipe**(int n);
int	**pfGetHyperpipe**(pfPipe *pipe);
int	**pfGetPipeHyperId**(const pfPipe *pipe);

```
typedef void (*pfStageFuncType)(int pipe, uint stage);
```

DESCRIPTION

An IRIS Performer application renders images using one or more pfPipes. A pfPipe is a software rendering pipeline that traverses, culls, and draws one or more pfChannels into a single graphics context. The software rendering pipeline is composed of three functional *stages*:

APP	Application processing
CULL	Database culling and level-of-detail selection
DRAW	Drawing geometry produced by **CULL**

In addition, IRIS Performer has a separate intersection stage which can operate either synchronously or asynchronously with the rendering pipeline (see **pfIsectFunc**).

All stages may be combined into a single process or split into multiple processes for enhanced performance on multiprocessing systems. **pfMultiprocess** controls the partitioning of functional stages into processes. *mode* is a bitwise OR of the following tokens:

PFMP_FORK_ISECT
PFMP_FORK_CULL
PFMP_FORK_DRAW
PFMP_FORK_DBASE
PFMP_CULLoDRAW
PFMP_CULL_DL_DRAW

These tokens specify which stages to fork into separate processes and what multiprocessing communication mechanism to use between the cull and draw processes.

The process from which all other processes are spawned is known as the application process, or **APP**. This process is the one that invokes **pfConfig** and controls the rendering and intersection pipelines through **pfFrame**.

User code in the intersection, database, cull, and draw processes are "triggered" by calling **pfFrame**. **pfFrame** causes IRIS Performer to invoke the user callbacks associated with each process. These callbacks are established by **pfIsectFunc**, **pfDBaseFunc**, **pfChanTravFunc** respectively. See **pfFrame** for more details.

Each pfPipe has a **CULL** and **DRAW** stage which may be configured as either one or two processes. The **ISECT** and **DBASE** stages are independent of any pfPipe and may run in the same process as the application process or as separate processes (**PFMP_FORK_ISECT**, **PFMP_FORK_DBASE**). In the latter case, the user may further multiprocess intersection traversals through any IRIX multiprocessing mechanism such as **fork**, **sproc**, or **m_fork**. Database processing utilizing the pfBuffer mechanism may be further parallelized through **fork** only (See **pfBuffer**).

For additional performance gains when a pfPipe contains multiple pfChannels, the **CULL** stage may be further parallelized on a per-pfChannel basis. When the *stage* argument to **pfMultithread** is **PFPROC_CULL**, the **CULL** stage of the *pipe*th rendering pipeline is split into *nprocs*, **fork**ed, processes each of which operates singly on a pfChannel. Thus this extra parallelization is only effective when both

nprocs and the number of pfChannels on *pipe* are greater than 1. *nprocs* need not be equal to the number of pfChannels. Currently, **pfMultithread** only accepts a *stage* argument of **PFPROC_CULL**, returns 1 on success and -1 otherwise. The **CULL** is not automatically multithreaded if **PFMP_DEFAULT** is specified as the **pfMultiprocess** mode.

When multithreading the **CULL**, care must be taken to avoid data collisions in user callback functions. In particular, pfChannel and pfNode CULL callbacks (**pfChanTravFunc**, **pfNodeTravFuncs**) may be invoked in parallel.

pfGetMultithread returns the number of processes in the processing stage identified by *stage* on the *pipe*th rendering pipeline. Currently, **pfGetMultithread** only accepts a *stage* argument of **PFPROC_CULL** and returns -1 otherwise.

Thus, the number of processes an application uses is dependent on:

1. The multiprocessing modes set by **pfMultiprocess** and **pfMultithread**.
2. The number of rendering pipelines set by **pfMultipipe**.
3. The number of user-spawned processes.

The following table indicates the number of processes that are implied by each multiprocessing mode combination as a function of the number of IRIS Performer pfPipes specified.

FORK_ISECT	FORK_CULL	FORK_DRAW	# Processes
No	No	No	1
No	No	Yes	2
No	Yes	No	1 + numPipes
No	Yes	Yes	1 + 2*numPipes
Yes	No	No	2
Yes	No	Yes	3
Yes	Yes	No	2 + numPipes
Yes	Yes	Yes	2 + 2*numPipes

Here is an example configuration which would be used to generate a high-performance stereo display using two pfPipes, each associated with a hardware graphics pipeline. In this situation the output of one pipeline will be displayed for the viewer's left eye, and the other will go to the right eye. Here, multithreading the **CULL** is of no use since each pfChannel is handled by its own pfPipe.

Example 1: Two pfPipe stereo configuration

```
/* configure two hardware pipelines */
pfMultipipe(2);

/* operate all processing tasks in parallel */
```

```
pfMultiprocess(PFMP_FORK_CULL | PFMP_FORK_DRAW | PFMP_FORK_ISECT);
```

The processing mode configured by this example looks like:

```
            CULL ---> DRAW                  left eye
           /
          /
        APP
       /  \
      /    \
    ISECT   CULL ---> DRAW                  right eye
```

Example 2: One pfPipe stereo configuration using multithreaded CULL

```
/* operate all processing tasks in parallel */
pfMultiprocess(PFMP_FORK_CULL | PFMP_FORK_DRAW | PFMP_FORK_ISECT);

pfMultithread(0, PFPROC_CULL, 2);
```

The processing mode configured by this example looks like:

```
                      CULL                  left eye
             /    \
         APP-----      ---------> DRAW
        /        \    /
       /          CULL                     right eye
     ISECT
```

PFMP_CULL_DL_DRAW and **PFMP_CULLoDRAW** specify how the cull and draw stages should communicate.

If **PFMP_CULL_DL_DRAW** is set the cull stage will build up an IRIS Performer display list (pfDispList) which contains the entire frame's worth of data. The draw stage then traverses this pfDispList when **pfDraw** is called and sends commands to the graphics hardware. When the cull and draw stages are different processes (**PFMP_FORK_DRAW**) this mode is always enabled. However, when the cull and draw stages are the same process, the display list construction may add some overhead. If, in this case, **PFMP_CULL_DL_DRAW** is not specified, the cull stage will be delayed until **pfDraw** is called. **pfDraw** will then cull and draw the scene in immediate mode and not use a pfDispList.

PFMP_CULL_DL_DRAW is disabled by default but should be used for applications which use multipass rendering techniques that require multiple calls to **pfDraw**.

The 'o' in **PFMP_CULLoDRAW** is short for 'overlap' and when this bit is set, the multiprocessed cull and draw stages of the same frame will be overlapped. The cull process (the producer) writes to a FIFO (implemented as a ring buffer) while the draw process (the consumer) simultaneously reads commands from the ring buffer.

The main benefit of this configuration is that latency will be reduced a full frame time over the pipelined (non-overlapped) case. A disadvantage is that the draw process may suffer from reduced throughput if the cull process cannot keep up. This condition is exacerbated when the cull sorts the database by draw bin or by graphics state. In each case, the cull retains the database in internal data structures and does not add drawing commands to the display list until the cull is completed. Consequently, to get the best throughput from **PFMP_CULLoDRAW**, database mode sorting and ordering should be disabled.

Example 3: Reasonable sorting setup for **PFMP_CULLoDRAW**

```
pfMultiprocess(PFMP_APP_CULL_DRAW | PFMP_CULLoDRAW);

/* Draw opaque geometry immediately into CULLoDRAW's pfDispList
 * Transparent geometry is still saved and drawn after opaque. */
pfChanBinOrder(chan, PFSORT_OPAQUE_BIN, PFSORT_NO_ORDER);

/* PFCULL_SORT must be enabled for transparent geometry to be
ordered, i.e. - drawn last. */
pfChanTravMode(chan, PFTRAV_CULL, PFCULL_ALL);
```

PFMP_CULLoDRAW is ignored if the cull and draw stages are in the same process.

For convenience, other tokens are provided for common multiprocessing modes:

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**.

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_APPCULL_DRAW
The application and cull stages are combined in a process that is separate from the draw process. Equivalent to (**PFMP_FORK_DRAW**).

PFMP_APPCULLoDRAW
The application 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 application, cull, and draw stages are each separate processes. Equivalent to (-**PFMP_FORK_CULL | PFMP_FORK_DRAW**).

PFMP_APP_CULLoDRAW
The application, 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_DEFAULT
IRIS Performer will choose a multiprocessing mode based on the number of pipelines required and the number of unrestricted processors available. This is also the default mode if **pfMultiprocess** is not called. **PFMP_DEFAULT** will attempt to use as many available processors as possible except the **CULL** will not be automatically multithreaded.

By default IRIS Performer uses a single pfPipe. If multiple rendering pipelines are required (in most cases this will be for machines with multiple hardware pipelines), use **pfMultipipe** to specify the number of pfPipes that are created by **pfConfig**. Multipipe operation absolutely requires that all participating hardware pipelines be genlocked. Otherwise reduced throughput and increased latency will result.

The multiprocessing mode set by **pfMultiprocess** is used for all rendering pipelines. However, IRIS Performer never multi-threads the application process although the application may choose to do so. If the application itself multiprocesses, all IRIS Performer calls must be made from the process which calls **pfConfig** or results are undefined. When using multiple pipelines, the cull stage must be forked (-**PFMP_FORK_CULL**). If not, IRIS Performer defaults to **PFMP_APP_CULL_DRAW**.

pfMultiprocess, **pfMultithread**, and **pfMultipipe** must be called after **pfInit** but before **pfConfig**. **pfConfig** configures IRIS Performer according to the required number of pipelines and multiprocessing

modes, forks the appropriate number of IRIS Performer processes and returns control to the single-threaded application. **pfConfig** should be called only once between **pfInit** and **pfExit**.

IRIS Performer uses **fork** to split off processes and will create the specified number of separate processes only when **pfConfig** is called. Forked processes do not share the same address space as **sproc**'ed processes so the application must establish shared memory communication mechanisms between processes or use the shared memory features provided by IRIS Performer (see **pfPassChanData**, **pfMalloc**, **pfGetSharedArena**, **pfDataPool**).

In particular, care must be taken when the **DBASE** stage is configured as a separate process. Although deletion requests (**pfDelete**) may be made in any process, **DBASE** frees all the memory so if **DBASE** is forked it can only free memory that was allocated out of IRIS Performer's shared memory arena (-**pfGetSharedArena**) or from some other memory arena that is visible to the **DBASE** process. Consequently it is safest to allocate all objects from a shared memory arena when using a forked **DBASE** process.

In addition to forking processes, **pfConfig** initializes the number of pfCycleBuffer copies (-**pfCBufferConfig**) appropriate to the multiprocessing mode and also initializes the video clock (-**pfInitVClock**) to 0.

After **pfConfig** is called, **pfGetPipe** should be used to get handles to pfPipes for subsequent use in IRIS Performer routines. *pipe* identifies a pipe and ranges from 0 to numPipes - 1 where numPipes is the number of pipes specified in **pfMultipipe**.

After **pfConfig** spawns other processes, **pfGetPID** will return the process id of a specific pipeline stage or -1 to indicate error. *pipe* specifies which pipeline the stage is in and ranges from 0 to numPipes - 1. *stage* is a bitmask which identifies one or more stages in the multiprocessing pipeline and may consist of:

Token	Stage Description
PFPROC_ISECT	The intersection stage
PFPROC_DBASE	The database stage
PFPROC_APP	The application stage
PFPROC_CULL	The cull stage
PFPROC_DRAW	The draw stage
PFPROC_CLOCK	The clock process

If *stage* identifies multiple stages, such as (**PFPROC_CULL | PFPROC_DRAW**), then the process id will be returned only if an exact match is made which in this example is only possible if the multiprocessing mode is **PFMP_APP_CULLDRAW**. Otherwise a -1 is returned.

pipe is ignored if *stage* identifies the **PFPROC_ISECT**, **PFPROC_DBASE**, or **PFPROC_APP** stages since these stages are not associated with any IRIS Performer pipe.

pfGetStage is the "inverse" of **pfGetPID**. Given a process id, *pid*, **pfGetStage** will return a bitmask which identifies the stages that are performed by process *pid* and will copy into *pipe* the number of the pipeline that *pid* is in if *pipe* is not **NULL**. **pfGetStage** returns -1 if *pid* is not a known IRIS Performer process.

The stage bitmask used in **pfGetPID** and **pfGetStage** identifies the thread number (**pfMultithread**) as well as the processing stage(s). The thread ID is OR'ed into the upper bits of the stage bitmask as follows:

```
threadId = (stage & PFPROC_THREAD_MASK) >> PFPROC_THREAD_SHIFT;
```

The **PFPROC_THREAD1-7** tokens are provided as a convenience (more than 8 threads are supported).

pfGetMultiprocess and **pfGetMultipipe** return the multiprocess mode and number of pfPipes configured.

pfInitPipe is an obsolete routine for initializing the graphics subsystem for a pfPipe. A callback function *configFunc* could be provided for initializing *pipe* in the draw process and was used for opening windows in the draw process for the pfPipe. This function has been obsoleted by the pfPipeWindow primitive which can be used to configure windows in either or both the application process and draw process, and by **pfConfigStage** which provides a mechanism for initializing any IRIS Performer process or pfPipe stage. See the **pfNewPWin** man page for more information on creating and opening IRIS Performer windows.

After **pfConfig**, stage configuration callbacks may be specified with **pfStageConfigFunc** and triggered with **pfConfigStage**. Configuration callbacks are typically used for process initialization, e.g, assign non-degrading priorities and locking processes to processors or downloading textures in the **DRAW** stage callback. The *stageMask* argument to **pfStageConfigFunc** is a bitmask which identifies one or more IRIS Performer stages (see **pfGetPID** above). If >= 0, the *pipe* argument to **pfStageConfigFunc** selects stage(s) on a particular pfPipe (**pfGetPipe**(*pipe*)). If *pipe* is < 0 it selects stages of all pfPipes. Note that *pipe* is ignored for the **PFPROC_ISECT**, **PFPROC_APP**, and **PFPROC_DBASE** stages since they are not associated with any pfPipe. *configFunc* is the callback function to be invoked for the indicated stages. **pfGetStageConfigFunc** returns the configuration function used for the stage identified by *pipe* and *stageMask*.

pfConfigStage causes the callback functions to be invoked for the identified stages at the start of processing the current application frame. The current application frame gets to the next stage at the next call to **pfFrame**. *pipe* and *stageMask* are treated identically as in **pfStageConfigFunc**. When multiprocessing, the callback functions are invoked in the appropriate processes.

Example 4: Stage configuration

```
void
configFunc(int pipe, uint stage)
{
    /* Fix CULL processes to processor 1 and 3 */
    if (stage == PFPROC_CULL)
                sysmp(MP_MUSTRUN, 2*pipe+1);

    /* Fix DRAW processes to processor 2 and 4 */
    else if (stage == PFPROC_DRAW)
                sysmp(MP_MUSTRUN, 2*pipe+2);
}

:

pfMultipipe(2);
pfMultiprocess(PFMP_APP_CULL_DRAW);
pfConfig();

pfStageConfigFunc(-1, PFPROC_CULL|PFPROC_DRAW, configFunc);
pfConfigStage(-1, PFPROC_CULL|PFPROC_DRAW);

pfFrame();
```

pfHyperpipe supports the hyperpipe hardware feature of VGXT/Skywriter and Onyx/RealityEngine2 research systems. *n* indicates the number of pfPipes that should be configured together in hyperpipe mode. Hyperpipes will run at a fraction of the system frame rate as defined by **pfFrameRate**. For example, if *n* is 2, then each pfPipe in the hyperpipe group will run at half the system frame rate so their aggregate rate will be equal to the system frame rate.

pfGetHyperpipe returns the total number of pfPipes in the hyperpipe group that *pipe* belongs to. **pfGetPipeHyperId** returns the position of *pipe* in its hyperpipe group. The following example configures a two-pipeline hyperpipe system:

Example 5: Hyperpipe Example

```
pfHyperpipe(2);
pfConfig();

pfGetHyperpipe(pfGetPipe(0));           /* This returns 2 */
pfGetPipeHyperId(pfGetPipe(1));         /* This returns 1 */
```

NOTES

In practice, user callbacks in the intersection process call only **pfNodeIsectSegs** and user callbacks in the database process uses the pfBuffer mechanism to asynchronously create and delete scene graphs to implement database paging.

If **PFMP_DEFAULT** is not used, it is up to the application to tailor the number of IRIS Performer processes to the number of processors. Care must be taken to avoid thrashing, starvation, and deadlock.

If **pfIsectFunc** is called before **pfConfig** and the multiprocessing mode is **PFMP_DEFAULT**, then **pfConfig** will fork the intersection process if there are enough processors. Otherwise, you must explicitly fork the intersection process by setting the **PFMP_FORK_ISECT** bit in the argument passed to **pfMultiprocess**.

When using **PFMP_CULLoDRAW**, multipass algorithms (e.g. - landing lights on RealityEngine) which call **pfDraw** more than once per frame will not work.

BUGS

If **PFMP_CULLoDRAW** is used, modifications to pfChannel passthrough data (see **pfPassChanData**) made by the cull callback will not be passed along to the draw callback. However, modifications made by the application process will still make it to both cull and draw callbacks.

PFMP_CULLoDRAW usually has no effect when IRIS Performer is in the free-running frame rate control mode specified by **pfPhase(PFPHASE_FREE_RUN)**. Instead, use **PFPHASE_FLOAT** or **PFPHASE_LOCK**.

When in **PFMP_CULLoDRAW** mode, the draw time recorded by IRIS Performer statistics does not include the time the draw process spends waiting for the cull process to begin filling the ring buffer.

pfHyperpipe assumes that the pfPipe to hardware pipe association is ordered, e.g. that pipe 0 renders to screen 0, pipe 1 renders to screen 1, and so on.

SEE ALSO

fork, m_fork, pfChannel, pfCycleBuffer, pfInit, pfIsectFunc, pfDBaseFunc, pfPipe, sproc

NAME

pfDBaseFunc, pfGetDBaseFunc, pfAllocDBaseData, pfGetDBaseData, pfPassDBaseData, pfDBase – Set database callback, allocate and pass database data.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

void	**pfDBaseFunc**(pfDBaseFuncType func);
pfDBaseFuncType	**pfGetDBaseFunc**(void);
void *	**pfAllocDBaseData**(int bytes);
void *	**pfGetDBaseData**(void);
void	**pfPassDBaseData**(void);
void	**pfDBase**(void);

```
typedef void (*pfDBaseFuncType)(void *userData);
```

DESCRIPTION

The *func* argument to **pfDBaseFunc** specifies the database callback function. This function will be invoked by **pfFrame** and will be passed a pointer to a data buffer allocated by **pfAllocDBaseData**. If a separate process is allocated for database processing by the **PFMP_FORK_DBASE** mode to **pfMultiprocess**, then **pfFrame** will cause *func* to be called in the separate (**DBASE**) process. **pfGetDBaseFunc** returns the database callback or **NULL** if none is set.

The database function's primary purpose is to provide asynchronous database creation and deletion when using the pfBuffer mechanism and a forked **DBASE** process (see **PFMP_FORK_DBASE**, **pfMultiprocess**, and **pfNewBuffer**).

When the database function is in a separate process, it will run asynchronously with the rest of the rendering pipeline. Specifically, if the database function takes more than a frame time, the rendering pipeline will not be affected.

If a database function has been specified by **pfDBaseFunc**, it must call **pfDBase** to carry out default IRIS Performer database processing. **pfDBase** should only be called from within the **DBASE** callback in the **DBASE** process just like **pfCull** and **pfDraw** should only be called in the pfChannel **CULL** and **DRAW** callbacks (**pfChanTravFunc**) respectively. If a database function has not been specified or is **NULL**, IRIS Performer automatically calls **pfDBase** from **pfFrame**.

pfAllocDBaseData returns a pointer to a chunk of shared memory of *bytes* bytes. This memory buffer may be used to communicate information between the database function and application. Database data should only be allocated once. **pfGetDBaseData** returns the previously allocated database data.

When the database function is forked, **pfPassDBaseData** should be used to copy the database data into internal IRIS Performer memory when the next **pfFrame** is called. Once **pfFrame** is called, the application may modify data in the database data buffer without fear of colliding with the forked database function. However, modifications to the database data chunk made by the **DBASE** process will not be visible to the **APP** process, i.e, there is no "upstream" propagation of passthrough data.

NOTES

Currently, **pfDBase** carries out asynchronous deletion requests made with **pfAsyncDelete**.

SEE ALSO

pfAsyncDelete, pfConfig, pfFrame, pfMultiprocess, pfNewBuffer

NAME

pfNewDCS, pfGetDCSClassType, pfDCSTrans, pfDCSRot, pfDCSCoord, pfDCSScale, pfDCSScaleXYZ, pfDCSMat, pfGetDCSMat, pfGetDCSMatPtr, pfDCSMatType, pfGetDCSMatType – Create, modify and get the matrix of a dynamic coordinate system.

FUNCTION SPECIFICATION

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

pfDCS *          pfNewDCS(void);
pfType *         pfGetDCSClassType(void);
void             pfDCSTrans(pfDCS *dcs, float x, float y, float z);
void             pfDCSRot(pfDCS *dcs, float h, float p, float r);
void             pfDCSCoord(pfDCS *dcs, pfCoord *coord);
void             pfDCSScale(pfDCS *dcs, float s);
void             pfDCSScaleXYZ(pfDCS *dcs, float x, float y, float z);
void             pfDCSMat(pfDCS *dcs, pfMatrix m);
void             pfGetDCSMat(pfDCS *dcs, pfMatrix m);
const pfMatrix*  pfGetDCSMatPtr(pfDCS *dcs);
void             pfDCSMatType(pfDCS *dcs, uint val);
uint             pfGetDCSMatType(pfDCS *dcs);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfDCS** is derived from the parent class **pfSCS**, so each of these member functions of class **pfSCS** are also directly usable with objects of class **pfDCS**. Casting an object of class **pfDCS** to an object of class **pfSCS** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfSCS**.

```
void             pfGetSCSMat(pfSCS *scs, pfMatrix mat);
const pfMatrix*  pfGetSCSMatPtr(pfSCS *scs);
```

Since the class **pfSCS** is itself derived from the parent class **pfGroup**, objects of class **pfDCS** can also be used with these functions designed for objects of class **pfGroup**.

```
int      pfAddChild(pfGroup *group, pfNode *child);
int      pfInsertChild(pfGroup *group, int index, pfNode *child);
int      pfReplaceChild(pfGroup *group, pfNode *old, pfNode *new);
int      pfRemoveChild(pfGroup *group, pfNode* child);
```

int **pfSearchChild**(pfGroup *group, pfNode* child);
pfNode * **pfGetChild**(const pfGroup *group, int index);
int **pfGetNumChildren**(const pfGroup *group);
int **pfBufferAddChild**(pfGroup *group, pfNode *child);
int **pfBufferRemoveChild**(pfGroup *group, pfNode *child);

Since the class **pfGroup** is itself derived from the parent class **pfNode**, objects of class **pfDCS** can also be used with these functions designed for objects of class **pfNode**.

pfGroup * **pfGetParent**(const pfNode *node, int i);
int **pfGetNumParents**(const pfNode *node);
void **pfNodeBSphere**(pfNode *node, pfSphere *bsph, int mode);
int **pfGetNodeBSphere**(pfNode *node, pfSphere *bsph);
pfNode* **pfClone**(pfNode *node, int mode);
pfNode* **pfBufferClone**(pfNode *node, int mode, pfBuffer *buf);
int **pfFlatten**(pfNode *node, int mode);
int **pfNodeName**(pfNode *node, const char *name);
const char * **pfGetNodeName**(const pfNode *node);
pfNode* **pfFindNode**(pfNode *node, const char *pathName, pfType *type);
pfNode* **pfLookupNode**(const char *name, pfType* type);
int **pfNodeIsectSegs**(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void **pfNodeTravMask**(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint **pfGetNodeTravMask**(const pfNode *node, int which);
void **pfNodeTravFuncs**(pfNode* node, int which, pfNodeTravFuncType pre,
pfNodeTravFuncType post);
void **pfGetNodeTravFuncs**(const pfNode* node, int which, pfNodeTravFuncType *pre,
pfNodeTravFuncType *post);
void **pfNodeTravData**(pfNode *node, int which, void *data);
void * **pfGetNodeTravData**(const pfNode *node, int which);

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfDCS** can also be used with these functions designed for objects of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfDCS** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *        pfGetType(const void *ptr);
int             pfIsOfType(const void *ptr, pfType *type);
int             pfIsExactType(const void *ptr, pfType *type);
const char *    pfGetTypeName(const void *ptr);
int             pfRef(void *ptr);
int             pfUnref(void *ptr);
int             pfUnrefDelete(void *ptr);
int             pfGetRef(const void *ptr);
int             pfCopy(void *dst, void *src);
int             pfDelete(void *ptr);
int             pfCompare(const void *ptr1, const void *ptr2);
void            pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *          pfGetArena(void *ptr);
```

PARAMETERS

dcs identifies a pfDCS

DESCRIPTION

A pfDCS (Dynamic Coordinate System) is a pfSCS whose matrix can be modified.

pfNewDCS creates and returns a handle to a pfDCS. Like other pfNodes, pfDCSes are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetDCSClassType returns the **pfType*** for the class **pfDCS**. The **pfType*** returned by **pfGetDCSClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfDCS**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

The initial transformation is the identity matrix. The transformation of a pfDCS can be set by specifying a matrix or translation, scale and rotation. When independently setting translation, rotation, and scale, the pfDCS matrix is computed as S*R*T, where S is the scale, R is the rotation, and T is the translation. The order of effect is then scale followed by rotation followed by translation.

pfDCS operations are absolute rather than cumulative. For example:

```
pfDCSTrans(dcs, 2.0f, 0.0f, 0.0f);
pfDCSTrans(dcs, 1.0f, 0.0f, 0.0f);
```

specifies a translation by 1 unit along the X coordinate axis, not 3 units.

By default a pfDCS uses a bounding sphere which is dynamic, so it is automatically updated when the pfDCS transformation is changed or when children are added, deleted or changed. This behavior may be changed using **pfNodeBSphere**. The bound for a pfDCS encompasses all B(i)*S*R*T, where B(i) is the

bound for the child 'i' and S*R*T represents the scale, rotation, and translation transformation of the pfDCS.

pfDCSTrans sets the translation part of the pfDCS to (*x*, *y*, *z*). The rotational portion of the matrix is unchanged.

pfDCSScale sets the scale portion of the pfDCS to scale uniformly by a scale factor *s*. This supersedes the previous scale leaving the rotation and translation unchanged. **pfDCSScaleXYZ** specifies a non-uniform scale of *x*, *y*, *z*.

pfDCSRot sets the rotation portion of the matrix:

- *h* Specifies heading, the rotation about the Z axis.
- *p* Specifies pitch, the rotation about the X axis.
- *r* Specifies roll, rotation about the Y axis.

The matrix created is R*P*H, where R is the roll transform, P is the pitch transform and H is the heading transform. The new (h,p,r) combination replaces the previous specification, leaving the scale and translation unchanged. The convention is natural for a model in which +Y is "forward," +Z is "up" and +X is "right". To maintain 1/1000 degree resolution in the single precision arithmetic used internally for sine and cosine calculations, the angles *h*, *p*, *r* should be in the range of -7500 to +7500 degrees.

pfDCSCoord sets the rotation and translation portion of the pfDCS according to *coord*. This is equivalent to:

```
pfDCSRot(dcs, coord->hpr[0], coord->hpr[1], coord->hpr[2]);
pfDCSTrans(dcs, coord->xyz[0], coord->xyz[1], coord->xyz[2]);
```

pfDCSMat sets the transformation matrix for *dcs* to *m*.

Normally **pfDCSMat** is used as a replacement for the above routines which individually set the scale, rotation and translational components. The mechanisms can be combined but only if the supplied matrix can be represented as scale followed by a rotation followed by a translation (e.g. a point pt is transformed by the matrix as: pt' = pt*S*R*T), which implies that no shearing or non-uniform scaling is present.

pfDCSMatType allows the specification of information about the type of transformation the matrix represents. This information allows Performer to speed up some operations. The matrix type is specified as the OR of

PFMAT_TRANS:
matrix may include a translational component in the 4th row.

PFMAT_ROT
matrix may include a rotational component in the left upper 3X3 submatrix.

PFMAT_SCALE
matrix may include a uniform scale in the left upper 3X3 submatrix.

PFMAT_NONORTHO
matrix may include a non-uniform scale in the left upper 3X3 submatrix.

PFMAT_PROJ
matrix may include projections.

PFMAT_HOM_SCALE
matrix may include have mat[4][4] != 1.

PFMAT_MIRROR
matrix may include mirroring transformation that switches between right handed and left handed coordinate systems.

pfGetDCSMatType returns the matrix type as
set by **pfDCSMatType**. If no matrix type is set the default is ~0, corresponding to a general matrix.

The transformation of a pfDCS affects all its children. As the hierarchy is traversed from top to bottom, each new matrix is pre-multiplied to create the new transformation. For example, if DCSb is below DCSa in the scene graph, any geometry G below DCSa is transformed as G*DCSb*DCSa.

pfFlatten cannot flatten pfDCSes since they may change at run-time. In this case **pfFlatten** will compute a pfSCS representing the accumulated static transformation that the pfDCS inherits and insert it above the pfDCS. Static transformations below a pfDCS are flattened as usual. See **pfFlatten** for more details.

The presence of transformations in the scene graph impacts the performance of intersection, culling and drawing. pfGeoSet culling (see **PFCULL_GSET** in **pfChanTravMode**) is disabled in portions of the scene graph below pfDCSes.

Both pre and post CULL and DRAW callbacks attached to a pfDCS (**pfNodeTravFuncs**) will be affected by the transformation represented by the pfDCS, i.e. - the pfDCS matrix will already have been applied to the matrix stack before the pre callback is called and will be popped only after the post callback is called.

pfGetDCSMat copies the transformation matrix value from *dcs* into the matrix *m*. For faster matrix access, **pfGetDCSMatPtr** can be used to get a const pointer to *dcs*'s matrix.

SEE ALSO

pfCoord, pfGroup, pfChanTravMode, pfLookupNode, pfFlatten, pfMatrix, pfNode, pfSCS, pfScene, pfTraverser, pfDelete

NAME

pfNewESky, pfGetESkyClassType, pfESkyMode, pfGetESkyMode, pfESkyAttr, pfGetESkyAttr, pfESkyColor, pfGetESkyColor, pfESkyFog, pfGetESkyFog – Create and control weather, Earth-Sky model, and screen clearing.

FUNCTION SPECIFICATION

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

pfEarthSky *  pfNewESky(void);
pfType *      pfGetESkyClassType(void);
void          pfESkyMode(pfEarthSky *esky, int mode, int val);
int           pfGetESkyMode(pfEarthSky *esky, int  mode);
void          pfESkyAttr(pfEarthSky *esky, int attr, float val);
float         pfGetESkyAttr(pfEarthSky *esky, int  mode);
void          pfESkyColor(pfEarthSky *esky, int which, float r, float g, float b, float a);
void          pfGetESkyColor(pfEarthSky *esky, int  which, float *r, float *g, float *b, float *a);
void          pfESkyFog(pfEarthSky *esky, int which, pfFog *fog);
pfFog *       pfGetESkyFog(pfEarthSky *esky, int  which);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfEarthSky** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfEarthSky**. Casting an object of class **pfEarthSky** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
void*   pfGetUserData(pfObject *obj);
```

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfEarthSky** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *      pfGetType(const void *ptr);
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
```

int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

esky identifies a pfEarthSky.

DESCRIPTION

These functions provide a means to clear the frame and Z-buffer, draw a sky, horizon and ground plane, and to implement various weather effects. Once the earth-sky is set in a channel, it should be the first thing drawn when a scene is rendered.

pfNewESky creates and returns a handle to a pfEarthSky. Like other pfNodes, pfEarthSkies are always allocated from shared memory and can be deleted using **pfDelete**.

pfNewESky creates a pfEarthSky and sets up reasonable defaults. To render the earth and sky model, it must be added to a pfChannel. By default, the mode is to render a full screen clear unless either the sky or ground is turned on. pfEarthSky is called automatically in the draw process, unless a draw callback is present, in which case, it must be explicitly called using **pfClearChan**.

pfGetESkyClassType returns the **pfType*** for the class **pfEarthSky**. The **pfType*** returned by **pfGetESkyClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfEarthSky**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfESkyMode is used to set the earth-sky rendering mode. **pfGetESkyMode** is used to obtain the earth-sky rendering mode. These functions currently accept the two mode arguments **PFES_BUFFER_CLEAR**, and **PFES_CLOUDS**.

PFES_BUFFER_CLEAR may have the following values:

PFES_FAST
The default mode. This simply clears the color and Z buffers. The clear color can be set using **pfESkyColor**. Dithering is turned off during the clear.

PFES_TAG Initializes the framebuffer to a known state very rapidly. Has an effect only when multisampling. Often, this mode is used as an optimization before rendering a background that covers the entire screen. See **pfClear** for the details and restrictions of the mode **PFCL_MSDEPTH**.

PFES_SKY
Causes a sky and horizon backdrop to be drawn. These are drawn using large polygons that are recalculated each frame, using information about the clipping planes, field of view, and eyepoint vertical position for the selected channel. They are drawn instead of a screen clear, forcing the Z buffer to a known state. If the viewpoint goes below the ground plane, the area below the horizon will not be cleared. In the case of **PFES_SKY**, the screen is never cleared below the lower edge of the horizon.

PFES_SKY_GRND
Add a ground plane to the sky and horizon model drawn by **PFES_SKY**.

PFES_SKY_CLEAR
Draw the sky and horizon, and clear the screen below the edge of the horizon.

PFES_CLOUDS is used to set the type of cloud layer. Currently, the only value supported is:

PFES_OVERCAST
This cloud type is a non-textured, opaque region that has a color and both top and bottom dimensions. This, being the only choice at present, is the default type.

pfESkyColor is used to set the colors referenced by the earth-sky rendering routines. **pfGetESkyColor** returns the indicated color component of the earth-sky mode. The components are:

PFES_SKY_TOP	The color of the sky directly above the viewpoint.
PFES_SKY_BOT	The color of the sky where it joins the horizon.
PFES_HORIZ	The color of the bottom edge of the horizon.
PFES_GRND_FAR	The color of the ground plane where it meets the horizon.
PFES_GRND_NEAR	The color of the ground plane directly below the viewer.
PFES_CLOUD_BOT	The color of the bottom of the opaque cloud layer.
PFES_CLOUD_TOP	The color of the top of the opaque cloud layer.
PFES_CLEAR	The color for simple screen clearing.

The fog color is set as explained in the **pfFog** reference page.

pfESkyAttr is used to set a number of attributes. The companion function **pfGetESkyAttr** is used to return these same attribute values. The tokens and their meanings are listed below:

PFES_GRND_HT Set the ground height for the ground plane that is used when **PFES_SKY_GRND** is enabled and defines the bottom edge of the horizon which is used in all of the modes that draw a sky. The ground plane extends from the eyepoint to the horizon with a width greater than the field of view. Note that objects placed on the ground with the same height may not Z buffer correctly. Also, as objects move into the

distance, the Z buffer resolution for those pixels will decrease, making proper priority resolution of small distances between the ground plane and objects less likely.

PFES_HORIZ_ANGLE Set the vertical displacement of the horizon band in degrees. The horizon band is blended into the sky bottom color so it may appear to be less than this angle. This angle remains constant for any heading. To simulate directional horizon glow, the angle and color can be changed each frame to achieve the correct appearance.

PFES_CLOUD_TOP Set the cloud layer upper position. The cloud layer is enabled when the cloud base is less than the cloud top. By default, it is disabled (base > top). Each token is followed by a height value. The cloud layer is opaque. The cloud layer thickness is simply (top - bottom).

PFES_CLOUD_BOT Set the cloud layer lower position. The cloud layer is enabled when the cloud base is less than the cloud top. By default, it is disabled (base > top). Each token is followed by a height value. The cloud layer is opaque. The cloud layer thickness is simply (top - bottom).

PFES_TZONE_TOP Set the transition zone for exiting a cloud layer. Provided to allow a smooth transition out of clouds. This transition is enabled by making the transition height greater than the cloud top. It is disabled by doing the opposite or by disabling the cloud layer. By default, the transition zone is disabled.

PFES_TZONE_BOT Set the transition zone for entering a cloud layer. Provided to allow a smooth transition into clouds. This transition is enabled by making the transition height less than the cloud bottom. It is disabled by doing the opposite or by disabling the cloud layer. By default, the transition zone is disabled.

PFES_GRND_FOG_TOP Set the height of the ground fog layer. Ground fog is enabled when a valid pfFog is set. By default ground fog is disabled.

pfESkyFog sets which type of fog to use when in ground fog or general visibility. The token may be one of the following values:

PFES_GRND
PFES_GENERAL

pfGetESkyFog returns the indicated fog selection.

Several different fog functions may be defined at initialization, then just switched in using this routine. Distant haze and different curves would be done this way. If ground fog is enabled, and the viewer is transitioning out of the ground fog layer, the fog will be blended into clear visibility or **PFES_GENERAL** fog.

Due to the design of the graphics library, fog would be discontinuous in adjacent channels which use rotational viewing offsets (See **pfChanViewOffsets**). However, when attached to a pfChannel (see **pfChanESky**) that has a rotational viewing offset, a pfEarthSky will automatically adjust the ranges of the

pfFog set by **pfESkyFog** to account for any rotational offsets so that fog is continuous across adjacent channels.

NOTES

pfEarthSky does not work properly for off-axis viewing frusta.

Because **PFES_TAG** only has effect when multisampling, care must be taken for cross-platform portability. Background renderings that rely on the depth buffer having been reset (e.g. backgrounds that do not disable z buffering with **zfunction(ZF_ALWAYS)** in IRIS GL or **glDepthFunc(GL_ALWAYS)** in OpenGL) may need to request a normal depth buffer clear when not multisampling.

When multisamling, **PFES_SKY_GND** and **PFES_SKY** are significantly faster than **PFES_SKY_CLEAR**.

In IRIX 5.3 IRIS GL on Indigo2/Extreme systems the Z-buffer is not fully updated after a window is moved unless a full Z-clear operation is performed. In such cases your software must detect REDRAW events and fully clear the Z-buffer.

SEE ALSO

pfChanViewOffsets, pfClear, pfFog, pfNewChan, zfunction, glDepthFunc, pfDelete

NAME

pfFrameRate, pfGetFrameRate, pfFieldRate, pfGetFieldRate, pfVideoRate, pfGetVideoRate, pfSync, pfFrame, pfAppFrame, pfGetFrameCount, pfFrameTimeStamp, pfGetFrameTimeStamp, pfPhase, pfGetPhase – Set and get system frame and video rate, phase, and frame count. Synchronize and initiate frame.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

float **pfFrameRate**(float rate);

float **pfGetFrameRate**(void);

int **pfFieldRate**(int fields);

int **pfGetFieldRate**(void);

void **pfVideoRate**(float vrate);

float **pfGetVideoRate**(void);

int **pfSync**(void);

int **pfFrame**(void);

int **pfAppFrame**(void);

int **pfGetFrameCount**(void);

void **pfFrameTimeStamp**(double time);

double **pfGetFrameTimeStamp**(void);

void **pfPhase**(int phase);

int **pfGetPhase**(void);

DESCRIPTION

IRIS Performer is designed to run at a fixed frame rate. The *rate* argument to **pfFrameRate** specifies the desired rate in units of frames per second. The actual rate used is based on the video timing of the display hardware. *rate* is rounded to the nearest frame rate which corresponds to an integral multiple of video fields.

For a 60Hz video rate, possible frame rates are (in Hz) 60.0, 30.0, 20.0, 15.0, 12.0, 10.0, 8.57, 7.5, 6.67, and 6.0. These rates would mean that the number of fields per frame would range from 1 (for 60Hz) to 10 (for 6Hz). **pfFrameRate** returns the actual frame rate used or -1.0 if it is called before **pfConfig**.

pfVideoRate specifies the system video rate as *vrate* fields per second. If **pfVideoRate** is not called, then IRIS Performer determines the video field rate at **pfConfig** time and will not be aware of changes in video timing made during application run-time until **pfVideoRate** is called.

pfGetVideoRate returns the video timing in number of video fields per second or -1.0 if it is called before

the video rate has been determined. The IRIS Performer video clock (see **pfInitVClock**) runs at this video field rate and is initialized to 0 by **pfConfig**.

An alternate way of specifying a desired frame rate is **pfFieldRate**. *fields* is the number of video fields per simulation frame. The corresponding frame rate will then be the video field rate (see **pfGetVideoRate**) divided by *fields*. **pfGetFieldRate** returns the number of video fields per simulation frame.

Frame rate is a per-machine metric and is used by all pfPipes. It controls the rate at which multiprocessing pipelines run and affects computed system load and related stress metrics (see **pfChanStress**). Since frame rate is global it follows that all hardware pipelines used by a single IRIS Performer application should be genlocked, i.e., the video signals are synchronized by hardware. Otherwise the video signals of the pipes will be out of phase, reducing graphics throughput and increasing latency. Genlock is crucial for proper multipipe operation and requires some simple, platform-specific cabling and software configuration through the **setmon** call.

Depending on the phase as is discussed below, **pfSync** synchronizes the application process with the frame rate specified by **pfFrameRate** (when phase is **PFPHASE_LOCK** or **PFPHASE_FLOAT**), or to the system rendering rate (when phase is **PFPHASE_FREE_RUN** or **PFPHASE_LIMIT**). In the first case, **pfSync** sleeps until the next frame boundary, then awakens and returns control to the application. In the second case, **pfSync** sleeps until the draw process begins rendering a new frame or returns immediately if in single-process operation. **pfSync** returns the current frame count and should only be called by the application process when multiprocessing.

pfFrame initiates a new frame of IRIS Performer processing by doing the following:

1. Triggers all processing stages that are configured as a separate process.
2. Inlines all processing stages that are not configured as a separate process.
3. Sets the current, global pfCycleBuffer index (see **pfCurCBufferIndex**) which is guaranteed not to be in use by any other IRIS Performer process.
4. Sets the frame's time stamp (**pfFrameTimeStamp**).

pfFrame triggers all IRIS Performer processing stages (**APP**, **ISECT**, **DBASE**, **CULL**, and **DRAW**). If a stage is partitioned into a separate process, **pfFrame** will allow that process to run. Otherwise, **pfFrame** itself will carry out the processing associated with the stage. **pfFrame** will directly invoke all user callbacks that are in the same process as that which called **pfFrame**. Otherwise, a callback will be invoked by the process of which it is a part, e.g., the **ISECT** callback will be invoked by the **ISECT** process if **PFMP_FORK_ISECT** is set in the argument to **pfMultiprocess**.

All IRIS Performer stage callbacks have a block of associated data known as "user data." User data is passed as an argument to the stage callback. To simplify data flow in a multiprocessing environment, IRIS Performer copies user data into internal buffers and propagates the data down multiprocessing pipelines.

To restrict data copying to only those frames in which user data changes, use the **pfPass<*>Data** functions. **pfPass<*>Data** signifys that the user data has changed and needs to be copied. **pfFrame** will then copy the data into its internal buffer and the stage callback will receive the updated user data. Stage callbacks and user data functions are listed below.

Stage	Callback	Allocation	Pass
APP	pfChanTravFunc	pfAllocChanData	pfPassChanData
CULL	pfChanTravFunc	pfAllocChanData	pfPassChanData
DRAW	pfChanTravFunc	pfAllocChanData	pfPassChanData
ISECT	pfIsectFunc	pfAllocIsectData	pfPassIsectData
DBASE	pfDBaseFunc	pfAllocDBaseData	pfPassDBaseData

pfFrame triggers the APP, CULL and DRAW stages of all pfPipes so it must be called every frame a new display is desired. IRIS Performer will attempt to cull and draw all active pfChannels on all pfPipes within a single frame period. Multiple pfChannels on a single pfPipe will be processed in the order they were added to the pfPipe. **pfFrame** returns the current frame count and should only be called by the application process when multiprocessing.

If specified, pfChannel cull and draw callbacks (**pfChanTravFunc**) will be invoked by the appropriate process which may or may not be the same process that called **pfFrame**. If these callbacks are not specified, **pfCull** and **pfDraw** will be called instead. pfChannel passthrough data which is passed to pfChannel function callbacks (see **pfPassChanData**) is copied into internal memory at **pfFrame** time.

In typical operation, **pfFrame** should closely follow **pfSync** in the main application loop. Since the **CULL** does not start until **pfFrame** is called, considerable processing between **pfSync** and **pfFrame** can reduce system throughput. However, any updates to the database or view made at this time will be applied to the current frame so latency is reduced for these updates. Updates made after **pfFrame** will be applied to the next frame. **pfFrame** returns the current frame count.

pfFrame will automatically call **pfSync** if the application did not call **pfSync** before calling **pfFrame**. This means the application need not call **pfSync**.

It is crucial to keep the time spent in the application process less than a frame's time so the system can meet the desired frame rate. If the application process exceeds a single frame's time, **pfFrame** will not be called often enough to meet the frame rate.

The following code fragment is an example of an application's main processing loop:

Example 1: Main simulation loop.

```
pfFrameRate(30.0f);     /* Set desired frame rate to 30Hz */

while (!done)
{
    app_funcs();        /* Perform application-specific functions */
    update_positions(); /* Update moving models for frame N */

    pfSync();           /* Sleep until next frame boundary */

    update_view();      /* Set view for frame N */
    pfFrame();          /* Trigger cull and draw for frame N */
}
```

pfAppFrame triggers a traversal that updates the state of the scene graph for the next frame. This includes updating the state of pfSequence nodes and invoking APP callbacks on nodes in the scene graph. If **pfAppFrame** is not invoked directly, **pfSync** or **pfFrame** invokes it automatically. Note that when the view is not set until after **pfSync**, as in the example above, the view point in the channel during the application traversal contains the eye point from the previous frame.

pfGetFrameCount returns the current frame count. The frame count is initialized to 0 by **pfConfig** and is incremented by each call to **pfFrame**.

pfGetFrameRate returns the current system frame rate (possibly rounded) previously set by **pfFrameRate**. Note that this is not necessarily the same as the achieved frame rate.

pfSync synchronizes the application process to a particular rate. This rate may be fixed, for example a steady 20Hz or may vary with the rendering rate. In addition, the drawing process may be synchronized to either a steady or a varying rate. **pfPhase** specifies the synchronization methods used by **pfSync** and the drawing process (if it is a separate process). *phase* is a symbolic constant that specifies the phase of all process pipeline(s). It can take on the following values:

PFPHASE_LOCK
: **pfSync** synchronizes to the next frame boundary and the drawing process begins drawing and swaps its rendering buffers only at fixed frame boundaries.

PFPHASE_FLOAT
: **pfSync** synchronizes to the next frame boundary but the drawing process can begin drawing and swap its rendering buffers at non-frame boundaries.

PFPHASE_FREE_RUN
: **pfSync** synchronizes to the rendering rate so the application runs at its peak (and usually non-constant) capability.

PFPHASE_LIMIT
pfSync synchronizes to the rendering rate but the rendering rate is limited to that frame rate specified by **pfFrameRate**.

If locked, the drawing process will swap buffers only on frame boundaries. A benefit of locking is that such pipelines are self-regulating so synchronizing two pfPipes together is simple, even across different machines. Another benefit is that latency is minimized and predictable. The major drawback is that if a view takes slightly longer than a frame to render (it has 'frame-extended'), then an entire frame is skipped rather than a single vertical retrace period. However, if minimal distraction is crucial, the phase can float so that buffer swapping may happen on non-frame boundaries. In this case it is not guaranteed that the windows on pfPipes will swap together; they may get out of phase resulting in inconsistent images if the displays are adjacent and are displaying the same scene.

The difference between phase lock and phase float becomes less apparent with increasing frame rate. At a rate equal to the vertical retrace rate, there is no difference. Also, if pfPipes do not 'frame extend', then there is no difference.

Applications which do not require a fixed frame rate may use **PFPHASE_FREE_RUN** or **PFPHASE_LIMIT**. **PFPHASE_FREE_RUN** essentially disables IRIS Performer's fixed frame rate mechanisms and will cause the application to run at its rendering rate so it slows down when rendering complex scenes and speeds up when rendering simple scenes. In this case, the frame rate specified by **pfFrameRate** no longer affects the system frame rate but is still used to compute system load and stress.

PFPHASE_LIMIT is equivalent to **PFPHASE_FREE_RUN** except that the application can go no faster than the frame rate specified by **pfFrameRate** although it may go slower. Thus fixed frame rate behavior is achieved if the time required to process a frame never takes longer than that specified by **pfFrameRate**.

pfPhase may be called any time after **pfConfig**.

pfGetPhase returns the current phase. The default phase is **PFPHASE_FREE_RUN**.

pfFrameTimeStamp sets the time stamp of the current frame to *time*. The frame time stamp is used when evaluating all pfSequences. Normally, **pfFrame** sets the frame time stamp immediately before returning control to the application although the application may set it to account for varying latency in a non-constant frame rate situation. Time is relative to **pfInit** when the system clock is initialized to 0.

SEE ALSO

pfChanStress, pfConfig, pfIsectFunc, pfInitVClock, pfCycleBuffer, pfGetTime

NAME

pfNewFStats, pfGetFStatsClassType, pfDrawFStats, pfCopyFStats, pfGetOpenFStats, pfOpenFStats, pfCloseFStats, pfFStatsCountNode, pfFStatsClass, pfGetFStatsClass, pfFStatsClassMode, pfGetFStatsClassMode, pfFStatsAttr, pfGetFStatsAttr, pfResetFStats, pfClearFStats, pfFStatsCountGSet, pfAccumulateFStats, pfAverageFStats, pfQueryFStats, pfMQueryFStats – Specify pfFrameStats modes and get collected values.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

#include <Performer/pfstats.h>

pfFrameStats * **pfNewFStats**(void);

pfType* **pfGetFStatsClassType**(void);

void **pfDrawFStats**(pfFrameStats *fstats, pfChannel *chan);

void **pfCopyFStats**(pfFrameStats *dst, pfFrameStats *src, uint dSel, uint sSel, uint classes);

uint **pfGetOpenFStats**(pfFrameStats *fstats, uint emask);

uint **pfOpenFStats**(pfFrameStats *fstats, uint enmask);

uint **pfCloseFStats**(uint enmask);

void **pfFStatsCountNode**(pfFrameStats *fstats, int class, uint mode, pfNode * node);

uint **pfFStatsClass**(pfFrameStats *fstats, uint enmask, int val);

uint **pfGetFStatsClass**(pfFrameStats *fstats, uint enmask);

uint **pfFStatsClassMode**(pfFrameStats *fstats, int class, uint mask, int val);

uint **pfGetFStatsClassMode**(pfFrameStats *fstats, int class);

void **pfFStatsAttr**(pfFrameStats *fstats, int attr, float val);

float **pfGetFStatsAttr**(pfFrameStats *fstats, int attr);

void **pfResetFStats**(pfFrameStats *fstats);

void **pfClearFStats**(pfFrameStats *fstats, uint which);

void **pfFStatsCountGSet**(pfFrameStats * fstats, pfGeoSet * gset);

void **pfAccumulateFStats**(pfFrameStats* dst, pfFrameStats* src, uint which);

void **pfAverageFStats**(pfFrameStats* dst, pfFrameStats* src, uint which, int num);

int **pfQueryFStats**(pfFrameStats *fstats, uint which, float *dst, int size);

int **pfMQueryFStats**(pfFrameStats *fstats, uint *which, float *dst, int size);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfFrameStats** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfFrameStats**. Casting an object of class **pfFrameStats** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfFrameStats** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

fstats identifies a pfFrameStats.

DESCRIPTION

The pfFrameStats utilities provide for the collection of statistics about all parts of IRIS Performer processing of a scene for a given frame. These statistics can be kept automatically on every pfChannel or Users may accumulate and store their own statistics. Routines for operating on, displaying, and printing statistics are also provided.

The frame statistics for a channel are gotten by first getting the pointer to the channel's statistics structure with **pfGetChanFStats**, and then enabling the desired statistics classes. When a channel is automatically accumulating frame statistics, it enables the necessary statistics hardware and statistics accumulation in the correct processes.

The resulting collected statistics can then be displayed in a channel, queried, or printed. These statistics may be accumulated and averaged over a specified number of frames or seconds. The pfFrameStats declarations are contained in pfstats.h. The class of process frame timing statistics for each of the IRIS Performer processes of application, cull and draw, is enabled by default.

pfNewFStats creates and returns a handle to a pfFrameStats. pfFrameStats are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetFStatsClassType returns the **pfType*** for the class **pfFrameStats**. The **pfType*** returned by **pfGetFStatsClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfFrameStats**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

A pfFrameStats structure contains pfStats statistics as well as additional statistics classes and support for tracking frame related tasks. Many pfFrameStats routines are borrowed from pfStats. These routines have the identical function as the pfStats routines but operate on a pfFrameStats rather than a pfStats. The routine correspondence is listed in the following table.

pfFrameStats routine	pfStats routine
pfAccumulateFStats	pfAccumulateStats
pfAverageFStats	pfAverageStats
pfClearFStats	pfClearStats
pfCloseFStats	pfCloseStats
pfFStatsAttr	pfStatsAttr
pfFStatsClass	pfStatsClass
pfFStatsClassMode	pfStatsClassMode
pfGetFStatsClassMode	pfGetStatsClassMode
pfFStatsCountGSet	pfStatsCountGSet
pfGetFStatsAttr	pfGetStatsAttr
pfGetFStatsClass	pfGetStatsClass
pfOpenFStats	pfOpenStats
pfCloseFStats	pfCloseStats
pfGetOpenFStats	pfGetOpenStats
pfMQueryFStats	pfMQueryStats
pfOpenFStats	pfOpenStats
pfGetOpenFStats	pfGetOpenStats
pfQueryFStats	pfQueryStats
pfResetFStats	pfResetStats

The reader is referred to the pfStats man page for details on the routine description.

Only the additional support for pfFrameStats above and beyond that of **pfNewStats** is discussed here. There are also examples showing different basic operations with pfFrameStats at the end of this reference page. The pfFrameStats structure stores accumulated statistics in several buffers. The following is a list of the frame statistics buffers:

PFFSTATS_BUF_PREV	Statistics for previous completed frame
PFFSTATS_BUF_CUR	Buffer for current statistics collection
PFFSTATS_BUF_CUM	Statistics accumulated since last update
PFFSTATS_BUF_AVG	Statistics averaged over previous update period

These different buffers can be queried with **pfQueryFStats** and printed with **pfPrint**. The desired **PFFSTATS_BUF_*** token is simply bitwise OR-ed with the desired statistics value token.

The following table of additional frame statistics classes, their naming token, and their enable token for forming bitmasks. Notice that pfFrameStats tokens start with **PFFSTATS***.

Frame Statistics Class Table

Class	PFSTATS_* Token	PFSTATS_EN* token
Process frame times	**PFFSTATS_PFTIMES**	**PFFSTATS_ENPFTIMES**
Database	**PFFSTATS_DB**	**PFFSTATS_ENDB**
Cull	**PFFSTATS_CULL**	**PFFSTATS_ENCULL**

This table lists the frame statistics modes and tokens.

Frame Statistics Class Mode Table

Class	PFSTATS_ Token	Modes
Process frame times	**PFFSTATS_PFTIMES**	**PFFSTATS_PFTIMES_BASIC** **PFFSTATS_PFTIMES_HIST**
Database	**PFFSTATS_DB**	**PFFSTATS_DB_VIS** **PFFSTATS_DB_EVAL**
Cull	**PFFSTATS_CULL**	**PFFSTATS_CULL_TRAV**

pfDrawFStats displays the pfFrameStats structure *fstats* in the channel specified by *chan*. This is useful for displaying the statistics in a special channel separate from the main scene channel. **pfDrawChanStats** may be called from IRIS Performer's application, cull, or draw processes and must be called each frame a statistics display is desired. See **pfDrawChanStats** for a detailed explanation of the channel statistics display.

pfFStatsClass takes a pointer to a statistics structure, *fstats*, and will set the classes specified in the bitmask, *enmask*, according to the *val*, which is one of the following:

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*.

All stats collection can be set at once to on, off, or the default by using **PFSTATS_ALL** for the bitmask and the appropriate value for the enable flag. For example, the following function call will enable all frame statistics, as well as basic statistics classes, with their current class mode settings.

```
pfFStatsClass(fstats, PFSTATS_ALL, PFSTATS_ON);
```

Only statistics classes that are enabled with pfFStatsClass are able to be printed with **pfPrint**, collected, copied, accumulated, averaged, and queried.

pfGetFStatsClass takes a pointer to a statistics structure, *fstats*, and the statistics classes of interest specified in the bitmask, *enmask*. The frame statistics classes are enabled through **pfFStatsClass** and the frame statistics class bitmasks may be combined with the basic statistics classes. If any of the statistics classes specified in *enmask* are enabled, then **pfGetFStatsClass** will return the bitmask of those classes, and will otherwise return zero.

pfFStatsClassMode takes a pointer to a pfFrameStats structure, *fstats*, the name of the class to set, *class*, a mask of class modes, *mask*, and the value for those modes, *val*. The pfFrameStats classes include all of the pfStats classes. If *class* is **PFSTATS_CLASSES**, then all pfFrameStats classes will have their modes set according to *mask* and *val*. Each statistics class has its own mode tokens that may be used for *mask*. *mask* may also be one of **PFSTATS_ALL** or 0x0. *val* is one of the statistics value tokens: **PFSTATS_ON**, **PFSTATS_OFF**, **PFSTATS_SET**, or **PFSTATS_DEFAULT**. See the pfStats reference page for more general information on pfStats statistics classes and value tokens under **pfStatsClassMode**. The following describes the additional classes for frame statistics and their corresponding modes.

Process Frame Times Modes:

PFFSTATS_PFTIMES_BASIC
This mode enables a running average of the time for each IRIS Performer process of application, cull, and draw to complete the tasks for a single frame. This mode is enabled by default.

PFFSTATS_PFTIMES_HIST
In this mode, a history of time stamps for different tasks within each of the IRIS Performer process of application, cull, draw, and the intersection process, is maintained. Examples of time stamps include when each processes starts and ends processing a frame, and the

application frame number for that frame for that processes. There are special additional time stamps for each process. For the application processes there are time stamps to mark when the application starts and finishes cleaning the scene in pfSync, a time stamp when the application wakes up to sync to the next frame boundary (done when the application is running with phase set to **PFPHASE_LOCK** or **PFPHASE_FLOAT**), and a time stamp to mark when the application returns after setting off a forked CULL or ISECT process. The time stamps for each process are defined in the pfFStatsValPFTimes* data type and queried by providing the corresponding **PFFSTATSVAL_PFTIMES_HIST_*** tokens to **pfQueryFStats**.

Database Statistics Modes:

PFFSTATS_DB_VIS
This mode enables tracking of how many pfNodes of each different type are visible and drawn in a given frame. This mode is enabled by default. These statistics are queried by providing the desired **PFFSTATSVAL_VISIBLE*** token to **pfQueryFStats**.

PFFSTATS_DB_EVAL
This mode enables tracking of how many pfNodes of each different type have special evaluations in a given frame. Node types that require special evaluation steps include pfBillboard, pfSCS, pfDCS, pfLayer, pfLightPoint, pfLightSource, pfPartition, and pfSequence. There are also query tokens to query what processes the evaluation step for a given node type is done in. This mode is enabled by default. These statistics are queried by providing the desired **PFFSTATSVAL_EVALUATED*** token to **pfQueryFStats**.

Cull Statistics Modes:

PFFSTATS_CULL_TRAV
There is only one cull frame statistics mode and it tracks culling traversal statistics: how many pfGeoSets and pfNodes of each type are traversed in the cull operation, how many pfNodes are trivially in or out of the viewing frustum, and how many must pass through a bounding sphere or bounding box test. These statistics are queried by providing one of the **PFFSTATSVAL_CULLTRAV** tokens to **pfQueryFStats**. There are also statistics on the test results of the cull traversal, queried with the **PFFSTATSVAL_CULLTEST*** tokens.

pfGetFStatsClassMode takes a pointer to a statistics structure, *fstats*, and the name of the class to query, *class*. The return value is the mode of *class*.

pfFStatsAttr takes a pointer to a statistics structure, *fstats*, the name of the attribute to set, *attr*, and the attribute value, *val*. Frame statistics provide additional attributes beyond the basic pfStats attributes. These attributes are only relevant when automatic statistics collection is being done by a parent channel. These attributes are:

PFFSTATS_UPDATE_FRAMES
The number of frames over which statistics should be averaged. The default value is 2. If *val* is 0, statistics accumulation and averaging is disabled and only the **CUR** and **PREV** statistics for enabled classes will be maintained. This is recommended for applications that are not using the averaged statistics and require a high, fixed frame rate.

PFFSTATS_UPDATE_SECS
The number of seconds, over which statistics should be averaged. The default uses the number of frames. As with **PFFSTATS_UPDATE_FRAMES**, if *val* is 0, statistics accumulation and averaging is disabled and only the **CUR** and **PREV** statistics for enabled classes will be maintained.

PFFSTATS_PFTIMES_HIST_FRAMES
For the Process Frame Times Statistics, **PFFSTATS_PFTIMES**, the number of frames of time-stamp history to keep. The default value is 4.

pfGetFStatsAttr takes a pointer to a statistics structure, *fstats*, and the name of the attribute to query, *attr*. The return value is that of attribute *attr*.

pfQueryFStats takes a pointer to a statistics structure, *fstats*. *which* is a **PFSTATSVAL_*** or **PFFSTATSVAL_*** token that specifies the value or values to query in *which*, and *dst* destination buffer that is a pointer to a float, a pfStatsVal* or pfFStatsVal* structure. The size of the expected return data is specified by *size* and if non-zero, will prevent **pfQueryFStats** from writing beyond a buffer that is too small. The return value is the number of bytes written to the destination buffer. The return value is the number of bytes written to the destination buffer. A single **PFFSTATS_BUF_*** token should be bitwise OR-ed into the *which* flag to select a frame stats buffer: **PREV**, **CUR**, **AVG**, or **CUM**. If no frame statistics buffer is selected, then the query accesses the CUR buffer by default. If multiple stats buffers are selected, no results will be written and a warning message will be printed. In a running application, one should query frame statistics in the application process and query the **PREV** and **AVG** statistics buffers. The pfFrameStats query structures and tokens are all defined in pfstats.h. Frame statistics queries may be mixed with standard statistics queries. There are tokens for getting back all of the statistics, entire sub-structures, and individual values.

pfMQueryFStats takes a pointer to a statistics structure, *fstats*, a pointer to the start of an array of query tokens in *which*, and a destination buffer *dst*. The size of the expected return data is specified by *size* and if non-zero, will prevent **pfQueryFStats** from writing beyond a buffer that is too small. The return value is the number of bytes written to the destination buffer. The return value is the number of bytes written to the destination buffer. If at any point in the query, an error is encountered, the query will return and not finish the rest of the requests.

pfCopyFStats: The *dSel* and *sSel* arguments explicitly specify the statistics buffers for both source and destination pfFrameStats structures. If either of these values are 0, then the current pfFrameStats buffer is used for the corresponding pfFrameStats structure. The *classes* argument is a **_EN*** statistics class enable

bitmask. Any buffer select token is included with the class bitmask is ignored.

pfFStatsCountNode will count *node* in the specified stats *class* for the specified *mode* of the pfFrameStats structure *fstats*. Only one class and mode may be specified, and children of *node* are not traversed.

pfFStatsCountGSet works as documented for the pfStats statistics structure and accumulates the statistics into the **CUR** statistics buffer.

The **pfClearFStats**, **pfAccumulateFStats**, **pfAverageFStats** routines all take pointers to a pfFrameStats structure, *fstats*, and work as documented for the basic pfStats statistics structure. However, for operating on a pfFrameStats structure, these routines need to know which pfFrameStats buffer to access. A pfFrameStats buffer is selected by OR-ing in a **_BUF_** token with the statistics class enable. The same pfFrameStats buffer is used for both source and destination pfFrameStats structures. If no pfFrameStats buffer is selected with a **_BUF_** token, the current pfFrameStats buffer is used.

EXAMPLES

For a class of statistics to be collected, the following must be true:

1. A pfFrameStats structure must be gotten from the channel of interest, or created.
2. The corresponding statistics class must be enabled with **pfFStatsClass**. No statistics classes are enabled by default.
3. The corresponding statistics class mode must be enabled with **pfFStatsClassMode** However, each statistics class does have a reasonable set of statistics modes enabled by default.

Here a pfFrameStats structure is obtained by the channel of interest and then database, cull, and graphics statistics are enabled.

```
pfFrameStats *fstats = NULL;
fstats = pfGetChanFStats(chan);
pfFStatsClass(stats, PFSTATS_ENGFX | PFFSTATS_ENDB | PFFSTATS_ENCULL, PFSTATS_ON);
```

This example shows how to enable and display just the frame times and the number of triangles per frame. This is a very efficient configuration.

```
pfFrameStats *fstats = NULL;
fstats = pfGetChanFStats(chan);

/* first, turn off the frame history stats */
pfFStatsClassMode(fstats, PFFSTATS_PFTIMES, PFFSTATS_PFTIMES_HIST, PFSTATS_OFF);

/* Only enable the geometry counts in the graphics stats */
pfFStatsClassMode(fstats, PFSTATS_GFX, PFSTATS_GFX_GEOM, PFSTATS_SET);

/* disable the display of the verbose graphics stats
```

```
 * and just have the total tris number at the top of your display.
 */
pfChanFStatsMode(chan, PFCSTATS_DRAW, PFFSTATS_ENPFTIMES);
```

The following is an example of querying a few specific statistics. Note that if the corresponding stats class and mode is not enabled then the query will simply return 0 for that value.

```
uint qtmp[5];
float ftmp[5];
pfFrameStats *fstats = NULL;

fstats = pfGetChanFStats(chan);

qtmp[0] = PFFSTATS_BUF_AVG | PFSTATSVAL_GFX_GEOM_TRIS;
qtmp[1] = PFFSTATS_BUF_AVG | PFFSTATSVAL_PFTIMES_PROC_TOTAL;
qtmp[2] = PFFSTATS_BUF_AVG | PFSTATSVAL_CPU_SYS_BUSY;
qtmp[3] = NULL;

pfMQueryFStats(fstats, qtmp, ftmp, sizeof(ftmp));

fprintf(stderr, "Query num tris: %.0f\n", ftmp[0]);
fprintf(stderr, "Query frame time: %.0f msecs\n", ftmp[1]*1000.0f);
fprintf(stderr, "Query sys busy: %.0f%%\n", ftmp[2]);
```

This example shows using a very inexpensive pfFrameStats mode to track frame rates and frames that missed the goal frame rate.

```
/* enable only the most minimal stats - tracking of process frame times */
pfFrameStats *fstats = pfGetChanFStats(chan);
pfFStatsClass(fstats, PFFSTATS_ENPFTIMES, PFSTATS_SET);
pfFStatsClassMode(fstats, PFFSTATS_PFTIMES, PFFSTATS_PFTIMES_BASIC, PFSTATS_SET);

/* turn off accumulation and averaging of stats */
pfFStatsAttr(fstats, PFFSTATS_UPDATE_FRAMES, 0.0f);

#define STAMPS 0
#define TIMES  1
#define MISSES 2
static uint query[] = {
    PFFSTATS_BUF_PREV | PFFSTATSVAL_PFTIMES_APPSTAMP,
    PFFSTATS_BUF_PREV | PFFSTATSVAL_PFTIMES_PROC,
    PFFSTATS_BUF_PREV | PFFSTATSVAL_PFTIMES_MISSES, NULL
```

```
};

static pfFStatsValProc dst[3];
int i;

if (!FrameStats)
    initFrameStats();

/* get the prev frame times and corresponding app frame stamps */
pfMQueryFStats(fstats, query, dst, sizeof(dst));

/* record the collected data here */
```

NOTES

pfDrawFStats does not actually draw the diagram but sets a flag so that the diagram is drawn just before IRIS Performer swaps buffers.

The CPU statistics from the pfStats class **PFSTATSHW_CPU** are obtained from IRIX process accounting data at the start and end of the update period. They are then copied into the **CUR** and **AVG** buffers.

pfOpenFStats and **pfCloseFStats** cannot be executed on a pfFrameStats structure. All actual frame statistics collection is done only by individual pfChannels. Frame statistics can be copied and accumulated into additional pfFrameStats structures.

The **pfDrawChanStats** manual page gives some pointers on how to interpret the statistics to help in tuning your database. Refer to the IRIS Performer Programming Guide for more detailed information.

BUGS

The checking of *size* in **pfQueryFStats** and **pfMQueryFStats** is not yet implemented.

SEE ALSO

pfChanStatsMode, pfDrawChanStats, pfGetChanFStats, pfStats, pfDelete

NAME

pfNewGeode, pfGetGeodeClassType, pfAddGSet, pfRemoveGSet, pfInsertGSet, pfReplaceGSet, pfGetGSet, pfGetNumGSets – Create, modify, and query a geometry node.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

pfGeode * **pfNewGeode**(void);

pfType * **pfGetGeodeClassType**(void);

int **pfAddGSet**(pfGeode* geode, pfGeoSet* gset);

int **pfRemoveGSet**(pfGeode* geode, pfGeoSet* gset);

int **pfInsertGSet**(pfGeode* geode, int index, pfGeoSet* gset);

int **pfReplaceGSet**(pfGeode* geode, pfGeoSet* old, pfGeoSet* new);

pfGeoSet * **pfGetGSet**(const pfGeode* geode, int index);

int **pfGetNumGSets**(const pfGeode* geode);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfGeode** is derived from the parent class **pfNode**, so each of these member functions of class **pfNode** are also directly usable with objects of class **pfGeode**. Casting an object of class **pfGeode** to an object of class **pfNode** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfNode**.

pfGroup * **pfGetParent**(const pfNode *node, int i);
int **pfGetNumParents**(const pfNode *node);
void **pfNodeBSphere**(pfNode *node, pfSphere *bsph, int mode);
int **pfGetNodeBSphere**(pfNode *node, pfSphere *bsph);
pfNode* **pfClone**(pfNode *node, int mode);
pfNode* **pfBufferClone**(pfNode *node, int mode, pfBuffer *buf);
int **pfFlatten**(pfNode *node, int mode);
int **pfNodeName**(pfNode *node, const char *name);
const char * **pfGetNodeName**(const pfNode *node);
pfNode* **pfFindNode**(pfNode *node, const char *pathName, pfType *type);
pfNode* **pfLookupNode**(const char *name, pfType* type);
int **pfNodeIsectSegs**(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void **pfNodeTravMask**(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint **pfGetNodeTravMask**(const pfNode *node, int which);
void **pfNodeTravFuncs**(pfNode* node, int which, pfNodeTravFuncType pre, pfNodeTravFuncType post);
void **pfGetNodeTravFuncs**(const pfNode* node, int which, pfNodeTravFuncType *pre, pfNodeTravFuncType *post);

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

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfGeode** can also be used with these functions designed for objects of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfGeode** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

geode identifies a pfGeode.

DESCRIPTION

The name "pfGeode" is short for Geometry Node. A pfGeode is a leaf node in the IRIS Performer scene graph hierarchy and is derived from pfNode so it can use pfNode API. A pfGeode is simply a list of pfGeoSets which it draws and intersects with. A pfGeode is the smallest cullable unit unless **PFCULL_GSET** is set by **pfChanTravMode** in which case IRIS Performer will cull individual pfGeoSets within pfGeodes.

The bounding volume of a pfGeode is that which surrounds all its pfGeoSets. Unless the bounding volume is considered static (see **pfNodeBSphere**), IRIS Performer will compute a new volume when the list of pfGeoSets is modified by **pfAddGSet**, **pfRemoveGSet**, **pfInsertGSet** or **pfReplaceGSet**. If the bounding box of a child pfGeoSet changes, call **pfNodeBSphere** to tell IRIS Performer to update the bounding volume of the pfGeode.

pfNewGeode creates and returns a handle to a pfGeode. Like other pfNodes, pfGeodes are always

allocated from shared memory and can be deleted using **pfDelete**.

pfGetGeodeClassType returns the **pfType*** for the class **pfGeode**. The **pfType*** returned by **pfGetGeodeClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfGeode**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfAddGSet appends *gset* to *geode*'s pfGeoSet list. **pfRemoveGSet** removes *gset* from the list and shifts the list down over the vacant spot. For example, if *gset* had index 0, then index 1 becomes index 0, index 2 becomes index 1 and so on. **pfRemoveGSet** returns a 1 if *gset* was actually removed and 0 if it was not found in the list. **pfAddGSet** and **pfRemoveGSet** will cause IRIS Performer to recompute new bounding volumes for *geode* unless it is configured to use static bounding volumes.

pfInsertGSet will insert *gset* before the pfGeoSet with index *index*. *index* must be within the range 0 to **pfGetNumGSets**(*geode*). **pfReplaceGSet** replaces *old* with *new* and returns 1 if the operation was successful or 0 if *old* was not found in the list. **pfInsertGSet** and **pfReplaceGSet** will cause IRIS Performer to recompute new bounding volumes for *geode* unless it is configured to use static bounding volumes.

pfGetNumGSets returns the number of pfGeoSets in *geode*. **pfGetGSet** returns a handle to the pfGeoSet with index *index* or **NULL** if the index is out of range.

If database sorting is disabled, that is if the **PFCULL_SORT** mode of **pfChanTravMode** is not set, the pfGeoSets in a pfGeode will be drawn in the order they appear on the list. If sorting is enabled, there is no guarantee about the drawing order, since the reordering of GeoSets for minimum state-changing overhead is one of the primary design motivations of IRIS Performer's libpf and libpr.

NOTES

pfGeode geometry is not multibuffered by IRIS Performer when in multiprocessing mode in order to save memory. Therefore there are some restrictions on dynamic geometry. Modified vertex positions will be culled properly only if a static bound is defined which surrounds all possible excursions of the dynamic geometry. Since the draw process may be drawing the geometry at the same time the application process is modifying it, cracks may appear between polygons which share a dynamic vertex. Creation and deletion of vertices are not currently supported by IRIS Performer. However, the application may handle its own multibuffering of pfGeodes through mutual exclusion with locks or through the use of parallel data structures and pfSwitch nodes to achieve any kind of dynamic geometry.

The shifting behavior of **pfRemoveGSet** can cause some confusion. The following sample code shows how to remove all pfGeoSets from *geode*:

```
int     i;
int     n = pfGetNumGSets(geode);

for (i = 0; i < n; i++)
    pfRemoveGSet(geode, pfGetGSet(geode, 0)); /* 0, not i */
```

Alternately, you can traverse the list from back to front, in which case the shift never hits the fan.

```
int     i;
int     n = pfGetNumGSets(geode);

for (i = n - 1; i >= 0; i--)
    pfRemoveGSet(geode, pfGetGSet(geode, i)); /* i, not 0 */
```

When sorting is enabled (see **pfChanTravMode** and **PFCULL_SORT**), transparent pfGeoSets are drawn last unless the pfGeode has a pre or post draw callback (see **pfNodeTravFuncs**). Drawing transparent pfGeoSets after opaque geometry reduces artifacts when blended transparency (see **pfTransparency**) is used and can improve fill rate performance.

SEE ALSO

pfChanTravMode, pfGeoSet, pfNode, pfNodeTravFuncs, pfTransparency, pfDelete

NAME

pfGetId – Get unique id of libpf object.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

int **pfGetId**(void *unknown);

PARAMETERS

unknown is a pointer to a libpf object

DESCRIPTION

All IRIS Performer objects defined in the **libpf** library have a unique integer identifier. **pfGetId** returns the identifier of *unknown* or -1 if *unknown* is not a **libpf** data type.

SEE ALSO

pfNode, pfUpdatable

NAME

pfNewGroup, pfGetGroupClassType, pfAddChild, pfInsertChild, pfReplaceChild, pfRemoveChild, pfSearchChild, pfGetChild, pfGetNumChildren, pfBufferAddChild, pfBufferRemoveChild – Create, modify, and query a group node.

FUNCTION SPECIFICATION

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

pfGroup *   pfNewGroup(void);
pfType *    pfGetGroupClassType(void);
int         pfAddChild(pfGroup *group, pfNode *child);
int         pfInsertChild(pfGroup *group, int index, pfNode *child);
int         pfReplaceChild(pfGroup *group, pfNode *old, pfNode *new);
int         pfRemoveChild(pfGroup *group, pfNode* child);
int         pfSearchChild(pfGroup *group, pfNode* child);
pfNode *    pfGetChild(const pfGroup *group, int index);
int         pfGetNumChildren(const pfGroup *group);
int         pfBufferAddChild(pfGroup *group, pfNode *child);
int         pfBufferRemoveChild(pfGroup *group, pfNode *child);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfGroup** is derived from the parent class **pfNode**, so each of these member functions of class **pfNode** are also directly usable with objects of class **pfGroup**. Casting an object of class **pfGroup** to an object of class **pfNode** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfNode**.

```
pfGroup *    pfGetParent(const pfNode *node, int i);
int          pfGetNumParents(const pfNode *node);
void         pfNodeBSphere(pfNode *node, pfSphere *bsph, int mode);
int          pfGetNodeBSphere(pfNode *node, pfSphere *bsph);
pfNode*      pfClone(pfNode *node, int mode);
pfNode*      pfBufferClone(pfNode *node, int mode, pfBuffer *buf);
int          pfFlatten(pfNode *node, int mode);
int          pfNodeName(pfNode *node, const char *name);
const char * pfGetNodeName(const pfNode *node);
pfNode*      pfFindNode(pfNode *node, const char *pathName, pfType *type);
pfNode*      pfLookupNode(const char *name, pfType* type);
int          pfNodeIsectSegs(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
```

```
void        pfNodeTravMask(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint        pfGetNodeTravMask(const pfNode *node, int which);
void        pfNodeTravFuncs(pfNode* node, int which, pfNodeTravFuncType pre,
                pfNodeTravFuncType post);
void        pfGetNodeTravFuncs(const pfNode* node, int which, pfNodeTravFuncType *pre,
                pfNodeTravFuncType *post);
void        pfNodeTravData(pfNode *node, int which, void *data);
void *      pfGetNodeTravData(const pfNode *node, int which);
```

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfGroup** can also be used with these functions designed for objects of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
void*   pfGetUserData(pfObject *obj);
```

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfGroup** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *      pfGetType(const void *ptr);
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

PARAMETERS

group identifies a pfGroup.

DESCRIPTION

A pfGroup is the internal node type of the IRIS Performer hierarchy and is derived from pfNode. A pfGroup has a list of children which are traversed when *group* is traversed. Children may be any pfNode which includes both internal nodes (pfGroups) and leaf nodes (pfNodes). Other nodes which are derived from pfGroup may use pfGroup API. IRIS Performer nodes derived from pfGroup are:

pfScene
pfSwitch
pfLOD

pfSequence
pfLayer
pfSCS
pfDCS
pfMorph

pfNewGroup creates and returns a handle to a pfGroup. Like other pfNodes, pfGroups are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetGroupClassType returns the **pfType*** for the class **pfGroup**. The **pfType*** returned by **pfGetGroupClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfGroup**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfAddChild appends *child* to *group* and increments the reference count of *child*. **pfRemoveChild** removes *child* from the list and shifts the list down over the vacant spot, e.g. - if *child* had index 0, then index 1 becomes index 0, index 2 becomes index 1 and so on. **pfRemoveChild** returns a 1 if *child* was actually removed and 0 if it was not found in the list. **pfRemoveChild** decrements the reference count of *child* but does not delete *child* if its reference count reaches 0.

pfInsertChild inserts *child* before the child with index *index*. *index* must be within the range 0 to **pfGetNumChildren**(*group*).

pfReplaceChild replaces *old* with *new* and returns 1 if the operation was successful or 0 if *old* is not a child of *group*.

pfSearchChild returns the index of *child* if it was found in the children list of *group* or -1 if it was not found.

pfGetNumChildren returns the number of children in *group*. **pfGetChild** returns a handle to the child with index *index* or NULL if the index is out of range.

The bounding volume of a pfGroup encompasses all its children. Modifications to the child list of a pfGroup will cause IRIS Performer to recompute new bounding volumes for the pfGroup unless it is configured to use static bounding volumes (see **pfNodeBSphere**).

pfBufferAddChild and **pfBufferRemoveChild** provide access to nodes that do not exist in the current pfBuffer (See the pfBuffer man page). Either, none, or both of *group* and *node* may exist outside the current pfBuffer. **pfBufferAddChild** and **pfBufferRemoveChild** act just like their non-buffered counterparts **pfAddChild** and **pfRemoveChild** except that the addition or removal request is not carried out immediately but is recorded by the current pfBuffer. The request is delayed until the first **pfMergeBuffer** when both *group* and *node* are found in the main IRIS Performer pfBuffer. The list of **pfBufferAddChild** and

pfBufferRemoveChild requests is traversed in **pfMergeBuffer** after all nodes have been merged. **pfBufferAddChild** and **pfBufferRemoveChild** return **TRUE** if the request was recorded and **FALSE** otherwise.

SEE ALSO

pfLookupNode, pfNode, pfBuffer, pfDelete

NAME

pfInit, pfExit – Initialize and terminate IRIS Performer processes.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

int **pfInit**(void);

void **pfExit**(void);

DESCRIPTION

pfInit initializes internal IRIS Performer data structures and must be the first IRIS Performer call in an application except for the following:

pfNotifyLevel

pfSharedArenaSize

pfSharedArenaBase

pfTmpDir

pfInit is required by all Performer applications whether they use libpf or not. But **pfInit** has slightly different behavior applications that only use libpr and do not include pf.h. for these applications, **pfInit** does not set up any shared memory arenas. If shared memory is required, it should be explicitly set up by calling **pfInitArenas** before **pfInit**.

pfExit closes graphics windows, frees all IRIS Performer data structures, deletes all IRIS Performer shared memory arenas (see **pfGetSharedArena**), kills all spawned IRIS Performer processes, then returns control to the application. **pfExit** also turns off the video retrace clock (see **pfVClock**). After calling **pfExit** an application may restart IRIS Performer with **pfInit**.

User processes forked or sproced after **pfConfig** will be terminated by **pfExit**. Those forked or sproced before **pfConfig** will be sent a **SIGCLD** signal.

NOTES

Since **pfExit** deletes all shared memory arenas, any memory used by the application that was created out of IRIS Performer shared memory is now invalid.

BUGS

Currently **pfExit** returns directly to the operating system, terminating the simulation application as well. However, it does turn off video retrace CPU interrupts while exiting (see **pfVClock**).

SEE ALSO

pfConfig, pfGetSharedArena, pfMalloc, pfVClock

NAME

pfIsectFunc, pfGetIsectFunc, pfAllocIsectData, pfGetIsectData, pfPassIsectData – Set intersection callback, allocate and pass intersection data.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

void	**pfIsectFunc**(pfIsectFuncType func);
pfIsectFuncType	**pfGetIsectFunc**(void);
void *	**pfAllocIsectData**(int bytes);
void *	**pfGetIsectData**(void);
void	**pfPassIsectData**(void);

```
typedef void (*pfIsectFuncType)(void *userData);
```

DESCRIPTION

The *func* argument to **pfIsectFunc** specifies the intersection callback function. This function will be invoked by **pfFrame** and will be passed a pointer to a data buffer allocated by **pfAllocIsectData**. If a separate process is allocated for intersections by the **PFMP_FORK_ISECT** mode to **pfMultiprocess**, then **pfFrame** will cause *func* to be called in the separate process. **pfGetIsectFunc** returns the intersection callback or **NULL** if none is set.

Within the intersection callback, the user may further multiprocess intersection queries through any IRIX multiprocessing mechanism such as **fork**, **sproc**, or **m_fork**. All of these processes may call **pfNodeIsectSegs** in parallel.

When the intersection function is in a separate process, it will run asynchronously with the rest of the rendering pipeline. Specifically, if the intersection function takes more than a frame time, the rendering pipeline will not be affected and the next invocation of the intersection function will be delayed until triggered by the next **pfFrame**. Changes to the scene graph made by the application process are only propagated to the intersection process after the intersection function returns.

Any modifications made to the scene graph by a forked intersection function will not be reflected in the scene graph that is seen by any other IRIS Performer functions. To be safe, only **pfNodeIsectSegs** (which does not modify the scene graph) should be called from within the intersection function.

pfAllocIsectData returns a pointer to a chunk of shared memory of *bytes* bytes. This memory buffer may be used to communicate information between the intersection function and application. Intersection data should only be allocated once. **pfGetIsectData** returns the previously allocated intersection data.

When the intersection function is forked, **pfPassIsectData** should be used to copy the intersection data

into internal IRIS Performer memory when the next **pfFrame** is called. Once **pfFrame** is called, the application may modify data in the intersection data buffer without fear of colliding with the forked intersection function.

Example 1: Multiprocessed intersections.

```
typedef struct
{
    int         frameCount;   /* For frame stamping collisions */
    pfNode      *collidee;    /* pfNode to collide with */
    int         numCollisions;/* Number of collision vectors */
    pfSeg       *collisionVecs[MAXCOLLISIONS];
} IsectStuff;

void
isectFunc(void *data)
{
    IsectStuff istuff = (IsectStuff*) data;

    pfNodeIsectSegs(istuff->collidee, etc...);
}
     :
    pfMultiprocess(PFMP_FORK_ISECT | PFMP_APP_CULL_DRAW);
    pfConfig();
     :
    pfIsectFunc(isectFunc);
    isectData = (IsectStuff*) pfAllocIsectData(sizeof(IsectStuff));

    isectData->collidee = (pfNode*) scene;

    while (!done)
    {
        pfSync();           /* Sleep until next frame boundary */

        update_view();      /* Set view for frame N */

        isectData->frameCount = pfGetFrameCount();

        pfPassIsectData();  /* Pass intersection data to */
                            /* intersection process */

        pfFrame();          /* Trigger cull, intersection for frame N */
```

```
    app_funcs();        /* Perform application-specific functions */
    update_positions(); /* Update moving models for frame N + 1 */

    /*
     * Act on result of previous collisions and set up isectData
     * for more collisions.
     */
    update_collisions(isectData);
}
```

If **pfIsectFunc** is called before **pfConfig** and the multiprocessing mode is **PFMP_DEFAULT**, then **pfConfig** will fork the intersection process if there are enough processors. Otherwise, you must explicitly fork the intersection process by setting the **PFMP_FORK_ISECT** bit in the argument passed to **pfMultiprocess**.

SEE ALSO

pfConfig, pfMultiprocess, pfNodeIsectSegs

NAME

pfNewLOD, pfGetLODClassType, pfLODRange, pfGetLODRange, pfGetLODNumRanges, pfLODTransition, pfGetLODTransition, pfGetLODNumTransitions, pfLODCenter, pfGetLODCenter, pfLODLODState, pfGetLODLODState, pfLODLODStateIndex, pfGetLODLODStateIndex, pfEvaluateLOD – Create, modify, and query level of detail nodes.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

pfLOD * **pfNewLOD**(void);

pfType * **pfGetLODClassType**(void);

void **pfLODRange**(pfLOD *lod, int index, float range);

float **pfGetLODRange**(const pfLOD *lod, int index);

int **pfGetLODNumRanges**(const pfLOD *lod);

void **pfLODTransition**(pfLOD *lod, int index, float distance);

float **pfGetLODTransition**(const pfLOD *lod, int index);

int **pfGetLODNumTransitions**(const pfLOD *lod);

void **pfLODCenter**(pfLOD *lod, pfVec3 center);

void **pfGetLODCenter**(const pfLOD *lod, pfVec3 center);

void **pfLODLODState**(pfLOD *lod, pfLODState *ls);

void **pfGetLODLODState**(pfLOD *lod);

void **pfLODLODStateIndex**(pfLOD *lod, int index);

void **pfGetLODLODStateIndex**(pfLOD *lod);

float **pfEvaluateLOD**(pfLOD *lod, const pfChannel *chan, const pfMatrix *offset);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfLOD** is derived from the parent class **pfGroup**, so each of these member functions of class **pfGroup** are also directly usable with objects of class **pfLOD**. Casting an object of class **pfLOD** to an object of class **pfGroup** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfGroup**.

int **pfAddChild**(pfGroup *group, pfNode *child);
int **pfInsertChild**(pfGroup *group, int index, pfNode *child);
int **pfReplaceChild**(pfGroup *group, pfNode *old, pfNode *new);
int **pfRemoveChild**(pfGroup *group, pfNode* child);
int **pfSearchChild**(pfGroup *group, pfNode* child);

pfNode * **pfGetChild**(const pfGroup *group, int index);
int **pfGetNumChildren**(const pfGroup *group);
int **pfBufferAddChild**(pfGroup *group, pfNode *child);
int **pfBufferRemoveChild**(pfGroup *group, pfNode *child);

Since the class **pfGroup** is itself derived from the parent class **pfNode**, objects of class **pfLOD** can also be used with these functions designed for objects of class **pfNode**.

pfGroup * **pfGetParent**(const pfNode *node, int i);
int **pfGetNumParents**(const pfNode *node);
void **pfNodeBSphere**(pfNode *node, pfSphere *bsph, int mode);
int **pfGetNodeBSphere**(pfNode *node, pfSphere *bsph);
pfNode* **pfClone**(pfNode *node, int mode);
pfNode* **pfBufferClone**(pfNode *node, int mode, pfBuffer *buf);
int **pfFlatten**(pfNode *node, int mode);
int **pfNodeName**(pfNode *node, const char *name);
const char * **pfGetNodeName**(const pfNode *node);
pfNode* **pfFindNode**(pfNode *node, const char *pathName, pfType *type);
pfNode* **pfLookupNode**(const char *name, pfType* type);
int **pfNodeIsectSegs**(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void **pfNodeTravMask**(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint **pfGetNodeTravMask**(const pfNode *node, int which);
void **pfNodeTravFuncs**(pfNode* node, int which, pfNodeTravFuncType pre,
pfNodeTravFuncType post);
void **pfGetNodeTravFuncs**(const pfNode* node, int which, pfNodeTravFuncType *pre,
pfNodeTravFuncType *post);
void **pfNodeTravData**(pfNode *node, int which, void *data);
void * **pfGetNodeTravData**(const pfNode *node, int which);

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfLOD** can also be used with these functions designed for objects of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfLOD** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);

```
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

PARAMETERS

lod identifies a pfLOD.

DESCRIPTION

A pfLOD is a level-of-detail (LOD) node. Level-of-detail is a technique for manipulating model complexity based on image quality and rendering speed. Typically, a model is drawn in finer detail when close to the viewer (occupies large screen area) than when it is far away (occupies little screen area). In this way, costly detail is drawn only when necessary.

Additionally, IRIS Performer can adjust LODs based on rendering load. If a scene is taking too long to draw, IRIS Performer can globally modify LODs so that they are drawn coarser and render time is reduced (see **pfChanStress**).

IRIS Performer uses range-based LOD and adjusts for field-of-view and viewport pixel size. Range is computed as the distance from the pfChannel eyepoint which is drawing the scene to a point designated as the center of a pfLOD. This range is then potentially modified by pfChannel attributes (see **pfChanLODAttr**, **pfChanStress**). This range indexes the pfLOD range list to select a single child to draw.

pfLOD is derived from pfGroup so it can have children and use pfGroup API to manipulate its child list. In addition to a list of children, a pfLOD has a list of ranges which specify the transition points between levels-of-detail. **pfNewLOD** creates and returns a handle to a pfLOD. Like other pfNodes, pfLODs are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetLODClassType returns the **pfType*** for the class **pfLOD**. The **pfType*** returned by **pfGetLODClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfLOD**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfLODCenter sets the object-space point which defines the center of *lod*. *center* is affected by any transforms in the hierarchy above *lod* (see pfSCS). **pfGetLODCenter** copies the LOD center point into

center.

pfLODRange sets the value of range list element *index* to *range* which is a floating point distance specified in world coordinates. A child is selected based on the computed range (LODRange) from the eyepoint to the pfLOD center and the range list (*Ranges*) according to the following pseudocode decision test:

```
if (LODRange < Ranges[0])
    draw nothing;
else
if (LODRange >= Ranges[i] && LODRange < Ranges[i+1])
    draw Child[i];
else
if (LODRange >= Ranges[N-1] where N is length of Ranges)
    draw nothing;
```

Ranges specified by **pfLODRange** must be positive and increasing with index or results are undefined. **pfGetLODRange** returns the range with index *index* and **pfGetLODNumRanges** returns the number of ranges currently set.

Normally, LOD transitions are abrupt switches that can cause distracting visual artifacts. On hardware which supports it, IRIS Performer can blend between levels-of-detail for a smooth transition. Blended level-of-detail transitions are enabled by setting a non-zero transition range with **pfChanLODAttr**. Blending is discussed in greater depth in the **pfChanLODAttr** reference page.

pfLODTransition sets the distance over which IRIS Performer should transition or "fade" between an lod's children. The number of transitions is equal to the number of LOD children + 1. Thus Transitions[0] specifies the distance over which LOD child 0 should fade in. Transitions[1] specifies the distance over which IRIS Performer will fade between child 0 and child 1. Transitions[N] specifies the distance over which the last lod child will fade out. Note that performer will regulate the transition such that the fade will be centered based on the ranges specified by pfLODRange. It is also important to note the pfLODTransition distances should be specified such that there is no overlap between transitions or reasonable, but undefined, behavior will result. Thus, it is important to consider pfLODRanges when specifying transition distances. **pfGetLODTransition** returns the range with index *index* and **pfGetLODNumRanges** returns the number of ranges currently set.

Note that in practice IRIS Performer will multiply this transition distance by a global transition scale (this scale is set by calls to **pfChanLODAttr** with the **PFLOD_FADE** token).

The default behavior of pfLODTransition is that each transition is set to a distance of 1.0 (except Transitions[0] which is set to 0.0 by default). This makes it easy to specify a "global fade range" by controlling a **pfChanLODAttr** attribute - **PFLOD_FADE**. By setting **PFLOD_FADE** to 10.0, all transitions that have not be explicitly set will use 10.0 * 1.0 = 10.0 as their fade distance (except Transitions[0] which will not

fade at all).

Note that if one does not desire control over individual lod transitions, it is not necessary to call **pfLODTransition**.

pfLODLODState associates the given pfLOD and pfLODState. This enables the control of how a particular pfLOD responds to stress and range. **pfGetLODLODState** returns the pfLODState associated with lod if there is one or NULL if one does not exist.

pfLODLODStateIndex allows pfLODStates to be indexed on a per channel basis. *index* is an index into an pfList of pfLODStates specified via **pfChanLODStateList**. **pfGetLODLODStateIndex** returns the index currently specified for *lod* or -1 if no index has been specified.

Note that if an out of range index is specified for a given pfLOD then the pfLODState specified as the global pfLODState for that channel will be used.

pfEvaluateLOD returns the index of the child that the Performer Cull traversal would produce given a specific channel and matrix offset. The integer portion of the return value represents the selected child, while the floating point portion of the return is used to distinguish the fade ratio between two visible lods if lod fading is turned on for the given channel (see **pfChanLODAttr**). Thus an index of 1.0 would correspond to Performer's decision to draw only child one. A value of 1.25 would mean Performer would be 25% across the FADE transition between child one and child two - meaning that child one would be 75% opaque while child two would be 25% opaque. Similarly a value of 3.9 would represent child three being 10% opaque (solid) while child four was 90% opaque. The value -1.0 is returned when no children are visible. Note that negative floating point values (like -.3) mean that Performer is currently fading in child 0 and that it is 70% opaque. Thus return values will range from -1.0 <= return value < N+1 where N is the number of children for the LOD. (See **pfChannel** and **pfLODState**)

NOTES

Intersection traversals currently always intersect with an LODRange of 0. To intersect with other ranges, a pfSwitch with the same parent and children as the pfLOD can be created with the pfLOD used for drawing and the pfSwitch used for intersecting (see **pfChanTravMask**).

SEE ALSO

pfChanLODAttr, pfChanStress, pfChannel, pfGroup, pfLODState, pfLookupNode, pfDelete

NAME

pfNewLODState, pfGetLODStateClassType, pfLODStateAttr, pfGetLODStateAttr, pfLODStateName, pfGetLODStateName, pfFindLODState – Create, modify, and query level of detail state.

FUNCTION SPECIFICATION

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

pfLODState *    pfNewLODState(void);
pfType *        pfGetLODStateClassType(void);
void            pfLODStateAttr(pfLODState *ls, long attr, float val);
float           pfGetLODStateAttr(pfLODState *ls, long attr);
int             pfLODStateName(pfLODState *ls, const char *name);
const char *    pfGetLODStateName(const pfLODState *ls);
pfLODState *    pfFindLODState(const char *name);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfLODState** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfLODState**. Casting an object of class **pfLODState** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
void*   pfGetUserData(pfObject *obj);
```

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfLODState** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *        pfGetType(const void *ptr);
int             pfIsOfType(const void *ptr, pfType *type);
int             pfIsExactType(const void *ptr, pfType *type);
const char *    pfGetTypeName(const void *ptr);
int             pfRef(void *ptr);
int             pfUnref(void *ptr);
int             pfUnrefDelete(void *ptr);
int             pfGetRef(const void *ptr);
int             pfCopy(void *dst, void *src);
int             pfDelete(void *ptr);
int             pfCompare(const void *ptr1, const void *ptr2);
void            pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
```

void * **pfGetArena**(void *ptr);

PARAMETERS

ls identifies a pfLODState.

DESCRIPTION

pfLODState encapsulates a definition of how an LOD or group of LODs should respond to distance from the eyepoint and stress. Currently, there are 8 attributes that can be used to define LOD child selection and child transition distance based on a LOD's distance from the channel's viewpoint and the channel's stress (see **pfNewChan** and **pfChanStress**).

pfNewLODState creates and returns a handle to a pfLODState. Like other pfNodes, pfLODStates are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetLODStateClassType returns the **pfType*** for the class **pfLODState**. The **pfType*** returned by **pfGetLODStateClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfLODState**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfLODStateAttr and **pfGetLODStateAttr** are used to set and get the following attributes:

PFLODSTATE_RANGE_RANGESCALE, **PFLODSTATE_RANGE_RANGEOFFSET**
directly modify the geometric range used to determine the current LOD child.

PFLODSTATE_RANGE_STRESSSCALE, **PFLODSTATE_RANGE_STRESSOFFSET**
modify the way the current channel stress affects the range computation.

PFLODSTATE_TRANSITION_RANGESCALE, **PFLODSTATE_TRANSITION_RANGEOFFSET**
directly modify the transition widths set by **pfLODTransition**.

PFLODSTATE_TRANSITION_STRESSSCALE, **PFLODSTATE_TRANSITION_STRESSOFFSET**
modify the way transition widths are adjusted by the channel stress value.

These scale and offset values adjust the LOD selection process in the following way, presented in pseudocode:

```
effectiveRange =
    OverallLODScale *
    (Range * RANGE_RANGESCALE + RANGE_RANGEOFFSET) *
    (Stress * RANGE_STRESSSCALE + RANGE_STRESSOFFSET);

effectiveTransitionWidth[i] =
    OverallFadeScale *
    (Trans[i] * TRANSITION_RANGESCALE + TRANSITION_RANGEOFFSET) /
    (Stress * TRANSITION_STRESSSCALE + TRANSITION_STRESSOFFSET);
```

OverallLODScale and OverallFadeScale are the **PFLOD_SCALE** and **PFLOD_FADE** attributes set with **pfChanLODAttr**. Both are global values that affect the switching and transition ranges of all pfLODs in the scene.

The default values for all **SCALE** and **OFFSET** attributes are 1.0 and 0.0 respectively except **TRANSITION_STRESSSCALE** and **TRANSITION_STRESSOFFSET** which are 0.0 and 1.0 respectively, i.e., transition ranges are not scaled by stress by default.

A pfLODState influences a pfLOD's behavior in one of 3 ways:

1. Direct reference. A pfLOD may directly reference a pfLODState with **pfLODLODState**.
2. Indexed. A pfLOD may index a pfLODState with **pfLODLODStateIndex**. When the LOD is evaluated, the *index*th entry of the evaluating pfChannel's pfLODState table is used. A pfChannel's pfLODState table is set with (**pfChanLODStateList**). With indexed pfLODStates, different pfChannels can have different LOD behavior by using different pfLODState tables, e.g., an infrared channel may not "see" cold objects as well as a visual channel so "cold" pfLODs will index a different pfLODState in the infrared channel than in the visual channel.
3. Inherited from pfChannel. A pfLOD which does not directly reference or index a pfLODState will use the pfLODState of the evaluating pfChannel (**pfChanLODState**). This is the default pfLOD behavior.

When a pfLOD references or indexes a pfLODState, the **SCALE** and **OFFSET** parameters of the pfLODState are multiplied and added, respectively, to the corresponding **SCALE** and **OFFSET** parameters of the evaluating pfChannel's pfLODState, e.g., effective **RANGE_RANGESCALE** = pfLODState's **RANGE_RANGESCALE** * pfChannel's **RANGE_RANGESCALE**.

Multiple pfLODs may share the same pfLODState reference or index.

pfLODStateName and **pfGetLODStateName** set and get the name of a particular pfLODState while **pfFindLODState** will return the first pfLODState defined with the given name.

SEE ALSO

pfLOD, pfChannel, pfChanLODAttr, pfChanStress

NAME

pfNewLayer, pfGetLayerClassType, pfLayerMode, pfGetLayerMode, pfLayerBase, pfGetLayerBase, pfLayerDecal, pfGetLayerDecal – Create, modify, and query layer nodes for decals and coplanar polygons.

FUNCTION SPECIFICATION

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

pfLayer *   pfNewLayer(void);
pfType *    pfGetLayerClassType(void);
void        pfLayerMode(pfLayer *layer, int mode);
int         pfGetLayerMode(const pfLayer *layer);
void        pfLayerBase(pfLayer *layer, pfNode *base);
pfNode *    pfGetLayerBase(const pfLayer *layer);
void        pfLayerDecal(pfLayer *layer, pfNode *decal);
pfNode *    pfGetLayerDecal(const pfLayer *layer);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfLayer** is derived from the parent class **pfGroup**, so each of these member functions of class **pfGroup** are also directly usable with objects of class **pfLayer**. Casting an object of class **pfLayer** to an object of class **pfGroup** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfGroup**.

```
int         pfAddChild(pfGroup *group, pfNode *child);
int         pfInsertChild(pfGroup *group, int index, pfNode *child);
int         pfReplaceChild(pfGroup *group, pfNode *old, pfNode *new);
int         pfRemoveChild(pfGroup *group, pfNode* child);
int         pfSearchChild(pfGroup *group, pfNode* child);
pfNode *    pfGetChild(const pfGroup *group, int index);
int         pfGetNumChildren(const pfGroup *group);
int         pfBufferAddChild(pfGroup *group, pfNode *child);
int         pfBufferRemoveChild(pfGroup *group, pfNode *child);
```

Since the class **pfGroup** is itself derived from the parent class **pfNode**, objects of class **pfLayer** can also be used with these functions designed for objects of class **pfNode**.

```
pfGroup *   pfGetParent(const pfNode *node, int i);
int         pfGetNumParents(const pfNode *node);
```

```
void          pfNodeBSphere(pfNode *node, pfSphere *bsph, int mode);
int           pfGetNodeBSphere(pfNode *node, pfSphere *bsph);
pfNode*       pfClone(pfNode *node, int mode);
pfNode*       pfBufferClone(pfNode *node, int mode, pfBuffer *buf);
int           pfFlatten(pfNode *node, int mode);
int           pfNodeName(pfNode *node, const char *name);
const char *  pfGetNodeName(const pfNode *node);
pfNode*       pfFindNode(pfNode *node, const char *pathName, pfType *type);
pfNode*       pfLookupNode(const char *name, pfType* type);
int           pfNodeIsectSegs(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void          pfNodeTravMask(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint          pfGetNodeTravMask(const pfNode *node, int which);
void          pfNodeTravFuncs(pfNode* node, int which, pfNodeTravFuncType pre,
                  pfNodeTravFuncType post);
void          pfGetNodeTravFuncs(const pfNode* node, int which, pfNodeTravFuncType *pre,
                  pfNodeTravFuncType *post);
void          pfNodeTravData(pfNode *node, int which, void *data);
void *        pfGetNodeTravData(const pfNode *node, int which);
```

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfLayer** can also be used with these functions designed for objects of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
void*   pfGetUserData(pfObject *obj);
```

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfLayer** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *      pfGetType(const void *ptr);
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
```

void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

layer identifies a pfLayer.

DESCRIPTION

On Z-buffer based machines, numerical precision can cause distracting artifacts when rendering coplanar geometry. A pfLayer is a node derived from pfGroup that supports proper drawing of coplanar geometry on IRIS platforms.

A pfLayer can be thought of as a stack of geometry where each layer has visual priority over the geometry beneath it in the stack. An example of a 3 layer stack consists of stripes which are layered over a runway which is layered over the ground. The bottommost layer is called the "base" while the other layers are called "decals". When using certain hardware mechanisms (**PFDECAL_BASE_STENCIL**) to implement pfLayers, the "base" is special because it defines the depth values which are used to determine pfLayer visibility with respect to other scene geometry and which are written to the depth buffer.

pfNewLayer creates and returns a handle to a pfLayer. Like other pfNodes, pfLayers are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetLayerClassType returns the **pfType*** for the class **pfLayer**. The **pfType*** returned by **pfGetLayerClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfLayer**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

Since pfLayer is derived from pfGroup, pfGroup API may be used to manipulate its child list. IRIS Performer considers child 0 to be the base geometry and children 1 through N-1 to be decals. Decals are rendered in order such that decal[i+1] is drawn atop decal[i]. In other words, decal[i+1] has visual priority over decal[i] even though they are coplanar. **pfLayerBase** and **pfLayerDecal** are convenience routines for setting the base and decal children of *layer* in the common case where there is only one decal child. **pfGetLayerBase** and **pfGetLayerDecal** return the base and first child of *layer*.

The *mode* argument to **pfLayerMode** specifies which hardware mechanism to use and is one of:

PFDECAL_BASE_DISPLACE
Use slope-based polygon displacement to slightly displace the depth values of decal geometry closer to the eye so they have visual priority. Each decal is displaced more than its predecessor to properly resolve priority between decals. The maximum number of decals is 8.

PFDECAL_BASE_DISPLACE | PFDECAL_LAYER_OFFSET
Use slope-based polygon displacement to slightly displace the depth values of decal geometry closer to the eye so they have visual priority. In addition, decal geometry is offset a constant amount to eliminate anomalies caused by geometry which is nearly perpendicular to the view. Each decal is displaced and offset more than its predecessor to properly resolve priority between decals. The maximum number of decals is 8.

PFDECAL_BASE_STENCIL
Use the stencil-buffer logic to determine visibility of decal geometry. There is no limit to the number of decals.

PFDECAL_BASE_FAST
Use a decaling mechanism appropriate to the hardware that produces the fastest, but not necessarily the highest quality, decaling.

PFDECAL_BASE_HIGH_QUALITY
Use a decaling mechanism appropriate to the hardware that produces the highest quality, but not necessarily the fastest, decaling.

The default layer mode is **PFDECAL_BASE_FAST**. **pfGetLayerMode** returns the mode of *layer*.

The different pfLayer modes offer quality-feature tradeoffs listed in the table below:

	DISPLACE	**STENCIL**	**(DISPLACE | OFFSET)**
Quality	medium	high	high
Sorting	enabled	disabled	enabled
Coplanarity	not required	required	not required
Multipass	ok	not ok	ok
Containment	not required	required	not required

The **STENCIL** mechanism offers the best image quality but at a performance cost since the base and layer geometry must be rendered in order, obviating any benefits of sorting by graphics state offered by **pfChanBinSort**. When multisampling on RealityEngine, this mechanism also significantly reduces pixel fill performance. An additional constraint is that **STENCIL**ed layers must be coplanar or decal geometry may incorrectly show through base geometry. A subtle but important issue with **STENCIL**ed layers is that they are unsuitable for multipass renderings (projected textures) since multiple surfaces are visible at a given pixel. For proper results, each layer in the "stack" must be completely contained within the boundaries of the base geometry.

The **DISPLACE** mechanism offers the best performance since layers can be sorted by graphics state, because the displace call itself is usually faster than other mode changes, and because there is no pixel fill rate penalty when it is in use. However, in IRIS GL the displace mechanism is only slope-based, so when geometry becomes nearly perpendicular to the view, i.e., has little or no slope, the displacement is too little to conclusively determine visibility. To solve this problem, the **OFFSET** mechanism adds a constant

offset to the decal geometry. This mode can be very expensive (RealityEngine) so when using it the database should be sorted with **PFSTATE_DECAL** as the first sorting key (see **pfChanBinSort**). Both **DISPLACE** mechanisms do not require that geometry within a single layer be coplanar and also produce a single visible surface at each pixel for multipass renderings. The main disadvantage is that decal geometry may incorrectly poke through other geometry due to the displacement of the decal geometry. Another disadvantage is that the maximum number of decals is 8.

The performance differences between **STENCIL** and **DISPLACE** modes are hardware-dependent so some experimentation and benchmarking is required to determine the most suitable method for your application.

NOTES

Using **PFDECAL_BASE_STENCIL** for pfLayer nodes requires several steps for proper operation. First, the graphics hardware must support stencil plane rendering. Secondly, the graphics context must be configured with at least one stencil plane, and the lowest order bit of the allocated stencil planes be reserved for IRIS Performer use. **pfInitGfx** configures the graphics context in just this way.

The use of displacements for rendering coplanar geometry can cause visual artifacts such as decals "Z fighting" or "flimmering" when viewed perpendicularly, and the "punching through" of decals that should mask base geometry when both are viewed obliquely. The former artifact can be eliminated by specifying **PFDECAL_BASE_DISPLACE | PFDECAL_LAYER_OFFSET** as the layer mode. If unacceptable artifacts still persist, the database should be modified to eliminate the need for coplanar rendering or **PFDECAL_BASE_HIGH_QUALITY** should be used.

When using **PFDECAL_LAYER_OFFSET**, the minimum depth buffer range set with **lsetdepth** must be incremented an extra 1024 * max layers so the negative displacement of the layers does not wrap. **pfInitGfx** does this automatically.

BUGS

IRIS Performer properly renders coplanar geometry only on machines that have a hardware stencil buffer allocated or which support displaced polygon rendering.

SEE ALSO

pfChanBinSort, pfDecal, pfGroup, pfInitGfx, pfLookupNode, pfNode, pfDelete

NAME

pfNewLPoint, pfGetLPointClassType, pfGetNumLPoints, pfLPointSize, pfGetLPointSize, pfLPointColor, pfGetLPointColor, pfLPointRot, pfGetLPointRot, pfLPointShape, pfGetLPointShape, pfLPointFogScale, pfGetLPointFogScale, pfLPointPos, pfGetLPointPos, pfGetLPointGSet – Set and get pfLightPoint size, color, shape, rotation and position.

FUNCTION SPECIFICATION

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

pfLightPoint *  pfNewLPoint(int num);
pfType *        pfGetLPointClassType(void);
int             pfGetNumLPoints(const pfLightPoint *lpoint);
void            pfLPointSize(pfLightPoint *lpoint, float size);
float           pfGetLPointSize(const pfLightPoint *lpoint);
void            pfLPointColor(pfLightPoint *lpoint, int index, pfVec4 clr);
void            pfGetLPointColor(const pfLightPoint *lpoint, int index, pfVec4 clr);
void            pfLPointRot(pfLightPoint *lpoint, float azim, float elev, float roll);
void            pfGetLPointRot(const pfLightPoint *lpoint, float *azim, float *elev, float *roll);
void            pfLPointShape(pfLightPoint *lpoint, int dir, float henv, float venv, float falloff);
void            pfGetLPointShape(const pfLightPoint *lpoint, int *dir, float *henv, float *venv,
                    float *falloff);
void            pfLPointFogScale(pfLightPoint *lpoint, float onsetScale, float opaqueScale);
void            pfGetLPointFogScale(const pfLightPoint *lpoint, float *onsetScale, float *opaqueScale);
void            pfLPointPos(pfLightPoint *lpoint, int index, pfVec3 pos);
void            pfGetLPointPos(const pfLightPoint *lpoint, int index, pfVec3 pos);
pfGeoSet*       pfGetLPointGSet(const pfLightPoint *lpoint);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfLightPoint** is derived from the parent class **pfNode**, so each of these member functions of class **pfNode** are also directly usable with objects of class **pfLightPoint**. Casting an object of class **pfLightPoint** to an object of class **pfNode** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfNode**.

```
pfGroup *   pfGetParent(const pfNode *node, int i);
int         pfGetNumParents(const pfNode *node);
void        pfNodeBSphere(pfNode *node, pfSphere *bsph, int mode);
```

int **pfGetNodeBSphere**(pfNode *node, pfSphere *bsph);
pfNode* **pfClone**(pfNode *node, int mode);
pfNode* **pfBufferClone**(pfNode *node, int mode, pfBuffer *buf);
int **pfFlatten**(pfNode *node, int mode);
int **pfNodeName**(pfNode *node, const char *name);
const char * **pfGetNodeName**(const pfNode *node);
pfNode* **pfFindNode**(pfNode *node, const char *pathName, pfType *type);
pfNode* **pfLookupNode**(const char *name, pfType* type);
int **pfNodeIsectSegs**(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void **pfNodeTravMask**(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint **pfGetNodeTravMask**(const pfNode *node, int which);
void **pfNodeTravFuncs**(pfNode* node, int which, pfNodeTravFuncType pre,
pfNodeTravFuncType post);
void **pfGetNodeTravFuncs**(const pfNode* node, int which, pfNodeTravFuncType *pre,
pfNodeTravFuncType *post);
void **pfNodeTravData**(pfNode *node, int which, void *data);
void * **pfGetNodeTravData**(const pfNode *node, int which);

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfLightPoint** can also be used with these functions designed for objects of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfLightPoint** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);

void * **pfGetArena**(void *ptr);

PARAMETERS

lpoint identifies a pfLightPoint.

DESCRIPTION

pfLightPoint is now obsoleted in favor of the **libpr** primitive pfLPointState. **pfGetLPointGSet** returns the underlying pfGeoSet of *lpoint* from which the pfLPointState can be found:

```
gset = pfGetLPointGSet(lpoint);
gstate = pfGetGSetGState(gset);
lpstate = pfGetGStateAttr(gstate, PFSTATE_LPOINTSTATE);
```

A pfLightPoint is a pfNode that contains one or more light points. The light point node is quite different from a pfLightSource; it is visible as one or more self-illuminated small points but these points do not illuminate surrounding objects. In contrast to this, a pfLightSource does illuminate scene contents but is itself not a visible object. All the light points in a pfLightPoint node share all their attributes except point location and color.

pfNewLPoint creates and returns a handle to a pfLightPoint. Like other pfNodes, pfLightPoints are always allocated from shared memory and can be deleted using **pfDelete**. *num* specifies the maximum number of individual light points the node may contain. The function **pfGetNumLPoints** returns this maximum number of light points that the pfLightPoint node *lpoint* can hold. This is the value set when the light point node was created using **pfNewLPoint** and is the size of the internal position and color arrays used to represent the light points.

pfGetLPointClassType returns the **pfType*** for the class **pfLightPoint**. The **pfType*** returned by **pfGetLPointClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfLightPoint**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfLPointSize sets the screen size of each point of light in *lpoint*. *size* is specified in pixels and is used as the argument to **pntsizef**. Whenever possible, antialiased points are used but the actual representation of a light point depends on the hardware being used. See the **pntsizef** man page for a description of available light point sizes on IRIS hardware. **pfGetLPointSize** returns the size of *lpoint*.

pfLPointColor sets the color of light point *index* in *lpoint* to *clr*. The actual color displayed depends on light point direction, shape, position, and fog. *clr* specifies red, green, blue and alpha in the range 0.0 to 1.0. A pfLightPoint is turned off with an alpha of 0.0 since it will be rendered as completely transparent. **pfGetLPointColor** copies the *index*th color into *clr*.

pfLPointRot is used for directional lights. The direction of all light points in *lpoint* is the positive Y axis, rotated about the X axis by *elev* then rotated about the Z axis by *azim*. *roll* only affects the light envelope as described below. The direction vector is rotated by any transformations (see pfSCS, pfDCS) above *lpoint* in the hierarchy.

pfGetLPointRot copies *lpoint*'s rotation into *azim*, *elev*, and *roll*.

pfLPointShape describes the intensity distribution of a light point about its direction vector. *dir* is a symbolic token:

PFLP_OMNIDIRECTIONAL
lpoint will be drawn as omnidirectional light points. Light distribution is equal in all directions. All other arguments are ignored.

PFLP_UNIDIRECTIONAL
lpoint will be drawn as unidirectional point lights. Light distribution is an elliptical cone centered about the light direction vector.

PFLP_BIDIRECTIONAL
lpoint will be drawn as bidirectional light points. Light distribution is two elliptical cones centered about the positive and negative light direction vectors.

henv and *venv* are total angles (not half-angles) in degrees which specify the horizontal and vertical envelopes about the direction vector. An envelope is a symmetric angular spread in a specific plane about the light direction vector. The default direction is along the positive Y axis so the horizontal envelope is in the X plane and the vertical in the Z plane. Both direction and envelopes are rotated by the **pfLPointRot** and any inherited transformations. The default envelope angles are 360.0 degrees which is equivalent to an omnidirectional light.

When the vector from the eyepoint to the light position is outside a light's envelope, the light point is not displayed. If within, the intensity of the light point is computed based on the location of the eye within the elliptical cone. Intensity ranges from 1.0 when the eye lies on the light direction vector to 0.0 on the edge of the cone. *falloff* is an exponent which modifies the intensity. A value of 0 indicates that there is no falloff and values > 0 increase the falloff rate. The default *falloff* is 4. As intensity decreases, the light point's transparency increases.

pfGetLPointShape copies *lpoint*'s shape parameters into *dir*, *henv*, *venv*, and *falloff*.

In general, the real world intensity of emissive light points is much greater than that of reflective surfaces. Consequently, when fog is active, light points should be more visible through the fog. **pfLPointFogScale** sets the fog range scale factors that affects all light points in *lpoint*. *onsetScale* and *opaqueScale* multiply the onset and opaque ranges (**pfFogRange**) of the currently active fog. Thus if the scale factors are greater than 1.0, the light points will be more visible through fog than reflective surfaces. The default fog scale factors are both 4.0. **pfGetLPointFogScale** copies the fog scale factors of *lpoint* into *onsetScale* and *opaqueScale*.

pfLPointPos sets the position of light point with index *index* to *pos*. *index* is clamped to the range [0, *num*-1]. All positions are transformed by any inherited transformations. The final position and orientation of a light point *i* is transformed by **R * T[index] * M** where **R** is a rotation matrix defined by **pfLPointRot**, **T[i]** is the position of light point *i*, and **M** is the transformation inherited by *lpoint* from its hierarchy.

pfGetLPointPos copies the *index*th position into *pos*.

NOTES

Light point processing in IRIS Performer has been subsumed by the new **pfLPointState** mechanism, which is both more powerful and more efficient. Application developers are encouraged to transition to these new light point facilities.

pfLightPoint nodes, unlike **pfLPointState** GeoSets, do not provide size or intensity modulation based on distance to the viewer and the viewport size. Also, directional lights are significantly more expensive to cull than omnidirectional lights.

Falloff distribution is cosine(incidence angle) ^ falloff.

When sorting is enabled (see **pfChanTravMode** and **PFCULL_SORT**), light points are drawn after opaque geometry unless the pfLightPoint node has a pre-draw or post-draw callback (see **pfNodeTravFuncs**).

SEE ALSO

pfLookupNode, pfNode, pfLPointState

NAME

pfNewLSource, pfGetLSourceClassType, pfLSourceAmbient, pfGetLSourceAmbient, pfLSourceColor, pfGetLSourceColor, pfLSourceAtten, pfGetLSourceAtten, pfSpotLSourceDir, pfGetSpotLSourceDir, pfSpotLSourceCone, pfGetSpotLSourceCone, pfLSourcePos, pfGetLSourcePos, pfLSourceOn, pfLSourceOff, pfIsLSourceOn, pfLSourceMode, pfGetLSourceMode, pfLSourceVal, pfGetLSourceVal, pfLSourceAttr, pfGetLSourceAttr – Create pfLightSource, specify pfLightSource properties.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

pfLightSource * **pfNewLSource**(void);

pfType * **pfGetLSourceClassType**(void);

void **pfLSourceAmbient**(pfLightSource* lsource, float r, float g, float b);

void **pfGetLSourceAmbient**(pfLightSource* lsource, float* r, float* g, float* b);

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

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

void **pfLSourceAtten**(pfLightSource* light, float constant, float linear, float quadratic);

void **pfGetLSourceAtten**(pfLightSource* light, float *constant, float *linear, float *quadratic);

void **pfSpotLSourceDir**(pfLightSource* lsource, float x, float y, float z);

void **pfGetSpotLSourceDir**(pfLightSource* lsource, float* x, float* y, float* z);

void **pfSpotLSourceCone**(pfLightSource* lsource, float f1, float f2);

void **pfGetSpotLSourceCone**(pfLightSource* lsource, float* f1, float* f2);

void **pfLSourcePos**(pfLightSource* lsource, float x, float y, float z, float w);

void **pfGetLSourcePos**(pfLightSource* lsource, float* x, float* y, float* z, float* w);

void **pfLSourceOn**(pfLightSource* lsource);

void **pfLSourceOff**(pfLightSource* lsource);

int **pfIsLSourceOn**(pfLightSource* lsource);

void **pfLSourceMode**(pfLightSource *lsource, int mode, int val);

int **pfGetLSourceMode**(const pfLightSource *lsource, int mode);

void **pfLSourceVal**(pfLightSource *lsource, int mode, float val);

float **pfGetLSourceVal**(const pfLightSource *lsource, int mode);

void **pfLSourceAttr**(pfLightSource *lsource, int attr, void *obj);

void* **pfGetLSourceAttr**(const pfLightSource *lsource, int attr);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfLightSource** is derived from the parent class **pfNode**, so each of these member functions of class **pfNode** are also directly usable with objects of class **pfLightSource**. Casting an object of class **pfLightSource** to an object of class **pfNode** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfNode**.

pfGroup * **pfGetParent**(const pfNode *node, int i);
int **pfGetNumParents**(const pfNode *node);
void **pfNodeBSphere**(pfNode *node, pfSphere *bsph, int mode);
int **pfGetNodeBSphere**(pfNode *node, pfSphere *bsph);
pfNode* **pfClone**(pfNode *node, int mode);
pfNode* **pfBufferClone**(pfNode *node, int mode, pfBuffer *buf);
int **pfFlatten**(pfNode *node, int mode);
int **pfNodeName**(pfNode *node, const char *name);
const char * **pfGetNodeName**(const pfNode *node);
pfNode* **pfFindNode**(pfNode *node, const char *pathName, pfType *type);
pfNode* **pfLookupNode**(const char *name, pfType* type);
int **pfNodeIsectSegs**(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void **pfNodeTravMask**(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint **pfGetNodeTravMask**(const pfNode *node, int which);
void **pfNodeTravFuncs**(pfNode* node, int which, pfNodeTravFuncType pre,
pfNodeTravFuncType post);
void **pfGetNodeTravFuncs**(const pfNode* node, int which, pfNodeTravFuncType *pre,
pfNodeTravFuncType *post);
void **pfNodeTravData**(pfNode *node, int which, void *data);
void * **pfGetNodeTravData**(const pfNode *node, int which);

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfLightSource** can also be used with these functions designed for objects of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfLightSource** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);

```
int             pfIsExactType(const void *ptr, pfType *type);
const char *    pfGetTypeName(const void *ptr);
int             pfRef(void *ptr);
int             pfUnref(void *ptr);
int             pfUnrefDelete(void *ptr);
int             pfGetRef(const void *ptr);
int             pfCopy(void *dst, void *src);
int             pfDelete(void *ptr);
int             pfCompare(const void *ptr1, const void *ptr2);
void            pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *          pfGetArena(void *ptr);
```

PARAMETERS

lsource identifies a pfLightSource.

DESCRIPTION

A pfLightSource is a pfNode which can illuminate geometry in a pfScene. In addition, pfLightSource supports a technique known as "projected texturing" which can simulate high quality, real time spotlights and shadows on certain graphics hardware.

pfNewLSource creates and returns a handle to a pfLightSource. Like other pfNodes, pfLightSources are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetLSourceClassType returns the **pfType*** for the class **pfLightSource**. The **pfType*** returned by **pfGetLSourceClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfLightSource**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

Most pfLightSource routines are borrowed from pfLight. These routines have the identical function as the pfLight routines but operate on a pfLightSource rather than a pfLight. The routine correspondence is listed in the following table.

pfLightSource routine	pfLight routine
pfLSourceAmbient	pfLightAmbient
pfGetLSourceAmbient	pfGetLightAmbient
pfLSourceColor	pfLightColor
pfLSourceAtten	pfLightAtten
pfGetLSourceColor	pfGetLightColor
pfLSourcePos	pfLightPos
pfGetLSourcePos	pfGetLightPos
pfSpotLSourceCone	pfSpotLightCone

pfGetSpotLSourceCone	pfGetSpotLightCone
pfSpotLSourceDir	pfSpotLightDir
pfGetSpotLSourceDir	pfGetSpotLightDir
pfLSourceOn	pfLightOn
pfLSourceOff	pfLightOff
pfIsLSourceOn	pfIsLightOn

The reader is referred to the pfLight man page for details on the routine description.

When enabled by **pfLSourceOn**, a pfLightSource influences all geometry that is in the same pfScene if it is not culled during the cull traversal. Its position in the hierarchy does not affect its area of influence. A pfLightSource is enabled by default and is explicitly disabled with **pfLSourceOff**.

pfLightSources are processed somewhat differently than other nodes. If the **PFCULL_IGNORE_LSOURCES** mode is not enabled by **pfChanTravMode**, the cull stage will begin with a special traversal of all paths which lead from the current pfScene to pfLightSources before it traverses the pfScene geometry. This initial traversal is no different from the ordinary cull traversal except that the traversal order is path-directed rather than an in-order traversal. Specifically, all switches (pfSwitch, pfLOD, pfSequence) and transformations (pfSCS, pfDCS) will affect the traversal. Note that nodes that lie on paths to pfLightSource nodes will be traversed multiple times; specifically, any cull or draw callbacks (**pfNodeTravFuncs**) will be invoked multiple times.

pfLightSources are culled to the viewing frustum only if they have been assigned a non-null bounding volume (**pfNodeBSphere**). If a pfLightSource has a null bounding volume (radius < 0) then it is not culled and has global effect over its pfScene. By default pfLightSources have null bounding volumes. After the pfLightSource traversal comes the database traversal which (usually) visually culls the current pfScene and ignores pfLightSources.

A pfLightSource inherits the current transformation from any pfSCSes and pfDCSes above it in the hierarchy. This matrix transforms the light source's position and direction depending on the light's type, i.e.- if it is a local, infinite, or spotlight.

All hardware lights corresponding to pfLightSources in a pfScene will be properly configured before the pfChannel's draw callback is invoked (see **pfChanTravFunc**). Consequently, all geometry rendered in the pfChannel draw callback will be illuminated by the pfScene's light sources. However, any draw callback assigned to the pfLightSource node by **pfNodeTravFuncs** will be invoked before the pfChannel draw callback is invoked so that anything drawn in the node callback will be obscured if the channel viewport is cleared (see **pfClearChan**). Example 1: Adding a pfLightSource to a pfScene.

```
sun = pfNewLSource();

/* Set slightly yellow color */
pfLSourceColor(sun, PFLT_DIFFUSE, 1.0f, 1.0f, .8f);
```

```
/* Set a high ambient level */
pfLSourceColor(sun, PFLT_AMBIENT, .4f, .4f, .3f);

/* Time of day is high noon */
pfLSourcePos(sun, 0.0f, 1.0f, 0.0f, 0.0f);

pfAddChild(scene, sun);
```

A pfLightSource supports 3 different lighting mechanisms as listed in the following table:

Lighting Method	Normals Used	Texture Required	Effects Are Per-?	Shadows	Extra Draw Pass(es)
pfLight	Yes	No	Vertex	No	None
PROJTEX	No	Yes	Pixel	No	+(0-1)
SHADOW	No	Auto	Pixel	Yes	+(0-2)

The normal use of a pfLightSource is as a pfLight which computes lighting at geometry vertices, taking into account the surface curvature as represented by geometry normals. This kind of lighting offers the highest performance but does not produce per-pixel effects or shadows. Lighting using projected textures, referred to as PROJTEX, produces high quality spotlights since the spotlight boundary is computed on a per-pixel, rather than a per-vertex basis as it is with pfLight. However, PROJTEX lighting does not take surface normals into account, requires hardware texture mapping for decent performance, and requires that textured geometry be rendered twice, once with their normal texture and once with the projected texture. SHADOW lighting is similar to PROJTEX but adds shadows at the cost of an additional rendering pass. In this case a special texture map, called a shadow map, is automatically generated by the pfLightSource and then projected onto the scene. Typically, pfLight-type lighting is used in conjunction with PROJTEX or SHADOW so that lighting is a function of both per-pixel projected texturing and per-vertex surface curvature.

SHADOW and PROJTEX lighting are separately enabled and disabled with the **PFLS_SHADOW_ENABLE** and **PFLS_PROJTEX_ENABLE** tokens to **pfLSourceMode**. *val* should be either **PF_ON** or **PF_OFF**. When either is enabled, pfChannels rendering the pfLightSource's scene automatically enter "multipass mode" since multiple renderings of the scene are usually required.

pfChanTravMode with the **PFTRAV_MULTIPASS** traversal token offers some control over the multiple renderings of the scene. The **PFMPASS_GLOBAL_AMBIENT** bit indicates that the alpha bitplanes of the pfChannel's viewport contain the ambient intensity of the scene. Note that the pfChannel will not clear the viewport alpha to this intensity but expects it to have already been properly cleared. If using a pfEarthSky to clear the viewport, you can specify the ambient alpha with **pfESkyColor**. Global ambient is not required and does have some extra cost. It is not particularly useful for PROJTEX lighting since ambient intensity can be easily incorporated in the projected texture (instead of black, just use gray

outside the spotlight) but is useful for SHADOWS which otherwise would be completely black.

By default, emissive surfaces (including light points) are attenuated by PROJTEX and SHADOW lighting which is not correct since emissive surfaces should shine even if in shadow or outside the cone of a projected spotlight. If a scene has emissive surfaces, set the **PFMPASS_EMISSIVE_PASS** bit in the **PFTRAV_MULTIPASS** mode and the emissive surfaces will be properly rendered. Note that the emissive rendering pass is not a full pass - rather it is a pass of only the emissive surfaces.

In situations where the scene is entirely non-textured, **PFMPASS_NONTEX_SCENE** can be specified as part of the **PFTRAV_MULTIPASS** traversal mode of a pfChannel. In this case a complete rendering pass is eliminated so that the total number of rendering passes is **numProjLights** + 2 * **numNonFrozenShadowLights**.

PROJTEX lighting requires that a pfTexture be specified with the **PFLS_PROJ_TEX** token to **pfLSourceAttr**. *obj* should be an intensity-alpha (2-component) pfTexture* with identical intensity and alpha components. If *lsource* is the only pfLightSource in the scene using PROJTEX lighting, the texture may be a full-color, 4-component texture.

SHADOW lighting does not require a pfTexture, rather one is automatically created and configured by the pfLightSource. The size of the texture(shadow) map may be specified with the **PFLS_SHADOW_SIZE** token to **pfLSourceVal**. *val* is then the square size of the texture map. The size of the shadow map greatly influences the quality and performance of SHADOW lighting. Large shadow map sizes increase quality but decrease performance. The default shadow map size is 256. SHADOW lighting requires that the viewport of each pfChannel which renders the pfLightSource's scene be at least as big as the shadow map. Otherwise, shadows will be clipped and visual anomalies will occur.

Both SHADOW and PROJTEX lighting require that a pfFrustum be specified with the **PFLS_PROJ_FRUSTUM** token to **pfLSourceAttr**. *obj* defines the projection of the texture (shadow) map and should be a nominal, i.e., non-transformed pfFrustum*. For SHADOW lighting, the field-of-view and near and far clipping planes should bracket the scene to be shadowed as tightly as possible for best results. A sloppy fit of pfFrustum to scene will result in blocky, poor-quality shadows.

By default, SHADOW lighting requires that the scene be rendered from the point of view of the pfLightSource to produce a shadow map. By default, pfChannels automatically do this for each SHADOW pfLightSource in their scene. However, a new shadow map is only required if the pfLightSource or objects in the scene change. In the special case where the pfLightSource and scene are totally static (e.g., the sun illuminating a sleepy town), the shadow map need not be recomputed. In this case **pfLSourceMode(PFLS_FREEZE_SHADOWS, PF_ON)** will disable the automatic recomputation of the shadow map, increasing performance.

For best results, SHADOW lighting requires that the scene be slightly displaced in depth when rendering the shadow map. This reduces artifacts such as "self-shadowing". The **PFLS_SHADOW_DISPLACE_SCALE** and **PFLS_SHADOW_DISPLACE_OFFSET** tokens to

pfLSourceVal specify displacement values. The default values are 1.0 and 256.0 respectively but experimentation is required for best results (both values should be positive).

For pfLightSources which are near the eye, a pfFog can be used to simulate range-attenuation of the light. Range-attenuation is enabled with the **PFLS_FOG_ENABLE** token to **pfLSourceMode** and by specifying a pfFog with the **PFLS_PROJ_FOG** token to **pfLSourceAttr**. The pfFog color should be the ambient color of the projected texture. Only a single range-attenuated projected pfLightSource is supported for a given pfChannel.

A pfLightSource's intensity is set with the **PFLS_INTENSITY** token to **pfLSourceVal**. *val* simply scales the color(s) of all 3 lighting types: pfLight, PROJTEX, SHADOW. A scene containing multiple, full-intensity pfLightSources can be easily saturated so setting pfLightSource intensities is a simple way to "normalize" lighting within a scene. For example, when using 3 pfLightSources to illuminate a scene, an intensity of .33 would be reasonable. Example 2: Range-attenuated, projected texture lighting for landing light

```
pfLightSource        *spot;
pfTexture *spotTex;
pfFrustum *spotFrust;
pfFog                *spotFog;
pfDCS                *spotDCS;
pfChannel *chan;
pfEarthSky           *esky;

// Create and load 2-component spotlight
spotTex = pfNewTex(arena);
pfLoadTexFile(spotTex, "spot.inta");

// Create and configure projected texture frustum
spotFrust = pfNewFrust(arena);
pfMakeSimpleFrust(spotFrust, 60.0f);
pfFrustNearFar(spotFrust, 1.0f, 100.0f);

// Create and configure range-attenuation fog model
spotFog = pfNewFog(arena);
pfFogColor(spotFog, 0.1f, 0.1f, 0.1f);
pfFogRange(spotFog, 0.0f, 100.0f);

// Create and configure projected texture light source
spot = pfNewLSource();
pfLSourceAttr(spot, PFLS_PROJ_TEX, spotTex);
pfLSourceAttr(spot, PFLS_PROJ_FRUST, spotFrust);
pfLSourceAttr(spot, PFLS_PROJ_FOG, spotFog);
```

```
pfLSourceMode(spot, PFLS_PROJTEX_ENABLE, 1);

// Set spotDCS to viewing matrix to move light around with eye
spotDCS = pfNewDCS();
pfAddChild(spotDCS, spot);
pfAddChild(scene, spotDCS);

// Enable emissive pass since scene has emissive surfaces
pfChanTravMode(chan, PFTRAV_MULTIPASS,
                PFMPASS_EMISSIVE_PASS|PFMPASS_GLOBAL_AMBIENT);

// Set ambient intensity to .1
pfESkyColor(esky, PFES_CLEAR, r, g, b, .1f);
pfChanESky(chan, esky);
```

Example 3: Multiple, shadow-casting, colored pfLightSources

```
pfLightSource        *shad0, *shad1;
pfDCS                *shadDCS0, *shadDCS1;
pfFrustum *shadFrust;
pfChannel *chan;
pfEarthSky           *esky;

// Create and configure shadow frustum
shadFrust = pfNewFrust(arena);
pfMakeSimpleFrust(shadFrust, 60.0f);
pfFrustNearFar(shadFrust, 1.0f, 100.0f);

// Create and configure shadow casting light sources
shad0 = pfNewLSource();
pfLSourceMode(shad0, PFLS_SHADOW_ENABLE, 1);
pfLSourceAttr(shad0, PFLS_PROJ_FRUST, shadFrust);
pfLSourceColor(shad0, PFLT_DIFFUSE, 1.0f, 0.0f, 0.0f);
pfLSourceVal(shad0, PFLS_INTENSITY, .5f);

shad1 = pfNewLSource();
pfLSourceMode(shad1, PFLS_SHADOW_ENABLE, 1);
pfLSourceAttr(shad1, PFLS_PROJ_FRUST, shadFrust);
pfLSourceColor(shad1, PFLT_DIFFUSE, 0.0f, 0.0f, 1.0f);
pfLSourceVal(shad1, PFLS_INTENSITY, .5f);

// Set DCSes to move lights around
```

```
shadDCS0 = pfNewDCS();
pfAddChild(shadDCS0, shad0);
pfAddChild(scene, shadDCS0);

shadDCS1 = pfNewDCS();
pfAddChild(shadDCS1, shad1);
pfAddChild(scene, shadDCS1);

// Enable global ambient
pfChanTravMode(chan, PFTRAV_MULTIPASS, PFMPASS_GLOBAL_AMBIENT);

// Set ambient intensity to .1
pfESkyColor(esky, PFES_CLEAR, r, g, b, .1f);
pfChanESky(chan, esky);
```

NOTES

To respect the limited number of active light sources allowed by graphics library implementations, IRIS Performer supports at most **PF_MAX_LIGHTS** active light sources.

If you want light sources to affect only portions of the scene, then set one or more pfLights on the pfGeoStates which are attached to the pfGeoSets that you wish to illuminate (see **pfGStateAttr** and **PFSTATE_LIGHTS** for further details).

Shadows are supported only by RealityEngine when using IRIS GL.

PROJTEX and SHADOW lighting on RealityEngine require local lighting for proper effects (-**pfLModelLocal**).

SHADOW lighting on RealityEngine requires the depth buffer to be configured with 32 bits (**zbsize()**). Note that it is legal to have multisample buffers allocated in addition, the only requirement is that the non-multisampled depth buffer be 32 bits. Also note that on RealityEngine, a 32-bit depth buffer requires 12-bit color.

On RealityEngine, shadows and projected textures are not clipped or properly computed behind the pfLightSource. Instead, geometry behind the pfLightSource will be textured randomly. The only workaround is to ensure that all geometry behind the pfLightSource is not visible to the pfChannel.

Local lighting results in improper shading of flat-shaded triangle and line strips (-**PFGS_FLAT_TRISTRIPS**, **PFGS_LINE_TRISTRIPS**) which often manifests itself as "faceting" of planar polygons. The only solution is either to use infinite lighting or not use FLAT primitives. Note that when using the IRIS Performer triangle meshing routine, **pfdMeshGSet**, the construction of non-FLAT strips is

easily enforced with **pfdMesherMode(PFDMESH_LOCAL_LIGHTING, 1)**.

SEE ALSO

pfChanTravFunc, pfChanTravMode, pfNode, pfSCS, pfDCS, pfGeoSet, pfGeoState, pfLight, pfDelete

NAME

pfNewMorph, pfGetMorphClassType, pfMorphAttr, pfMorphWeights, pfGetMorphWeights, pfGetMorphNumAttrs, pfGetMorphSrc, pfGetMorphNumSrcs, pfGetMorphDst, pfEvaluateMorph – Create, modify, and query a pfMorph node.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

pfMorph * **pfNewMorph**(void);

pfType * **pfGetMorphClassType**(void);

int **pfMorphAttr**(pfMorph *morph, int index, int floatsPerElt, int nelts, void *dst, int nsrcs, float *alist[], ushort *ilist[], int nlist[]);

int **pfMorphWeights**(pfMorph *morph, int index, float *weights);

int **pfGetMorphWeights**(const pfMorph *morph, int index, float *weights);

int **pfGetMorphNumAttrs**(const pfMorph *morph);

int **pfGetMorphSrc**(const pfMorph *morph, int index, int src, float **alist, ushort **ilist, int *nlist);

int **pfGetMorphNumSrcs**(const pfMorph *morph, int index);

void * **pfGetMorphDst**(const pfMorph *morph, int index);

void **pfEvaluateMorph**(pfMorph *morph);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfMorph** is derived from the parent class **pfGroup**, so each of these member functions of class **pfGroup** are also directly usable with objects of class **pfMorph**. Casting an object of class **pfMorph** to an object of class **pfGroup** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfGroup**.

int **pfAddChild**(pfGroup *group, pfNode *child);
int **pfInsertChild**(pfGroup *group, int index, pfNode *child);
int **pfReplaceChild**(pfGroup *group, pfNode *old, pfNode *new);
int **pfRemoveChild**(pfGroup *group, pfNode* child);
int **pfSearchChild**(pfGroup *group, pfNode* child);
pfNode * **pfGetChild**(const pfGroup *group, int index);
int **pfGetNumChildren**(const pfGroup *group);
int **pfBufferAddChild**(pfGroup *group, pfNode *child);
int **pfBufferRemoveChild**(pfGroup *group, pfNode *child);

Since the class **pfGroup** is itself derived from the parent class **pfNode**, objects of class **pfMorph** can also be used with these functions designed for objects of class **pfNode**.

```
pfGroup *     pfGetParent(const pfNode *node, int i);
int           pfGetNumParents(const pfNode *node);
void          pfNodeBSphere(pfNode *node, pfSphere *bsph, int mode);
int           pfGetNodeBSphere(pfNode *node, pfSphere *bsph);
pfNode*       pfClone(pfNode *node, int mode);
pfNode*       pfBufferClone(pfNode *node, int mode, pfBuffer *buf);
int           pfFlatten(pfNode *node, int mode);
int           pfNodeName(pfNode *node, const char *name);
const char *  pfGetNodeName(const pfNode *node);
pfNode*       pfFindNode(pfNode *node, const char *pathName, pfType *type);
pfNode*       pfLookupNode(const char *name, pfType* type);
int           pfNodeIsectSegs(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void          pfNodeTravMask(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint          pfGetNodeTravMask(const pfNode *node, int which);
void          pfNodeTravFuncs(pfNode* node, int which, pfNodeTravFuncType pre,
                  pfNodeTravFuncType post);
void          pfGetNodeTravFuncs(const pfNode* node, int which, pfNodeTravFuncType *pre,
                  pfNodeTravFuncType *post);
void          pfNodeTravData(pfNode *node, int which, void *data);
void *        pfGetNodeTravData(const pfNode *node, int which);
```

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfMorph** can also be used with these functions designed for objects of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
void*   pfGetUserData(pfObject *obj);
```

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfMorph** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *      pfGetType(const void *ptr);
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
```

int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

morph identifies a pfMorph.

DESCRIPTION

A pfMorph node does not define geometry; rather, it manipulates geometric attributes of pfGeoSets and other geometric primitives. While pfMorph is very general, its primary use is for geometric morphing where the colors, normals, texture coordinates and coordinates of geometry are smoothly changed over time to simulate actions such as facial and skeletal animation, ocean waves, continuous level-of-detail, and advanced special effects. In these situations, the rigid body transformations provided by matrices do not suffice - instead, efficient per-vertex manipulations are required.

A pfMorph consists of one or more "sources" and a single "destination" which together are termed an "attribute". Both sources and destination are arrays of "elements" where each element consists of 1 or more floating point numbers, e.g., an array of pfVec3 coordinates. The pfMorph node produces the destination by computing a weighted sum of the sources. By varying the source weights and using the morph destination as a pfGeoSet attribute array, the application can achieve smooth, geometric animation. A pfMorph can "morph" multiple attributes.

pfNewMorph creates and returns a handle to a pfMorph. Like other pfNodes, pfMorphs are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetMorphClassType returns the **pfType*** for the class **pfMorph**. The **pfType*** returned by **pfGetMorphClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfMorph**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfMorphAttr configures the *index*th attribute of *morph*. *floatsPerElt* specifies how many floating point numbers comprise a single attribute element. For example, when morphing pfGeoSet coordinate and texture coordinate arrays (**PFGS_COORD3**, **PFGS_TEXCOORD2**), *floatsPerElt* would be 3 and 2 respectively. *nelts* specifies how many attribute elements are in the destination array. If the required number of pfGeoSet coordinates is 33, then *nelts* would be 33, *not* 33 * 3 = 99. *dst* is a pointer to the destination array which should be at least of size *floatsPerElt* * *nelts* * sizeof(float). If *dst* is **NULL**, then *morph* will automatically create and use a pfCycleBuffer of appropriate size. (pfCycleBuffers are useful when IRIS Performer is configured to multiprocess.)

There are 2 distinct methods of accessing the source arrays of a pfMorph attribute: non-indexed and indexed. Indexing provides a means of efficiently applying sparse changes to the destination array. The *nsrcs* argument to **pfMorphAttr** specifies how many source arrays are provided in *alist*, i.e., *alist*[i] is the

i'th source and is treated as an array of elements where each element consists of *floatsPerElt* floating point numbers. Index arrays and their lengths are provided in *ilist* and *nlist* respectively. If *ilist* is **NULL** then all sources are non-indexed. If *ilist* is non-**NULL**, it contains a list of index lists corresponding to the source lists in *alist*. If *nlist* is **NULL**, then the index lists are assumed to be *nelts* long and if non-**NULL**, the length of each index list is specified in *nlist*. *ilist* may contain **NULL** pointers to mix indexed and non-indexed source arrays.

All source arrays referenced in *alist* and *ilist* are reference counted by **pfMorphAttr**.

pfMorphWeights specifies the source weights of the *index*th attribute of *morph* in the array *weights*. *weights* should consist of *nsrcs* floating point numbers where *nsrcs* is the number of attribute sources specified in **pfMorphAttr**. If *index* is < 0, then *weights* is used for all attributes of *morph*. **pfGetMorphWeights** copies the weights of the *index*th attribute of *morph* into *weights*. *weights* should be an array of at least *nsrcs* floats.

A pfMorph node is evaluated, i.e., its destination array is computed, during the **APP** traversal which is triggered directly by the application through **pfAppFrame** (see **pfAppFrame**) or indirectly by **pfSync**. Alternately, the pfMorph node may be explicitly evaluated by calling the function **pfEvaluateMorph**. In all cases, destination elements are computed as in the following pseudocode:

```
zero destination array;

for (s=0; s<nsrcs; s++)
{
    if (ilist == NULL || ilist[s] == NULL)
    {
        /* Source is non-indexed */
        for (i=0; i<nelts; i++)
            for (e=0; e<floatsPerElt; e++)
               dst[i][e] += weights[s] * alist[s][i][e];
    }
    else
    {
        /* Source is indexed */

        int     nindex;

        if (nlist == NULL)
            nindex = nelts;
        else
            nindex = nlist[s];

        for (i=0; i<nindex; i++)
```

```
            for (e=0; e<floatsPerElt; e++)
               dst[ilist[s][i]][e] += weights[s] * alist[s][i][e];
      }
   }
```

Note that the actual implementation is much more efficient than above, particularly for the special weights of 0 and 1.

Since pfMorph is a pfGroup, it is guaranteed to be evaluated before its children in the APP traversal. The pfMorph is only evaluated by the APP traversal when its weights change.

pfGetMorphNumAttrs returns the number of *morph*'s attributes.

pfGetMorphSrc returns the *src*th source parameters of the *index*th attribute of *morph*. The source attribute array and index array pointers are copied into *alist* and *ilist* respectively. The size of the *src*th index array is copied into *nlist* and the number of floats per element is returned by **pfGetMorphSrc**.

pfGetMorphNumSrcs returns the number of sources of the *index*th attribute of *morph*.

pfGetMorphDst returns the *index*th destination array of *morph*. The destination array is either that provided earlier by **pfMorphAttr** or the pfCycleBuffer automatically created when **NULL** was passed as the *dst* argument to **pfMorphAttr**.

SEE ALSO

pfAppFrame, pfCycleBuffer, pfGroup, pfDelete, pfLookupNode

NAME

pfGetNodeClassType, pfGetParent, pfGetNumParents, pfNodeBSphere, pfGetNodeBSphere, pfClone, pfBufferClone, pfFlatten, pfNodeName, pfGetNodeName, pfFindNode, pfLookupNode, pfNodeIsectSegs, pfNodeTravMask, pfGetNodeTravMask, pfNodeTravFuncs, pfGetNodeTravFuncs, pfNodeTravData, pfGetNodeTravData – Set and get pfNode parents and bounding spheres.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

pfType * **pfGetNodeClassType**(void);

pfGroup * **pfGetParent**(const pfNode *node, int i);

int **pfGetNumParents**(const pfNode *node);

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

int **pfGetNodeBSphere**(pfNode *node, pfSphere *bsph);

pfNode* **pfClone**(pfNode *node, int mode);

pfNode* **pfBufferClone**(pfNode *node, int mode, pfBuffer *buf);

int **pfFlatten**(pfNode *node, int mode);

int **pfNodeName**(pfNode *node, const char *name);

const char * **pfGetNodeName**(const pfNode *node);

pfNode* **pfFindNode**(pfNode *node, const char *pathName, pfType *type);

pfNode* **pfLookupNode**(const char *name, pfType* type);

int **pfNodeIsectSegs**(pfNode *node, pfSegSet *segSet, pfHit **hits[]);

void **pfNodeTravMask**(pfNode *node, int which, uint mask, int setMode, int bitOp);

uint **pfGetNodeTravMask**(const pfNode *node, int which);

void **pfNodeTravFuncs**(pfNode* node, int which, pfNodeTravFuncType pre, pfNodeTravFuncType post);

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

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

void * **pfGetNodeTravData**(const pfNode *node, int which);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfNode** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfNode**. Casting an object of class **pfNode** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfNode** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

node identifies a pfNode

```
typedef struct _pfSegSet
{
    int         mode;
    void *      userData;
    pfSeg       segs[PFIS_MAX_SEGS];
    uint        activeMask;
    uint        isectMask;
    void *      bound;
    int        (*discFunc)(pfHit*);
} pfSegSet;

typedef int (*pfNodeTravFuncType)(pfTraverser *trav, void *userData);
```

which identifies the traversal: **PFTRAV_ISECT**, **PFTRAV_APP**, **PFTRAV_CULL** or **PFTRAV_DRAW**, denoting the intersection, application,

DESCRIPTION

A pfNode is an abstract type. IRIS Performer does not provide any means to explicitly create a pfNode. Rather, the pfNode routines operate on the common aspects of other IRIS Performer node types.

The complete list of IRIS Performer nodes (all derived from pfNode) is:

pfLightPoint

pfText

pfGeode

pfBillboard

pfLightSource

pfGroup

pfSCS

pfDCS

pfPartition

pfScene

pfSwitch

pfLOD

pfSequence

pfLayer

Any IRIS Performer node is implicitly a pfNode, and a pointer to any of the above nodes may be used wherever a pfNode* is required as an argument.

The various pfNode types have certain common properties such as a set of parents, a name, an intersection mask, bounding geometry, callback functions and callback data.

pfGetNodeClassType returns the **pfType*** for the class **pfNode**. The **pfType*** returned by **pfGetNodeClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfNode**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfGetNumParents returns the number of parents *node* has in the scene graph. A node may have multiple parents because it was explicitly added to multiple parents with **pfAddChild**. In such cases it said to be 'instanced'. Also, leaf geometry nodes such as pfGeodes, pfLightPoints, and pfBillboards, may have multiple parents as a result of a **pfClone**. **pfGetParent** returns the *i*th parent of *node* or **NULL** if *i* is out of the range 0 to **pfGetNumParents** - 1.

pfNodeBSphere sets the bounding volume of *node*. Each pfNode has an associated bounding volume used for culling and intersection testing and a bounding mode, either static or dynamic. By definition, the bounding volume of a node encloses all the geometry parented by node, which means that the node and all its children fit within the node's bounding volume.

Only a subset of the pfNode types actually contain geometry. These are known as "leaf nodes" in IRIS Performer. They are:

pfBillboard
pfGeode
pfLightPoint

These and other nodes may indirectly contain geometry through user-supplied function callbacks set by **pfNodeTravFuncs**.

Normally IRIS Performer automatically computes bounding volumes but provides routines to explicitly set bounding volumes. This is useful for pfNodes which draw custom geometry through node callbacks (**pfNodeTravFuncs**).

The *bsph* argument to **pfNodeBSphere** is the bounding sphere of *node*. If the *bsph* is **NULL**, IRIS Performer will compute the bounding sphere of *node*.

The *mode* argument to **pfNodeBSphere** specifies whether or not the bounding volume for *node* should be recomputed when an attribute of *node* changes or something in the scene graph below *node* changes (if *node* is a pfGroup). If the mode is **PFBOUND_STATIC**, IRIS Performer will not modify the bound once it is set or computed. If the mode is **PFBOUND_DYNAMIC**, IRIS Performer will recompute the bound after children are added or deleted or after the matrix in a pfDCS changes. Changes in pfSwitches, pfLODs and pfSequences do not affect bounds above them in the scene graph.

pfGetNodeBSphere returns the current bounding mode and copies into *bsph* a pfSphere which encloses *node* and its children The return value is the bounding mode which is either **PFBOUND_DYNAMIC** or **PFBOUND_STATIC** indicating whether or not the bounding volume is updated automatically when its children change.

IRIS Performer supports two methods of node instancing. The first method is to simply add a node to more than one parent using **pfAddChild** or **pfReplaceChild** (see pfGroup). In this case the graph rooted by the instanced node is shared by all its parents. This type of instancing is called *shared* instancing.

pfClone provides instancing which shares geometry but not variable state like transformations (pfDCS) and switches (pfSwitch). **pfClone** copies the entire scene graph from *node* down to, but not including, leaf geometry nodes such as pfGeodes, pfBillboards and pfLightPoints. These leaf nodes are instanced by reference in the cloned scene graph. **pfClone** returns the root pfNode of the cloned graph or **NULL** to indicate error. This type of instancing is called *common geometry* instancing. An attempt to clone a leaf geometry node simply returns the handle to that node.

Cloning is recommended for instances of dynamic and articulated models. For example: Shared instances of a model with pfDCSes in its hierarchy will share the pfDCSes as well as the geometry. This means that all instances will have the exact same articulation. However, a common geometry instance will share only geometry and as a result of the cloning process will have its own pfDCSes allowing manipulation independently of any other instances. This example creates a cloned instance:

```
if ((clone = pfClone(carModel, 0)) != NULL)
    pfAddChild(carDCS_3, clone);
```

The *mode* argument to **pfClone** is reserved for future extensions and must be 0 in this release of IRIS Performer.

When cloning, if the global copy function (**pfCopyFunc**) is **NULL**, user data pointers (**pfUserData**) are copied to each new node and the reference counts of pfMemory-derived user data are incremented. If **pfCopyFunc** is not **NULL**, it will be invoked with the destination and source nodes as arguments. It is then the responsibility of the copy function to handle the copy of user data.

pfBufferClone is identical to **pfClone** but allows cloning across pfBuffers. *buf* identifies the pfBuffer which contains *node* and its subtree. The clone of *node* and its subtree is placed in the current buffer set by **pfSelectBuffer**. See the pfBuffer man page for more details.

pfFlatten is a database pre-processing step which 'flattens' the transformation hierarchy of the scene graph rooted by *node*. Coordinates and normals contained in leaf geometry nodes such as pfGeodes, pfBillboards and pfLightPoints are transformed by any inherited static transformations (pfSCS). **pfFlatten** automatically clones any pfNode or pfGeoSet that is multiply referenced. Specifically, if *node* has multiple parents, *node* and its entire subtree will be cloned. If a pfDCS is encountered, **pfFlatten** inserts a pfSCS in between the pfDCS and its parent.

Flattening can substantially improve performance, especially when pfSCSes are being used to instance a relatively small amount of geometry since the cost of the transformation approaches the cost of drawing the geometry. However, it can also increase the size of the database since it copies instanced nodes and geometry. Flattening is highly recommended for pfBillboards. Flattening also increases the ability of IRIS Performer to sort the database by mode (see **pfChanBinSort**), often a major performance enhancement, since sorting does not cross transformation boundaries.

pfFlatten does not remove pfSCSes from the hierarchy; instead it sets their transformations to the identity matrix. For improved traversal performance, these flattened pfSCS nodes should be removed from the hierarchy.

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

All IRIS Performer database nodes may be assigned a character string name. Individual node names need

not be unique but to access a node with a non-unique name, an unambiguous pathname to the node must be given. The pathname doesn't need to be a full path. All that's required is enough to distinguish the node from others with the same name.

pfNodeName sets the name of *node* to the string *name*. If the name is unique a 1 will be returned and if the name is not unique, a 0 will be returned. Node names are kept in a global table which is used for resolving the first path component of a path name by **pfLookupNode**. In this case, unambiguous resolution is only possible if the first path component is unique. **pfGetNodeName** returns the name of the node or NULL if the name has not been set.

pfFindNode is a general search routine for finding named pfNodes. **pfFindNode** begins searching for the node of type *type* and identified by a '/'-separated path name *pathName*. The search begins at *node* and uses a depth-first traversal. **pfFindNode** returns NULL if it cannot find the node. Note that the type checking performed by **pfFindNode** is equivalent to **pfIsOfType**, not **pfIsExactType**, e.g. searching for a pfGroup includes derived classes such as pfSwitch.

The string *pathName* can be either a name or a '/'-separated pathname. If the name contains no '/' characters, it is assumed to be unique and the global name table is searched. If *pathName* contains '/' characters, it is assumed to be a path. Paths are searched by first finding the node corresponding to the first component of the path in a global name table. The find routine then traverses the subtree rooted at that node, searching for the rest of the path. The first node encountered during the search traversal which matches *pathName* is returned.

Example 1:

```
pfNode  *newhouse, *newdoor;
pfDCS   *door;

/* Create "house" model with named subparts including "door" */

/* Create a new instance of "house" */
newhouse = pfClone(house, 0);

/* Give cloned house a new name */
pfNodeName(newhouse, "newhouse");

/* Find the door part of the new house */
door = (pfDCS*) pfFindNode(newhouse, "door", pfGetDCSClassType());
```

pfNodeIsectSegs intersects a group of line segments with a scene or portion thereof. The intersection operation traverses the scene graph, testing a group of segments against bounding geometry and eventually model geometry within pfGeoSets. *node* specifies the node at which intersection traversal starts.

pfNodeIsectSegs returns the number of segments which intersected something. *hits* is an empty array supplied by the user through which results are returned. The array must have an entry for each segment in *segSet*. Upon return, *hits*[i][0] is a pfHit* which gives the intersection result for the *i*th segment in *segSet*. The pfHit objects come from an internally maintained pool and are reused on subsequent requests. Hence, the contents are only valid until the next invocation of **pfGSetIsectSegs** in the current process. They should not be freed by the application.

segSet is a pfSegSet structure specifying the intersection request. In the structure, *segs* is an array of line segments to be intersected against the pfGeoSet. *activeMask* is a bit vector specifying which segments in the pfSegSet are to be active for the current request. If bit[i] of the *activeMask* is set to 1, it indicates the corresponding segment in the *segs* array is active.

The bit vector *mode* specifies the behavior of the intersection operation and is a bitwise OR of the following:

PFTRAV_IS_PRIM
Intersect with quads or triangle geometry.

PFTRAV_IS_GSET
Intersect with pfGeoSet 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.

For several types of pfGroups, the traversal of children can be controlled for the traversal.

For pfSwitches, the default is to traverse only the child or children specified by the current switch value. This can be changed OR-ing one of the following into the *mode* argument.

PFTRAV_SW_ALL
Traverse all children of pfSwitches.

PFTRAV_SW_NONE
Don't traverse any children of pfSwitches.

For pfSequences, the default is to traverse only the current child in the sequence. This can be changed OR-ing one of the following into the *mode* argument.

PFTRAV_SEQ_ALL
Intersect with all children of pfSequences.

PFTRAV_SEQ_NONE
Intersect with no children of pfSequences.

For pfLODs, the default is to traverse only the child that would be active at range 0. This can be changed OR-ing one of the following into the *mode* argument. Also, see **pfChanNodeIsectSegs** for child selection based on range.

PFTRAV_LOD_ALL
Intersect with all children of pfLODs (default is range 0).

PFTRAV_LOD_NONE
Intersect with no children of pfLODs (default is range 0).

For pfLayers, the default is to traverse all children. This can be changed OR-ing one of the following into the *mode* argument.

PFTRAV_LAYER_NONE
Intersect with no children of pfLayers (default is all).

PFTRAV_LAYER_BASE
Intersect with no children of pfLayers (default is all).

PFTRAV_LAYER_DECAL
Intersect with no children of pfLayers (default is all).

The bit fields **PFTRAV_IS_PRIM**, **PFTRAV_IS_GSET**, and **PFTRAV_IS_GEODE** indicate the level at which intersections should be evaluated and discriminator callbacks, if any, invoked. If none of these three fields are specified, no intersection testing is done.

In the pfSegSet, *isectMask* is another bit vector which directs the intersection traversal. At each stage of the intersection operation, the mask is bit-wise AND-ed with the mask of the pfNode or pfGeoSet. If the mask is non-zero the intersection continues with the next object, either a pfNode within a pfGroup or a primitive within a pfGeoSet. The mask of a pfNode is set using **pfNodeTravMask** and that of a pfGeoSet by **pfGSetIsectMask**. The mask can be used to distinguish parts of the scene graph which might respond differently to vision or collision. For example, as a wall would stop a truck but shrubbery would not.

The *bound* field in a pfSegSet is an optional user-provided bounding volume around the set of segments. Currently, the only supported volume is a cylinder. To use a bounding cylinder, perform a bitwise OR of

PFTRAV_IS_BCYL into the **mode** field of the pfSegSet and assign the pointer to the bounding volume to the **bound** field.

pfCylAroundSegs will construct a cylinder around the segments. When a bounding volume is supplied, the intersection traversal may use the cylinder to improve performance. The largest improvement is for groups of at least several segments which are closely grouped segments. Placing a bounding cylinder around small groups or widely dispersed segments can decrease performance.

The *userData* pointer allows an application to associate other data with the pfSegSet. Upon return and in discriminator callbacks, the pfSegSet's *userData* pointer can be obtained from the returned pfHit with **pfGetUserData**.

discFunc is a user supplied callback function which provides a more powerful means for controlling intersections than the simple mask test.

If *discFunc* is NULL, the default behavior clips the end of the segment after each successful intersection at the finest resolution (pfGeode bounding volume , pfGeoSet bounding box, pfGeoSet geometry) specified in *mode*. Thus, the segment is clipped by each successful intersection so that the intersection point nearest the starting point of the segment is returned upon completion.

If a discriminator callback is specified, whenever an intersection occurs, the *discFunc* callback is invoked with a pfHit structure containing information about the intersection. The discriminator may then return a value which indicates whether and how the intersection should continue. The continuation selectors are **PFTRAV_CONT**, **PFTRAV_PRUNE**, and **PFTRAV_TERM**.

PFTRAV_CONT
: Indicates that the traversal should continue traversing the pfGeoSets beneath a pfGeode. The discriminator function can examine information about candidate intersections and judge their validity and control the continuation of the traversal with its return value.

PFTRAV_PRUNE
: Indicates the traversal should return from the current level of the search and continue. If returned on a pfGeoSet primitive or bounding box test, **PFTRAV_PRUNE** stops further testing of the line segment against that pfGeoSet. If returned on the test against a pfGeode bounding volume, the pfGeode is not traversed for that line segment.

PFTRAV_TERM
: Indicates that the search should terminate for this segment of the pfSegSet. To have **PFTRAV_TERM** or **PFTRAV_PRUNE** apply to all segments, **PFTRAV_IS_ALL_SEGS** can be OR-ed into the discriminator return value. This causes the entire traversal to be terminated or pruned.

The callback may OR other bitfields into the status return value:

PFTRAV_IS_IGNORE
Indicates that the current intersection should be ignored, otherwise the intersection is taken as valid.

PFTRAV_IS_CLIP_START
Indicates for pruned and continued traversals that before proceeding the segment should be clipped to start at the current intersection point.

PFTRAV_IS_CLIP_END
Indicates for pruned and continued traversals that before proceeding the segment should be clipped to end at the current intersection point.

If *discFunc* is NULL, the behavior is the same as if the discriminator returned **PFTRAV_CONT | PFTRAV_IS_CLIP_END**, so that the intersection nearest the start of the segment will be returned.

In addition to the discriminator callback, pre- and post- intersection callbacks are available for each node. These behave identically to the pre- and post-callbacks for the cull traversal and can be used to prune, continue or terminate the traversal at any node.

Both **pfNodeIsectSegs** and the discriminator callback return information about an intersection in a pfHit object which can be examined using the **pfQueryHit** and **pfMQueryHit calls**. The information includes the intersection point, current matrix transformation, scene graph, and path. See the reference page for pfHit for further details.

In multiprocess applications, **pfNodeIsectSegs** should be called from the APP process or from the ISECT process (in the callback specified by **pfIsectFunc**). When called in the APP process, **pfNodeIsectSegs** should be called after **pfFrame** and before **pfSync** for best system throughput.

pfNodeTravMask sets the traversal masks of *node* which are used to control traversal during the intersection, cull, and draw traversals. If the bitwise AND of the node's mask for that traversal type and the mask for the current traversal is zero, the traversal is disabled at that node. By default, the node masks are all 1's. Traverser masks are set by **pfNodeIsectSegs/pfChanNodeIsectSegs** for the intersection traversal and **pfChanTravMask** for the CULL and DRAW traversals. **pfGetNodeTravMask** returns the specified traversal mask for the node.

Bits in the *setMode* argument indicate whether the set operation should be carried out for just the specified pfNode (**PFTRAV_SELF**), just its descendents (**PFTRAV_DESCEND**) or both itself and descendents. The descendent traversal goes down into pfGeoSets.

The *bitOp* argument is one of **PF_AND**, **PF_OR**, or **PF_SET** and indicates whether the new mask should be AND-ed with the old mask, OR-ed with the old mask or set outright, respectively.

Efficient intersections require that information be cached for each pfGeoSet to be intersected with. To create this cache, **PFTRAV_IS_CACHE** should be OR-ed into the *setMode* when first setting the

intersection mask. Because of the computation involved, the cache is best created at setup time. Subsequent changes to the masks themselves do not require **PFTRAV_IS_CACHE** to be specified. However, for dynamic objects whose geometry changes (e.g. pfGeoSets whose vertex arrays are being changed), additional calls with the **PFTRAV_IS_CACHE** in *setMode* should be used to recompute the cached information. **PFTRAV_IS_UNCACHE** can be OR-ed into the *setMode* to disable caching. **PFTRAV_IS_CACHE** and **PFTRAV_IS_UNCACHE** can only be specified when *which* is **PFTRAV_ISECT**.

pfNodeTravFuncs specify the user supplied functions which are to be invoked during the traversal indicated by *which*. For each traversal, there is a *pre* and *post* traversal callback. *pre* is invoked before *node* and its children are processed while *post* is invoked after. The pre- and post- methodology supports save and restore or push and pop programming constructs. Node callbacks are passed pointers to the user supplied traversal data pointer for that node and a pfTraverser which defines the current traversal state. **pfGetNodeTravFuncs** copies *node*'s pre and post callbacks of traversal type *which* into *pre* and *post* respectively.

The *data* argument to **pfNodeTravData** is the pointer which is passed to the traversal callbacks indicated by *which*. Both pre- and post-callbacks will be passed *data* in addition to a pfTraverser*. When multiprocessing, *data* should point to memory in a shared arena. **pfGetNodeTravData** returns the current data pointer for the specified traversal.

NOTES

When instanced geometry is flattened, the copy created by **pfFlatten** shares pfGeoSet attribute arrays with the original when possible. This means that the newly flattened pfGeoSet may share some arrays (e.g. color array), but not other arrays (e.g. the vertex array) with the original.

The post-cull callback is a good place to implement custom level-of-detail mechanisms.

Currently, nodes use spheres as the default bounding volume. This may change in a future release. **libpfutil** contains sample code for computing the bounding box for a subgraph of the scene.

It's an interesting fact that although a node's bounding volume completely contains the geometry of the nodes that it parents, it may well *not* completely contain the bounding volumes of those same nodes. Do you understand when this situation would occur?

Finding a node by name can be expensive, particularly for path based searches. These functions are primarily intended to get handles to nodes which are loaded from disk and should be used sparingly at simulation time.

In Performer 2.0, **pfLookupNode** replaces a number of functions from 1.2, e.g. **pfLookupBboard**. See the scripts in /usr/share/Performer/src/tools for help in porting code.

BUGS

If the graph under a node cloned by **pfClone** contains an object instanced within the graph, (i.e. a node having two or more parents within the graph), the new graph will contain multiple copies of the instanced node rather than duplicating the connectivity of the original graph.

pfFlatten transforms the vertex arrays of non-instanced geometry in place. If a pfGeoSet belongs to multiple pfGeodes or a vertex array is shared between pfGeoSets the array is still flattened in place.

It is not possible to get multiple intersection results per segment without a discriminator callback.

Bounding cylinders do not work when non-orthonormal transformations are present in the pfDCS and pfSCS nodes of a scene graph.

The path returned by **pfGetTravPath** is valid only when invoked from a cull callback.

SEE ALSO

pfAddChild, pfCylAroundSegs, pfDelete, pfClone, pfFlatten, pfGetType, pfFindNode, pfLookupNode, pfNodeIsectSegs, pfNodeName, pfNodeTravFuncs, pfCopyFunc, pfChanBinSort, pfGSetIsectMask, pfQueryHit, pfMQueryHit, pfBillboard, pfDCS, pfFrame, pfGeode, pfIsectFunc, pfLightPoint, pfScene, pfSCS, pfSeg, pfGSetIsectSegs, pfSync, pfTraverser

NAME

pfNewPart, pfGetPartClassType, pfPartVal, pfGetPartVal, pfPartAttr, pfGetPartAttr, pfBuildPart, pfUpdatePart – Create and update pfPartition spatial partitioning node.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

pfPartition * **pfNewPart**(void);

pfType * **pfGetPartClassType**(void);

void **pfPartVal**(pfPartition* part, int which, float val);

float **pfGetPartVal**(pfPartition *part, int which);

void **pfPartAttr**(pfPartition* part, int which, void *attr);

void * **pfGetPartAttr**(pfPartition *part, int which);

void **pfBuildPart**(pfPartition* cset);

void **pfUpdatePart**(pfPartition* cset);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfPartition** is derived from the parent class **pfGroup**, so each of these member functions of class **pfGroup** are also directly usable with objects of class **pfPartition**. Casting an object of class **pfPartition** to an object of class **pfGroup** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfGroup**.

int **pfAddChild**(pfGroup *group, pfNode *child);
int **pfInsertChild**(pfGroup *group, int index, pfNode *child);
int **pfReplaceChild**(pfGroup *group, pfNode *old, pfNode *new);
int **pfRemoveChild**(pfGroup *group, pfNode* child);
int **pfSearchChild**(pfGroup *group, pfNode* child);
pfNode * **pfGetChild**(const pfGroup *group, int index);
int **pfGetNumChildren**(const pfGroup *group);
int **pfBufferAddChild**(pfGroup *group, pfNode *child);
int **pfBufferRemoveChild**(pfGroup *group, pfNode *child);

Since the class **pfGroup** is itself derived from the parent class **pfNode**, objects of class **pfPartition** can also be used with these functions designed for objects of class **pfNode**.

pfGroup * **pfGetParent**(const pfNode *node, int i);
int **pfGetNumParents**(const pfNode *node);
void **pfNodeBSphere**(pfNode *node, pfSphere *bsph, int mode);

```
int             pfGetNodeBSphere(pfNode *node, pfSphere *bsph);
pfNode*         pfClone(pfNode *node, int mode);
pfNode*         pfBufferClone(pfNode *node, int mode, pfBuffer *buf);
int             pfFlatten(pfNode *node, int mode);
int             pfNodeName(pfNode *node, const char *name);
const char *    pfGetNodeName(const pfNode *node);
pfNode*         pfFindNode(pfNode *node, const char *pathName, pfType *type);
pfNode*         pfLookupNode(const char *name, pfType* type);
int             pfNodeIsectSegs(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void            pfNodeTravMask(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint            pfGetNodeTravMask(const pfNode *node, int which);
void            pfNodeTravFuncs(pfNode* node, int which, pfNodeTravFuncType pre,
                    pfNodeTravFuncType post);
void            pfGetNodeTravFuncs(const pfNode* node, int which, pfNodeTravFuncType *pre,
                    pfNodeTravFuncType *post);
void            pfNodeTravData(pfNode *node, int which, void *data);
void *          pfGetNodeTravData(const pfNode *node, int which);
```

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfPartition** can also be used with these functions designed for objects of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
void*   pfGetUserData(pfObject *obj);
```

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfPartition** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *        pfGetType(const void *ptr);
int             pfIsOfType(const void *ptr, pfType *type);
int             pfIsExactType(const void *ptr, pfType *type);
const char *    pfGetTypeName(const void *ptr);
int             pfRef(void *ptr);
int             pfUnref(void *ptr);
int             pfUnrefDelete(void *ptr);
int             pfGetRef(const void *ptr);
int             pfCopy(void *dst, void *src);
int             pfDelete(void *ptr);
int             pfCompare(const void *ptr1, const void *ptr2);
void            pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
```

void * **pfGetArena**(void *ptr);

DESCRIPTION

A pfPartition is a type of pfGroup for organizing the subgraph of a scene into a static data structure which is more efficient for intersection testing with **pfNodeIsectSegs** for some databases. pfPartition does not affect culling performance nor does it improve intersection performance under transformation nodes, pfSwitch nodes, pfMorph nodes or pfSequence nodes.

pfNewPart creates and returns a handle to a pfPartition. Like other pfNodes, pfPartitions are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetPartClassType returns the **pfType*** for the class **pfPartition**. The **pfType*** returned by **pfGetPartClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfPartition**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfBuildPart constructs a 2D spatial partitioning based on the *type*.

Within the confines of the parameters set by **pfPartAttr**, IRIS Performer attempts to construct an optimal partition based on the distribution of vertices within the pfGeoSets in the subgraph of the scene rooted at the partition. Information about the selected partitioning is displayed when the **pfNotifyLevel** is debug or higher. Because the search for the optimal partitioning is compute intensive, once the partitioning has been determined for a particular database, the range of the search should be restricted using **pfPartAttr**.

pfUpdatePart causes the scene graph under the partition to be traversed and any changes incorporated into the spatial partitioning. The partitioning itself does not change.

pfPartAttr sets the partition attribute *attr* to the attribute *attr*. Partition attributes are:

PFPART_MIN_SPACING
attr points to a pfVec3 specifying the minimum spacing between partition dividers in each dimension. If not specified, the default is 1/20th of the bounding box diagonal. When a partition is built, a search is made between **PFPART_MAX_SPACING** and **PFPART_MIN_SPACING**.

PFPART_MAX_SPACING
attr points to a pfVec3 specifying the maximum spacing between partition dividers in each dimension. If not specified, the default is 1/10th of the bounding box diagonal. When a partition is built, a search is made between **PFPART_MAX_SPACING** and **PFPART_MIN_SPACING**.

PFPART_ORIGIN
attr points to a pfVec3 specifying an origin for the partition. If not specified, a search is done to find an optimal origin.

pfGetPartAttr returns the partition attribute *attr*.

pfPartVal sets the partition value *val* to the value *val*. Partition values are:

PFPART_FINE
A value between 0.0 and 1.0 which indicates how fine of a partitioning should be constructed. The subdivision is limited by **PFPART_MIN_SPACING** adn **PFPART_MAX_SPACING**. 1.0 causes extremely fine subdivision. 0.0 causes no subdivision. 0.5 is usually a good value and is the default.

pfGetPartVal returns the partition value *val*.

A pfPartition behaves like a pfGroup when the mode in the pfSegSet used with **pfNodeIsectSegs** includes **PFTRAV_IS_NO_PART**.

NOTES

pfPartitions are primarily useful for databases containing many axis-aligned objects for which bounding spheres are a poor fit and when only one or two segments are made per call to **pfNodeIsectSegs**. For example, terrain following on gridded terrain is likely to benefit. For databases such as this which themselves have a regular grid, it is also important for performance that the origin and spacing of the partition align exactly the terrain grid. pfPartitions do not currently help with the problem pfGeoSets containing too much geometry.

BUGS

The search for an optimal grid is very thorough so that it takes a *very* long time if the search domain is large. Once a good partitioning for a database is determined, the **PFPART_MIN_SPACING**, **PFPART_MAX_SPACING** and **PFPART_ORIGIN** can be set equal for much faster building.

Currently only partitionings in the XY plane are supported.

SEE ALSO

pfGroup, pfLookupNode, pfNode, pfNodeIsectSegs, pfNotifyLevel, pfScene

NAME

pfNewPath, pfGetPathClassType, pfCullPath – Create, modify, and maintain a node path.

FUNCTION SPECIFICATION

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

pfPath *    pfNewPath(void);

pfType *    pfGetPathClassType(void);

int         pfCullPath(pfPath *path, pfNode *node, int mode);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfPath** is derived from the parent class **pfList**, so each of these member functions of class **pfList** are also directly usable with objects of class **pfPath**. Casting an object of class **pfPath** to an object of class **pfList** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfList**.

```
void            pfAdd(pfList* list, void* elt);
void            pfCombineLists(pfList* dst, const pfList *a, const pfList *b);
int             pfFastRemove(pfList* list, void* elt);
void            pfFastRemoveIndex(pfList* list, int index);
void *          pfGet(const pfList* list, int index);
const void **   pfGetListArray(const pfList* list);
int             pfGetListArrayLen(const pfList* len);
int             pfGetListEltSize(const pfList* list);
int             pfGetNum(const pfList* list);
void            pfInsert(pfList* list, int index, void* elt);
int             pfMove(pfList* lists, int index, void *elt);
void            pfListArrayLen(pfList* list, int len);
void            pfNum(pfList *list, int num);
int             pfRemove(pfList* list, void* elt);
void            pfRemoveIndex(pfList* list, int index);
int             pfReplace(pfList* list, void* old, void* new);
void            pfResetList(pfList* list);
int             pfSearch(const pfList* list, void* elt);
void            pfSet(pfList* list, int index, void *elt);
```

Since the class **pfList** is itself derived from the parent class **pfObject**, objects of class **pfPath** can also be used with these functions designed for objects of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
```

void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfPath** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

DESCRIPTION

A pfPath is a dynamically-sized array of pointers. A pfPath consisting of pfNode pointers can define a specific path or chain of nodes through a scene graph.

pfNewPath creates and returns a handle to a pfPath. pfPaths are usually allocated from shared memory. The path element size is sizeof(void*) and the initial number of elements in the path is 4. pfPaths can be deleted using **pfDelete**.

pfGetPathClassType returns the **pfType*** for the class **pfPath**. The **pfType*** returned by **pfGetPathClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfPath**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfCullPath traverses and culls the chain of nodes specified in *path*, beginning at *root*. If *path* is NULL, then *root* will be traversed in-order. If *root* is NULL, then the exact chain of nodes specified in *path* will be traversed. If neither *root* nor *path* is NULL, then the paths traversed will be all paths emanating from *root* which reach the first node in *path* and then continue down the nodes specified in *path*.

mode is a bitmask indicating which type of "switching" nodes (pfLOD, pfSequence, pfSwitch) to evaluate and may be either:

PFPATH_IGNORE_SWITCHES
Do not evaluate any switches in the node path.

or else it is the bitwise OR of the following:

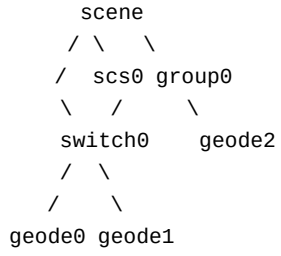
PFPATH_EVAL_LOD
Evaluate any pfLOD nodes in the node path.

PFPATH_EVAL_SEQUENCE
Evaluate any pfSequence nodes in the node path.

PFPATH_EVAL_SWITCH
Evaluate any pfSwitch nodes in the node path.

When an enabled switch node is encountered, traversal will terminate if the next node in the path is not one selected by the switch. As a convenience, **PFPATH_EVAL_SWITCHES** is defined to enable all three of these switchs (**PFPATH_EVAL_LOD**, **PFPATH_EVAL_SWITCH**, and **PFPATH_EVAL_SEQUENCE**).

Example 1: Path culling




```
path = pfNewPath();

pfAdd(path, switch0);
pfAdd(path, geode1);
 :
/*
 * In cull callback.  This will cull the following paths:
 *
 *      scene -> switch0 -> geode1
 *      scene -> scs0 -> switch0 -> geode1
 *
 * Note that both path traversals will terminate at switch0
 * if the pfSwitch's switch value is not 1.
*/
```

```
pfCullPath(path, scene, PFPATH_EVAL_SWITCHES);
```

pfCullPath should only be called in the cull callback function set by **pfChanTravFunc**. The pfChannel passed to the cull callback will be used to traverse the path, that is its LOD attributes will affect the pfLODs traversed and nodes will be culled to its viewing frustum.

SEE ALSO

pfChanTravFunc, pfCull, pfList

NAME

pfGetPipeSize, pfGetPipeClassType, pfPipeScreen, pfGetPipeScreen, pfPipeWSConnectionName, pfGetPipeWSConnectionName, pfGetPipePWin, pfMovePWin, pfPipeSwapFunc, pfGetPipeSwapFunc, pfGetPipeNumPWins, pfGetPipeNumChans, pfGetPipeChan – Initialize and get window information for a pfPipe.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

void	**pfGetPipeSize**(const pfPipe *pipe, int *xsize, int *ysize);
pfType *	**pfGetPipeClassType**(void);
void	**pfPipeScreen**(pfPipe *pipe, int screen);
int	**pfGetPipeScreen**(const pfPipe *pipe);
void	**pfPipeWSConnectionName**(pfPipe *pipe, const char *name);
const char *	**pfGetPipeWSConnectionName**(const pfPipe *pipe);
pfPipeWindow *	**pfGetPipePWin**(const pfPipe *pipe, int which);
int	**pfMovePWin**(pfPipe* pipe, int where, pfPipeWindow *pwin);
void	**pfPipeSwapFunc**(pfPipe* pipe, pfPipeSwapFuncType func);
pfPipeSwapFuncType	**pfGetPipeSwapFunc**(const pfPipe* pipe);
int	**pfGetPipeNumPWins**(const pfPipe *pipe);
int	**pfGetPipeNumChans**(const pfPipe *pipe);
pfChannel *	**pfGetPipeChan**(const pfPipe *pipe, int which);

```
/* pfPipe-specific types */
typedef void (*pfPipeFuncType)(pfPipe *p);
typedef void (*pfPipeSwapFuncType)(pfPipe *p, pfPipeWindow *pw);
```

PARAMETERS

pipe identifies a pfPipe.

DESCRIPTION

A pfPipe is a software rendering pipeline which renders one or more pfChannels into one or more pfPipeWindows. A pfPipe can be configured as multiple processes for increased throughput on multiprocessor systems. Multiple pfPipes can operate in parallel in support of platforms with multiple graphics pipelines. The number of pfPipes and the multiprocessing mode used are set by **pfMultipipe** and **pfMultiprocess** respectively (see **pfConfig**).

A pfPipe references one or more pfPipeWindows which in turn reference one or more pfChannels. A pfChannel is simply a view of a scene which is rendered into a viewport of a pfPipeWindow. A

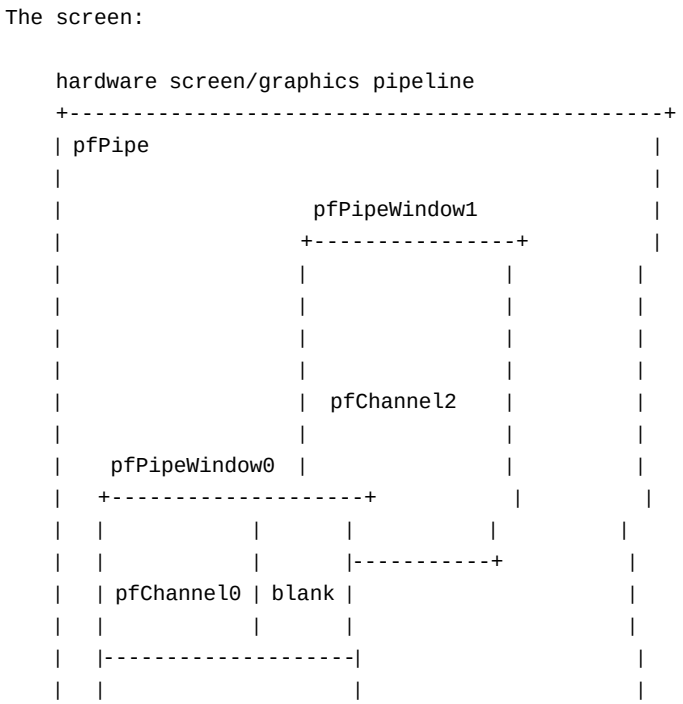
pfPipeWindow is a graphics window managed by its parent pfPipe.

pfPipes, pfPipeWindows, and pfChannels form a hierarchy with the following rules:

1. A screen (i.e. hardware graphics display) can have multiple pfPipes but should only have one drawing to it
2. A pfPipe may only draw to one screen
3. A pfPipe may render to multiple pfPipeWindows
4. A pfPipeWindow belongs to a single fixed pfPipe and thus also to a single fixed screen
5. A pfPipeWindow may have multiple pfChannels
6. A pfChannel always belongs to a pfPipe but may change pfPipeWindows or might not belong to any pfPipeWindow. a channel not assigned to a pfPipeWindow is culled but not drawn.

The following is an example pfPipe->pfPipeWindow->pfChannel configuration.

Example 1:



```
|  |    pfChannel1      |                    |
|  |                    |                    |
|  +--------------------+                     |
|                                             |
|                                             |
+---------------------------------------------+
```

The hierarchy:

```
              screen0 ....
                |
                |
              pfPipe0 ----------------------+
               /   \                        |
              /     \                       |
             /       \                      |
        pfPipeWindow0   pfPipeWindow1       |
          /     \          \                |
         /       \          \               |
    pfChannel0  pfChannel1   pfChannel2  pfChannel3
                                         (not drawn)
```

The code: (in application process)

```
    /* Calls that create the hierarchy: */
    pfPipe             *pipe = pfGetPipe(0);
    pfPipeWindow       *pwin0 = pfNewPWin(pipe);
    pfPipeWindow       *pwin1 = pfNewPWin(pipe);

    pfChannel  *chan0 = pfNewChan(pipe);
    pfChannel  *chan1 = pfNewChan(pipe);
    pfChannel  *chan2 = pfNewChan(pipe);

    pfAddChan(pwin0, chan0);
    pfAddChan(pwin0, chan1);
    pfAddChan(pwin1, chan2);

    /* Calls that cause the window to be opened at next pfFrame() */
    pfOpenPWin(pwin0);
    pfOpenPWin(pwin1);
    pfFrame();
```

If a pfPipe has no windows at the time **pfFrame** is called, a full screen pfPipeWindow will be opened for *pipe* and all pfChannels of *pipe* will be assigned to that pfPipeWindow.

pfGetPipeClassType returns the **pfType*** for the class **pfPipe**. The **pfType*** returned by **pfGetPipeClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfPipe**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfPipeScreen specifies the hardware screen, *screen*, (graphics pipeline) used for rendering by the pfPipe, *pipe*. The screen of the pfPipe may be specified in the application process before the call to open or configure any pfPipeWindows (**pfOpenPWin**, **pfConfigPWin**) on *pipe*, or may be specified implicitly by the screen of the first opened pfPipeWindow. A pfPipe is tied to a specific hardware pipeline and the screen of a pfPipe cannot be changed once determined. For single pipe operation, if the screen of a pfPipe or pfPipeWindow is never explicitly set in single pipe configuration, the screen will be taken from the default screen of the current pfWSConnection, or current X Display. For multipipe operation, if the screen of a pfPipe or pfPipeWindow is never explicitly set and pfMultipipe() has been used to configure multiple pfPipes, then pfPipes will automatically be assigned to hardware screens in order, i.e., pfGet-Pipe(0) -> screen 0, pfGetPipe(1) -> screen 1, etc. If a custom mapping of pfPipes to screens is desired, the screens of all pfPipes must be specified before the configuration of the first pfPipe which will happen at the first call to **pfFrame**. See the **pfGetCurWSConnection** reference page for more details on how to manage X display connections.

pfPipeWSConnectionName allows you to specify both a window server target and screen for the pfPipe. This is useful for doing remote rendering, or for running on a system with multiple window servers. This call should be made in the application process, before the first call to pfFrame.
pfGetPipeWSConnectionName will return the current window server target name. A window server target specified on a pfPipe will take precedence over any such targets specified on pfPipeWindows of that pfPipe. If the window server target of a pfPipe has not been set, it may be implicitly set from the first such setting on a child pfPipeWindow. The window server target of a pfPipe may not be changed after the first call to **pfFrame**. See the **pfGetCurWSConnection** reference page for more details on how to manage X display connections.

pfGetPipeScreen can be used to get the screen of a pfPipe. A return value of (-1) indicates that the screen of the pfPipe is undefined. **pfGetPipeSize** returns the size of the screen used by *pipe*.

For best performance only one pfPipe should render to a given hardware pipeline. If multiple views on a single screen are desired, use multiple pfChannels, and if necessary, multiple pfPipeWindows.

Normally a pfPipe swaps the color buffers at the end of each frame. However, if special control is needed over buffer swapping, **pfPipeSwapFunc** will register *func* as the buffer swapping function for *pipe*. Instead of swapping buffers, *func* will be called and will be expected to swap the color buffers of the pro-vided pfPipeWindow. **pfGetPipeSwapFunc** returns the buffer swapping function of *pipe* or **NULL** if

none is set.

If you wish to frame lock multiple pfPipes so that each pfPipe swaps its color buffers at the same time, then you should create a channel group consisting of one or more pfChannels on each pfPipe and make sure **PFCHAN_SWAPBUFFERS** is shared. In addition, separate hardware graphics pipelines *must* be genlocked for proper frame-locking.

pfGetPipePWin returns the pointer to the pfPipeWindow at the location specified by *which* in the pfPipeWindow list on *pipe*.

pfGetPipeNumPWins returns the number of pfPipeWindows that have been created on *pipe*.
pfGetPipeNumChans returns the number of pfChannels that have been created on *pipe*.

pfMovePWin moves the specified pfPipeWindow *pwin* to the location specified by *where* in the pfPipeWindow list on *pipe*. The move includes removing *pwin* from its current location by moving up the elements in the list that follow it and then inserting *pwin* into its new location. If *pwin* is attached to *pipe*, (-1) is returned and *pwin* is not inserted into the list. Otherwise, *where* is returned to indicate success. *where* must be within the range [0 .. n] where n is the number returned by **pfGetPipeNumPWins**(), or else (-1) is returned and no move is executed.

pfGetPipeChan returns the pointer to the pfChannel at location *which* in the list of pfChannels on *pipe*.

Example 2: How to frame lock pfPipes

```
leftChan  = pfNewChan(pfGetPipe(0));
rightChan = pfNewChan(pfGetPipe(1));

/* BPFCHAN_SWAPBUFFERS is shared by default */
pfAttachChan(leftChan, rightChan);

/* Pipe 0 and pipe 1 are now frame-locked */
```

NOTES

pfPipes cannot be deleted.

SEE ALSO

pfAttachChan, pfNewChan, pfConfig, pfMultipipe, pfMultiprocess, pfPipeWindow, pfGetCurWSConnection

NAME

pfNewPWin, pfGetPWinClassType, pfPWinAspect, pfPWinConfigFunc, pfPWinFBConfig, pfPWinFBConfigAttrs, pfPWinFBConfigData, pfPWinFBConfigId, pfPWinFullScreen, pfPWinGLCxt, pfPWinIndex, pfPWinList, pfPWinMode, pfPWinName, pfPWinOrigin, pfPWinOriginSize, pfPWinOverlayWin, pfPWinScreen, pfPWinShare, pfPWinSize, pfPWinStatsWin, pfPWinType, pfPWinWSConnectionName, pfPWinWSDrawable, pfPWinWSWindow, pfGetPWinAspect, pfGetPWinChanIndex, pfGetPWinConfigFunc, pfGetPWinCurOriginSize, pfGetPWinCurScreenOriginSize, pfGetPWinCurState, pfGetPWinCurWSDrawable, pfGetPWinFBConfig, pfGetPWinFBConfigAttrs, pfGetPWinFBConfigData, pfGetPWinFBConfigId, pfGetPWinGLCxt, pfGetPWinIndex, pfGetPWinList, pfGetPWinMode, pfGetPWinName, pfGetPWinOrigin, pfGetPWinOverlayWin, pfGetPWinPipe, pfGetPWinPipeIndex, pfGetPWinScreen, pfGetPWinSelect, pfGetPWinShare, pfGetPWinSize, pfGetPWinStatsWin, pfGetPWinType, pfGetPWinWSConnectionName, pfGetPWinWSDrawable, pfGetPWinWSWindow, pfAttachPWin, pfAttachPWinWin, pfDetachPWin, pfDetachPWinWin, pfClosePWin, pfClosePWinGL, pfConfigPWin, pfOpenPWin, pfIsPWinOpen, pfMQueryPWin, pfQueryPWin, pfChoosePWinFBConfig, pfSelectPWin, pfSwapPWinBuffers, pfGetNumChans, pfAddChan, pfGetChan, pfInsertChan, pfMoveChan, pfRemoveChan, pfInitGfx – Initialize and manipulate pfPipeWindows within a pfPipe

FUNCTION SPECIFICATION

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

pfPipeWindow*   pfNewPWin(pfPipe *p);
pfType*         pfGetPWinClassType(void);
void            pfPWinAspect(pfPipeWindow *pwin, int x, int y);
void            pfPWinConfigFunc(pfPipeWindow *pwin, pfPWinFuncType func);
void            pfPWinFBConfig(pfPipeWindow *pwin, XVisualInfo *vi);
void            pfPWinFBConfigAttrs(pfPipeWindow *pwin, int *attr);
void            pfPWinFBConfigData(pfPipeWindow *pwin, void *data);
void            pfPWinFBConfigId(pfWindow *win, int id);
void            pfPWinFullScreen(pfPipeWindow *pwin);
void            pfPWinGLCxt(pfPipeWindow *pwin, pfGLContext gc);
void            pfPWinIndex(pfPipeWindow *pwin, int index);
void            pfPWinList(pfPipeWindow *pwin, pfList *wlist);
void            pfPWinMode(pfPipeWindow *pwin, int mode, int val);
void            pfPWinName(pfPipeWindow *pwin, const char *name);
```

void	**pfPWinOrigin**(pfPipeWindow *pwin, int xo, int yo);
void	**pfPWinOriginSize**(pfPipeWindow *pwin, int xo, int yo, int xs, int ys);
void	**pfPWinOverlayWin**(pfPipeWindow *pwin, pfWindow *ow);
void	**pfPWinScreen**(pfPipeWindow *pwin, int screen);
void	**pfPWinShare**(pfPipeWindow *pwin, int mode);
void	**pfPWinSize**(pfPipeWindow *pwin, int xs, int ys);
void	**pfPWinStatsWin**(pfPipeWindow *pwin, pfWindow *sw);
void	**pfPWinType**(pfPipeWindow *pwin, uint type);
void	**pfPWinWSConnectionName**(const pfWindow *win, const char *name);
void	**pfPWinWSDrawable**(pfPipeWindow *pwin, pfWSConnection dsp, pfWSDrawable gxw);
void	**pfPWinWSWindow**(pfPipeWindow *pwin, pfWSConnection dsp, pfWSWindow wsw);
void	**pfGetPWinAspect**(pfPipeWindow *pwin, int *x, int *y);
int	**pfGetPWinChanIndex**(pfPipeWindow *pwin, pfChannel *chan);
pfPWinFuncType	**pfGetPWinConfigFunc**(pfPipeWindow *pwin);
void	**pfGetPWinCurOriginSize**(pfPipeWindow *pwin, int *xo, int *yo, int *xs, int *ys);
void	**pfGetPWinCurScreenOriginSize**(pfPipeWindow *pwin, int *xo, int *yo, int *xs, int *ys);
pfState*	**pfGetPWinCurState**(pfPipeWindow *pwin);
pfWSDrawable	**pfGetPWinCurWSDrawable**(pfPipeWindow *pwin);
XVisualInfo*	**pfGetPWinFBConfig**(pfPipeWindow *pwin);
int*	**pfGetPWinFBConfigAttrs**(pfPipeWindow *pwin);
void*	**pfGetPWinFBConfigData**(pfPipeWindow *pwin);
int	**pfGetPWinFBConfigId**(const pfWindow *win);
pfGLContext	**pfGetPWinGLCxt**(pfPipeWindow *pwin);
int	**pfGetPWinIndex**(pfPipeWindow *pwin);
pfList*	**pfGetPWinList**(const pfPipeWindow *pwin);
int	**pfGetPWinMode**(pfPipeWindow *pwin, int mode);
const char*	**pfGetPWinName**(pfPipeWindow *pwin);

void	**pfGetPWinOrigin**(pfPipeWindow *pwin, int *xo, int *yo);
pfWindow*	**pfGetPWinOverlayWin**(pfPipeWindow *pwin);
pfPipe*	**pfGetPWinPipe**(pfPipeWindow *pwin);
int	**pfGetPWinPipeIndex**(const pfPipeWindow *pwin);
int	**pfGetPWinScreen**(pfPipeWindow *pwin);
pfWindow*	**pfGetPWinSelect**(pfPipeWindow *pwin);
uint	**pfGetPWinShare**(pfPipeWindow *pwin);
void	**pfGetPWinSize**(pfPipeWindow *pwin, int *xs, int *ys);
pfWindow*	**pfGetPWinStatsWin**(pfPipeWindow *pwin);
uint	**pfGetPWinType**(pfPipeWindow *pwin);
const char*	**pfGetPWinWSConnectionName**(const pfWindow *win);
pfWSDrawable	**pfGetPWinWSDrawable**(pfPipeWindow *pwin);
Window	**pfGetPWinWSWindow**(pfPipeWindow *pwin);
int	**pfAttachPWin**(pfPipeWindow *pwin, pfPipeWindow *pw);
int	**pfAttachPWinWin**(pfPipeWindow *pwin, pfWindow *w);
int	**pfDetachPWin**(pfPipeWindow *pwin, pfPipeWindow *pw);
int	**pfDetachPWinWin**(pfPipeWindow *pwin, pfWindow *w);
void	**pfClosePWin**(pfPipeWindow *pwin);
void	**pfClosePWinGL**(pfPipeWindow *pwin);
void	**pfConfigPWin**(pfPipeWindow *pwin);
void	**pfOpenPWin**(pfPipeWindow *pwin);
int	**pfIsPWinOpen**(pfPipeWindow *pwin);
int	**pfMQueryPWin**(pfPipeWindow *pwin, int *which, int *dst);
int	**pfQueryPWin**(pfPipeWindow *pwin, int which, int *dst);
pfFBConfig	**pfChoosePWinFBConfig**(pfPipeWindow *pwin, pfWSConnection dsp, int screen, int *attr);
pfWindow*	**pfSelectPWin**(pfPipeWindow *pwin);
void	**pfSwapPWinBuffers**(pfPipeWindow *pwin);
int	**pfGetNumChans**(const pfPipeWindow *pwin);

void **pfAddChan**(pfPipeWindow *pwin, pfChannel *chan);
pfChannel* **pfGetChan**(pfPipeWindow *pwin, int which);
void **pfInsertChan**(pfPipeWindow *pwin, int where, pfChannel *chan);
void **pfMoveChan**(pfPipeWindow *pwin, int where, pfChannel *chan);
void **pfRemoveChan**(pfPipeWindow *pwin, pfChannel *chan);
extern void **pfInitGfx**(void);

```
/* pfPipeWindow-specific types */
typedef void (*pfPWinFuncType)(pfPipeWindow *pw);

/* X-Window system based Performer types */
typedef Display         *pfWSConnection;
typedef XVisualInfo     pfFBConfig;
typedef Window          pfWSWindow;
typedef Drawable        pfWSDrawable;

#ifdef IRISGL
typedef int             pfGLContext;
#else /* OPENGL */
typedef GLXContext      pfGLContext;
#endif
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfPipeWindow** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfPipeWindow**. Casting an object of class **pfPipeWindow** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfPipeWindow** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);

```
int             pfIsExactType(const void *ptr, pfType *type);
const char *    pfGetTypeName(const void *ptr);
int             pfRef(void *ptr);
int             pfUnref(void *ptr);
int             pfUnrefDelete(void *ptr);
int             pfGetRef(const void *ptr);
int             pfCopy(void *dst, void *src);
int             pfDelete(void *ptr);
int             pfCompare(const void *ptr1, const void *ptr2);
void            pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *          pfGetArena(void *ptr);
```

PARAMETERS

pwin identifies a pfPipeWindow.

dsp identifies a pfWSConnection.

wsw identifies a pfWSWindow.

gxw identifies a pfWSDrawable.

gc identifies a pfGLContext.

DESCRIPTION

IRIS Performer programs render a pfChannel to a pfPipeWindow of the same parent pfPipe. Multiple pfPipeWindows can be open on a single pfPipe. A pfPipe and all of its windows have the same screen, or hardware graphics pipeline. By default, pfChannels are assigned to the first pfPipeWindow of a pfPipe. pfChannels can be removed from the pfPipeWindow and assigned to other pfPipeWindows. pfPipeWindows can be opened/closed and created at any time. Refer to the **pfPipe** reference page for more information on how pfPipeWindows fit into the hierarchy of pfPipes, pfPipeWindows, and pfChannels.

pfPipeWindows are similar to pfWindows but are tracked/maintained by libpf and are needed by libpf to draw pfChannels. Because of their similarity, many of the pfPipeWindow routines are identical to pfWindow routines accept for the fact that the pfPWin<*> routines operate on a pfPipeWindow and the pfWin<*> routines operate on a pfWindow. These corresponding routines are listed in the table below and their functionality is documented in the pfWindow reference page.

pfPipeWindow routine	pfWindow routine
pfPWinAspect	pfWinAspect
pfPWinFBConfig	pfWinFBConfig
pfPWinFBConfigAttrs	pfWinFBConfigAttrs
pfPWinFBConfigData	pfWinFBConfigData
pfPWinFBConfigId	pfWinFBConfigId
pfPWinFullScreen	pfWinFullScreen
pfPWinGLCxt	pfWinGLCxt
pfPWinIndex	pfWinIndex
pfPWinMode	pfWinMode
pfPWinName	pfWinName
pfPWinOrigin	pfWinOrigin
pfPWinOriginSize	pfWinOriginSize
pfPWinOverlayWin	pfWinOverlayWin
pfPWinScreen	pfWinScreen
pfPWinShare	pfWinShare
pfPWinSize	pfWinSize
pfPWinStatsWin	pfWinStatsWin
pfPWinWSConnectionName	pfWinWSConnectionName
pfPWinWSDrawable	pfWinWSDrawable
pfPWinWSWindow	pfWinWSWindow
pfGetPWinAspect	pfGetWinAspect
pfGetPWinCurOriginSize	pfGetWinCurOriginSize
pfGetPWinCurScreenOriginSize	pfGetWinCurScreenOriginSize
pfGetPWinCurState	pfGetWinCurState
pfGetPWinCurWSDrawable	pfGetWinCurWSDrawable
pfGetPWinFBConfig	pfGetWinFBConfig
pfGetPWinFBConfigAttrs	pfGetWinFBConfigAttrs
pfGetPWinFBConfigData	pfGetWinFBConfigData
pfGetPWinFBConfigId	pfGetWinFBConfigId
pfGetPWinGLCxt	pfGetWinGLCxt
pfGetPWinIndex	pfGetWinIndex
pfGetPWinList	pfGetWinList
pfGetPWinMode	pfGetWinMode
pfGetPWinName	pfGetWinName
pfGetPWinOrigin	pfGetWinOrigin
pfGetPWinOverlayWin	pfGetWinOverlayWin
pfGetPWinScreen	pfGetWinScreen
pfGetPWinSelect	pfGetWinSelect
pfGetPWinShare	pfGetWinShare
pfGetPWinSize	pfGetWinSize

pfPipeWindow routine	pfWindow routine
pfGetPWinStatsWin	pfGetWinStatsWin
pfGetPWinType	pfGetWinType
pfGetPWinWSConnectionName	pfGetWinWSConnectionName
pfGetPWinWSDrawable	pfGetWinWSDrawable
pfGetPWinWSWindow	pfGetWinWSWindow
pfChoosePWinFBConfig	pfChooseWinFBConfig
pfAttachPWin	pfAttachWin
pfSelectPWin	pfSelectWin
pfSwapPWinBuffers	pfSwapWinBuffers
pfIsPWinOpen	pfIsWinOpen
pfQueryPWin	pfQueryWin
pfMQueryPWin	pfMQueryWin

pfNewPWin creates and returns a handle to a pfPipeWindow on the screen managed by *pipe*. Like other pfUpdatables, pfPipeWindows are always allocated from shared memory. The pipe of a pfPipeWindow cannot be changed. **pfGetPWinPipe** returns a pointer to the pfPipe of *pwin*. Like other pfObjects, pfPipeWindows must be created in the application process.

pfGetPWinClassType returns the **pfType*** for the class **pfPipeWindow**. The **pfType*** returned by **pfGetPWinClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfPipeWindow**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfWinFBConfigId allows you to directly set the OpenGL X visual id to be used in configuring the resulting OpenGL/X window. **pfGetWinFBConfigId** will return the current OpenGL visual id of the window (or -1 if the id is not known, or if running under IRIS GL). This routine is useful in multiprocess operation if you want to be able to directly specify the framebuffer configuration of an X window in the application process. See the **XVisualIDFromVisual(3X11)** and **XGetVisualInfo(3X11)** reference pages for more information about X visuals. This functionality is not supported under IRIS GL operation.

pfPWinScreen will set the screen of *pwin* and on the parent pfPipe. Once set, the screen cannot be changed. If the screen of the parent pfPipe had already been set when the pfPipeWindow was created, the pfPipeWindow will inherit that screen setting and will not accept another. The pfPipeWindow will direct all rendering comments to the hardware graphics pipeline specified by *screen*. As with pfWindows, if a screen is never set, the default screen of the current window system connection will be set as the screen when the window is opened with **pfOpenPWin::**. **pfGetPWinScreen** will return the screen of the pfPipeWindow. If the screen has not yet been set, (-1) will be returned. See the **pfGetCurWSConnection**

reference page for more information on the specification of a default screen. See the **pfPipeScreen** reference page for special restrictions and proper specification of pfPipe and pfPipeWindow screens in multipipe configurations.

pfPWinWSConnectionName allows you to specify the exact window server and default screen for the successive opening of the window. This can be used for specifying remote displays or on machines running more than one window server. **pfGetPWinWSConnectionName** will return the name specifying the current window server target. As with the setting of screens, a window server target specified on a pfPipe will take precedence over a target set on a pfPipeWindow. If a window server target is not specified for the parent pfPipe of a pfPipeWindow, the parent pfPipe will inherit the window setting. Because of these restrictions, this routine must be called in the application process, before the first call to **pfFrame**. See the **pfPipeScreen** reference page for special restrictions and proper specification of pfPipe and pfPipeWindow screens in multipipe configurations.

pfGetPWinIndex returns the index of *pwin* in the pfPipeWindow list of the parent pfPipe.

pfChannels are assigned to a pfPipeWindow upon their creation. pfPipeWindows also have list-style API for adding, removing, inserting, and reordering pfChannels on a pfPipeWindow: **pfAddChan** will append *chan* as the last pfChannel of *pwin*. **pfInsertChan** will insert *chan* as the *where*th pfChannel of *pwin*. **pfMoveChan** will move *chan* from its current position in the pfChannel list of *pwin* to position *where*. If *chan* does not belong to *pwin*, no action is taken and an error flag of (-1) is returned; otherwise, *where* is returned. **pfRemoveChan** will remove *chan* from *pwin*. If *chan* does not belong to *pwin*, no action is done and an error flag of (-1) is returned. Otherwise, the previous index of *chan* is returned. **pfGetChan** returns a pointer to the *index*th pfChannel of *pwin*. **pfGetNumChans** returns the number of pfChannels attached to *pwin*. **pfGetPWinChanIndex** returns the index of the *chan* in the channel list, or (-1) if the pfChannel is not attached to *pwin*.

pfClosePWin can be called from the application process to close a window. However, if additional draw process work is needed to be done, a **pfConfigPWin** draw process callback should be used.

pfConfigPWin, called from the application process, will trigger the configuration callback function to be called in the draw process for the current frame. If no user configuration callback function has been specified, a default configuration function will be called that will open and initialize *pwin*. **pfPWinConfigFunc**, called from the application process, specifies a draw process callback function, *func*, to configure *pwin*. The configure function can be used to make draw process calls to open, initialize, and close pfPipeWindows. In this window configuration callback function **pfOpenPWin** can be called on the pfPipeWindow, or an IRIS GL or OpenGL window can be created and assigned to the pfPipeWindow. **pfGetPWinConfigFunc** returns the pointer to the user-specified window configuration callback function, or NULL of no such function has been set.

pfOpenPWin will cause *pwin* to be opened and initialized via **pfInitGfx**. If called from the application process, the pfPipeWindow will be automatically opened in the draw process for the corresponding frame. If called in the draw process, the pfPipeWindow will be opened automatically. Similarly,

pfClosePWin and **pfClosePWinGL** can be called from either the application process or the draw process and will cause the *pwin* or the graphics context, respectively, to be closed in the draw process for the given frame. If application specific work needs to be done in the draw process for manipulating pfPipeWindows, **pfConfigPWin** should be used.

IRIS Performer automatically calls **pfInitGfx** for windows that it creates and opens. For pfPipeWindows, **pfInitGfx** does the same operations as for pfWindows, and in addition, will apply a default material and a default MODULATE texture environment (**pfApplyTEnv**), and enable backface culling (**pfCullFace**(-**PFCF_BACK**)).

pfPWinList can be used to specify a pfList of pfWindows, *wlist*, that can draw into a single pfPipeWindow. This enables a pfPipeWindow to maintain a list of alternate framebuffer configurations for the base pfPipeWindow. A pfPipeWindow always maintains a default main graphics pfWindow and a pfWindow list. Two of the windows in this list are so commonly needed that they have special names and can be created automatically for the user: OVERLAY and STATS. The user can also add his own pfWindows to the pfWindow list for additional configurations. This list may only hold pfWindows, NOT pfPipeWindows. With window lists, we have an effective pfWindow hierarchy of: screen->pfPipe->pfPipeWindow[graphics, stats, overlay, ...]->pfChannel(s). See the **pfWinList** reference page for more information on these alternate framebuffer configuration windows.

pfPWinIndex selects pfWindow *index* from the alternate configuration window list to be the current pfWindow the pfPipeWindow shall render to. All the pfChannels attached to the pfPipeWindow will automatically be drawn into this current pfWindow. See **pfWinIndex** for more details of this operation. **pfGetPWinIndex** will return the current index of the pfPipeWindow.

pfPWinType sets the type of a pfPipeWindow where *type* is an or-ed bitmask that may contain the type constants listed below. **pfGetPWinType** returns the type of a pfPipeWindow. A change in the type of a pfPipeWindow takes effect upon the call to **pfOpenPWin**. The type of an open pfPipeWindow cannot be changed. The pfWindow type attributes all start with **PFPWIN_TYPE_** and are:

PFPWIN_TYPE_X
: has identical characteristics to the **PFWIN_TYPE_X** specification for pfWindows. See the **pfWinType** reference page for more information.

PFPWIN_TYPE_SHARE
: Specifies that this window should be automatically attached to the first pfPipeWindow on the parent pfPipe. See the **pfAttachWin** reference page for more details.

PFPWIN_TYPE_STATS
: has identical characteristics to the **PFWIN_TYPE_STATS** specification for pfWindows. See the **pfWinType** reference page for more information.

Note that the pfWindow type settings of **PFWIN_TYPE_NOPORT** and **PFWIN_TYPE_OVERLAY** are not supported for pfPipeWindows. **pfGetPWinType** will return the type of *pwin*.

EXAMPLES

The following is an example of basic pfPipeWindow creation:

```
{ /* in the application process after pfConfig() */
                              pfPipeWindow *pw;
                              pw = pfNewPWin(pfGetPipe(0));
                              pfPWinName(pw,"PipeWin");
                              pfPWinOriginSize(pw, 0, 0, 500, 500);
                              pfPWinType(pw, PFPWIN_TYPE_X);
                              pfOpenPWin(pw);
                              /* set off the draw process to open window */
                              pfFrame();
}
```

If special draw process operations are to be done with the opening of the window, a pfConfigPWin callback function should be used.

```
{
                              /* in the application process pfPipeWindow init callback */
                              pfPWinConfigFunc(pw,OpenPipeWin);
                              /* trigger the draw process to call the config callback
                               * for this frame
                               */
                              pfConfigPWin(pw);
}
/* in the draw process pfPipeWindow init callback */
void OpenPipeWin(pfPipeWindow *pw)
{
                              pfOpenPWin(pw);
                              /* do other application specific draw process work,
                               * such as downloading scene textures, displaying
                               * welcome messages, etc.
                               */
}
```

The following is an example that shows the creation of multiple pfPipeWindows for a single pfPipe and the assignment of pfChannels to the different windows:

```
{
    pfChannel *chan[MAX_CHANS];
    pfPipeWindow *pwin[MAX_PWINS];
    pfPipe *p = pfGetPipe(0);
```

```
    for (int loop=0; loop < NumWins; loop++)
    {
        pfPipeWindow *pw;
        char str[PF_MAXSTRING];
        pwin[loop] = pfNewPWin(p);
        sprintf(str, "IRIS Performer - Win %d", loop);
        pfPWinName(pwin[loop], str);
        pfPWinOriginSize(pwin[loop], (loop&0x1)*315, ((loop&0x2)>>1)*340, 300, 300);
                            pfPWinConfigFunc(pwin[loop], OpenPipeWin);
        pfConfigPWin(pwin[loop]);
    }

    /* Create and configure a pfChannel for each pfPipeWindow. */
    for (int loop=0; loop < NumWins; loop++)
    {
        chan[loop] = pfNewChan(p);
        pfAddChan(pwin[loop], chan[loop]);
    }

    /* set off the draw process to config window */
    pfFrame();
}
```

pfOpenPWin and **pfClosePWin** can both be called from the application process, or from the draw process. The following example demonstrates using **pfConfigPWin** to close a pfPipeWindow:

```
{
    /* in the application process specify a close config func */
    pfPWinConfigFunc(pw,ClosePipeWin);
    pfConfigPWin(pw);

}

/* in the draw process pfPipeWindow init callback */
void ClosePipeWin(pfPipeWindow *pw)
{
    pfClosePWin(pw);
    /* do other application specific draw process calls */
}
```

NOTES

pfPipeWindows handle the multiprocessing details of IRIS Performer applications for pfWindows. pfPipeWindows must be created in the application process. However, with some minor exceptions, pfPipeWindows may be configured, opened, closed, and edited in either the application process or draw process. Typically, a pfPipeWindow is created and configured in the application process. Custom graphics state is initialized in a **pfPWinConfigFunc** callback function. The pfPipeWindow of a channel or a channel's position in a pfPipeWindow list may only be modified in the application process. The specification of the current drawing window with **pfSelectPWin** must be done in the drawing process. Explicit specification of the pfGLContext or pfFBConfig must be done in the drawing process. pfPipeWindow queries are also best done in the draw process as the query may have to access the graphics context to provide the requested information.

The following table shows from which process pfPipeWindow routines may be called.

pfPipeWindow routine	Application Process	Draw Process
pfNewPWin	Yes	No
pfPWinAspect	Yes	Yes
pfPWinConfigFunc	Yes	No
pfPWinFBConfig	Yes	No
pfPWinFBConfigAttrs	Yes	Yes
pfPWinFBConfigData	No	Yes
pfPWinFBConfigId	Yes	Yes
pfPWinFullScreen	Yes	Yes
pfPWinGLCxt	No	Yes
pfPWinIndex	Yes	Yes
pfPWinList	Yes	Yes
pfPWinMode	Yes	Yes
pfPWinName	Yes	Yes
pfPWinOrigin	Yes	Yes
pfPWinOriginSize	Yes	Yes
pfPWinOverlayWin	Yes	Yes
pfPWinScreen	Yes	Yes
pfPWinShare	Yes	Yes
pfPWinSize	Yes	Yes
pfPWinStatsWin	Yes	Yes
pfPWinType	Yes	Yes
pfPWinWSConnectionName	Yes	No
pfPWinWSDrawable	Yes	Yes
pfPWinWSWindow	Yes	Yes

pfPipeWindow routine	Application Process	Draw Process
pfAttachPWin	Yes	Yes
pfClosePWin	Yes	Yes
pfClosePWinGL	Yes	Yes
pfConfigPWin	Yes	Yes
pfOpenPWin	Yes	Yes
pfIsPWinOpen	Yes	Yes
pfMQueryPWin	No	Yes
pfQueryPWin	No	Yes
pfChoosePWinFBConfig	No	Yes
pfSelectPWin	No	Yes
pfSwapPWinBuffers	No	Yes
pfGetNumChans	Yes	Yes
pfAddChan	Yes	No
pfGetChan	Yes	Yes
pfInsertChan	Yes	No
pfMoveChan	Yes	No
pfRemoveChan	Yes	No

Note that whenever any pfObjects are given to a pfPipeWindow, such as **pfPWinList**, the data must be valid for access by the graphics process. This data, such as pfLists and pfWindows, should always be allocated from shared memory. Structures provided by X, such as that returned by **pfChoosePWinFBConfig**, or **pfChooseFBConfig**, will not have been allocated in shared memory. Therefore, those routines must be called from the draw process. Under OpenGL operation, **pfWinFBConfigId** can be used to set the framebuffer configuration of an X window in the application proceess.

pfPipeWindows support windows in the multiprocessed libpf environment and are the glue between pfChannels and pfPipes. There are times when you might want to use pfWindows, instead of pfPipeWindows, even in a libpf application. For example, popping up a simple dialog window in the draw process should use pfWindows and not pfPipeWindows. Additionally, if you want to maintain alternate windows with different visual (framebuffer) configurations for your pfPipeWindow, you use pfWindows that are alternate framebuffer configurations for the base pfPipeWindow. The PFWIN_STATS_WIN, PFWIN_OVERLAY_WIN, and other pfPWinList windows must themselves be pfWindows and not pfPipeWindows. See the pfPWinList routine below and the pfWindow man page for more information.

BUGS

pfPipeWindows cannot be deleted.

SEE ALSO

pfChannel, pfPipe, pfWindow, pfGetCurWSConnection, XGetVisualInfo, XVisualIDFromVisual

NAME

pfNewSCS, pfGetSCSClassType, pfGetSCSMat, pfGetSCSMatPtr – Create and get matrix for a static coordinate system node.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

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

pfType * **pfGetSCSClassType**(void);

void **pfGetSCSMat**(pfSCS *scs, pfMatrix mat);

const pfMatrix* **pfGetSCSMatPtr**(pfSCS *scs);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfSCS** is derived from the parent class **pfGroup**, so each of these member functions of class **pfGroup** are also directly usable with objects of class **pfSCS**. Casting an object of class **pfSCS** to an object of class **pfGroup** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfGroup**.

int **pfAddChild**(pfGroup *group, pfNode *child);
int **pfInsertChild**(pfGroup *group, int index, pfNode *child);
int **pfReplaceChild**(pfGroup *group, pfNode *old, pfNode *new);
int **pfRemoveChild**(pfGroup *group, pfNode* child);
int **pfSearchChild**(pfGroup *group, pfNode* child);
pfNode * **pfGetChild**(const pfGroup *group, int index);
int **pfGetNumChildren**(const pfGroup *group);
int **pfBufferAddChild**(pfGroup *group, pfNode *child);
int **pfBufferRemoveChild**(pfGroup *group, pfNode *child);

Since the class **pfGroup** is itself derived from the parent class **pfNode**, objects of class **pfSCS** can also be used with these functions designed for objects of class **pfNode**.

pfGroup * **pfGetParent**(const pfNode *node, int i);
int **pfGetNumParents**(const pfNode *node);
void **pfNodeBSphere**(pfNode *node, pfSphere *bsph, int mode);
int **pfGetNodeBSphere**(pfNode *node, pfSphere *bsph);
pfNode* **pfClone**(pfNode *node, int mode);
pfNode* **pfBufferClone**(pfNode *node, int mode, pfBuffer *buf);
int **pfFlatten**(pfNode *node, int mode);
int **pfNodeName**(pfNode *node, const char *name);
const char * **pfGetNodeName**(const pfNode *node);

pfNode* **pfFindNode**(pfNode *node, const char *pathName, pfType *type);
pfNode* **pfLookupNode**(const char *name, pfType* type);
int **pfNodeIsectSegs**(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void **pfNodeTravMask**(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint **pfGetNodeTravMask**(const pfNode *node, int which);
void **pfNodeTravFuncs**(pfNode* node, int which, pfNodeTravFuncType pre,
pfNodeTravFuncType post);
void **pfGetNodeTravFuncs**(const pfNode* node, int which, pfNodeTravFuncType *pre,
pfNodeTravFuncType *post);
void **pfNodeTravData**(pfNode *node, int which, void *data);
void * **pfGetNodeTravData**(const pfNode *node, int which);

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfSCS** can also be used with these functions designed for objects of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfSCS** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

scs identifies a pfSCS

DESCRIPTION

A pfSCS node represents a static coordinate system -- a modeling transform that cannot be changed once created. pfSCS nodes are similar to but less flexible than pfDCS nodes. What they lack in changeability they make up in performance.

pfNewSCS creates and returns a handle to a pfSCS. Like other pfNodes, pfSCSes are always allocated from shared memory and can be deleted using **pfDelete**.

pfNewSCS creates a pfSCS using *mat* as the transformation matrix.

By default a pfSCS uses a dynamic bounding volume so it is automatically updated when children are added, deleted or changed. This behavior may be changed using **pfNodeBSphere**. The bound for a pfSCS encompasses all B(i)**mat*, where B(i) is the bound for the child 'i' and *mat* is the transformation matrix of the pfSCS.

pfGetSCSClassType returns the **pfType*** for the class **pfSCS**. The **pfType*** returned by **pfGetSCSClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfSCS**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

The transformation of a pfSCS affects all its children. As the hierarchy is traversed from top to bottom, each new matrix is pre-multiplied to create the new transformation. For example, if SCSb is below SCSa in the scene graph, any geometry G below SCSa is transformed as G*SCSb*SCSa.

Static transformations represented by pfSCSes may be 'flattened' in a pre-processing step for improved intersection, culling, and drawing performance. **pfFlatten** accumulates transformations in a scene graph, applies them to geometry, and sets flattened pfSCSes to the identity matrix. Flattening is recommended when available memory and scene graph structure allow it. See **pfFlatten** for more details.

pfGetSCSMat copies the transformation matrix for *scs* into *mat*. For faster matrix access, **pfGetSCSMatPtr** returns a const pointer to *scs*'s matrix.

Both pre and post CULL and DRAW callbacks attached to a pfSCS (**pfNodeTravFuncs**) will be affected by the transformation represented by the pfSCS, i.e. - the pfSCS matrix will already have been applied to the matrix stack before the pre callback is called and will be popped only after the post callback is called.

SEE ALSO

pfGroup, pfLookupNode, pfFlatten, pfMatrix, pfNode, pfTraverser, pfDelete

NAME

pfNewScene, pfGetSceneClassType, pfSceneGState, pfGetSceneGState, pfSceneGStateIndex, pfGetSceneGStateIndex – Create a scene or root node, set and get scene pfGeoState or pfGeoState index.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

pfScene * **pfNewScene**(void);

pfType * **pfGetSceneClassType**(void);

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

pfGeoState * **pfGetSceneGState**(const pfScene *scene);

void **pfSceneGStateIndex**(pfScene *scene, int index);

int **pfGetSceneGStateIndex**(const pfScene *scene);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfScene** is derived from the parent class **pfGroup**, so each of these member functions of class **pfGroup** are also directly usable with objects of class **pfScene**. Casting an object of class **pfScene** to an object of class **pfGroup** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfGroup**.

int **pfAddChild**(pfGroup *group, pfNode *child);
int **pfInsertChild**(pfGroup *group, int index, pfNode *child);
int **pfReplaceChild**(pfGroup *group, pfNode *old, pfNode *new);
int **pfRemoveChild**(pfGroup *group, pfNode* child);
int **pfSearchChild**(pfGroup *group, pfNode* child);
pfNode * **pfGetChild**(const pfGroup *group, int index);
int **pfGetNumChildren**(const pfGroup *group);
int **pfBufferAddChild**(pfGroup *group, pfNode *child);
int **pfBufferRemoveChild**(pfGroup *group, pfNode *child);

Since the class **pfGroup** is itself derived from the parent class **pfNode**, objects of class **pfScene** can also be used with these functions designed for objects of class **pfNode**.

pfGroup * **pfGetParent**(const pfNode *node, int i);
int **pfGetNumParents**(const pfNode *node);
void **pfNodeBSphere**(pfNode *node, pfSphere *bsph, int mode);
int **pfGetNodeBSphere**(pfNode *node, pfSphere *bsph);
pfNode* **pfClone**(pfNode *node, int mode);
pfNode* **pfBufferClone**(pfNode *node, int mode, pfBuffer *buf);

int **pfFlatten**(pfNode *node, int mode);
int **pfNodeName**(pfNode *node, const char *name);
const char * **pfGetNodeName**(const pfNode *node);
pfNode* **pfFindNode**(pfNode *node, const char *pathName, pfType *type);
pfNode* **pfLookupNode**(const char *name, pfType* type);
int **pfNodeIsectSegs**(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void **pfNodeTravMask**(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint **pfGetNodeTravMask**(const pfNode *node, int which);
void **pfNodeTravFuncs**(pfNode* node, int which, pfNodeTravFuncType pre,
pfNodeTravFuncType post);
void **pfGetNodeTravFuncs**(const pfNode* node, int which, pfNodeTravFuncType *pre,
pfNodeTravFuncType *post);
void **pfNodeTravData**(pfNode *node, int which, void *data);
void * **pfGetNodeTravData**(const pfNode *node, int which);

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfScene** can also be used with these functions designed for objects of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfScene** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

scene identifies a pfScene.

gstate identifies a pfGeoState.

DESCRIPTION

A pfScene is the root of a hierarchical database which may be drawn or intersected with. pfScene is derived from pfGroup so it can use pfGroup and pfNode API. A pfScene may have children like a pfGroup but it cannot be a child of another node. Its special purpose is to serve as the root node of a scene graph.

pfNewScene creates and returns a handle to a pfScene. Like other pfNodes, pfScenes are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetSceneClassType returns the **pfType*** for the class **pfScene**. The **pfType*** returned by **pfGetSceneClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfScene**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

IRIS Performer will automatically carry out the APP, CULL, and DRAW traversals on pfScenes which are attached to pfChannels by **pfChanScene**. The CULL and DRAW traversals are directly or indirectly triggered by **pfFrame** while the APP traversal is triggered by **pfAppFrame**.

Multiple pfChannels may reference the same pfScene but each pfChannel references only a single pfScene.

pfSceneGState attaches *gstate* to *scene*. The pfGeoState of a pfScene defines the "global state" which may be inherited by other pfGeoStates. This state inheritance mechanism is further described in the pfGeoState man page.

The scene pfGeoState is defined as the global state by **pfLoadGState**. This pfGeoState will be loaded before the pfChannel DRAW callback (**pfChanTravFunc**) is invoked so any custom rendering in the callback will inherit the state set by the scene pfGeoState. **pfGetSceneGState** returns the directly referenced pfGeoState of *scene* or the appropriate pfGeoState in the global table if *scene* indexes its pfGeoState or NULL if the index cannot be resolved.

The scene pfGeoState may be indexed through a global table by assigning an index with **pfSceneGStateIndex** and specifying the table with **pfApplyGStateTable**. Usually this table is provided by the pfChannel (**pfChanGStateTable**). **pfGetSceneGStateIndex** returns the pfGeoState index of *scene* or -1 if *scene* directly references its pfGeoState.

It is not necessary to provide a scene pfGeoState, but it is a convenient way to specify the default inheritable values for all pfGeoState elements on a per-scene basis.

SEE ALSO

pfApplyGStateTable, pfChanScene, pfChanTravFunc, pfGeoState, pfGroup, pfDelete

NAME

pfNewSeq, pfGetSeqClassType, pfSeqTime, pfGetSeqTime, pfSeqInterval, pfGetSeqInterval, pfSeqDuration, pfGetSeqDuration, pfSeqMode, pfGetSeqMode, pfGetSeqFrame – Control animation sequence nodes.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

pfSequence * **pfNewSeq**(void);

pfType * **pfGetSeqClassType**(void);

void **pfSeqTime**(pfSequence *seq, int frame, double time);

double **pfGetSeqTime**(const pfSequence *seq, int frame);

void **pfSeqInterval**(pfSequence *seq, int mode, int begin, int end);

void **pfGetSeqInterval**(const pfSequence *seq, int *mode, int *begin, int *end);

void **pfSeqDuration**(pfSequence *seq, float speed, int nReps);

void **pfGetSeqDuration**(const pfSequence *seq, float *speed, int *nReps);

void **pfSeqMode**(pfSequence *seq, int mode);

int **pfGetSeqMode**(const pfSequence *seq);

int **pfGetSeqFrame**(const pfSequence *seq, int *repeat);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfSequence** is derived from the parent class **pfGroup**, so each of these member functions of class **pfGroup** are also directly usable with objects of class **pfSequence**. Casting an object of class **pfSequence** to an object of class **pfGroup** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfGroup**.

int **pfAddChild**(pfGroup *group, pfNode *child);
int **pfInsertChild**(pfGroup *group, int index, pfNode *child);
int **pfReplaceChild**(pfGroup *group, pfNode *old, pfNode *new);
int **pfRemoveChild**(pfGroup *group, pfNode* child);
int **pfSearchChild**(pfGroup *group, pfNode* child);
pfNode * **pfGetChild**(const pfGroup *group, int index);
int **pfGetNumChildren**(const pfGroup *group);
int **pfBufferAddChild**(pfGroup *group, pfNode *child);
int **pfBufferRemoveChild**(pfGroup *group, pfNode *child);

Since the class **pfGroup** is itself derived from the parent class **pfNode**, objects of class **pfSequence** can also be used with these functions designed for objects of class **pfNode**.

```
pfGroup *       pfGetParent(const pfNode *node, int i);
int             pfGetNumParents(const pfNode *node);
void            pfNodeBSphere(pfNode *node, pfSphere *bsph, int mode);
int             pfGetNodeBSphere(pfNode *node, pfSphere *bsph);
pfNode*         pfClone(pfNode *node, int mode);
pfNode*         pfBufferClone(pfNode *node, int mode, pfBuffer *buf);
int             pfFlatten(pfNode *node, int mode);
int             pfNodeName(pfNode *node, const char *name);
const char *    pfGetNodeName(const pfNode *node);
pfNode*         pfFindNode(pfNode *node, const char *pathName, pfType *type);
pfNode*         pfLookupNode(const char *name, pfType* type);
int             pfNodeIsectSegs(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void            pfNodeTravMask(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint            pfGetNodeTravMask(const pfNode *node, int which);
void            pfNodeTravFuncs(pfNode* node, int which, pfNodeTravFuncType pre,
                    pfNodeTravFuncType post);
void            pfGetNodeTravFuncs(const pfNode* node, int which, pfNodeTravFuncType *pre,
                    pfNodeTravFuncType *post);
void            pfNodeTravData(pfNode *node, int which, void *data);
void *          pfGetNodeTravData(const pfNode *node, int which);
```

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfSequence** can also be used with these functions designed for objects of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
void*   pfGetUserData(pfObject *obj);
```

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfSequence** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *        pfGetType(const void *ptr);
int             pfIsOfType(const void *ptr, pfType *type);
int             pfIsExactType(const void *ptr, pfType *type);
const char *    pfGetTypeName(const void *ptr);
int             pfRef(void *ptr);
int             pfUnref(void *ptr);
int             pfUnrefDelete(void *ptr);
int             pfGetRef(const void *ptr);
int             pfCopy(void *dst, void *src);
```

```
int         pfDelete(void *ptr);
int         pfCompare(const void *ptr1, const void *ptr2);
void        pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *      pfGetArena(void *ptr);
```

PARAMETERS

seq identifies a pfSequence.

DESCRIPTION

A pfSequence is a pfGroup that sequences through a range of its children, drawing each child for a certain length of time. Its primary use is for animations, where a *sequence* of objects or geometry (children) represent a desired visual event. **pfNewSeq** creates and returns a handle to a pfSequence. Like other pfNodes, pfSequences are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetSeqClassType returns the **pfType*** for the class **pfSequence**. The **pfType*** returned by **pfGetSeqClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfSequence**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

Children are added to a pfSequence using normal pfGroup API (**pfAddChild**). The length of time that a child is drawn is specified by **pfSeqTime**. *frame* is the index of a child that should be drawn for *time* seconds. If *frame* < 0, then all children will be displayed for *time* seconds. If *time* = 0.0 or time is not specified for a particular child, then it will not be drawn at all. If *time* < 0.0 the sequence will pause at child *frame* and draw it repeatedly until the sequence is resumed or stopped (see **pfSeqMode** below). **pfGetSeqTime** returns the time for frame *frame*.

pfSeqInterval specifies the interval or range of frames (children) to sequence. *begin* and *end* specify the beginning and ending indexes of *seq* respectively. Indexes are inclusive and should be in the range 0, numChildren - 1. An index < 0 is equivalent to numChildren - 1 for convenience. *end* may be less than *begin* for reverse sequences. The default sequence interval is 0, numChildren - 1.

mode specifies how *seq* is sequenced over the range from *begin* to *end* if it is a repeating sequence.

PFSEQ_CYCLE
: *seq* will go from *begin* to *end* then restart at *begin*.

PFSEQ_SWING
: *seq* will go back and forth from *begin* to *end*. The endpoint frames are drawn only once when the swing changes directions.

The default *mode* is **PFSEQ_CYCLE**. **pfGetSeqInterval** copies the interval parameters into *mode*, *begin*, and *end*.

pfSeqDuration controls the duration of an sequence. *speed* divides the time that each sequence frame is

displayed. Values < 1.0 slow down the sequence while values > 1.0 speed up the sequence. The default *speed* is 1.0. *nReps* is the number of times *seq* repeats before stopping. If *nReps* is < 0, *seq* will sequence indefinitely and if == 0 the sequence is disabled. If *nReps* is > 1, *seq* will sequence for *nReps* cycles or swings depending on the sequencing mode set by **pfSeqInterval**.

The number of repetitions for both **PFSEQ_CYCLE** and **PFSEQ_SWING** is increased by 1 every time an endpoint of the sequence is reached. Therefore **PFSEQ_CYCLE** begins to repeat itself after 1 repetition while **PFSEQ_SWING** repeats itself after 2 repetitions. Note that for 1 repetition, both modes are equivalent.

The default value for *nReps* is 1. **pfGetSeqDuration** copies the duration parameters into *speed* and *nReps*.

pfSeqMode controls the run-time execution of *seq*. *mode* is a symbolic token:

PFSEQ_START
Restarts the sequence from its beginning. Once started, a sequence may be stopped, paused, or started again in which case it is restarted from its beginning.

PFSEQ_STOP
Stops the sequence. After an sequence is stopped, it is reset so that further executions of the sequence begin from the starting index.

PFSEQ_PAUSE
Pauses the sequence without resetting it. When paused, the current child will be drawn until the sequence is either stopped or resumed.

PFSEQ_RESUME
Resumes a paused sequence.

Sequences are evaluated once per frame by **pfAppFrame**. The time used in the evaluation is that set by **pfFrameTimeStamp**. This time is automatically set by **pfFrame** but it may be overridden by the application to account for varying latency due to non-constant frame rates.

pfGetSeqMode returns the mode of *seq*. The mode will automatically be set to **PFSEQ_STOP** if the sequence completes the number of repetitions set by **pfSeqDuration**.

pfGetSeqFrame returns the index of the child which *seq* is currently drawing and also copies the number of repetitions it has completed into *repeat*.

SEE ALSO

pfAppFrame, pfFrame, pfFrameTimeStamp, pfGroup, pfNode, pfDelete

NAME

pfNewSwitch, pfGetSwitchClassType, pfSwitchVal, pfGetSwitchVal – Create, modify, and query a switch node.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

pfSwitch * **pfNewSwitch**(void);

pfType * **pfGetSwitchClassType**(void);

int **pfSwitchVal**(pfSwitch *sw, int val);

int **pfGetSwitchVal**(const pfSwitch *sw);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfSwitch** is derived from the parent class **pfGroup**, so each of these member functions of class **pfGroup** are also directly usable with objects of class **pfSwitch**. Casting an object of class **pfSwitch** to an object of class **pfGroup** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfGroup**.

int **pfAddChild**(pfGroup *group, pfNode *child);
int **pfInsertChild**(pfGroup *group, int index, pfNode *child);
int **pfReplaceChild**(pfGroup *group, pfNode *old, pfNode *new);
int **pfRemoveChild**(pfGroup *group, pfNode* child);
int **pfSearchChild**(pfGroup *group, pfNode* child);
pfNode * **pfGetChild**(const pfGroup *group, int index);
int **pfGetNumChildren**(const pfGroup *group);
int **pfBufferAddChild**(pfGroup *group, pfNode *child);
int **pfBufferRemoveChild**(pfGroup *group, pfNode *child);

Since the class **pfGroup** is itself derived from the parent class **pfNode**, objects of class **pfSwitch** can also be used with these functions designed for objects of class **pfNode**.

pfGroup * **pfGetParent**(const pfNode *node, int i);
int **pfGetNumParents**(const pfNode *node);
void **pfNodeBSphere**(pfNode *node, pfSphere *bsph, int mode);
int **pfGetNodeBSphere**(pfNode *node, pfSphere *bsph);
pfNode* **pfClone**(pfNode *node, int mode);
pfNode* **pfBufferClone**(pfNode *node, int mode, pfBuffer *buf);
int **pfFlatten**(pfNode *node, int mode);
int **pfNodeName**(pfNode *node, const char *name);
const char * **pfGetNodeName**(const pfNode *node);

pfNode* **pfFindNode**(pfNode *node, const char *pathName, pfType *type);
pfNode* **pfLookupNode**(const char *name, pfType* type);
int **pfNodeIsectSegs**(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void **pfNodeTravMask**(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint **pfGetNodeTravMask**(const pfNode *node, int which);
void **pfNodeTravFuncs**(pfNode* node, int which, pfNodeTravFuncType pre,
pfNodeTravFuncType post);
void **pfGetNodeTravFuncs**(const pfNode* node, int which, pfNodeTravFuncType *pre,
pfNodeTravFuncType *post);
void **pfNodeTravData**(pfNode *node, int which, void *data);
void * **pfGetNodeTravData**(const pfNode *node, int which);

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfSwitch** can also be used with these functions designed for objects of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfSwitch** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

sw identifies a pfSwitch.

DESCRIPTION

A pfSwitch is an interior node in the IRIS Performer node hierarchy that selects one, all, or none of its children. It is derived from pfGroup so it can use pfGroup API to manipulate its child list.

pfNewSwitch creates and returns a handle to a pfSwitch. Like other pfNodes, pfSwitches are always

allocated from shared memory and can be deleted using **pfDelete**.

pfGetSwitchClassType returns the **pfType*** for the class **pfSwitch**. The **pfType*** returned by **pfGetSwitchClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfSwitch**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfSwitchVal sets the switch value of *sw* to *val*. *val* may be an integer ranging from 0 to N-1 with N being the number of children of *sw* or it may be a symbolic token: **PFSWITCH_ON** or **PFSWITCH_OFF** in which case all children or no children are selected. **pfGetSwitchVal** returns the current switch value.

The validity of the switch value delayed until the switch is actually evaluated (usually by a traversal such as CULL). For example, it is legal to set a switch value of 2 on a pfSwitch node with no children, provided at least 2 children are added before the pfSwitch is evaluated.

NOTES

PF_ON and **PF_OFF** tokens will NOT work with **pfSwitchVal**.

SEE ALSO

pfGroup, pfLookupNode, pfNode, pfScene, pfDelete

NAME

pfNewText, pfGetTextClassType, pfAddString, pfRemoveString, pfInsertString, pfReplaceString, pfGetString, pfGetNumStrings – Create, modify, and query a 3D text node.

FUNCTION SPECIFICATION

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

pfText *    pfNewText(void);
pfType *    pfGetTextClassType(void);
int         pfAddString(pfText* text, pfString* string);
int         pfRemoveString(pfText* text, pfString* str);
int         pfInsertString(pfText* text, int index, pfString* str);
int         pfReplaceString(pfText* text, pfString* old, pfString* new);
pfString *  pfGetString(const pfText* text, int index);
int         pfGetNumStrings(const pfString* string);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfText** is derived from the parent class **pfNode**, so each of these member functions of class **pfNode** are also directly usable with objects of class **pfText**. Casting an object of class **pfText** to an object of class **pfNode** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfNode**.

```
pfGroup *    pfGetParent(const pfNode *node, int i);
int          pfGetNumParents(const pfNode *node);
void         pfNodeBSphere(pfNode *node, pfSphere *bsph, int mode);
int          pfGetNodeBSphere(pfNode *node, pfSphere *bsph);
pfNode*      pfClone(pfNode *node, int mode);
pfNode*      pfBufferClone(pfNode *node, int mode, pfBuffer *buf);
int          pfFlatten(pfNode *node, int mode);
int          pfNodeName(pfNode *node, const char *name);
const char * pfGetNodeName(const pfNode *node);
pfNode*      pfFindNode(pfNode *node, const char *pathName, pfType *type);
pfNode*      pfLookupNode(const char *name, pfType* type);
int          pfNodeIsectSegs(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
void         pfNodeTravMask(pfNode *node, int which, uint mask, int setMode, int bitOp);
uint         pfGetNodeTravMask(const pfNode *node, int which);
void         pfNodeTravFuncs(pfNode* node, int which, pfNodeTravFuncType pre,
                 pfNodeTravFuncType post);
void         pfGetNodeTravFuncs(const pfNode* node, int which, pfNodeTravFuncType *pre,
                 pfNodeTravFuncType *post);
```

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

Since the class **pfNode** is itself derived from the parent class **pfObject**, objects of class **pfText** can also be used with these functions designed for objects of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfText** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

text identifies a pfText.

string identifies a pfString.

DESCRIPTION

A pfText is analogous to a **pfGeode**. A pfText encapsulates pfStrings in a scene graph as a **pfGeode** encapsulates **pfGeoSet**s. A pfText is a leaf node in the IRIS Performer scene graph hierarchy and is derived from **pfNode** so it can use **pfNode** API. A pfText is simply a list of pfStrings.

The bounding volume of a pfText is that which surrounds all its pfStrings. Unless the bounding volume is considered static (see **pfNodeBSphere**), IRIS Performer will compute a new volume when the list of pfStrings is modified by **pfAddString**, **pfRemoveString**, **pfInsertString** or **pfReplaceString**. If the bounding box of a child pfString changes, call **pfNodeBSphere** to tell IRIS Performer to update the bounding volume of the pfText.

pfNewText creates and returns a handle to a pfText. Like other pfNodes, pfTexts are always allocated from shared memory and can be deleted using **pfDelete**.

pfGetTextClassType returns the **pfType*** for the class **pfText**. The **pfType*** returned by **pfGetTextClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfText**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfAddString appends *str* to *text*'s pfString list. **pfRemoveString** removes *str* from the list and shifts the list down over the vacant spot. For example, if *str* had index 0, then index 1 becomes index 0, index 2 becomes index 1 and so on. **pfRemoveString** returns a 1 if *str* was actually removed and 0 if it was not found in the list. **pfAddString** and **pfRemoveString** will cause IRIS Performer to recompute new bounding volumes for *text* unless it is configured to use static bounding volumes.

pfInsertString will insert *str* before the pfString with index *index*. *index* must be within the range 0 to **pfGetNumStrings**(*text*). **pfReplaceString** replaces *old* with *new* and returns 1 if the operation was successful or 0 if *old* was not found in the list. **pfInsertString** and **pfReplaceString** will cause IRIS Performer to recompute new bounding volumes for *text* unless it is configured to use static bounding volumes.

pfGetNumStrings returns the number of pfStrings in *text*. **pfGetString** returns a handle to the pfString with index *index* or **NULL** if the index is out of range.

Here is a sample code snippet demonstrating how to use pfText, pfFont, and pfString to add 3D text to a scene graph:

```
/* Initialize Performer and create pfScene "scene" */

/* Get shared memory arena */
arena = pfGetSharedArena();

/* Append standard directories to Performer search path, PFPATH */
pfFilePath(".:/usr/share/Performer/data");

/* Create 3D message and place in scene. */
text = pfNewText();
pfAddChild(scene, text);
if (pfFindFile("Times-Elfin.of", path, R_OK))
{
    str = pfNewString(arena);
    pfStringMode(str, PFSTR_DRAWSTYLE, PFSTR_EXTRUDED);
    pfStringMode(str, PFSTR_JUSTIFY, PFSTR_MIDDLE);
    pfStringColor(str, 1.0f, 0.0f, 0.8f, 1.0f);
    pfStringString(str, "Welcome to IRIS Performer");
    pfFlattenString(str);
```

```
        pfAddString(text, str);
    }
    else
    {
        pfNotify(PFNFY_WARN,PFNFY_PRINT,"Couldn't find font file.");
        exit(0);
    }
```

SEE ALSO

pfGeoSet, pfNode, pfNodeTravFuncs, pfString, pfFont, pfDelete

NAME

pfCullResult, pfGetCullResult, pfGetParentCullResult, pfGetTravChan, pfGetTravMat, pfGetTravNode, pfGetTravIndex, pfGetTravPath – Set and get traversal masks, callback functions and callback data, and get pfTraverser attributes.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

void	**pfCullResult**(int result);
int	**pfGetCullResult**(void);
int	**pfGetParentCullResult**(void);
pfChannel *	**pfGetTravChan**(const pfTraverser *trav);
void	**pfGetTravMat**(const pfTraverser *trav, pfMatrix mat);
pfNode *	**pfGetTravNode**(const pfTraverser *trav);
int	**pfGetTravIndex**(const pfTraverser *trav);
const pfPath *	**pfGetTravPath**(const pfTraverser *trav);

```
typedef int (*pfNodeTravFuncType)(pfTraverser *trav, void *userData);
```

PARAMETERS

node identifies a pfNode

which identifies the traversal: **PFTRAV_ISECT**, **PFTRAV_APP**, **PFTRAV_CULL** or **PFTRAV_DRAW**, denoting the intersection, application, cull or draw traversals respectively.

DESCRIPTION

IRIS Performer provides four major traversals: intersection, application, cull, and draw that are often abbreviated as **ISECT**, **APP**, **CULL**, and **DRAW**. A traversal is typically an in-order traversal of a directed acyclic graph of pfNodes otherwise known as a subgraph. The actual traversal method, traverser structure, and traversal initiation used depends on the traversal type as well as the multiprocessing mode as shown in the following table.

Traversal	Traverser	Traversee	Trigger
PFTRAV_ISECT	pfSegSet	subgraph	pfNodeIsectSegs(), pfChanNodeIsectSegs
PFTRAV_APP	pfTraverser	pfScene	pfApp()
CULL_DL_DRAW is set			
PFTRAV_CULL	pfChannel	pfScene	pfCull()
PFTRAV_DRAW	pfChannel	pfDispList	pfDraw()
CULL_DL_DRAW is not set			
PFTRAV_CULL	pfChannel	pfScene	pfDraw()
PFTRAV_DRAW			

Typical traversal callback usage:

ISECT
Collision detection, terrain following, line of sight

APP Application-specific behavior, motors

CULL
Custom level-of-detail selection, culling

DRAW
Custom rendering

When **PFMPCULL_DL_DRAW** is not set in the multiprocessing mode argument to **pfMultiprocess** (and the cull and draw stages are in the same process), then **pfDraw** simultaneously culls and draws the pfScene attached to the pfChannel by **pfChanScene**. Otherwise, **pfCull** culls and builds up a pfDispList which is later rendered by **pfDraw**.

If the traversal CULL mask and node CULL mask AND to zero at a node, the CULL traversal disables view culling and trivially accepts the node and all its descendents. Note that unlike other traversals, a mask result of 0 does not prune the node.

If the traversal DRAW mask and node DRAW mask AND to zero at a node, the CULL traversal prunes the node, so descendents are neither CULL-traversed nor drawn.

If the traversal APP mask and the node APP mask AND to zero, the APP traversal prunes the node and its descendents.

If the ISECT masks AND to zero, the ISECT traversal prunes the node. The intersection mask is typically used to control traversals of different types of objects, e.g. different bits may indicate ground, water, foliage, and buildings, so they may be intersected selectively. See (**pfNodeTravMask**).

In many respects a traversal appears to the user as an atomic action. The user configures a traverser, triggers it with the appropriate routine and awaits the results. Node callbacks are supported to provide user extensibility and configuration into this scenario. They are user-supplied routines that are invoked in the course of a traversal. Callbacks return a value which can control traversal on a coarse-grained basis. In addition, draw callbacks can render custom geometry and cull callbacks can substitute custom culling for the default IRIS Performer culling.

The pre- or post-callbacks for the cull and intersection traversals may return **PFTRAV_CONT**, **PFTRAV_PRUNE**, **PFTRAV_TERM** to indicate that traversal should continue normally, skip this node or terminate the traversal, respectively. **PFTRAV_PRUNE** is equivalent to **PFTRAV_CONT** for the post-callback. Currently, the return value from the draw callbacks is ignored.

pfCullResult, **pfGetCullResult**, and **pfGetParentCullResult** can all be called in the pre-cull callback and all but **pfCullResult** may be called in the post-cull callback. **pfGetCullResult** returns the result of the cull for the node that the cull callback is associated with. **pfGetParentCullResult** returns the cull result for the parent of the node that the cull callback is associated with. When called within the pre-cull callback, **pfCullResult** specifies the result of cull for the node that the pre-cull callback is associated with. This essentially replaces default IRIS Performer cull processing with user-defined culling. *result* is a token which specifies the result of the cull test and should be one of:

PFIS_FALSE
Node is entirely outside the viewing frustum and should be pruned.

PFIS_MAYBE | PFIS_TRUE
Node is partially inside the viewing frustum and the children of the node should be cull-tested.

PFIS_MAYBE | PFIS_TRUE | PFIS_ALL_IN
Node is totally inside the viewing frustum so all the children of the node should be trivially accepted without further cull testing.

If **pfCullResult** is not called within the pre-cull callback, IRIS Performer will use its default geometric culling mechanism that compares node bounding volumes to the current culling frustum to determine if the node may be within view.

In the post-cull callback **pfGetCullResult** will return the result of the cull set by **pfCullResult** or the result of the default cull if **pfCullResult** was not called.

The evaluation order of the cull and draw traversal masks and callbacks is illustrated in the following pseudo-code:

Example 1: Cull and draw traversal mask and callback evaluation order.

```
/* Return if draw mask test fails */
if ((drawMask & nodeDrawMask) == 0)
    return PFTRAV_CONT;

/* Call pre-cull callback */
if (preCull)
{
    rtn = (*preCull)(traverser, cullData);

    if (rtn == PFTRAV_PRUNE)
        return PFTRAV_CONT;
    else if (rtn == PFTRAV_TERM)
        return PFTRAV_TERM;
}

/* Disable view culling if cull mask test fails */
if ((cullMask & nodeCullMask) == 0)
    disableViewCulling();

/* Perform default culling if pfCullResult was not called */
if (!userCalledpfCullResultInThePreCullCallback)
    cullResult = cullTest(node);

if (cullResult == PFIS_FALSE)
{
    /* Call post-cull callback */
    if (postCull)
    {
        rtn = (*postCull)(traverser, cullData);

        if (rtn == PFTRAV_PRUNE)
            return PFTRAV_CONT;
        else if (rtn == PFTRAV_TERM)
            return PFTRAV_TERM;
    }
    return PFTRAV_CONT;
}
else
/* Trivially accept node and all its children */
if (cullResult == PFIS_ALL_IN)
    disableViewCulling();

/* Call pre-draw callback */
```

```
if (preDraw)
    (*preDraw)(traverser, drawData);

evaluateNodeAndItsChildren();

/* Call post-draw callback */
if (postDraw)
    (*postDraw)(traverser, drawData);

/* Call post-cull callback */
if (postCull)
{
    rtn = (*postCull)(traverser, cullData);

    if (rtn == PFTRAV_PRUNE)
        return PFTRAV_CONT;
    else
    if (rtn == PFTRAV_TERM)
        return PFTRAV_TERM;
}

return PFTRAV_CONT;
```

Example 2: Use of DRAW callbacks to save and restore state.

```
extern int
preDraw(pfTraverser *trav, void *data)
{
    pfPushState();
    pfEnable(PFEN_TEXGEN);
    pfApplyTGen((pfTexGen*)data);

    return PFTRAV_CONT;
}

extern int
postDraw(pfTraverser *trav, void *data)
{
    pfPopState();

    return PFTRAV_CONT;
}
```

```
/*
 * Set up draw callbacks and user data to draw 'geode' in
 * EYE_LINEAR texgen mode.
 */


pfTexGen                  *tgen;

tgen = pfNewTGen(pfGetSharedArena());

pfTGenMode(tgen, PF_S, PFTG_EYE_LINEAR);
pfTGenMode(tgen, PF_T, PFTG_EYE_LINEAR);

pfNodeTravFuncs(geode, PFTRAV_DRAW, preDraw, postDraw);
pfNodeTravData(geode, PFTRAV_DRAW, tgen);
```

libpr graphics calls like **pfApplyTGen** should be made in a DRAW callback only. Specifically, **libpr** graphics calls made in a CULL callback are not legal and have undefined behavior.

The intersection, application, cull and draw callbacks are passed a pfTraverser which can be used to query the channel, current transformation matrix and current node. **pfGetTravChan** returns the current channel for the cull, and draw traversal. It returns the current channel for intersection traversals initiated with **pfChanNodeIsectSegs** and **NULL** for intersection traversals initiated with **pfNodeIsectSegs**.

pfGetTravMat sets *mat* to the current transformation matrix, which is the concatenation of the matrices from the root of the scene down to and including the current node. Since no transformation hierarchy is retained in the draw process, in a draw callback, the current matrix should be queried using the **getmatrix** or **pfGetModelMat**/**pfGetViewMat** routines.

pfGetTravNode returns the current node being traversed and **pfGetTravIndex** returns the child index of the current node, i.e.- the index of the current node in its parent's list of children. **pfGetTravPath** returns a pointer to the list of nodes which defines the path from the scene graph root to the current node.

NOTES

The post-cull callback is a good place to implement custom level-of-detail mechanisms.

BUGS

The path returned by **pfGetTravPath** is valid only when invoked from a cull callback.

SEE ALSO

pfAddChild, pfClone, pfFrame, pfNode

libpr

libpr is a low-level library for high-performance graphics applications.

This library provides a wide range of functions useful functions including optimized rendering, graphics state management, math functions, and shared memory utilities.

NAME

pfAlphaFunc, pfGetAlphaFunc – Specify alpha function and reference value

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfAlphaFunc**(float ref, int mode);

void **pfGetAlphaFunc**(float *ref, int *mode);

PARAMETERS

ref is a reference value with which to compare source alpha at each pixel. This value should be a float in the range 0 through 1.

mode is a symbolic constant that specifies the conditional comparison that source alpha and *ref* must pass for a pixel to be drawn.

DESCRIPTION

pfAlphaFunc sets the alpha function mode and reference value which affects all subsequent geometry. *mode* is a symbolic constant that specifies the conditional comparison that source alpha and *ref* must pass for a pixel to be drawn. For example:

```
if (source alpha mode ref)
    draw the pixel
```

where the alpha value boolean function *mode* is be one of:

PFAF_ALWAYS
PFAF_EQUAL
PFAF_GEQUAL
PFAF_GREATER
PFAF_LEQUAL
PFAF_LESS
PFAF_NEVER
PFAF_NOTEQUAL
PFAF_OFF

If it was desired to only draw pixels whose alpha value was greater than or equal to 50% of the representable range, then a *mode* of **PFAF_GEQUAL** and a *ref* of 0.5 would produce the hardware pixel rendering conditional:

```
if (source alpha PFAF_GEQUAL 0.5)
    draw the pixel
```

The the default *mode* is **PFAF_OFF** and default *ref* value is 0. The alpha function and reference value state elements are identified by the **PFSTATE_ALPHAFUNC** and **PFSTATE_ALPHAREF** tokens respectively. Use these tokens with **pfGStateMode** and **pfGStateVal**, to set the alpha function and reference value of a

pfGeoState and with **pfOverride** to override subsequent alpha function and reference value changes.

Here is an example:

```
/*
 * Setup pfGeoState so that only pixels whose alpha is > 40
 * are drawn once the pfGeoState is applied with pfApplyGState.
 */
pfGStateMode(gstate, PFSTATE_ALPHAFUNC, PFAF_GREATER);
pfGStateVal(gstate, PFSTATE_ALPHAREF, (40.0f/255.0f));

/*
 * Override alpha function.  The alpha reference value can still
 * be changed.
 */
pfOverride(PFSTATE_ALPHAFUNC, PF_ON);

/*
 * All subsequent attempts to set alpha function will be ignored
 * until pfOverride is called to unlock it.
 */
```

pfAlphaFunc is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfAlphaFunc** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

pfGetAlphaFunc copies the current alpha function reference value and mode into *ref* and *mode* respectively.

NOTES

pfAlphaFunc is typically used for textures with alpha that simulate trees and other complicated geometry having many holes. See the IRIS GL **afunction(3g)** or OpenGL **glAlphaFunc** manual page for further details.

SEE ALSO

afunction, glAlphaFunc, pfDispList, pfGeoState, pfState

NAME

pfAntialias, pfGetAntialias – Specify antialiasing mode

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfAntialias**(int mode);

int **pfGetAntialias**(void);

PARAMETERS

mode is a symbolic constant and is one of:

PFAA_OFF	Antialiasing will be disabled.
PFAA_ON	Antialiasing will be enabled. The antialiasing mechanism used depends on the machine type.

DESCRIPTION

pfAntialias sets the hardware antialiasing mode. Geometry drawn subsequent to calling **pfAntialias** will be antialiased according to *mode*. The antialiasing mechanism used is machine-dependent: multisampling on RealityEngine systems and non-multisampling on all others. In addition, if available, **pfAntialias** will enable a special hardware mode that efficiently renders points using multisampled circles rather than squares. See the IRIS GL **multisample(3g)** reference page and the **SGIS_multisample** section of the OpenGL **glIntro(3g)** reference page for more detailed information on multisampled antialiasing.

If *mode* is **PFAA_ON**, then antialiasing will be enabled. On machines which do not support multisampling, **PFAA_ON** will enable line and point antialiasing. Polygons will not be antialiased. In this case it is recommended that **pfAntialias** be enabled only for points and lines since it may reduce the speed of polygon rendering.

In pure IRIS GL windows (not GLX), the framebuffer will be reconfigured as needed and as possible to support multisampling. Since **pfAntialias** may configure hardware buffers, it is best called at initialization time for performance reasons. On RealityEngine systems, multisample buffers are configured and multisampling is enabled if the combination of Video Output Format and Raster Manager count support multisampling. Specifically, **pfAntialias** will attempt to configure the IRIS GL window with 12 bit color buffers, 8 subsamples, 24 bits of depth buffer, and 4 bits of stencil. If this is not available, 1 bit of stencil will be used. Non-multisample buffers, configured by such IRIS GL calls as **zbsize(3g)** and **stensize(3g)** are all deallocated. If the hardware configuration does not support 8 subsamples then **pfAntialias** will attempt to acquire 4 subsamples.

If *mode* is **PFAA_OFF**, for pure IRIS GL windows, **pfAntialias** will deallocate all multisample buffers and allocate non-multisample buffers accordingly: 12-bit color buffer, 32 bit depth buffer, 4 bit stencil buffer.

X windows cannot have their framebuffer resources reconfigured. X windows for both IRIS GL and OpenGL are, by default, created with multisample buffers if they are available in the current hardware configuration. The default configuration, if available will be 8 subsamples, 24 bits of depth buffer, and 4

bits of stencil. If this is not available, 1 bit of stencil will be used, and then 4 subsamples will be allocated if 8 are not still available. The exact framebuffer configuration of windows can be specified via **pfWinFBConfigAttrs**.

For X windows, if *mode* is **PFAA_OFF**, the antialiasing mode will be disabled but the buffers cannot be deallocated and there might be associated framebuffer operations that are not truly disabled. Because of this, the full performance benefit expected by turning off antialiasing may not be achieved.

The antialiasing mode state element is identified by the **PFSTATE_ANTIALIAS** token. Use this token with **pfGStateMode** to set the antialiasing mode of a pfGeoState and with **pfOverride** to override subsequent antialiasing mode changes.

pfAntialias is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfAntialias** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

pfGetAntialias returns the current antialiasing mode.

Example 1:

```
/* Set up 'antialiased' pfGeoState */
pfGStateMode(gstate, PFSTATE_ANTIALIAS, PFAA_ON);

/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/* Draw antialiased gset */
pfDrawGSet(gset);
```

Example 2:

```
/* Override antialiasing mode to PFAA_OFF */
pfAntialias(PFAA_OFF);
pfOverride(PFSTATE_ANTIALIAS, PF_ON);
```

NOTES

pfQueryFeature can be used to determined what features are available on the current hardware configuration. **pfQuerySys** can be used to query the exact extent of hardware resources, such as number of subsamples available for multisampling.

When using antialiasing without multisampling, blending is used which may conflict with other transparency modes. Specifically, all geometry will be blended which may cause artifacts and may

substantially reduce performance. For this reason **pfAntialias** should be used with discretion on all but RealityEngine systems.

For pure IRIS GL windows, since **pfAntialias** may configure hardware buffers, it is best called at initialization time for performance reasons.

In the default framebuffer configurations, the 4 bit of stencil buffer is allocated to support depth complexity fill statistics; see the **pfStats** reference man page for more information. 1 bit of stencil is required for the support of high quality decals; see the **pfDecal** reference page for more information.

Not all machines support stencil planes and in these cases, stencil bits will not be allocated. Indy platforms under IRIS GL operation do not support stencil. Additionally, the Extreme graphics platforms only support stencil with reduced depth buffer resolution and so stencil will not be allocated by default.

Under OpenGL operation, if a window has been configured with multisample buffers, the state of pfAntialias() is used internally to track whether or not multisampling is being done. This knowledge is used for doing the fast TAG clear **pfClear()**, and for drawing multisampled points. IRIS Performer will not detect a GL call made to enable or disable multisampling so if you do this you must return state to match IRIS Performer's internal state or the results will be undefined.

SEE ALSO

blendfunction, glBlendFunc, linesmooth, glHint(GL_LINE_SMOOTH_HINT), pntsmooth, glHint(GL_POINT_SMOOTH_HINT), mssize, multisample, glIntro, pfQueryFeature, pfQuerySys, pfWindow, pfChooseFBConfig, pfDispList, pfGeoState, pfOverride, pfState

NAME

pfMakeEmptyBox, pfBoxExtendByPt, pfBoxExtendByBox, pfBoxAroundPts, pfBoxAroundBoxes, pfBoxAroundSpheres, pfBoxAroundCyls, pfBoxContainsPt, pfBoxContainsBox, pfBoxIsectSeg, pfXformBox – Operate on axis-aligned bounding boxes

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfMakeEmptyBox**(pfBox *box);

void **pfBoxExtendByPt**(pfBox *dst, const pfVec3 pt);

void **pfBoxExtendByBox**(pfBox *dst, const pfBox *box);

void **pfBoxAroundPts**(pfBox *dst, const pfVec3 *pts, int npt);

void **pfBoxAroundBoxes**(pfBox* dst, const pfBox **boxes, int nbox);

void **pfBoxAroundSpheres**(pfBox *dst, const pfSphere **sphs, int nsph);

void **pfBoxAroundCyls**(pfBox *dst, const pfCylinder **sphs, int ncyl);

int **pfBoxContainsPt**(const pfBox* box, const pfVec3 pt);

int **pfBoxContainsBox**(const pfBox *box1, const pfBox *box2);

int **pfBoxIsectSeg**(const pfBox* box, const pfSeg* seg, float* d1, float* d2);

void **pfXformBox**(pfBox *dst, const pfBox *box, const pfMatrix xform);

```
typedef struct
{
    pfVec3      min;
    pfVec3      max;
} pfBox;
```

DESCRIPTION

A pfBox is an axis-aligned box which can be used for intersection tests and for maintaining bounding information about geometry. A box represents the axis-aligned hexahedral volume: (x, y, z) where min[0] <= x <= max[0], min[1] <= y <= max[1] and min[2] <= z <= max[2]. pfBox is a public struct whose data members *min* and *max* may be operated on directly.

pfMakeEmptyBox sets *dst* to appear empty to extend operations.

pfBoxExtendByPt extends the size of box *dst* to include the point *pt*.

pfBoxExtendByBox extends the size of box *dst* to include the box *box*.

pfBoxAroundPts, **pfBoxAroundBoxes pfBoxAroundCyls** and **pfBoxAroundSpheres** set *dst* to be an axis-aligned box encompassing the given primitives. *npt*, *nbox*, *ncyls* and *nsph* are the number of points, boxes, and spheres in the respective primitive lists.

pfBoxContainsPt returns **TRUE** or **FALSE** depending on whether the point *pt* is in the interior of the specified box.

The return value from **pfBoxContainsBox** is the OR of one or more bit fields. The returned value may be:

PFIS_FALSE:
The intersection of the second box argument and the first box is empty.

PFIS_MAYBE:
The intersection of the second box argument and the first box might be non-empty.

PFIS_MAYBE | PFIS_TRUE:
The intersection of the second box argument and the first box is definitely non-empty.

PFIS_MAYBE | PFIS_TRUE | PFIS_ALL_IN:
The second box argument is non-empty and lies entirely inside the first box.

pfBoxIsectSeg intersect the line segment *seg* with the volume of an axis-aligned box *box*. The possible return values include all of the above as well as:

PFIS_FALSE:
seg lies entirely in the exterior.

PFIS_MAYBE | PFIS_TRUE | PFIS_START_IN:
The starting point of *seg* lies in the interior.

PFIS_MAYBE | PFIS_TRUE | PFIS_END_IN:
The ending point of *seg* lies in the interior.

PFIS_MAYBE | PFIS_TRUE | PFIS_ALL_IN | PFIS_START_IN | PFIS_END_IN:
Both end points of *seg* lie in the interior.

If *d1* and *d2* are non-NULL, on return from **pfBoxIsectSeg** they contain the starting and ending positions of the line segment (0 <= d1 <= d2 <= seg->length) intersected with the specified volume.

pfXformBox sets *dst* to a box which contains *box* as transformed by the matrix *xform*, i.e. a box around (*box* * *xform*). Because transformed boxes must be axis-aligned, most rotations cause the box to grow, and the transformation is not reversed by the inverse rotation.

NOTES

The bit fields returned by the contains functions are structured so that bitwise AND-ing the results of sequential tests can be used to compute composite results, e.g. testing exclusion against a number of half spaces.

Because pfBoxes are axially aligned, they tend to grow when transformed. Hence, they are best for static geometry or other cases in which the bounding geometry does not need to be transformed.

SEE ALSO

pfSeg, pfSphere

NAME

pfClear – Clear specified graphics buffers

FUNCTION SPECIFICATION

void **pfClear**(int which, const pfVec4 color);

PARAMETERS

which is a mask that specifies which buffers are to be cleared. *which* is a bitwise OR of:

PFCL_COLOR	Clear color buffer to *color*.
PFCL_DEPTH	Clear depth buffer to maximum value of our defined depth range.
PFCL_MSDEPTH	Fast clear of the multisample depth buffer.
PFCL_STENCIL	Clear stencil buffer to 0.
PFCL_DITHER	Enable dithering during the color clear. By default, **pfClear** turns off dithering for color clears.

color specifies the red, green, blue, and alpha components of the color buffer clear color. Each component is defined in the range 0.0 to 1.0. If *color* is NULL then a black fully opaque color will be used.

DESCRIPTION

pfClear clears the buffers specified by *which* in the current graphics window. The actual screen area cleared depends on many GL state settings including viewport and screen or scissor mask (IRIS GL **scrmask** or OpenGL **glScissor**), current draw buffer (front, back, left, right, overlay, etc.), and the existence of a depth buffer for **PFCL_DEPTH** and stencil buffer for **PFCL_STENCIL**. See the IRIS GL **clear(3g)** or OpenGL **glClear(3g)** reference page for more details.

If *which* includes **PFCL_COLOR** and *color* is NULL, then any selected color buffer will be cleared to black fully opaque pixels using **cpack(0xff000000)** in IRIS GL and **glColor4f(0,0,0,1)** in OpenGL.

PFCL_MSDEPTH has effect only when multisampling (See **pfAntialias**). In this case, instead of writing the maximum depth value into each individual pixel subsample, each pixel is "tagged" as having the maximum depth value. This clear is much faster than a full depth buffer clear; however, the color buffer is not cleared so results from previous frames will be left in the color buffer if not redrawn. This requires that each pixel in the viewport be covered by geometry. Often this is accomplished by drawing one or more large background polygons (often textured) at the far clip plane to "clear" the framebuffer to an interesting background rather than depth buffer and then incurring the additional cost of clearing drawing background polygons. This requires that the background rendering disable depth buffer testing (e.g. **zfunction(ZF_ALWAYS)** in IRIS GL or **glDepthFunc(GL_ALWAYS)** in OpenGL). Otherwise, a normal depth buffer clear will be required if multisampling is not in use or not supported in the current framebuffer configuration. Note that the background drawing should leave depth buffering enabled so that it's depth values will be written.

The follow example shows how to clear all buffers with one **pfClear** call:

```
/*
 * Clear color buffer to black, depth buffer to the maximum depth value,
 * and stencil buffer to 0.
 */
pfClear(PFCL_DEPTH | PFCL_COLOR | PFCL_STENCIL, NULL);
```

pfClear is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfClear** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

NOTES

PFCL_MSDEPTH is only available on RealityEngine systems, and then only in the multisample antialiasing mode. For performance reasons, the depth buffer for the entire window rather than just the current viewport is cleared with OpenGL on Indy, i.e. scissoring is disabled. Also, Indy depth buffer clears are significantly slower under IRIS GL than under OpenGL.

SEE ALSO

pfAntialias, pfDispList, glClear, glDepthFunc, clear, multisample, gconfig, zclear, zfunction, czclear

NAME

pfNewCtab, pfGetCtabClassType, pfGetCtabSize, pfApplyCtab, pfCtabColor, pfGetCtabColor, pfGetCtabColors, pfGetCurCtab – Specify color table properties

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfColortable * **pfNewCtab**(int size, void *arena);

pfType * **pfGetCtabClassType**(void);

int **pfGetCtabSize**(const pfColortable *ctab);

void **pfApplyCtab**(pfColortable *ctab);

int **pfCtabColor**(pfColortable *ctab, int index, pfVec4 color);

int **pfGetCtabColor**(const pfColortable *ctab, int index, pfVec4 color);

pfVec4 * **pfGetCtabColors**(const pfColortable *ctab);

pfColortable * **pfGetCurCtab**(void);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfColortable** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfColortable**. Casting an object of class **pfColortable** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfColortable** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);

```
int         pfDelete(void *ptr);
int         pfCompare(const void *ptr1, const void *ptr2);
void        pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *      pfGetArena(void *ptr);
```

DESCRIPTION

A pfColortable is a 'color indexing' mechanism used by pfGeoSets. It is *not* related to the graphics library hardware rendering notion of *color index mode*. If pfColortable operation is enabled, pfGeoSets will be drawn with the colors defined in the current globally active pfColortable rather than using the pfGeoSet's own local color list. This facility can be used for instant large-scale color manipulation of geometry in a scene.

pfNewCtab creates and returns a handle to a pfColortable. *arena* specifies a malloc arena out of which the pfColortable is allocated or **NULL** for allocation off the process heap. *size* is the number of pfVec4 color elements to allocate for the pfColortable. pfColortables can be deleted with **pfDelete**.

The number of color elements in the pfColortable is returned by **pfGetCtabSize**.

pfGetCtabClassType returns the **pfType*** for the class **pfColortable**. The **pfType*** returned by **pfGetCtabClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfColortable**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType**to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfApplyCtab selects *ctab* as the current, global pfColortable. If colorindex mode is enabled (**pfEnable**(-**PFEN_COLORTABLE**)), then all subsequent pfGeoSets will use the pfVec4 array supplied by the global color table rather than their own local color array. Colorindex mode works for both indexed and non-indexed pfGeoSets.

pfApplyCtab is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfApplyCtab** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

pfGetCurCtab returns the currently active pfColortable or **NULL** if there is none active.

Colors in a pfColortable are pfVec4's which specify red, green, blue, and alpha in the range [0..1]. **pfCtabColor** and **pfGetCtabColor** respectively set and get the color at index *index*. To support high performance manipulation of colortables, IRIS Performer allows direct access to the array of pfVec4 colors of a pfColortable. **pfGetCtabColors** returns a pointer to this array which may be manipulated directly. However care must be taken not to write data outside the array limits.

The pfColortable state element is identified by the **PFSTATE_COLORTABLE** token. Use this token with **pfGStateAttr** to set the pfColortable of a pfGeoState and with **pfOverride** to override subsequent

colortable changes.

Example 1:

```
/* Set up 'colorindexed' pfGeoState */
pfGStateAttr(gstate, PFSTATE_COLORTABLE, ctab);
pfGStateMode(gstate, PFSTATE_ENCOLORTABLE, PF_ON);

/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/* Draw gset colorindexed with ctab */
pfDrawGSet(gset);
```

Example 2:

```
pfEnable(PFEN_COLORTABLE);
pfApplyCtab(ctab);

/*
 * Override active pfColortable to 'ctab' and colorindex enable
 * to PF_ON.
 */
pfOverride(PFSTATE_COLORTABLE | PFSTATE_ENCOLORTABLE, PF_ON);
```

NOTES

pfColortables can be used to simulate FLIR (Forward Looking Infrared) and NVG (Night Vision Goggles) and for monochrome display devices which separate video components for stereo display purposes. More flexible FLIR and NVG simulation is available through the use of indexed pfGeoStates.

SEE ALSO

pfDelete, pfDispList, pfEnable, pfGeoSet, pfGeoState, pfOverride, pfState

NAME

pfCullFace, pfGetCullFace – Specify face culling mode

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfCullFace**(int mode);

int **pfGetCullFace**(void);

PARAMETERS

mode is a symbolic constant and is one of:

PFCF_OFF	Face culling is off,
PFCF_BACK	Polygons that are back-facing will be culled.
PFCF_FRONT	Polygons that are front-facing will be culled.
PFCF_BOTH	Polygons that are front and back-facing will be culled.

DESCRIPTION

pfCullFace sets the face culling mode used to cull all subsequent polygons. A polygon is considered to be backfacing if its vertices are in clockwise order (screen coordinates). Frontfacing polygon vertex ordering is counterclockwise.

pfGetCullFace returns the current face culling mode.

The face culling mode state element is identified by the **PFSTATE_CULLFACE** token. Use this token with **pfGStateMode** to set the face culling mode of a pfGeoState and with **pfOverride** to override subsequent face culling mode changes.

pfCullFace is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfCullFace** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

Example 1:

```
/* Set up 'face-culled' pfGeoState */
pfGStateMode(gstate, PFSTATE_CULLFACE, PFCF_BACK);

/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/* Draw face-culled gset */
pfDrawGSet(gset);
```

Example 2:

```
/* Override face culling mode to PFCF_OFF */
pfCullFace(PFCF_OFF);
pfOverride(PFSTATE_CULLFACE, PF_ON);
```

NOTES

Backface culling with **pfCullFace**(**PFCF_BACK**) can significantly improve performance for "solid" databases whose polygons are oriented consistently and where objects are closed. With these databases you cannot see backfacing polygons since they are always obscured by nearer front-facing ones. The graphics hardware can quickly reject backfacing polygons so use of backface culling is *strongly* encouraged to increase performance.

Face culling should be disabled when using two-sided lighting, since the two-sided lighting is only useful for distinguishing backfacing objects.

SEE ALSO

backface, frontface, pfDispList, pfGeoState, pfOverride, pfState

NAME

pfNewCBuffer, pfGetCBufferClassType, pfGetCurCBufferData, pfGetCBufferCMem, pfCBufferChanged, pfInitCBuffer, pfCBufferConfig, pfGetCBufferConfig, pfCBufferFrame, pfGetCBufferFrameCount, pfGetCurCBufferIndex, pfCurCBufferIndex, pfGetCBuffer, pfGetCMemFrame, pfGetCMemCBuffer, pfGetCMemClassType – Create, initialize, manage pfCycleBuffer and pfCycleMemory memory

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfCycleBuffer * **pfNewCBuffer**(size_t nbytes, void *arena);

pfType * **pfGetCBufferClassType**(void);

void * **pfGetCurCBufferData**(const pfCycleBuffer *cbuf);

pfCycleMemory * **pfGetCBufferCMem**(const pfCycleBuffer *cbuf, int index);

void **pfCBufferChanged**(pfCycleBuffer *cbuf);

void **pfInitCBuffer**(pfCycleBuffer *cbuf, void *data);

int **pfCBufferConfig**(int numBuffers);

int **pfGetCBufferConfig**(void);

int **pfCBufferFrame**(void);

int **pfGetCBufferFrameCount**(void);

int **pfGetCurCBufferIndex**(void);

void **pfCurCBufferIndex**(int index);

pfCycleBuffer * **pfGetCBuffer**(void *data);

int **pfGetCMemFrame**(const pfCycleMemory *cmem);

pfCycleBuffer * **pfGetCMemCBuffer**(pfCycleMemory *cmem);

pfType * **pfGetCMemClassType**(void);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfCycleBuffer** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfCycleBuffer**. Casting an object of class **pfCycleBuffer** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfCycleBuffer** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

DESCRIPTION

Together, pfCycleBuffer and pfCycleMemory provide an automated mechanism for managing dynamic data in a pipelined, multiprocessing environment. In this kind of environment, data is typically modified at the head of the pipeline and must propagate down it in a "frame-accurate" fashion. For example, assume the coordinates of a pfGeoSet are modified for facial animation. If a two-stage rendering pipeline is used, then it is likely that the coordinates will be modified in the head of the pipeline, at the same time they are being rendered in the tail of the pipeline. If only a single memory buffer is used, then the pfGeoSet might be rendered when its coordinates are only partially updated, potentially resulting in cracks in the facial mesh or other anomalies.

A solution to this problem is to use two memory buffers for the coordinates, one written to by the head and one read from by the tail of the pipeline. In order for the new coordinates to propagate to the rendering stage we could copy the newly updated buffer into the renderer's buffer during a handshake period between the two stages. However, if the buffer is large, the copy time could become objectionable. Another alternative is to simply swap pointers to the two buffers - the classic "double-buffering" approach. This is much more efficient but requires that the contents of the buffer be completely updated each frame. Otherwise the render stage will access a "stale" buffer that represents the facial expression at a previous time so that the animation will appear to go backwards.

The pfCycleBuffer/pfCycleMemory combination supports efficient dynamic data management in an N-stage pipeline. A pfCycleBuffer logically contains multiple pfCycleMemorys. Each process has a global index which selects the currently active pfCycleMemory in each pfCycleBuffer. This index can be advanced once a frame by **pfCurCBufferIndex** so that the buffers "cycle". By advancing the index appropriately in each pipeline stage, dynamic data can be frame-accurately propagated down the

pipeline.

While pfCycleBuffers can be used for generic dynamic data, a prominent use is as attribute arrays for pfGeoSets. **pfGSetAttr** accepts pfCycleBuffer memory for attribute arrays and the pfGeoSet will index the appropriate pfCycleMemory when rendering and intersection testing. Currently, pfGeoSets do not support pfCycleBuffer index lists.

pfNewCBuffer returns a pfCycleBuffer allocated out of *arena* or off the heap if *arena* is **NULL**. The argument *nbytes* specifies the length of each associated pfCycleMemory. pfCycleBuffers can be deleted with **pfDelete**.

The number of pfCycleMemorys allocated for each pfCycleBuffer is specified by **pfCBufferConfig** which is typically called only once at initialization time. **pfGetCBufferConfig** returns the number set by **pfCBufferConfig**.

pfGetCBufferClassType returns the **pfType*** for the class **pfCycleBuffer**. The **pfType*** returned by **pfGetCBufferClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfCycleBuffer**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfGetCMemClassType returns the **pfType*** for the class **pfCycleMemory**.

pfInitCBuffer initializes all pfCycleMemorys of *cbuf* to the data referenced by *data*. *data* should be at least of size *nbytes*.

pfCycleMemory is derived from pfMemory and also provides access to its raw data in the form of a void* pointer through the **pfGetData** call. Thus pfCycleBuffer memory is arranged in a hierarchy: pfCycleBuffer -> pfCycleMemory -> void* and various routines exist which convert one handle into another. These routines are listed in the following table.

	pfCycleBuffer*	pfCycleMemory*	void*
pfCycleBuffer*	NA	pfGetCBufferCMem	pfGetCurCBufferData
pfCycleMemory*	pfGetCMemCBuffer	NA	pfGetData
void*	pfGetCBuffer	pfGetMemory	NA

The currently active pfCycleMemory portion of a pfCycleBuffer is selected by the global index specified by **pfCurCBufferIndex**, and is returned by **pfGetCurCBufferIndex.** One can think of this in pseudocode as

```
current pfCycleMemory = pfCycleBuffer[pfCurCBufferIndex]
```

Thus one should always get a new handle to the currently active data whenever the global index changes. Data modification that is incremental, (such as a += .2) must retain a handle to the previous data for proper results (current a = previous a + .2).

As mentioned above, cycling buffer pointers is efficient but requires that the buffers be completely updated each frame. If the data at some time becomes static, it must then be copied into those buffers that are out of date. pfCycleBuffer supports this copying automatically with **pfCBufferChanged** in conjunction with **pfCBufferFrame**. **pfCBufferFrame** advances a global frame counter that is used to frame-stamp pfCycleMemorys. After *cbuf* has been updated, **pfCBufferChanged** frame-stamps *cbuf* with the current frame count. Then if *cbuf* is not changed in a later frame, **pfCBufferFrame** will automatically copy the latest pfCycleMemory into its currently active, sibling pfCycleMemory. This copying will continue until all selected pfCycleMemorys contain the latest data. **pfGetCMemFrame** returns the frame stamp of *cmem*. **pfGetCBufferFrameCount** returns the current, global, pfCycleBuffer frame count.

The following are examples of pfCycleBuffer usage for **libpr**-only and **libpf** applications. When using **libpf**, **pfConfig** and **pfFrame** call **pfCBufferConfig** and **pfCBufferFrame** respectively so the application should not call the latter routines. In addition, **libpf** calls **pfCurCBufferIndex** in each process so that pfCycleBuffer changes are properly propagated down the processing pipelines.

Example 1: **libpr**-only pfCycleBuffer example

```
pfVec3                                *prevVerts, *curVerts;

/*
 * Configure number of pfCycleMemorys per pfCycleBuffer
*/
pfCBufferConfig(numBuffers);

verts = pfNewCBuffer(sizeof(pfVec3) * numVerts, arena);
pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX, verts, NULL);

while(!done)
{
    static int  index = 0;

    pfCurCBufferIndex(index);

    curVerts = pfGetCurCBufferData(verts);

    /* Compute new positions of mass-spring system */
```

```
    for (i=0; i<numVerts; i++)
                curVerts[i] = prevVerts[i] + netForceVector * deltaTime;

    /* Indicate that 'verts' has changed */
    pfCBufferChanged(verts);

    prevVerts = curVerts;

    /* Advance cyclebuffer frame count */
    pfCBufferFrame();

    /* Advance buffer index. */
    index = (index + 1) % numBuffers;
}
```

Example 2: **libpf** pfCycleBuffer example

```
pfVec3                              *prevVerts, *curVerts;

pfInit();

pfMultiprocess(mpMode);

/*
 * This calls pfCBufferConfig() with the number of buffers
 * appropriate to the multiprocessing mode.
*/
pfConfig();

verts = pfNewCBuffer(sizeof(pfVec3) * numVerts, pfGetSharedArena());
pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX, verts, NULL);

while(!done)
{
    curVerts = pfGetCurCBufferData(verts);

    /* Compute new positions of mass-spring system */
    for (i=0; i<numVerts; i++)
                curVerts[i] = prevVerts[i] + netForceVector * deltaTime;
```

```
        /* Indicate that 'verts' has changed */
        pfCBufferChanged(verts);

        prevVerts = curVerts;

        /* This calls pfCBufferFrame() */
        pfFrame();
    }
```

NOTES

The global index which selects the currently active pfCycleMemory is unique for a given address space. Specifically, share group processes like those spawned by **sproc** will share the same global index.

SEE ALSO

pfDelete, pfGetData, pfGetMemory, pfMemory

NAME

pfMakeEmptyCyl, pfCylAroundSegs, pfCylAroundPts, pfCylAroundBoxes, pfCylAroundSpheres, pfCylExtendByBox, pfCylExtendBySphere, pfCylExtendByCyl, pfCylContainsPt, pfCylIsectSeg, pfOrthoXformCyl – Operations on cylinder definitions.

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfMakeEmptyCyl**(pfCylinder *cyl);

void **pfCylAroundSegs**(pfCylinder *dst, const pfSeg **segs, int nseg);

void **pfCylAroundPts**(pfCylinder *dst, const pfVec3 *pts, int npt);

void **pfCylAroundBoxes**(pfCylinder *dst, const pfBox **boxes, int nbox);

void **pfCylAroundSpheres**(pfCylinder *dst, const pfSphere **sphs, int nsph);

void **pfCylExtendByBox**(pfCylinder *dst, const pfBox *box);

void **pfCylExtendBySphere**(pfCylinder *dst, const pfSphere *sph);

void **pfCylExtendByCyl**(pfCylinder *dst, const pfCylinder *cyl);

int **pfCylContainsPt**(const pfCylinder* cyl, const pfVec3 pt);

int **pfCylIsectSeg**(const pfCylinder* cyl, const pfSeg* seg, float* d1, float* d2);

void **pfOrthoXformCyl**(pfCylinder *dst, const pfCylinder *cyl, const pfMatrix xform);

```
typedef struct
{
    pfVec3   center;
    float    radius;
    pfVec3   axis;
    float    halfLength;
} pfCylinder;
```

DESCRIPTION

A pfCylinder represents a cylinder of finite length. The routines listed here provide means of creating and extending cylinders for use as bounding geometry around groups of line segments. The cylinder is defined by its *center*, *radius*, *axis* and *halfLength*. The routines assume *axis* is a vector of unit length, otherwise results are undefined. pfCylinder is a public struct whose data members *center*, *radius*, *axis* and *halfLength* may be operated on directly.

pfMakeEmptyCyl sets *dst* so that it appears empty to other operations.

pfCylAroundSegs, **pfCylAroundPts**, **pfCylAroundSpheres** and **pfCylAroundBoxes** set *dst* to a cylinder

which contains a set of line segments, points, spheres or boxes, respectively. These routines are passed the address of an array of pointers to the objects to be encompassed along with the number of objects.

pfCylExtendByBox, **pfCylExtendBySphere**, and **pfCylExtendByCyl** set *dst* to a cylinder which contains both the pfSphere *dst* and the box *box*, the sphere *sph* or the cylinder *cyl*, respectively.

pfCylContainsPt returns **TRUE** or **FALSE** depending on whether the point *pt* is in the interior of the specified pfCylinder.

pfCylIsectSeg intersects the line segment *seg* with the volume of the cylinder *cyl*. The possible return values are:

PFIS_FALSE:
seg lies entirely in the exterior.

PFIS_MAYBE | PFIS_TRUE | PFIS_START_IN:
The starting point of *seg* lies in the interior.

PFIS_MAYBE | PFIS_TRUE | PFIS_END_IN:
The ending point of *seg* lies in the interior.

PFIS_MAYBE | PFIS_TRUE | PFIS_ALL_IN | PFIS_START_IN | PFIS_END_IN:
Both end points of *seg* lie in the interior.

If *d1* and *d2* are non-NULL, on return from **pfCylIsectSeg** they contain the starting and ending positions of the line segment (0 <= d1 <= d2 <= seg->length) intersected with the cyl.

pfOrthoXformCyl sets *dst* to the cylinder *cyl* transformed by the orthogonal transformation *xform*. *dst* = *cyl* * *xform*. If *xform* is not an orthogonal transformation the results are undefined.

SEE ALSO

pfBox, pfCylinder, pfSeg, pfSphere, pfVec3

NAME

pfCreateDPool, pfGetDPoolSize, pfGetDPoolName, pfReleaseDPool, pfAttachDPool, pfDPoolAlloc, pfDPoolFree, pfDPoolFind, pfDPoolLock, pfDPoolSpinLock, pfDPoolUnlock, pfDPoolTest, pfDPoolAttachAddr, pfGetDPoolAttachAddr, pfGetDPoolClassType – Create, control and allocate from locked memory pools.

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfDataPool*	**pfCreateDPool**(uint size, char *name);
int	**pfGetDPoolSize**(pfDataPool *dpool);
const char*	**pfGetDPoolName**(pfDataPool* dpool);
int	**pfReleaseDPool**(pfDataPool *dpool);
pfDataPool*	**pfAttachDPool**(char *name);
volatile void*	**pfDPoolAlloc**(pfDataPool *dpool, uint size, int id);
int	**pfDPoolFree**(pfDataPool *dpool, void *dpmem);
volatile void*	**pfDPoolFind**(pfDataPool *dpool, int id);
int	**pfDPoolLock**(void *dpmem);
int	**pfDPoolSpinLock**(void* dpmem, int spins, int block);
void	**pfDPoolUnlock**(void *dpmem);
int	**pfDPoolTest**(void *dpmem);
void	**pfDPoolAttachAddr**(void *addr);
void*	**pfGetDPoolAttachAddr**(void);
pfType *	**pfGetDPoolClassType**(void);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfDataPool** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfDataPool**. Casting an object of class **pfDataPool** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void	**pfUserData**(pfObject *obj, void *data);
void*	**pfGetUserData**(pfObject *obj);
int	**pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfDataPool** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *      pfGetType(const void *ptr);
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

PARAMETERS

dpool identifies a pfDataPool.

DESCRIPTION

A pfDataPool is similar to a shared memory malloc arena but adds the ability to lock/unlock pfDataPool memory for multiprocessing applications. The datapool functions allow related or unrelated processes to share data and provide a means for locking data blocks to eliminate data collision. These functions use the shared arena functions (see **usinit**).

pfCreateDPool creates and returns a handle to a pfDataPool. *size* is the size in bytes of the pfDataPool. *name* is the name of the pfDataPool and is also the name of the memory-mapped file used by the pfDataPool. This file is created in the directory "/usr/tmp" unless the environment variable **PFTMPDIR** is defined, in which case the file is created in the directory named in the **PFTMPDIR** environment variable. *name* should be unique among all pfDataPool names and only a single process should create a given pfDataPool with name *name*.

pfGetDPoolClassType return the **pfType*** for the class **pfDataPool**. The **pfType*** returned is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfDataPool**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfGetDPoolSize and **pfGetDPoolName** return the size in bytes and the string name of *dpool* respectively.

pfAttachDPool allows the calling process to attach to a pfDataPool with name *name* that may have been created by another process. A handle to the found pfDataPool is returned or **NULL** if it was not found or could not be accessed.

pfReleaseDPool hides *dpool* so that no other processes may attach to it although all previously attached processes may still access it. Additionally, a released pfDataPool will be removed from the file system

(deleted) once all attached processes exit. **pfReleaseDPool** returns **TRUE** if successful and **FALSE** otherwise.

pfDPoolAlloc returns a pointer to a block of memory of *size* bytes that was allocated out of *dpool* or **NULL** if there is not enough available memory in *dpool*. *size* is in bytes and can range from 1 to the size of the pfDataPool. The actual size allocated is always rounded up to the next 16 byte boundary. *id* is an integer id assigned to the block of memory that is used to reference it by **pfDPoolFind**. Block id's should be unique or the results are undefined.

pfDPoolFind returns a pointer to a block of pfDataPool memory which is identified by *id* or **NULL** if *id* was not found. The calling process must be attached to the datapool memory.

pfDPoolFree frees the memory block previously allocated by **pfDPoolAlloc** and makes it available to be reallocated.

pfDPoolLock, **pfDPoolSpinLock** and **pfDPoolUnlock** lock and unlock access to a block of pfDataPool memory that was allocated by **pfDPoolAlloc**. When the lock cannot be acquired, **pfDPoolLock** yields the processor causing the current thread to block until the lock is available. **pfDPoolSpinLock** provides more control by accepting arguments to control the spinning and blocking. When *block* is *FALSE*, **pfDPoolSpinLock** returns rather than yielding the processor if the lock cannot be acquired. *spins* specifies the number of times to spin before yielding or returning. A *spins* value of -1 invokes the default, currently 600. **pfDPoolLock** and **pfDPoolSpinLock** return 1 upon acquisition of the lock, 0 upon failure to acquire the lock and -1 upon error. **pfDPoolUnlock** relinquishes the lock on the block of memory.

There are a fixed number of locks (currently 4096) allocated for each pfDataPool and a new lock is consumed when an allocation in that pfDataPool is first locked. Subsequent releases and locks do not require further lock allocations.

Example:

```
typedef struct SharedData
{
    float a, b, c;
} SharedData;

pfDataPool *pool;
SharedData *data;
 :
/* create a DataPool with room for 4 SharedData structures */
pool = pfCreateDPool(4*sizeof(SharedData), "dpoolForSharedData");

/* allocate SharedData structure in the data pool with ID=153 */
data = (SharedData*)pfDPoolAlloc(pool, sizeof(SharedData), 153);
```

```
 :
/* write to the DataPool with cooperative mutual exclusion */
pfDPoolLock((void*)data);
data->a = 370.0;
data->b = 371.0;
data->c = 407.0;
pfDPoolUnlock((void*)data);
```

pfDPoolLock attempts to acquire a hardware lock associated with *dpmem*. If another process has already acquired the lock, the calling process will not return until the lock is acquired. Whether the process blocks or spins is a function of the machine configuration. (see **usconfig**). **pfDPoolUnlock** unlocks *dpmem*. A process which double-trips a lock by calling **pfDPoolLock** twice in succession will block until the lock is unset by another process. A process may unlock a lock that was locked by a different process. **pfDPoolTest** returns 0 if *dpmem* is unlocked and 1 if it is locked.

pfDataPool memory may be accessed without using the lock and unlock feature; however this defeats the mutual exclusion feature provided by pfDataPool functions.

A data pool must occupy the same range of virtual memory addresses in all processes that attach to it. **pfAttachDPool** will fail if something else has already been mapped into the required address space, e.g. as a result of mmap or sbrk. To minimize this risk, **pfCreateDPool** tries to place new datapools above the main shared memory arena created by **pfInitArenas**. The address at which the next datapool will be created can be overridden by calling **pfDPoolAttachAddr** with the *addr* argument specifying the desired address. An *addr* of NULL tells Performer to return to its normal placement efforts. The next attachment address is returned by **pfGetDPoolAttachAddr**.

In the absence of a shared memory arena created by **pfInitArenas**, **pfCreateDPool** lets the kernel choose the data pool placement.

Deleting a data pool with **pfDelete** or unmaps the data pool from virtual memory as well as deleting the pfDataPool data structure.

NOTES

When a datapool is created, a file is created in "/usr/tmp" or **PFTMPDIR**. The file name will end with the string ".pfdpool". If **pfReleaseDPool** is not called to unlink the datapool, this file will remain in the file system after the program exits, taking up disk space.

When using pfDataPools between unrelated processes, you can reduce memory conflicts by having the application that uses more virtual memory create the datapool and having the smaller application attach to the datapool before allocating memory that might cause conflicts. Alternately, if an address is known to be safe for both applications, it can be specified using **pfDPoolAttachAddr**.

SEE ALSO

amalloc, pfInitArenas, usconfig, usinit, ussetlock, ustestlock, usunsetlock

NAME

pfDecal, pfGetDecal – Set and get decal mode for drawing coplanar polygons

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfDecal**(int mode);

int **pfGetDecal**(void);

PARAMETERS

mode is a symbolic constant specifying a decaling mode and is one of:

PFDECAL_OFF
Decaling is off

PFDECAL_BASE
Subsequent drawn geometry is considered to be 'base' geometry. Use the default decaling mechanism.

PFDECAL_LAYER
Subsequent drawn geometry is considered to be 'layered' geometry. Use the default decaling mechanism.

PFDECAL_BASE_FAST, **PFDECAL_LAYER_FAST**
Use a decaling mechanism appropriate to the hardware that produces the fastest, but not necessarily the highest quality, decaling.

PFDECAL_BASE_HIGH_QUALITY, **PFDECAL_LAYER_HIGH_QUALITY**
Use a decaling mechanism appropriate to the hardware that produces the highest quality, but not necessarily the fastest, decaling.

PFDECAL_BASE_DISPLACE, **PFDECAL_LAYER_DISPLACE**
Use the polygon displacement technique (**displacepolygon** in IRIS GL and **glPolygonOffsetEXT** in OpenGL) to slightly displace the depth values of layer geometry *toward* the eyepoint.

PFDECAL_BASE_DISPLACE, **PFDECAL_LAYER_DISPLACE_AWAY**
Use the polygon displacement technique (**displacepolygon** in IRIS GL and **glPolygonOffsetEXT** in OpenGL) to slightly displace the depth values of layer geometry *away* from the eyepoint.

PFDECAL_BASE_STENCIL, **PFDECAL_LAYER_STENCIL**
Use the stencil buffer technique (**stencil** in IRIS GL; **glStencilFunc**, **glStencilOp**, and **glEnable(GL_STENCIL_TEST)** in OpenGL) to determine visual priority.

DESCRIPTION

In some cases, such as when drawing stripes on a runway, it is easier to draw coplanar polygons than it is to model the geometry without coplanar faces. However, on Z-buffer based machines, coplanar polygons can cause unwanted visual artifacts because the visual priorities of the coplanar polygons are subject to the finite numerical precision of the graphics pipeline. This results in a "torn" appearance and "twinkling"

from frame to frame.

Decaled geometry can be thought of as a stack where each layer has visual priority over the geometry beneath it in the stack. An example of a 3 layer stack consists of stripes which are layered over a runway which is layered over the ground. The bottommost layer is called the "base" while the other layers are called "decals" or "layers". When using certain hardware mechanisms (**PFDECAL_BASE_STENCIL**) to implement pfDecal, the "base" is special because it defines the depth values which are used to determine layer visibility with respect to other scene geometry and which are written to the depth buffer.

Certain decaling mechanisms (currently only **DISPLACE**) require that each layer in the layer stack be identified for proper rendering. The **PFDECAL_LAYER_1** - **PFDECAL_LAYER_7** tokens are provided for this purpose and should be logically OR'ed into the layer mode, e.g., **PFDECAL_LAYER_DISPLACE** | **PFDECAL_LAYER_2**. Note that the layer identifier is extracted from the mode as follows:

```
layerId = (mode & PFDECAL_LAYER_MASK) >> PFDECAL_LAYER_SHIFT;
```

pfDecal is used to draw visually correct coplanar polygons that are arranged as 'base' and 'layer' polygons as shown here:

```
/* Prepare to draw base polygons */
pfDecal(PFDECAL_BASE_DISPLACE);
 :
/* draw base geometry using IRIS Performer or graphics library */
 :
/* Prepare to draw first layer polygons */
pfDecal(PFDECAL_LAYER_DISPLACE);
 :
/* draw layer geometry using IRIS Performer or graphics library */
 :
/* Prepare to draw second layer polygons */
pfDecal(PFDECAL_LAYER_DISPLACE | PFDECAL_LAYER_1);
 :
/* draw layer geometry using IRIS Performer or graphics library */
 :
/* exit decal mode */
pfDecal(PFDECAL_OFF);
```

The different **pfDecal** modes offer quality-feature tradeoffs listed in the table below:

	DISPLACE	STENCIL	(DISPLACE | OFFSET)
Quality	medium	high	high
Order	not required	required	not required
Coplanarity	not required	required	not required
Containment	not required	required	not required

The **STENCIL** mechanism offers the best image quality but at a performance cost since the base and layer geometry must be rendered in strict order. When multisampling on RealityEngine, this mechanism also significantly reduces pixel fill performance. An additional constraint is that **STENCIL**ed layers must be coplanar or decal geometry may incorrectly show through base geometry. For proper results, each layer in the "stack" must be completely contained within the boundaries of the base geometry.

The **DISPLACE** mechanism offers the best performance since layers can be sorted by graphics state, because the displace call itself is usually faster than other mode changes, and because there is no pixel fill rate penalty when it is in use. However, in IRIS GL the displace mechanism is only slope-based, so when geometry becomes nearly perpendicular to the view, i.e., has little or no slope, the displacement is too little to conclusively determine visibility. To solve this problem, logically OR-ing the **PFDECAL_LAYER_OFFSET** bit into the layer mode will add a constant offset to the decal geometry. This mode can be very expensive (RealityEngine) so when using it the database should be sorted so that all layers are drawn at the same time, i.e., draw all **PFDECAL_LAYER_1** layers together etc. Both **DISPLACE** mechanisms do not require that geometry within a single layer be coplanar. The main disadvantage is that decal geometry may incorrectly poke through other geometry due to the displacement of the decal geometry.

The performance differences between **STENCIL** and **DISPLACE** modes are hardware-dependent so some experimentation and benchmarking is required to determine the most suitable method for your application.

pfDecal is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfDecal** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

pfGetDecal returns the current decal mode.

The decaling mode state element is identified by the **PFSTATE_DECAL** token. Use this token with **pfGStateMode** to set the decaling mode of a pfGeoState and with **pfOverride** to override subsequent decaling mode changes.

EXAMPLES

Example 1:

```
/* Set up 'base' pfGeoState */
pfGStateMode(gstate, PFSTATE_DECAL, PFDECAL_BASE);

/* Attach pfGeoState to pfGeoSet */
pfGSetGState(gset, gstate);

/* Draw base pfGeoState */
pfDrawGSet(gset);
```

Example 2:

```
/* Override decaling mode to PFDECAL_OFF */
pfDecal(PFDECAL_OFF);
pfOverride(PFSTATE_DECAL, PF_ON);
```

NOTES

PFDECAL_BASE_FAST currently implies **displacepolygon** on machines that support this feature. The use of displacements for rendering coplanar geometry can cause visual artifacts such as decals "Z fighting" or "flimmering" when viewed perpendicularly and punching through geometry that should be in front of them when viewed obliquely. In these cases, use **PFDECAL_LAYER_OFFSET**, modify the database should by cutting away overlapping polygons to eliminate the need for coplanar rendering or use **PFDECAL_BASE_HIGH_QUALITY** or **PFDECAL_BASE_STENCIL**.

PFDECAL_BASE_STENCIL is implemented with stencil planes and requires the framebuffer to be configured with at least one stencil bit (see **stensize(3g)** and **mssize(3g)**). The first stencil bit should be considered as reserved for **pfDecal**.

When using **PFDECAL_LAYER_OFFSET**, the minimum depth buffer range set with **lsetdepth** must be incremented an extra 1024 * max layers so the negative displacement of the layers does not wrap. **pfInitGfx** does this automatically.

SEE ALSO

mssize, pfDispList, pfGStateMode, pfGeoState, pfOverride, pfState, pfInitGfx, stencil, stensize

NAME

pfNewDList, pfGetDListClassType, pfOpenDList, pfResetDList, pfCloseDList, pfGetDListSize, pfGetDListType, pfAddDListCmd, pfDListCallback, pfDrawDList, pfDrawGLObj, pfGetCurDList – Create and control a display list

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfDispList * **pfNewDList**(int type, int size, void *arena);

pfType * **pfGetDListClassType**(void);

void **pfOpenDList**(pfDispList *dlist);

void **pfResetDList**(pfDispList *dlist);

void **pfCloseDList**(void);

int **pfGetDListSize**(const pfDispList *dlist);

int **pfGetDListType**(const pfDispList *dlist);

void **pfAddDListCmd**(int cmd);

void **pfDListCallback**(pfDListFuncType callback, int nbytes, void *data);

int **pfDrawDList**(pfDispList *dlist);

void **pfDrawGLObj**(GLOBJECT obj);

pfDispList * **pfGetCurDList**(void);

```
typedef void (*pfDListFuncType)(void *data);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfDispList** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfDispList**. Casting an object of class **pfDispList** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfDispList** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *        pfGetType(const void *ptr);
int             pfIsOfType(const void *ptr, pfType *type);
int             pfIsExactType(const void *ptr, pfType *type);
const char *    pfGetTypeName(const void *ptr);
int             pfRef(void *ptr);
int             pfUnref(void *ptr);
int             pfUnrefDelete(void *ptr);
int             pfGetRef(const void *ptr);
int             pfCopy(void *dst, void *src);
int             pfDelete(void *ptr);
int             pfCompare(const void *ptr1, const void *ptr2);
void            pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *          pfGetArena(void *ptr);
```

PARAMETERS

dlist identifies a pfDispList.

DESCRIPTION

A pfDispList is a reusable display list that captures certain libpr commands. **pfNewDList** creates and returns a handle to a new pfDispList. The arguments specify the type and size of the display list. pfDispLists can be deleted with **pfDelete**.

type is a symbolic token that specifies a type of pfDispList and is either **PFDL_FLAT** or **PFDL_RING**. A **PFDL_FLAT** display list is a linear list of commands and data while a **PFDL_RING** is configured as a ring buffer (FIFO). A ring buffer is provided for multiprocessed paired producer and consumer applications where the producer writes to the buffer while the consumer simultaneously reads from, and draws the buffer. IRIS Performer automatically ensures ring buffer consistency by providing synchronization and mutual exclusion to processes on ring buffer full or empty conditions. **pfGetDListType** returns the type of *dlist*.

pfGetDListClassType returns the **pfType*** for the class **pfDispList**. The **pfType*** returned by **pfGetDListClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfDispList**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType**to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

The *size* argument to **pfNewDList** gives a hint in words about how much storage the pfDispList will require. If more storage is required, IRIS Performer will automatically grow the pfDispList by *size* words at a time. *arena* specifies the malloc arena out of which the pfDispList is allocated or NULL for allocation off the heap. **pfGetDListSize** returns the size of *dlist* that was requested by **pfNewDList**, not its current size.

pfOpenDList opens *dlist* for appending and puts the calling process into display list mode. When in display list mode, display-listable **libpr** commands are recorded in the currently active display list rather

than being executed immediately. libpr commands that may be recorded in a pfDispList say so in their respective man pages. Only one pfDispList may be open at a time. **pfGetCurDList** returns the currently active display list or **NULL** if the calling process is in immediate mode.

The currently active pfDispList is a global value but is stored in the PRDA process header so that *share group processes* (see **sproc**) need not share the same currently active pfDispList.

Each pfDispList maintains head and tail pointers that indicate where in the list commands are to be appended and evaluated respectively. Commands entered into the display list are appended after the head pointer and increment the head pointer appropriately. Commands drawn by **pfDrawDList** increment the tail pointer but do not remove commands from the list. In the **PFDL_RING** case, IRIS Performer ensures that the tail pointer does not overrun the head pointer and vice versa by spinning processes.

Both head and tail pointers are reset to the beginning of the pfDispList by **pfResetDList** so that any additions to the current pfDispList will overwrite previously entered commands. The tail pointers of flat lists are automatically reset by **pfDrawDList** when the tail pointer reaches the head pointer so that the pfDispList may be rendered again from the beginning.

pfCloseDList 'closes' the active pfDispList and returns the application to immediate mode.

For **PFDL_FLAT** display lists, **pfDrawDList** traverses *dlist* from the tail to the head pointer, and then resets the tail pointer to the beginning of *dlist*. If the pfDispList is a **PFDL_RING**, **pfDrawDList** will continually draw the display list, returning control to the application only on **PFDL_END_OF_FRAME** or **PFDL_RETURN** commands (see **pfAddDListCmd**). After returning, a subsequent call to **pfDrawDList** will restart drawing from the previous position in the list.

pfDrawDList interprets the commands and data in *dlist* and executes libpr state routines that in turn execute graphics library commands that send command tokens down the graphics pipeline. **pfDrawDList** is itself a display-listable command provided *dlist* is not the currently active pfDispList.

The following example draws a pfGeoSet into a pfDispList and then subsequently draws the pfDispList:

```
/* Open DList and append GSet */
pfOpenDList(dlist);
pfEnable(PFEN_WIREFRAME);
pfDrawGSet(gset);

/* Close DList and return to immediate mode */
pfCloseDList();

/* Draw 'gset' in wireframe */
pfDrawDList(dlist);
```

pfDListCallback allows custom rendering in the middle of a display list by putting a function callback and data in the current display list. Up to 64 bytes of user-data may be copied into the display list. *nbytes* specifies the length of data that *data* references. When a callback token is encountered while drawing a display list, the function *callback* will be called with a pointer to the user data that is cached in the display list. A callback may call **pfPushState** upon entering and **pfPopState** when leaving to ensure that any state changes made in the callback will be not inherited by subsequent geometry.

pfAddDListCmd adds *cmd* to the currently active display list. *cmd* is one of the following symbolic tokens, both of which return control to the application but indicate different situations.

PFDL_RETURN
PFDL_END_OF_FRAME

pfDrawGLObj will directly draw the graphics library display list object identified by *obj* (via **callobj** in IRIS GL or **glCallList** in OpenGL) if there is no active pfDispList. If there is an open pfDispList, then **pfDrawGLObj** will simply add the identifier and command to the active pfDispList.

IRIS Performer optimizes pfDispList's when they are being built by eliminating redundant mode changes and by unwinding pfGeoStates into their component parts. As a result, modifications to objects after they are placed in a pfDispList may be ignored by the pfDispList. To be safe, do not modify any objects that were placed in one or more pfDispLists.

Here is an example of this phenomenon:

```
/* Attach gstate0 to gset */
pfGSetGState(gset, gstate0);

/* Open dlist as current pfDispList */
pfOpenDList(dlist);
pfDrawGSet(gset);

/* Return to immediate mode */
pfCloseDList();

/* Attach gstate1 to gset */
pfGSetGState(gset, gstate1);

/*
 * dlist will use gstate0 and not be aware that gset was modified to
 * use gstate1.
 */
pfDrawDList(dlist);
```

SEE ALSO

pfDelete, pfGeoState, pfObject, pfState, callobj, sproc

NAME

pfEnable, pfDisable, pfGetEnable – Enable and disable graphics modes.

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfEnable**(int mode);

void **pfDisable**(int mode);

int **pfGetEnable**(int mode);

PARAMETERS

mode is a symbolic constant that specifies the enable target which is to be enabled or disabled. The enable targets are:

PFEN_LIGHTING	Hardware lighting
PFEN_TEXTURE	Hardware texturing
PFEN_FOG	Hardware fogging
PFEN_WIREFRAME	Wireframe display mode
PFEN_COLORTABLE	Colortable display mode
PFEN_HIGHLIGHTING	Highlight display mode
PFEN_LPOINTSTATE	Light point mode
PFEN_TEXGEN	Automatic texture coordinate generation

DESCRIPTION

pfEnable and **pfDisable** enable and disable various graphics library and IRIS Performer modes. By default all modes listed above are disabled, i.e., when a pfState is first created, its enable modes are all **PF_OFF**.

pfGetEnable returns the enable status of the indicated*mode*.

Each **pfEnable**/**pfDisable** mode token corresponds to a **PFSTATE_** token that identifies the state element and is used in pfGeoState routines and **pfOverride**. The **PFEN_** / **PFSTATE_** correspondence is illustrated in the following table:

Enable Token	State Token
PFEN_LIGHTING	**PFSTATE_ENLIGHTING**
PFEN_TEXTURE	**PFSTATE_ENTEXTURE**
PFEN_FOG	**PFSTATE_ENFOG**
PFEN_WIREFRAME	**PFSTATE_ENWIREFRAME**
PFEN_COLORTABLE	**PFSTATE_ENCOLORTABLE**
PFEN_HIGHLIGHTING	**PFSTATE_ENHIGHLIGHTING**
PFEN_LPOINTSTATE	**PFSTATE_ENLPOINTSTATE**
PFEN_TEXGEN	**PFSTATE_ENTEXGEN**

Once enabled or disabled, these mode changes will not take effect until their associated IRIS Performer state elements are applied. The following table lists the modes, state objects, and provoking activation routine for each mode.

Enable Mode	Required State	Activation Routine
PFEN_LIGHTING	pfLightModel	**pfApplyLModel** pfMaterial pfLight normals
PFEN_TEXTURE	pfTexture	**pfApplyTex** pfTexEnv texture coordinates
PFEN_FOG	pfFog	**pfApplyFog**
PFEN_COLORTABLE	pfColortable	**pfApplyCtab**
PFEN_HIGHLIGHTING	pfHighlight	**pfApplyHlight**
PFEN_LPOINTSTATE	pfLPointState	**pfApplyLPState**
PFEN_TEXGEN	pfTexGen	**pfApplyTGen**
PFEN_WIREFRAME	none	none

Use the **PFSTATE_** tokens with **pfGStateMode** to set the enable modes of a pfGeoState and with **pfOverride** to override subsequent enable mode changes:

Example 1:

```
/* Set up textured pfGeoState */
pfGStateMode(gstate, PFSTATE_ENTEXTURE, PF_ON);
pfGStateAttr(gstate, PFSTATE_TEXTURE, tex);
pfGStateAttr(gstate, PFSTATE_TEXENV, texEnv);

/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/* Draw textured gset */
pfDrawGSet(gset);
```

Example 2:

```
/* Override lighting and texturing enable modes to PF_OFF */
pfDisable(PFEN_LIGHTING);
pfDisable(PFEN_TEXTURE);
pfOverride(PFSTATE_ENLIGHTING | PFSTATE_ENTEXTURE, PF_ON);
```

pfEnable and **pfDisable** are display-listable commands. If a pfDispList has been opened by **pfOpenDList**, then **pfEnable** and **pfDisable** will not have immediate effect. They will be captured by the pfDispList and will take effect when that pfDispList is later drawn with **pfDrawDList**.

pfBasicState disables all of the above modes.

notes When lighting is disabled, lmcolor mode is set to **LMC_COLOR** which effectively turns it off. Then when enabled, the lmcolor mode is restored to that of the current front material if there is one.

SEE ALSO

pfDispList, pfGeoSet, pfGeoState, pfOverride, pfState

NAME

pfFPConfig, pfGetFPConfig – Specify floating-point tolerances and exception handling

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfFPConfig**(int which, float val);

float **pfGetFPConfig**(int which);

DESCRIPTION

pfFPConfig allows you to set some tolerances used by the IRIS Performer math routines. The *which* field specifies which floating point attribute is to be set. *val* specifies the value that attribute should take. Supported values are:

PFFP_UNIT_ROUNDOFF
Specifies the tolerance applied to testing for equality, usually scaled by the magnitude of the operand. The default is 1.0e-5. For performance reasons, the appropriately scaled tolerance is not used pervasively.

PFFP_ZERO_THRESH
Specifies how large in magnitude a floating point number can be before it should be considered non-zero. The default is 1.0e-15.

PFFP_TRAP_FPES
Enables and disables trapping of floating point exceptions with *value*s 1.0 and 0.0, respectively. Normally, these floating point errors are handled through kernel exceptions or by the floating point hardware, and may be nearly invisible to an application except from performance degradation, sometimes very significant, which they can cause. When enabled, pfNotify events are generated for the floating point exceptions mentioned above and messages displayed or passed to the user supplied pfNotify handler.

An application can also turn on floating point exception handling via the general Performer pfNotify notification mechanism. Calling **pfNotifyLevel** with the **PFNFY_FP_DEBUG** notification level configures **pfFPConfig** to enable exceptions. Alternately, an appropriate setting of the environment variable **PFNFYLEVEL** will enable this processing as well. Use of the **pfNotifyLevel** function is preferable because then the PFNFYLEVEL environment variable can be used when necessary to override the specification.

BUGS

Enabling floating point exceptions may cause the values returned from exceptions to be different from the system defaults. Once a **PFNFY_FP_INVALID** exception has been reported, all subsequent exceptions will generate incorrect return values.

SEE ALSO

handle_sigfpes, pfNotifyLevel

NAME

pfQueryFeature, pfMQueryFeature, pfFeature – Graphics feature availability routines

FUNCTION SPECIFICATION

#include <Performer/pr.h>

int **pfQueryFeature**(int which, int *dst);

int **pfMQueryFeature**(int *which, int *dst);

void **pfFeature**(int which, int val);

DESCRIPTION

IRIS Performer make runtime determinations regarding the existence and relative speed of certain graphics features for the current operating graphics library (IRIS GL or OpenGL) on the current hardware configuration. These functions provide the ability to both query these results, and to override the existence of a given feature. IRIS Performer makes use of the following useful routines in determining its information: **getgdesc(3g)**, **XGetVisualInfo(3X11)**, **glGetString(3g)**, **glXQueryExtensionsString(3g)**, and **getinvent(3)**.

pfQueryFeature takes a **PFQFTR_** token and returns in *dst* a token that indicates whether or not that feature exists and whether the feature is reasonable to use in a real-time application. The return value is the number of bytes successfully written. The tokens are documented below and special note is made where performance is commonly an issue. These **pfQueryFeature** return values will be one of:

PFQFTR_FALSE
indicates that the feature is not available on the current hardware configuration.

PFQFTR_TRUE
indicates that the feature is available on the current hardware configuration.

PFQFTR_FAST
indicates that the feature is reasonable for real-time on the current hardware configuration.

The **pfQueryFeature** tokens must be one of:

PFQFTR_VSYNC
queries the status of the graphics video clock. See the **pfInitVClock** reference page for more information.

PFQFTR_VSYNC_SET
queries the write-ability of the graphics video clock. See the **pfInitVClock** reference page for more information.

PFQFTR_GANGDRAW
queries the availability of "gang" swapbuffers where multiple graphics pipelines may be forced to swap framebuffers simultaneously. See the **pfChanShare** reference page for more information.

PFQFTR_HYPERPIPE
queries the support for hyperpipe hardware. See the **pfHyperpipe** reference page for more information.

PFQFTR_STEREO_IN_WINDOW
queries the support for doing stereo with multiple buffers in a single window. See the IRIS GL **stereobuffer(3g)** and OpenGL **glDrawBufferMode(3g)** reference pages for more information.

PFQFTR_MULTISAMPLE
queries the support and relative performance of multisampled antialiasing. See the **pfAntialias** reference page for more information.

PFQFTR_MULTISAMPLE_TRANSP
queries the support and relative performance of multisampled transparency. See the **pfTransparency** reference page for more information.

PFQFTR_MULTISAMPLE_ROUND_POINTS
queries the support and relative performance of round multisampled light points.

PFQFTR_MULTISAMPLE_STENCIL
queries the support and relative performance of multisampled stencil.

PFQFTR_COLOR_ABGR
queries the support and relative performance of image data in the IRIS GL style ABGR format. This format may be slow or unsupported in some OpenGL implementations. This is relevant to the OpenGL **glDrawPixels(3g)** command.

PFQFTR_DISPLACE_POLYGON
queries the support for polygon displacement in screen Z used for doing decals. See the **pfDecal** reference page for more information.

PFQFTR_POLYMODE
queries the support for polygon fill modes. See the IRIS GL **polymode(3g)** and OpenGL **glPolygonMode(3g)** reference pages for more information.

PFQFTR_TRANSPARENCY
queries the support for and relative performance of transparency.

PFQFTR_FOG_SPLINE
queries the support for spline fog. See the **pfFog** reference page for more information.

PFQFTR_ALPHA_FUNC
queries the support for alpha functions. See the **pfAlphaFunc** reference page and the IRIS GL **blendfunction(3g)** and OpenGL **glBlendFunc(3g)** reference pages for more information.

PFQFTR_ALPHA_FUNC_COMPARE_REF
queries the support for comparative alpha functions. Some graphics platforms under IRIS GL do not support the comparison alpha functions. See the **pfAlphaFunc** reference page for more information.

PFQFTR_BLENDCOLOR
queries the support for specification of a blend color to use with alpha functions. Refer to the IRIS GL **blendcolor(3g)** and the OpenGL extension **glBlendColor(3g)** for more information.

PFQFTR_BLEND_FUNC_SUBTRACT
queries the support for additional differencing alpha blending functions.

PFQFTR_BLEND_FUNC_MINMAX
queries the support for additional min/max alpha blending functions.

PFQFTR_TEXTURE
queries the support and relative performance of texture mapping.

PFQFTR_TEXTURE_16BIT_IFMTS
queries the support and relative performance of 16-bit texel formats. These formats take up less texture memory and can provide a significant performance improvement at the cost of some loss of image quality. See the **pfTexFormat** reference page for more information.

PFQFTR_TEXTURE_SUBTEXTURE
queries the support for dynamic loading of parts or all of textures after the texture has been defined.

PFQFTR_TEXTURE_TRILINEAR
queries the support for trilinear MIPmapping for minification filtering of texture maps.

PFQFTR_TEXTURE_DETAIL
queries the support for detailing of magnified texture maps. See the **pfTexFilter** and **pfTexDetail** reference pages for more information.

PFQFTR_TEXTURE_SHARPEN
queries the support for sharpening of magnified texture maps. See the **pfTexFilter** reference page for more information.

PFQFTR_TEXTURE_3D
queries the support for three-dimensional textures.

PFQFTR_TEXTURE_PROJECTIVE
queries the support for projected textures.

PFQFTR_TEXTURE_MINFILTER_BILINEAR_CMP
queries the support for special bilinear LEQUAL/GEQUAL minification filters for doing real-time shadows.

PFQFTR_READ_MSDEPTH_BUFFER
queries the support for reading the multisample depth buffer.

PFQFTR_COPY_MSDEPTH_BUFFER
queries the support for copying to/from the multisample depth buffer.

PFQFTR_READ_TEXTURE_MEMORY
queries the support for reading texture memory.

PFQFTR_COPY_TEXTURE_MEMORY
queries the support for copying to/from texture memory.

PFQFTR_MTL_CMODE
queries the support and speed of material color mode. On most graphics platforms, this mode yields significant performance improvements for management of multiple materials. However, on some older low-end platforms, it can have additional cost and should not be used if multiple materials are not in used.

PFQFTR_LMODEL_ATTENUATION
queries the support for light attenuation on the light model. This is IRIS GL style light attenuation. See the IRIS GL **lmdef(3g)** reference page for more information.

PFQFTR_LIGHT_ATTENUATION
queries the support for light attenuation on the light. This is OpenGL style light attenuation. See the OpenGL **glLightModel(3g)** reference page for more information.

PFQFTR_LIGHT_CLR_SPECULAR
queries the support for specular color components on lights. This is only supported in OpenGL. See the OpenGL **glLight(3g)** reference page for more information.

pfMQueryFeature takes an NULL-terminated array of query tokens and a destination buffer and will do multiple queries. The return value is the number of bytes successfully written. This routine is more efficient than **pfQueryFeature** if multiple queries are desired.

pfFeature takes a **PFFTR_** token *which* and a boolean value *val* and allows the overriding of IRIS Performer's determination of the existence of certain features. This can force IRIS Performer to use, or to stop using, a specific feature. This is useful for running on new graphics platforms that may have considerations that IRIS Performer did not predict, or for making one machine behave like another for a specific feature. Note that if a particular feature is forced on and it happens to require hardware support that does not exist, the program may not run. The features that may be set are:

PFFTR_VSYNC

PFFTR_VSYNC_SET

PFFTR_MULTISAMPLE

PFFTR_MULTISAMPLE_ROUND_POINT

PFFTR_ALPHA_FUNC_ALL

PFFTR_DISPLACE_POLYGON

PFFTR_POLYMODE

PFFTR_FOG_SPLINE

PFFTR_GANGDRAW

PFFTR_HYPERPIPE

PFFTR_FAST_TRANSPARENCY

PFFTR_TEXTURE_FAST

PFFTR_TEXTURE_16BIT_IFMTS

PFFTR_TEXTURE_SUBTEXTURE

PFFTR_TEXTURE_3D

PFFTR_TEXTURE_DETAIL

PFFTR_TEXTURE_SHARPEN

PFFTR_TEXTURE_OBJECT

PFFTR_TEXTURE_PROJECTIVE

PFFTR_TEXTURE_TRILINEAR

NOTES

pfQueryWin can be used to query the configuration parameters of a given pfWindow. **pfQuerySys** can be used to query the specific configuration parameters of the current hardware configuration.

SEE ALSO

getgdesc, XGetVisualInfo, glGetString, glXQueryExtensionsString, getinvent

NAME

pfCreateFile, pfOpenFile, pfCloseFile, pfReadFile, pfSeekFile, pfWriteFile, pfGetFileStatus, pfGetFileClassType – Asynchronous real-time file access operations

FUNCTION SPECIFICATION

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

pfFile *    pfCreateFile(char *path, mode_t mode);

pfFile *    pfOpenFile(char *fname, int oflag,

int         pfCloseFile(pfFile *file);

int         pfReadFile(pfFile *file, char *buf, int nbyte);

off_t       pfSeekFile(pfFile *file, off_t off, int whence);

int         pfWriteFile(pfFile *file, char *buf, int nbyte);

int         pfGetFileStatus(const pfFile *file, int attr);

pfType *
            pfGetFileClassType(void);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfFile** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfFile**. Casting an object of class **pfFile** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
void*   pfGetUserData(pfObject *obj);
int     pfGetGLHandle(pfObject *obj);
```

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfFile** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *        pfGetType(const void *ptr);
int             pfIsOfType(const void *ptr, pfType *type);
int             pfIsExactType(const void *ptr, pfType *type);
const char *    pfGetTypeName(const void *ptr);
int             pfRef(void *ptr);
int             pfUnref(void *ptr);
int             pfUnrefDelete(void *ptr);
int             pfGetRef(const void *ptr);
```

```
int         pfCopy(void *dst, void *src);
int         pfDelete(void *ptr);
int         pfCompare(const void *ptr1, const void *ptr2);
void        pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *      pfGetArena(void *ptr);
```

DESCRIPTION

The functions **pfCreateFile**, **pfOpenFile**, **pfCloseFile**, **pfReadFile**, **pfWriteFile**, and **pfSeekFile** operate in an identical fashion and take similar arguments as the standard UNIX file I/O functions: **creat, open, close, read, write,** and **lseek**. The difference is that they return immediately without blocking while the physical file-system access operation completes and also that instead of an integer file descriptor, a pfFile handle is used. IRIS Performer supports a maximum of **PFRTF_MAXREQ** pending file I/O requests.

When called, **pfOpenFile** and **pfCreateFile** create a new process using the **sproc** mechanism that manages the file operations asynchronously with the calling process. If the calling process has super-user privileges, the new process will assign itself to processor 0, and lower its priority. The spawned process will exit when either its pfFile is closed via **pfCloseFile** or when its parent process (that which called **pfOpenFile** or **pfCreateFile**) exits.

pfCloseFile closes the open file and terminates the I/O process created by **pfOpenFile** or **pfCreateFile**. **pfCloseFile** does not free *file* - use **pfDelete** for that purpose.

pfGetFileClassType returns the **pfType*** for the class **pfFile**. The **pfType*** returned by **pfGetFileClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfFile**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfGetFileStatus returns the status of *file* corresponding to **attr** which may be one of:

PFRTF_STATUS	Return 0 if last action complete and no other actions pending. 1 if action in progress, and -1 if last action failed.
PFRTF_CMD	Return the current (or last) file I/O action, one of the following: **PFRTF_NOACTION** **PFRTF_CREATE** **PFRTF_OPEN** **PFRTF_READ** **PFRTF_WRITE** **PFRTF_SEEK** **PFRTF_CLOSE** **PFRTF_PENDING**

PFRTF_BYTES	Return the number of bytes from the last read or write action.
PFRTF_OFFSET	Return the offset from the last seek action.
PFRTF_PID	Return the process id of the I/O process associated with *file*.

NOTES

The need for the pfFile facility has been largely superseded by the IRIX 5 asynchronous I/O facility. These capabilities are accessible through **aio_cancel**, **aio_error**, **aio_init**, **aio_read**, **aio_return**, **aio_suspend**, and **aio_write**. Users are encouraged to use the aio functions for performing asynchronous file operations in programs now in development.

The calling process should always call **pfCloseFile** to close a file before exiting. If the calling program exits without doing so, the file will not be closed. Such files can be challenging to remove from the file system.

SEE ALSO

access, aio_cancel, aio_error, aio_init, aio_read, aio_return, aio_suspend, aio_write, close, creat, fcntl, lseek, open, read, write

NAME

pfFilePath, pfGetFilePath, pfFindFile – Locate files using a search path.

FUNCTION SPECIFICATION

```
#include <unistd.h>
#include <Performer/pr.h>
void          pfFilePath(const char *path);
const char *  pfGetFilePath(void);
int           pfFindFile(const char *file, char path[PF_MAXSTRING], int amode);
```

DESCRIPTION

pfFilePath specifies one or more search path locations. These locations are directories to be searched for data files by IRIS Performer applications. This information is used by the **pfFindFile** function. The *path* argument to **pfFilePath** is a colon-separated list of directory pathnames similar to the **PATH** environment variable. Here is a simple example:

```
pfFilePath("/usr/bin:/usr/sbin:/usr/local/bin");
```

pfGetFilePath returns the path list set using **pfFilePath** or **NULL** if no path has yet been set. The string returned is identical in format to the one set via **pfFilePath**; each of the directory names is colon separated.

pfFindFile attempts to find *file* amongst the paths set in the environment variable **PFPATH** and by **pfFilePath**. It also tests the file's access mode against the *amode* argument. IRIS Performer routines which access files use **pfFindFile**.

The **PFPATH** environment variable is interpreted in the same format used for the **pfFilePath**. A C-shell example that specifies the directories "/usr/data" and "/usr/share/Performer/data" is:

```
setenv PFPATH "/usr/data:/usr/share/Performer/data"
```

The search logic used by **pfFindFile** is this:

1. First the file is sought exactly as named by *file*. If it exists and passes the subsequent access test described below, then *file* will be returned in the *path* argument.
2. If the file was not found or was not accessible, then each of the locations defined by the **PFPATH** environment variable are prepended to the *file* argument and tested. If it exists and passes the subsequent access test described below, then the complete path name will be returned in the *path* argument.

3. If the file has still not been successfully located, then each of the locations defined by the most recent call to **pfFilePath** will be prepended to the *file* argument and tested. If it exists and passes the subsequent access test described below, then the complete path name will be returned in the *path* argument.
4. If all of these efforts are fruitless, then **pfFindFile** will give up and return a NULL string in the *path* argument.

The mere existence of *file* in one of the indicated directories is not sufficient, the file must also be accessible in the access mode defined by *amode*. This mode is a bitfield composed by OR-ing together the permission attributes defined in <*unistd.h*> and listed in the following table:

Mode Token	Mode Value	Action
R_OK	0x4	Read permission
W_OK	0x2	Write permission
X_OK	0x1	Execute and search
F_OK	0x0	Existence of file

If the bits set in *amode* are also set in the file's access permission mode, then the complete path is copied into *path* and **TRUE** is returned indicating success. If the access modes are not similar, then the search continues until there are no more paths to search and **FALSE** is returned indicating failure.

NOTES

pfCreateFile and **pfOpenFile** do not use **pfFindFile**. This is because the search implied can be unpredictably slow when remote directories are present in the search path.

SEE ALSO

access

NAME

pfNewFog, pfGetFogClassType, pfFogType, pfGetFogType, pfFogColor, pfGetFogColor, pfFogRange, pfGetFogRange, pfFogOffsets, pfGetFogOffsets, pfFogRamp, pfGetFogRamp, pfGetFogDensity, pfApplyFog, pfGetCurFog – Create, modify and query fog definition

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfFog * **pfNewFog**(void *arena);

pfType * **pfGetFogClassType**(void);

void **pfFogType**(pfFog *fog, int type);

int **pfGetFogType**(const pfFog *fog);

void **pfFogColor**(pfFog *fog, float r, float g, float b);

void **pfGetFogColor**(const pfFog *fog, float *r, float *g, float *b);

void **pfFogRange**(pfFog *fog, float onset, float opaque);

void **pfGetFogRange**(const pfFog *fog, float *onset, float *opaque);

void **pfFogOffsets**(pfFog *fog, float onset, float opaque);

void **pfGetFogOffsets**(const pfFog *fog, float *onset, float *opaque);

void **pfFogRamp**(pfFog *fog, int points, float *range, float *density, float bias);

void **pfGetFogRamp**(const pfFog *fog, int *points, float *range, float *density, float *bias);

float **pfGetFogDensity**(const pfFog *fog, float range);

void **pfApplyFog**(pfFog *fog);

pfFog * **pfGetCurFog**(void);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfFog** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfFog**. Casting an object of class **pfFog** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfFog** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *      pfGetType(const void *ptr);
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

DESCRIPTION

pfFog is used to simulate atmospheric phenomena such as fog and haze and for depthcueing. The fog color is blended with the color that is computed for rendered geometry based on the geometry's range from the eyepoint. Fog effects may be computed at geometric vertices and then interpolated or computed precisely at each individual pixel.

pfNewFog creates and returns a handle to a new pfFog. *arena* specifies a malloc arena out of which the pfFog is allocated or NULL for allocation from the calling process' heap. pfFogs can be deleted with **pfDelete**.

pfGetFogClassType returns the **pfType*** for the class **pfFog**. The **pfType*** returned by **pfGetFogClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfFog**. When decisions are made based on the type of an object, it is usually better to use **pfIsOfType**to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfFogType sets the fog type to be used when this pfFog is applied. *type* must be one of:

PFFOG_VTX_LIN
PFFOG_VTX_EXP
PFFOG_VTX_EXP2
PFFOG_PIX_LIN
PFFOG_PIX_EXP
PFFOG_PIX_EXP2
PFFOG_PIX_SPLINE

Detailed descriptions of these fog types are in the IRIS GL **fogvertex(3G)** and the OpenGL **glFog(3G)** reference pages, with the exception of **PFFOG_PIX_SPLINE**. This is an advanced fog type that allows the user to define fog densities as a **PFFOG_MAXPOINTS** point spline curve as described in **pfFogRamp**. When fog type **PFFOG_PIX_SPLINE** is specified the internal fog ramp table will be recomputed using the

current values of fog range, fog offsets, and fog ramp. The default fog type is **PFFOG_PIX_EXP2**. **pfGetFogType** returns the fog type as its value.

pfFogColor specifies the color to be used as the fog blend color. The default fog color is white, whose RGB value is [1.0, 1.0, 1.0]. **pfGetFogColor** returns the fog color in the variables specified.

pfFogRange sets the onset and opaque ranges in world coordinate distances. The onset is the range at which fog blending first occurs. The opaque range is the distance at which scene elements are completely opaque and appear as the fog color. For the fog types **PFFOG_VTX_EXP, PFFOG_VTX_EXP2, PFFOG_PIX_EXP, PFFOG_PIX_EXP2** only the opaque range is significant; the onset range is always 0.0 in world coordinates. If the fog type is **PFFOG_PIX_SPLINE** then the internal fog ramp table will be recomputed whenever the ranges are specified. **pfGetFogRange** returns the current fog range values.

pfFogOffsets sets the individual onset and opaque range offsets used to modify the fog range. These offsets are added to the fog range values when the pfFog is applied. Calling this function with offsets of zero causes the ranges defined by **pfFogRange** to be used directly. If the fog type is **PFFOG_PIX_SPLINE** then the internal fog ramp table will be recomputed whenever the offsets are specified. **pfGetFogOffsets** returns the current fog offset values.

pfFogRamp defines the fog density curve using a table rather than as an algebraic function of range. The fog ramp table is only used with the **PFFOG_PIX_SPLINE** fog type. From four to PFFOG_MAXPOINTS control points are used to describe this curve. If fewer than four control points are given, the last point will be replicated to create four points. Each point consists of a range and fog density pair. These are given in ascending order in the arrays *range* and *density*. The range value is specified in a normalized form in the numeric range [0..1], with 0.0 corresponding to the fog onset range (plus offset) and 1.0 the fog opaque range (plus offset). This allows the ranges to be changed while maintaining the same fog density curve. The fog density at each range point must also be in the [0..1] range, where 0.0 represents no fog and 1.0 means opaque fog. A Catmull-Rom spline interpolation is used to create hardware fog tables from this fog ramp table. If the fog type is **PFFOG_PIX_SPLINE** then the internal fog ramp table will be recomputed whenever the fog ramp, fog range, or fog offsets are specified. The default fog ramp table defines a linear interpolation between the onset and opaque ranges. Currently, the *bias* value must be set to zero. **pfGetFogRamp** returns the number of points, range and density arrays, and bias in the variables specified.

pfGetFogDensity returns the density, ranging from 0 to 1 of *fog* at range *range*.

pfApplyFog configures the graphics hardware with the fog parameters encapsulated by *fog*. Only the most recently applied pfFog is active although any number of pfFog definitions may be created. Fogging must also be enabled (**pfEnable**(**PFEN_FOG**)) for *fog* to take effect. Modifications made to this pfFog do not have effect until **pfApplyFog** is called. If a pfDispList has been opened by **pfOpenDList**, **pfApplyFog** will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

The fog state element is identified by the **PFSTATE_FOG** token. Use this token with **pfGStateAttr** to set the fog mode of a pfGeoState and with **pfOverride** to override subsequent fog changes:

Example 1:

```
/* Set up 'fogged' pfGeoState */
pfGStateMode(gstate, PFSTATE_ENFOG, PFFOG_ON);
pfGStateAttr(gstate, PFSTATE_FOG, fog);

/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/* Draw fogged gset */
pfDrawGSet(gset);
```

Example 2:

```
/* Override so that all geometry is fogged with 'fog' */
pfEnable(PFEN_FOG);
pfApplyFog(fog);
pfOverride(PFSTATE_FOG | PFSTATE_ENFOG, PF_ON);
```

pfGetCurFog returns the currently active pfFog.

NOTES

PFFOG_PIX_SPLINE is only effective on RealityEngine graphics systems. The visual quality of per-pixel fogging is influenced by the ratio of the distances from the eye to the far and the eye to the near clipping planes. This ratio should be minimized for best results.

BUGS

pfGetFogDensity does not properly evaluate **PFFOG_PIX_SPLINE**; instead it linearly interpolates the spline points.

SEE ALSO

pfDispList, pfEnable, pfGeoState, pfObject, pfOverride, fogvertex, lsetdepth, perspective, glFog

NAME

pfNewFont, pfGetFontClassType, pfFontCharGSet, pfGetFontCharGSet, pfFontCharSpacing, pfGetFontCharSpacing, pfFontAttr, pfGetFontAttr, pfFontVal, pfGetFontVal, pfFontMode, pfGetFontMode – Routines to load fonts for use in Performer.

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfFont*	**pfNewFont**(void *arena);
pfType*	**pfGetFontClassType**(void);
void	**pfFontCharGSet**(pfFont* font, int ascii, pfGeoSet* gset);
pfGeoSet*	**pfGetFontCharGSet**(pfFont* font, int ascii);
void	**pfFontCharSpacing**(pfFont* font, int ascii, pfVec3 spacing);
const pfVec3*	**pfGetFontCharSpacing**(pfFont* font, int ascii);
void	**pfFontAttr**(pfFont* font, int which, void *attr);
void*	**pfGetFontAttr**(pfFont* font, int which);
void	**pfFontVal**(pfFont* font, int which, float val);
float	**pfGetFontVal**(pfFont* font, int which);
void	**pfFontMode**(pfFont* font, int mode, int val);
int	**pfGetFontMode**(pfFont* font, int mode);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfFont** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfFont**. Casting an object of class **pfFont** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void	**pfUserData**(pfObject *obj, void *data);
void*	**pfGetUserData**(pfObject *obj);
int	**pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfFont** can also be used with these functions designed for objects of class **pfMemory**.

pfType *	**pfGetType**(const void *ptr);
int	**pfIsOfType**(const void *ptr, pfType *type);
int	**pfIsExactType**(const void *ptr, pfType *type);

```
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

DESCRIPTION

The **pfFont** facility provides the capability to load fonts for 3-D rendering with the string drawing routines from **pfString** and **pfText**. The basic methodology is the user provides individual GeoSets to be used as font characters. Likewise, the user provides 3-D spacings for each character so that Performer can correctly move the 'cursor' or '3-D origin' after drawing each character. Note that this facility is completely general and is independent of external font descriptions - see **pfdLoadFont** for a description of loading some PostScript Type I fonts into a pfFont structure.

pfNewFont returns a handle to a new **pfFont**. *arena* specifies the malloc arena out of which the **pfFont** is allocated or NULL for allocation off the heap. A NULL pointer is returned to indicate failure. pfFonts can be deleted with **pfDelete**.

pfGetFontClassType returns the **pfType*** for the class **pfFont**. The **pfType*** returned by **pfGetFontClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfFont**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfFont Definition:

Call **pfFontCharGSet** to set the pfGeoSet which Performer should use when drawing the character specified by *ascii* in a **pfString**. **pfGetFontCharGSet** returns the pfGeoSet currently being used for the character specified by *ascii*. Call **pfFontCharSpacing** to set the 3D spacing for the character specified by *ascii* to be the pfVec3 *spacing*. This spacing is used to update the cursor position of a **pfString** after this character has been drawn. **pfGetFontCharSpacing** returns a reference to a pfVec3 specifying the spacing of a given character of a font.

pfFont Attributes:

pfFontAttr sets a particular attribute for a given Performer font, while **pfGetFontAttr** will return a particular attribute corresponding to the *which* token.

Current valid pfFont Attributes: **PFFONT_GSTATE PFFONT_BBOX PFFONT_SPACING PFFONT_NAME**

PFFONT_GSTATE specifies a global pfGeoState to be used for every character of a pfFont. Note that pfGeoStates bound to GeoSets representing characters will take precedence over the PFFONT_GSTATE pfGeoState. A Font has NO pfGeoState by default.

PFFONT_BBOX specifies a bounding box that will enclose every individual character of a pfFont.

PFFONT_SPACING specifies a global spacing to use to simulate fixed width fonts. This spacing is used only if the pfFont mode PFFONT_CHAR_SPACING is set to PFFONT_CHAR_SPACING_FIXED or a spacing is not available (NULL) for a given character. **PFFONT_NAME** simply specifies a name associated with a pfFont.

pfFont Modes:

pfFontMode sets a particular mode for a given Performer font, while **pfGetFontMode** will return the current value of the mode corresponding to the *mode* token.

Current valid **pfFont** Modes: **PFFONT_CHAR_SPACING PFFONT_NUM_CHARS**

PFFONT_CHAR_SPACING specifies whether to use fixed or variable spacings for all characters of a pfFont. Possible values are **PFFONT_CHAR_SPACING_FIXED** and **PFFONT_CHAR_SPACING_VARIABLE** - the later being the default.

pfFont Values:

pfFontVal sets a particular value for a given Performer font, while **pfGetFontVal** will return the corresponding value associated with the *which* token.

NOTES

See **pfText** for sample code demonstrating **pfFont**.

SEE ALSO

pfBox, pfDelete, pfGeoSet, pfGeoState, pfString, pfText, pfdLoadFont

NAME

pfNewFrust, pfGetFrustClassType, pfMakePerspFrust, pfMakeOrthoFrust, pfMakeSimpleFrust, pfGetFrustType, pfGetFrustFOV, pfFrustAspect, pfGetFrustAspect, pfFrustNearFar, pfGetFrustNearFar, pfGetFrustNear, pfGetFrustFar, pfGetFrustEye, pfGetFrustPtope, pfGetFrustGLProjMat, pfApplyFrust, pfFrustContainsPt, pfFrustContainsBox, pfFrustContainsSphere, pfFrustContainsCyl, pfOrthoXformFrust – Operations on frusta

FUNCTION SPECIFICATION

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

pfFrustum *   pfNewFrust(void* arena);
pfType *      pfGetFrustClassType(void);
void          pfMakePerspFrust(pfFrustum* frust, float left, float right, float bottom, float top);
void          pfMakeOrthoFrust(pfFrustum* frust, float left, float right, float bottom, float top);
void          pfMakeSimpleFrust(pfFrustum* frust, float fov);
int           pfGetFrustType(const pfFrustum* frust);
void          pfGetFrustFOV(const pfFrustum* frust, float* fovh, float* fovv);
void          pfFrustAspect(pfFrustum* frust, int which, float widthHeightRatio);
float         pfGetFrustAspect(const pfFrustum* frust);
void          pfFrustNearFar(pfFrustum* frust, float near, float far);
void          pfGetFrustNearFar(const pfFrustum* frust, float* near, float* far);
void          pfGetFrustNear(const pfFrustum* frust, pfVec3 ll, pfVec3 lr, pfVec3 ul, pfVec3 ur);
void          pfGetFrustFar(const pfFrustum* frust, pfVec3 ll, pfVec3 lr, pfVec3 ul, pfVec3 ur);
int           pfGetFrustEye(const pfFrustum* frust, pfVec3 eye);
void          pfGetFrustPtope(const pfFrustum* frust, pfPolytope *ptope);
void          pfGetFrustGLProjMat(const pfFrustum* frust, pfMatrix mat);
void          pfApplyFrust(const pfFrustum *frust);
int           pfFrustContainsPt(const pfFrustum *fr, const pfVec3 pt);
int           pfFrustContainsBox(const pfFrustum *frust, const pfBox *box);
int           pfFrustContainsSphere(const pfFrustum *fr, const pfSphere *sph);
int           pfFrustContainsCyl(const pfFrustum *frust, const pfCylinder *cyl);
void          pfOrthoXformFrust(pfFrustum* dst, const pfFrustum* src, const pfMatrix mat);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfFrustum** is derived from the parent class **pfPolytope**, so each of these member functions of class **pfPolytope** are also directly usable with objects of class **pfFrustum**. Casting an object of class **pfFrustum** to an object of class **pfPolytope** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfPolytope**.

int **pfGetPtopeNumFacets**(pfPolytope *ptope);
int **pfPtopeFacet**(pfPolytope *ptope, int i, const pfPlane *facet);
int **pfGetPtopeFacet**(pfPolytope *ptope, int i, pfPlane *facet);
int **pfRemovePtopeFacet**(pfPolytope *ptope, int i);
void **pfOrthoXformPtope**(pfPolytope *ptope, const pfPolytope *src, const pfMatrix mat);
int **pfPtopeContainsPt**(const pfPolytope *ptope, const pfVec3 pt);
int **pfPtopeContainsSphere**(const pfPolytope *ptope, const pfSphere *sphere);
int **pfPtopeContainsBox**(const pfPolytope *ptope, const pfBox *box);
int **pfPtopeContainsCyl**(const pfPolytope *ptope, const pfCylinder *cyl);
int **pfPtopeContainsPtope**(const pfPolytope *ptope, const pfPolytope *ptope1);

Since the class **pfPolytope** is itself derived from the parent class **pfObject**, objects of class **pfFrustum** can also be used with these functions designed for objects of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfFrustum** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);

void * **pfGetArena**(void *ptr);

PARAMETERS

frust identifies a pfFrustum.

DESCRIPTION

A pfFrustum represents a viewing and or culling volume bounded by left, right, top, bottom, near and far planes.

pfNewFrust creates and returns a handle to a pfFrustum. *arena* specifies a malloc arena out of which the pfFrustum is allocated or **NULL** for allocation off the process heap. pfFrusta can be deleted with **pfDelete**.

A new pfFrustum defaults to a simple perspective frustum (see **pfMakeSimpleFrust**) with FOV = 45 degrees, and near and far distances of 1 and 1000.

pfGetFrustClassType returns the **pfType*** for the class **pfFrustum**. The **pfType*** returned by **pfGetFrustClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfFrustum**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfMakePerspFrust configures *frust* as a perspective frustum with the eye at (0,0,0) and the points (*left*, *near*, *bottom*) and (*right*, *near*, *top*) being the lower-left and upper-right corners of the viewing plane. The coordinate system used is: left -> right = +X axis, near -> far = +Y axis, bottom -> top = +Z axis. The far plane lies at Y = far. Note that the field of view of a frustum configured with **pfMakePerspFrust** is dependent on the current near plane distance. However, subsequent changes to the near plane distance with **pfFrustNearFar** do not affect the field of view, simplifying clip plane modification.

pfMakePerspFrust is similar to the IRIS GL **window(3g)** command and can generate off-axis projections that are often used for stereo and "video-wall" displays. With an off-axis frustum, the line from the eyepoint passing through the center of the image is not perpendicular to the projection plane.

Example 1:

```
/*
 * Make two off-axis projections which together provide
 * horizontal and vertical FOVs of 90 and 45 degrees.
 */
t = pfTan(22.5f);

pfFrustNearFar(left,  1.0f, 1000.0f);
pfMakePerspFrust(left, -1.0f, 0.0f, -t, t);

pfFrustNearFar(right, 1.0f, 1000.0f);
```

```
pfMakePerspFrust(right, 0.0f, 1.0f, -t, t);
```

pfMakeOrthoFrust configures *frust* as an orthogonal frustum. The 6 sides of the frustum are: x = left, x = right, z = bottom, z = top, y = near, y = far. **pfMakeOrthoFrust** is similar to the IRIS GL **ortho2(3g)** command. The near and far distances of an orthogonal frustum are set by **pfFrustNearFar**.

pfMakeSimpleFrust configures *frust* as an on-axis perspective frustum with horizontal and vertical fields-of-view of *fov* degrees. With an on-axis frustum, the line connecting the center of projection with the eyepoint is perpendicular to the projection plane. **pfMakeSimpleFrust** is similar to the IRIS GL **perspective(3g)** command. The near and far distances of a simple frustum are set by **pfFrustNearFar**. For viewports with non-square aspect ratios, **pfFrustAspect** may be used to automatically fit either the horizontal or vertical fields of view to the viewport (see below).

pfGetFrustType returns a symbolic token indicating the frustum type of *frust* and is one of: **PFFRUST_SIMPLE**, **PFFRUST_ORTHOGONAL**, or **PFFRUST_PERSPECTIVE**. The frustum type is set by the **pfMake<*>Frust** routines. Note that it is possible to make a simple frustum with **pfMakePerspFrust** if *left* == *-right* and *bottom* == *-top*.

pfFrustNearFar sets the near and far distances of *frust*. It will also recalculate the frustum's geometry based on the frustum type. If *frust* is perspective, its field of view will not be changed, only the near and far planes will be modified. **pfGetFrustNearFar** copies the near and far distances of *frust* into *near* and *far*.

pfFrustAspect adjusts the horizontal or vertical extent of *frust* to fit the aspect ratio specified by *widthHeightRatio*. *which* is a symbolic token specifying how to modify *frust* and is one of:

PFFRUST_CALC_NONE	Disable aspect ratio calculation
PFFRUST_CALC_HORIZ	Modify horizontal extent of frustum to match the aspect ratio
PFFRUST_CALC_VERT	Modify vertical extent of frustum to match the aspect ratio

pfFrustAspect is useful for matching a frustum to a viewport:

Example 2:

```
getviewport(&l, &r, &b, &t);
aspect = (float)(r - l) / (float)(t - b);

/*
 * Fit vertical frustum extent to viewport so that horizontal
 * FOV is 45 degrees and vertical is based on 'aspect'.
 */
pfMakeSimpleFrust(frust, 45.0f);
pfFrustAspect(frust, PFFRUST_CALC_VERT, aspect);
```

Frustum aspect ratio matching is not persistent. You must call **pfFrustAspect** each time the frustum changes shape in order to maintain matched frustum/viewport.

pfGetFrustAspect returns the aspect ratio of *frust*.

pfGetFrustFOV copies the total horizontal and vertical fields of view into *fovh* and *fovv* respectively. The fields of view for an orthogonal frustum are both 0.0.

pfGetFrustNear returns the four corners of the near (viewing or projection) plane putting the lower-left, lower-right, upper-left and upper-right vertices into *ll*, *lr*, *ul*, and *ur*, respectively.

pfGetFrustFar returns the four corners of the far plane putting the lower-left, lower-right, upper-left and upper-right vertices into *ll*, *lr*, *ul*, and *ur*, respectively.

pfGetFrustEye copies the eye position of the frustum *frust* into *eye*.

pfGetFrustPtope copies the 6 half spaces which define *frust* into the pfPolytope *ptope*.

pfGetFrustGLProjMat returns the projection matrix corresponding to *frust* in the coordinate system of the Graphics Library, ignoring any transformations applied to *frust* with **pfOrthoXformFrust**.

pfApplyFrust configures the hardware projection matrix with the projection defined by *frust*. Modifications made to *frust* do not have effect until **pfApplyFrust** is called.

pfApplyFrust is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfApplyFrust** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

pfFrustContainsPt returns **TRUE** or **FALSE** depending on whether the point *pt* is in the interior of the specified frustum.

pfFrustContainsBox, **pfFrustContainsSphere** and **pfFrustContainsCyl** test whether the specified pfFrustum contains a non-empty portion of the volume specified by the second argument, a box, sphere or cylinder, respectively.

The return value from these functions is the OR of one or more bit fields. The returned value may be:

PFIS_FALSE: The intersection of the primitive and the pfFrustum is empty.

PFIS_MAYBE: The intersection of the primitive and the pfFrustum might be non-empty.

PFIS_MAYBE | PFIS_TRUE: The intersection of the primitive and the pfFrustum is definitely non-empty.

PFIS_MAYBE | PFIS_TRUE | PFIS_ALL_IN: The primitive is non-empty and lies entirely inside the pfFrustum.

The primary use of **pfFrustContainsSphere** and **pfFrustContainsBox** within IRIS Performer is in culling the database to the view frustum each frame, where speed is paramount. If this computation cannot be done easily, the function returns **PFIS_MAYBE**.

pfOrthoXformFrust transforms the frustum using the matrix *mat*: *dst = src * mat*. If *mat* is not orthogonal the results are undefined.

NOTES

pfFrustum construction orients the frustum with +Z up, +X to the right, and +Y into the screen which is different than both the IRIS GL and OpenGL viewing coordinate systems which have +Y up, +X to the right and -Z into the screen.

SEE ALSO

pfBox, pfDelete, pfDispList, pfMatrix, pfObject, pfPlaneIsectSeg, pfPolytope, pfSphere, pfState, pfVec3, ortho, perspective, window

NAME

pfScale, pfTranslate, pfRotate, pfPushMatrix, pfPushIdentMatrix, pfPopMatrix, pfLoadMatrix, pfMultMatrix – Operate on graphics library matrix stack

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfScale**(float x, float y, float z);

void **pfTranslate**(float x, float y, float z);

void **pfRotate**(int axis, float degrees);

void **pfPushMatrix**(void);

void **pfPushIdentMatrix**(void);

void **pfPopMatrix**(void);

void **pfLoadMatrix**(pfMatrix m);

void **pfMultMatrix**(pfMatrix m);

DESCRIPTION

These functions are similar to the corresponding IRIS GL and OpenGL graphics library matrix stack commands. The only difference is that these IRIS Performer commands may be applied to and retained in a **pfDispList** for subsequent activation.

pfRotate accepts the values **PF_X**, **PF_Y**, and **PF_Z** to select the axis of rotation.

pfPushIdentMatrix is equivalent to calling **pfPushMatrix** followed with a call to **pfLoadMatrix** with an identity matrix.

These routines are all display-listable commands. If a pfDispList has been opened by **pfOpenDList**, these commands will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

SEE ALSO

loadmatrix, multmatrix, popmatrix, pushmatrix, rot, scale, translate

NAME

pfNewGSet, pfGetGSetClassType, pfDrawGSet, pfDrawHlightedGSet, pfGSetNumPrims, pfGetGSetNumPrims, pfGSetPrimType, pfGetGSetPrimType, pfGSetPrimLengths, pfGetGSetPrimLengths, pfGSetAttr, pfGetGSetAttrBind, pfGetGSetAttrLists, pfGetGSetAttrRange, pfGSetDrawMode, pfGetGSetDrawMode, pfGSetGState, pfGetGSetGState, pfGSetGStateIndex, pfGetGSetGStateIndex, pfGSetLineWidth, pfGetGSetLineWidth, pfGSetPntSize, pfGetGSetPntSize, pfGSetHlight, pfGetGSetHlight, pfGSetDrawBin, pfGetGSetDrawBin, pfGSetPassFilter, pfGetGSetPassFilter, pfQueryGSet, pfMQueryGSet, pfGSetBBox, pfGetGSetBBox, pfGSetIsectMask, pfGetGSetIsectMask, pfGSetIsectSegs – Create, modify and query geometry set objects

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfGeoSet * **pfNewGSet**(void *arena);

pfType * **pfGetGSetClassType**(void);

void **pfDrawGSet**(pfGeoSet *gset);

void **pfDrawHlightedGSet**(pfGeoSet* gset);

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

int **pfGetGSetNumPrims**(const pfGeoSet *gset);

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

int **pfGetGSetPrimType**(const pfGeoSet *gset);

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

int * **pfGetGSetPrimLengths**(const pfGeoSet* gset);

void **pfGSetAttr**(pfGeoSet *gset, int attr, int bind, void *alist, ushort *ilist);

int **pfGetGSetAttrBind**(const pfGeoSet *gset, int attr);

void **pfGetGSetAttrLists**(const pfGeoSet *gset, int attr, void **alist, ushort **ilist);

int **pfGetGSetAttrRange**(const pfGeoSet *gset, int attr, int *minIndex, int *maxIndex);

void **pfGSetDrawMode**(pfGeoSet *gset, int mode, int val);

int **pfGetGSetDrawMode**(const pfGeoSet *gset, int mode);

void **pfGSetGState**(pfGeoSet *gset, pfGeoState *gstate);

pfGeoState * **pfGetGSetGState**(const pfGeoSet *gset);

void **pfGSetGStateIndex**(pfGeoSet *gset, int id);

int **pfGetGSetGStateIndex**(const pfGeoSet *gset);

void **pfGSetLineWidth**(pfGeoSet *gset, float width);
float **pfGetGSetLineWidth**(const pfGeoSet *gset);
void **pfGSetPntSize**(pfGeoSet *gset, float size);
float **pfGetGSetPntSize**(const pfGeoSet *gset);
void **pfGSetHlight**(pfGeoSet* gset, pfHighlight *hlight);
pfHighlight * **pfGetGSetHlight**(const pfGeoSet* gset);
void **pfGSetDrawBin**(pfGeoSet *gset, short bin);
int **pfGetGSetDrawBin**(const pfGeoSet *gset);
void **pfGSetPassFilter**(uint mask);
uint **pfGetGSetPassFilter**(void);
int **pfQueryGSet**(const pfGeoSet* gset, uint which, void* dst);
int **pfMQueryGSet**(const pfGeoSet* gset, uint* which, void* dst);
void **pfGSetBBox**(pfGeoSet *gset, pfBox *box, int mode);
void **pfGetGSetBBox**(pfGeoSet *gset, pfBox *box);
void **pfGSetIsectMask**(pfGeoSet *gset, uint mask, int setMode, int bitOp);
uint **pfGetGSetIsectMask**(pfGeoSet *gset);
int **pfGSetIsectSegs**(pfGeoSet *gset, pfSegSet *segSet, pfHit **hits[]);

```
typedef struct
{
    int        mode;
    void*      userData;
    pfSeg      segs[PFIS_MAX_SEGS];
    uint       activeMask;
    uint       isectMask;
    void*      bound;
    int        (*discFunc)(pfHit*);
} pfSegSet;
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfGeoSet** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfGeoSet**. Casting an object of class **pfGeoSet** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
void*   pfGetUserData(pfObject *obj);
int     pfGetGLHandle(pfObject *obj);
```

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfGeoSet** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *      pfGetType(const void *ptr);
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

PARAMETERS

gset identifies a pfGeoSet.

attr is a symbolic token that identifies a specific attribute type and is one of:

PFGS_COLOR4	*alist* must be list of pfVec4 colors
PFGS_NORMAL3	*alist* must be list of pfVec3 normals,
PFGS_TEXCOORD2	*alist* must be list of pfVec2 texture coordinates,
PFGS_COORD3	*alist* must be list of pfVec3 coordinates.

bind is a symbolic token that specifies an attribute binding type and is one of:

PFGS_OFF	*attr* is not specified and is thus inherited,
PFGS_OVERALL	*attr* is specified once for the entire pfGeoSet,
PFGS_PER_PRIM	*attr* is specified once per primitive,
PFGS_PER_VERTEX	*attr* is specified once per vertex.

DESCRIPTION

The pfGeoSet (short for "Geometry Set") is a fundamental IRIS Performer data structure. Each pfGeoSet is a collection of geometry with one primitive type, such as points, lines, triangles. Each pfGeoSet also has a single combination of texture, normal, and color attribute bindings, such as "untextured with colors per vertex and normals per primitive". A pfGeoSet forms primitives out of lists of attributes which may or may not be indexed. An indexed pfGeoSet uses a list of unsigned shorts to index an attribute list.

Indexing provides a more general mechanism for specifying geometry than hardwired attribute lists and also has the potential for substantial memory savings due to shared attributes. Nonindexed pfGeoSet's are sometimes easier to construct and may save memory in situations where vertex sharing is not possible since index lists are not required. Nonindexed pfGeoSet's also require fewer CPU cycles to traverse and may exhibit better caching behavior. A pfGeoSet is either completely indexed or non-indexed; hybrid pfGeoSets that have some attributes indexed and others non-indexed are not supported. For theses cases, simply construct an identity-map index list and specify it with each "non-indexed" pfGeoSet attribute array.

pfNewGSet creates and returns a handle to a pfGeoSet. *arena* specifies a malloc arena out of which the pfGeoSet is allocated or **NULL** for allocation off the process heap. pfGeoSets can be deleted with **pfDelete**.

pfGeoSet Attributes

pfGSetAttr sets a pfGeoSet attribute binding type, attribute list, and attribute index list. An "attribute" is either coordinate, color, normal or texture coordinate which is supplied in list form to the pfGeoSet. The optional attribute index list is a list of unsigned short integers which index the attribute list. The attribute binding type specifies how the lists are interpreted to define geometry, for example, does the color attribute list provide a color for each vertex (PFGS_PER_VERTEX) or just an overall color for the entire pfGeoSet (PFGS_OVERALL)?

Only certain combinations of attributes and binding types make sense. For example, vertices clearly must be specified per-vertex and the utility of texture coordinates specified other than per-vertex is questionable. The following table shows the allowed combinations:

Binding	Attribute Type			
	COLOR4	**NORMAL3**	**TEXCOORD2**	**COORD3**
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

An **OVERALL** binding requires an index list of length 1 for indexed pfGeoSets. The value of *bind* is unimportant for *attr* = **PFGS_COORD3** since vertices are always specified on a per-vertex basis. Default bindings are OFF for all attributes except coordinates.

ilist, if not NULL, is an index array which indexes the attribute array, *alist*. If *ilist* is NULL, the pfGeoSet is non-indexed and accesses the attribute list in sequential order.

If any attribute's binding is not **PFGS_OFF** and the corresponding *ilist* is defined as NULL, the pfGeoSet is considered to be non-indexed and ALL other specified index lists will be ignored. Nonindexed interpretation of an attribute list is equivalent to using an index list whose elements are 0,1,2,...,N-1.

Consequently it is possible to emulate a pfGeoSet with mixed indexed and non-indexed attributes by using an index array whose elements are 0,1,2,...,N-1 with N being the largest possible index.

If attribute and index lists are allocated from the **pfMalloc** routines, **pfGSetAttr** will correctly update the reference counts of the lists. Specifically, **pfGSetAttr** will decrement the reference counts of the old lists and increment the reference counts of the new lists. It will not free any lists whose reference counts reach 0. When a pfGeoSet is deleted with **pfDelete**, all pfMalloc'ed lists will have their reference counts decremented by one and will be freed if their count reaches 0.

When pfGeoSets are copied with **pfCopy**, all pfMalloc'ed lists of the source pfGeoSet will have their reference counts incremented by one and those pfMalloc'ed lists of the destination pfGeoSet will have their reference counts decremented by one. **pfCopy** copies lists only by reference (only the pointer is copied) and will not free any lists whose reference counts reach 0.

Attribute lists may be any of the following types of memory:

1. Data allocated with **pfMalloc**. This is the usual, and recommended memory type for pfGeoSet index and attribute arrays.
2. Static, malloc(), amalloc(), usmalloc() etc, data subsequently referred to as non-pfMalloc'ed data. This type of memory is not generally recommended since it does not support reference counting or other features provided by **pfMalloc**. In particular, the use of static data is highly discouraged and may result in segmentation violations.
3. pfCycleBuffer memory. In a pipelined, multiprocessing environment, pfCycleBuffers provide multiple data buffers which allow frame-accurate data modifications to pfGeoSet attribute arrays like coordinates (facial animation), and texture coordinates (ocean waves, surf). **pfGSetAttr** will accept a pfCycleBuffer* or pfCycleMemory* for the attribute list (index lists do not yet support pfCycleBuffer) and *gset* will select the appropriate buffer when rendered or intersected with. See pfCycleBuffer for more details.

pfGetGSetAttrBind returns the binding type of *attr* and **pfGetGSetAttrLists** returns the attribute and index list base pointers. If the gset is non-indexed, send down a dummy ushort pointer instead of NULL as ilist.

pfGetGSetAttrRange returns the range of attributes in the attribute list identified by *attr* that are used by *gset*. (The total size, in bytes, of the list may be queried through **pfGetSize** if the list was allocated by **pfMalloc**.) If the list is non-indexed, **pfGetGSetAttrRange** returns the number of contiguous attributes accessed by *gset* (the range implicitly beginning at 0). If the list is indexed, **pfGetGSetAttrRange** returns the same value as in the non-indexed case but also copies the minimum and maximum indices into *minIndex* and *maxIndex*. If the attribute list is non-indexed, or the attribute binding type is **PFGSS_OFF**, 0 and -1 are returned in *minIndex* and *maxIndex*. **NULL** may be passed instead of *minIndex* and/or *maxIndex* when the min/max index is not required.

pfGetGSetAttrRange is typically used to allocate a new attribute array when cloning a pfGeoSet:

```
int numVerts = pfGetGSetAttrRange(gset, PFGS_COORD3, NULL, &max);
numVerts = PF_MAX2(numVerts, max + 1);
newVerts = (pfVec3*) pfMalloc(sizeof(pfVec3) * numVerts, arena);
```

pfGeoSet Primitive Types

pfGSetPrimType specifies the type of primitives found in a pfGeoSet. *type* is a symbolic token and is one of:

PFGS_POINTS
PFGS_LINES
PFGS_LINESTRIPS
PFGS_FLAT_LINESTRIPS
PFGS_TRIS
PFGS_QUADS
PFGS_TRISTRIPS
PFGS_FLAT_TRISTRIPS
PFGS_POLYS

The primitive type dictates how the coordinate and coordinate index lists are interpreted to form geometry. See below for a description of primitive types. **pfGetGSetPrimType** returns the primitive type of *gset*.

pfGSetNumPrims and **pfGetGSetNumPrims** sets/gets the number of primitives in *gset*. A primitive is a single point, line segment, line strip, triangle, quad, triangle strip, or polygon depending on the primitive type.

A single line strip, triangle strip, or polygon is considered to be a primitive so a pfGeoSet may contain multiple strips of differing lengths or multiple polygons with differing number of sides. Therefore, for strip primitives and **PFGS_POLYS**, a separate array is necessary which specifies the number of vertices in each strip or polygon. This array is set by **pfGSetPrimLengths**. *lengths* is an array of vertex counts such that lengths[0] = number of vertices in strip/polygon 0, lengths[1] = number of vertices in strip/polygon 1,..., lengths[n-1] = number of vertices in strip/polygon n-1 where n is the number of primitives set by **pfGSetNumPrims**. **pfGetGSetPrimLengths** returns a pointer to the lengths array of *gset*.

Assuming the coordinate index list is an array V indexed by i, *num* is the number of primitives, *lengths* is the array of strip or polygon lengths and Nv the size of the coordinate index list, the primitive type interprets V in the following ways:

PFGS_POINTS
The pfGeoSet is a set of *num* points. Each V[i] is a point, i = 0, 1, 2, ..., *num*-1. Nv = *num*.

PFGS_LINES
The pfGeoSet is a set of *num* disconnected line segments. Each line segment is drawn from V[i] to V[i+1], i = 0, 2, ..., 2*(*num*-1). Nv = 2 * *num*.

PFGS_LINESTRIPS
The pfGeoSet is a set of *num* line strips (also known as polylines). Linestrip[i] is drawn between V[p+j], j = 0, 1, ..., *lengths*[i]-1, where p is sum of all *lengths*[k], 0 <= k < i. Nv = sum of all *lengths*[k], k = 0, 1, ..., *num*-1. Note that all *lengths*[i] values should be >= 2.

PFGS_FLAT_LINESTRIPS
The pfGeoSet is a set of *num* line strips (also known as polylines). Linestrip[i] is drawn between V[p+j], j = 0, 1, ..., *lengths*[i]-1, where p is sum of all *lengths*[k], 0 <= k < i. Nv = sum of all *lengths*[k], k = 0, 1, ..., *num*-1. Note that all *lengths*[i] value should be >= 2.

PFGS_TRIS
The pfGeoSet is a set of *num* independent triangles. Each triangle is V[i], V[i+1], V[i+2], i = 0, 3, 6, ..., 3*(*num*-1). Nv = 3 * *num*.

PFGS_QUADS
The pfGeoSet is a set of *num* independent quads. Each quad is V[i], V[i+1], V[i+2], V[i+3], i = 0, 4, 8, ..., 4*(*num*-1). Nv = 4 * *num*.

PFGS_TRISTRIPS
The pfGeoSet is a set of *num* triangle strips. Tristrip[i] is drawn between V[p+j], j = 0, 1, ..., *lengths*[i]-1, where p is sum of all *lengths*[k], 0 <= k < i. Nv = sum of all *lengths*[k], k = 0, 1, ..., *num*-1. Note that all *lengths*[i] values should be >= 3.

PFGS_FLAT_TRISTRIPS
The pfGeoSet is a set of *num* triangle strips. Tristrip[i] is drawn between V[p+j], j = 0, 1, ..., *lengths*[i]-1, where p is sum of all *lengths*[k], 0 <= k < i. Nv = sum of all *lengths*[k], k = 0, 1, ..., *num*-1. Note that all *lengths*[i] should be >= 3.

PFGS_POLYS
The pfGeoSet is a set of *num* polygons. Polygon[i] is the convex hull of the vertices V[p+j], j = 0, 1, ..., *lengths*[i]-1 where p is sum of all *lengths*[k], 0 <= k < i. Nv = sum of all *lengths*[k], k = 0, 1, ..., *num*-1. Note that all *lengths*[i] should be >= 3.

PFGS_TRIS, **PFGS_QUADS**, **PFGS_TRISTRIPS**, **PFGS_FLAT_TRISTRIPS**, and **PFGS_POLYS** are rendered as filled polygons but will be rendered in wire-frame according to the following rules:

1. Always render in wireframe mode if **PFEN_WIREFRAME** mode is enabled through **pfGSetDrawMode**.

2. Use the wireframe mode set by the attached pfGeoState, if any, as described in **pfGSetGState** below.
3. Use the wireframe mode set by **pfEnable** or **pfDisable** with the **PFEN_WIREFRAME** argument.

A **PFGS_PER_VERTEX** binding for **PFGS_COLOR4** and **PFGS_NORMAL3** is interpreted differently for **PFGS_FLAT_LINESTRIPS** and **PFGS_FLAT_TRISTRIPS** primitive types. With flat-shaded strip primitives, only the last vertex in each primitive defines the shading of the primitive (see **pfShadeModel**.) Thus the first vertex in a FLAT_LINESTRIP and the first two vertices in a FLAT_TRISTRIP do not require normals or colors. Consequently when specifying a **PFGS_PER_VERTEX** binding for either colors or normals, you should not specify a color or normal for the first vertex of a line strip or for the first 2 vertices of a triangle strip. **pfDrawGSet** will automatically set the shading model to **FLAT** before rendering **PFGS_FLAT_** primitives.

Example 1:

```
/* Set up a non-indexed, FLAT_TRISTRIP pfGeoSet */
gset = pfNewGSet(NULL);
pfGSetPrimType(gset, PFGS_FLAT_TRISTRIPS);
pfGSetNumPrims(gset, 2);
lengths[0] = 4;
lengths[1] = 3;
pfGSetPrimLengths(gset, lengths);

/* Only need 3 colors: 2 for 1st strip and 1 for 2nd */
colors = (pfVec4*) pfMalloc(sizeof(pfVec4) * 3, NULL);

pfGSetAttr(gset, PFGS_COLOR4, PFGS_PER_VERTEX, colors, NULL);
pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX, coords, NULL);
```

pfGeoSet Special Rendering Characteristics

When colortable mode is enabled, either through **pfEnable** or through **pfApplyGState**, a pfGeoSet will not use its local color array but will use the color array supplied by the currently active pfColortable (See the **pfColortable** and **pfEnable** manual pages). pfColortables will affect both indexed and non-indexed pfGeoSets.

A pfGeoSet of type **PFGS_POINTS** will be rendered with the special characteristics of light points if a pfLPointState has been applied. Light point features include:

1. Perspective size.
2. Perspective fading.
3. Fog punch-through.
4. Directionality.
5. Intensity.

See pfLPointState for more details.

pfGSetPntSize and **pfGSetLineWidth** set the point size and line width of *gset*. Point size has effect only when the primitive type is **PFGS_POINTS** and line width is used only for primitive types **PFGS_LINES**, **PFGS_LINESTRIPS**, **PFGS_FLAT_LINESTRIPS** and for all primitives drawn in wireframe mode. A pfGeoSet sets point size and line width immediately before rendering only if the size/width is greater than zero. Otherwise it will inherit size/width through the Graphics Library.

pfGetGSetPntSize and **pfGetGSetLineWidth** return *gset*'s point size and line width respectively.

pfGSetDrawMode further characterizes a pfGeoSet's primitive type as flat-shaded, wireframe or compiled. *mode* is a symbolic token specifying the mode to set and is one of:

PFGS_FLATSHADE	Always render *gset* with a flat shading model.
PFGS_WIREFRAME	Always render and intersect *gset* in wireframe. For rendering in wireframe and intersection with solid geometry, enable wireframe on an attached pfGeoState (See **pfGSetGState**).
PFGS_COMPILE_GL	Compile *gset*'s geometry into a GL display list and subsequently render the display list.

val is **PF_ON** or **PF_OFF** to enable/disable the mode.

If a pfGeoSet has very few primitives, the CPU overhead in **pfDrawGSet** may become noticeable. In this situation, it is reasonable to compile the pfGeoSet into a GL display list which has very little CPU overhead. However, GL display lists have several drawbacks that must be considered:

Storage

GL display lists will increase memory usage because every vertex, color, etc is copied into the display list, thus duplicating the pfGeoSet's attribute arrays. Additionally, GL display lists cannot index and so do not benefit from vertex sharing.

While it is possible to delete the attribute arrays after the pfGeoSet has been compiled to free up some memory, it will no longer be possible to intersect with the pfGeoSet's geometry (see **pfGSetIsectSegs**).

Flexibility
: Once in a GL display list, attributes like coordinates and normals may not be modified. This precludes dynamic geometry like water and facial animation.

Coherency
: If any attribute of the pfGeoSet changes then the burden is on the user to regenerate the GL display list through **pfGSetDrawMode**.

In summary, applications with many very small pfGeoSets each of which defines static unchanging geometry may be suitable for pfGeoSet compilation into GL display lists.

The mechanism of **PFGS_COMPILE_GL** is illustrated in the following example:

```
/* We assume 'gset' is already "built" by this point */

/* Enable GL display list compilation and rendering */
pfGSetDrawMode(gset, PFGS_COMPILE_GL, PF_ON);

/*
 * The first pfDrawGSet after pfGSetDrawMode will compile
 * the pfGeoSet into a GL display list.  Note that this is
 * a very slow procedure and is generally done at
 * initialization time.
 */
pfDrawGSet(gset);
 :
/* This time we draw the GL display list */
pfDrawGSet(gset);
 :
/* Disable GL display list mode */
pfGSetDrawMode(gset, PFGS_COMPILE_GL, PF_OFF);

/* Free the GL display list and render 'gset' in immediate mode */
pfDrawGSet(gset);
```

Deciding which shading model to used when draw a pfGeoSet is performed with the following decision hierarchy:

1. Use flat shading if pfGeoSet consists of either **PFGS_FLAT_TRISTRIPS** or **PFGS_FLAT_LINESTRIPS** or if the mode **PFGS_FLATSHADE** is enabled with **pfGSetDrawMode**.

2. Use the shading model specified by the pfGeoState bound to the pfGeoSet. This is the typical case in IRIS Performer. See the **pfGSetGState** description below for further details.
3. Use the shading model set by **pfShadeModel**.

pfGetGSetDrawMode returns the value of *mode* or -1 if *mode* is an unknown mode.

pfGeoSets (Geometry) and pfGeoStates (Appearance)

A pfGeoState is an encapsulation of libpr graphics modes and attributes (see **pfState**). For example, a pfGeoState representing a glass surface may reference a shiny pfMaterial and enable transparency. A pfGeoState does not inherit state from other pfGeoStates. Consequently, when attached to a pfGeoSet via **pfGSetGState**, *gset* will always be rendered with the state encapsulated by *gstate*, regardless of the order in which pfGeoSet/pfGeoState pairs are rendered. This behavior greatly eases the burden of managing graphics state in the graphics library. A pfGeoSet may directly reference or indirectly index a pfGeoState through a global table.

pfGSetGState "attaches" *gstate* to *gset* so that *gset* may be drawn with a certain graphics state. When drawn by **pfDrawGSet**, a pfGeoSet will apply its pfGeoState (if it has one) with **pfApplyGState** and the graphics library will be initialized to the proper state for drawing *gset*. A *gstate* value of **NULL** will remove any previous pfGeoState and cause *gset* to inherit whatever graphics state is around at the time of rendering.

pfGSetGStateIndex allows a pfGeoSet to index its pfGeoState. Indexing is useful for efficiently managing a single database with multiple appearances, e.g., a normal vs. an infrared view of a scene would utilize 2 pfGeoState tables, each referencing a different set of pfGeoStates.

Indexed pfGeoStates use a global table of pfGeoState* specified by **pfApplyGStateTable**. When indexing a pfGeoState, **pfDrawGSet** calls **pfApplyGState** with the *index*th entry of this table if the index can be properly resolved. Otherwise no pfGeoState is applied. **pfGetGSetGStateIndex** returns the pfGeoState index of *gset* or -1 if *gset* directly references its pfGeoState.

pfGSetGState increments the reference count of the new pfGeoState by one and decrements the reference count of the previous pfGeoState by one but does not delete the previous pfGeoState if its reference count reaches zero. **pfGSetGStateIndex** does not affect pfGeoState reference counts.

It is important to understand and remember that any pfGeoSet without an associated pfGeoState will **not** be rendered with the global, default state but will be drawn with the current state. To inherit the global state, a pfGeoState which inherits all state elements should be attached to the pfGeoSet. pfGeoSets should share like pfGeoStates for space and rendering time savings. See the **pfGeoState** reference page for full details.

pfGetGSetGState returns the pfGeoState associated with *gset* or **NULL** if there is none. If *gset* indexes its pfGeoState, **pfGetGSetGState** will look up the pfGeoState index in the global pfGeoState table and return

the result or **NULL** if it cannot resolve the reference.

pfGSetHlight sets *hlight* to be the highlighting structure used for *gset*. When this flag is not **PFHL_OFF**, this gset will be drawn as highlighted unless highlighting has been overridden as off with **pfOverride**. See the **pfHighlight** manual page for information of creating and configuring a highlighting state structure. **pfGetGSetHlight** returns the current GeoSet highlight definition.

pfDrawHlightedGSet is a convenience routine for drawing ONLY the highlighting stage of *gset*, according to the currently active highlighting structure.

Drawing pfGeoSets

pfDrawGSet is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfDrawGSet** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

If *gset* has an attached pfGeoState, then **pfDrawGSet** first calls **pfApplyGState** before rendering the pfGeoSet geometry, as shown in the following examples.

Example 3a:

```
/* Make sure 'gset' has not attached pfGeoState */
pfGSetGState(gset, NULL);

/* Apply graphics state encapsulated by 'gstate' */
pfApplyGState(gstate);

/* Draw 'gset' with graphics state encapsulated by 'gstate' */
pfDrawGSet(gset);
```

Example 3b:

```
/* Attach 'gstate' to 'gset' */
pfGSetGState(gset, gstate);

/* Draw 'gset' with graphics state encapsulated by 'gstate' */
pfDrawGSet(gset);
```

Example 3c:

```
/* Use indexed pfGeoState */
pfGSetGStateIndex(gset, 2);

/* Set up and apply pfGeoState table */
pfSet(list, 2, gstate);
pfApplyGStateTable(list);

/* Draw 'gset' with graphics state encapsulated by 'gstate' */
pfDrawGSet(gset);
```

Examples 3a, 3b, and 3c are equivalent methods for drawing the same thing. Method 3b is recommended though since the pfGeoState and pfGeoSet pair can be set up at database initialization time.

pfGSetDrawBin sets *gset*'s draw bin identifier to *bin*. *bin* identifies a drawing bin to which *gset* belongs and is used for controlling the rendering order of a database. The pfGeoSet draw bin is currently used only by **libpf** applications (see **pfChanBinOrder**) and is ignored by **libpr**-only applications. The default pfGeoSet draw bin identifier is -1. **pfGetGSetDrawBin** returns the draw bin identifier of *gset*.

The *mask* argument to **pfGSetPassFilter** is a bitmask which specifies a pfGeoSet drawing "filter". Only pfGeoSets which pass the filter test are rendered by **pfDrawGSet**. *mask* consists of the logical OR of the following:

PFGS_TEX_GSET
Draw only textured pfGeoSets

PFGS_NONTEX_GSET
Draw only non-textured pfGeoSets

PFGS_EMISSIVE_GSET
Draw only pfGeoSets which use an emissive pfMaterial or pfLPointState.

PFGS_NONEMISSIVE_GSET
Draw only non-emissive pfGeoSets

PFGS_LAYER_GSET
Draw only pfGeoSets which are layer (as opposed to base) geometry.

PFGS_NONLAYER_GSET
Draw only pfGeoSets which are not layer geometry.

A *mask* of 0 disables pfGeoSet filtering. Filtering is useful for multipass rendering techniques. **pfGetGSetPassFilter** returns the current pfGeoSet filtering mask.

Intersecting with pfGeoSets

pfGSetIsectMask enables intersections and sets the intersection mask for *gset*. *mask* is a 32-bit mask used to determine whether a particular pfGeoSet should be examined during a particular intersection request. A non-zero bit-wise AND of the pfGeoSet's mask with the mask of the intersection request (-**pfGSetIsectSegs**) indicates that the pfGeoSet should be tested. The default mask is all 1's, i.e. 0xffffffff.

pfGetGSetIsectMask returns the intersection mask of the specified pfGeoSet.

Intersections for geometry whose vertex coordinates don't change are more efficient when information is cached for each pfGeoSet to be intersected with. When setting the mask or changing caching, **PFTRAV_SELF** should always be part of *setMode*. OR-ing **PFTRAV_IS_CACHE** into *setMode* causes the creation or update of the cache. Because creating the cache requires a moderate amount of computation, it is best done at setup time.

For objects whose geometry changes only occasionally, additional calls to **pfGSetIsectMask** with **PFTRAV_IS_CACHE** OR-ed into *setMode* will recompute the cached information. Alternately, OR-ing **PFTRAV_IS_UNCACHE** into *setMode* will disable caching.

The *bitOp* argument is one of **PF_AND**, **PF_OR**, or **PF_SET** and indicates, respectively, whether the new mask is derived from AND-ing with the old mask, OR-ing with the old mask or simply set.

pfGSetBBox sets the bounding volume of *gset*. Each pfGeoSet has an associated bounding volume used for culling and intersection testing and a bounding mode, either static or dynamic. By definition, the bounding volume of a node encloses all the geometry parented by node, which means that the node and all its children fit within the node's bounding volume.

The *mode* argument to **pfGSetBBox** specifies whether or not the bounding volume for *node* should be recomputed when an attribute of *gset* is changed. If the mode is **PFBOUND_STATIC**, IRIS Performer will not modify the bound once it is set or computed. If the mode is **PFBOUND_DYNAMIC**, IRIS Performer will recompute the bound if the number of primitives, the primitive lengths array or the vertex coordinate arrays are changed. Note that IRIS Performer does not know if the contents of these arrays changes, only when the pointer itself is set. Recomputation of the bounding box can be forced by calling **pfGSetBBox** with a *bbox* that is **NULL**.

pfGetGSetBBox copies the bounding box of *gset* into *bbox* and returns the current bounding mode.

pfGSetIsectSegs tests for intersection between the pfGeoSet *gset* and the group of line segments specified in *segSet*. The resulting intersections (if any) are returned in *hits*. The return value of **pfGSetIsectSegs** is the number of segments that intersected the pfGeoSet.

hits is an empty array supplied by the user through which results are returned. The array must have an entry for each segment in *segSet*. Upon return, *hits*[i][0] is a pfHit* which gives the intersection result for the *i*th segment in *segSet*. The pfHit objects come from an internally maintained pool and are reused on

subsequent requests. Hence, the contents are only valid until the next invocation of **pfGSetIsectSegs** in the current process. They should not be freed by the application.

segSet is a pfSegSet public structure specifying the intersection request. In the structure, *segs* is an array of line segments to be intersected against the pfGeoSet. *activeMask* is a bit vector specifying which segments in the SegSet are to be active for the current request. If the i'th bit is set to 1, it indicates the corresponding segment in the *segs* array is active.

The bit vector *mode* specifies the behavior of the intersection process and is a bitwise OR of the following:

PFTRAV_IS_PRIM	Intersect with primitives (quads or triangles)
PFTRAV_IS_GSET	Intersect with pfGeoSet bounding boxes
PFTRAV_IS_NORM	Return normal in the pfHit structure
PFTRAV_IS_CULL_BACK	Ignore backfacing polygons
PFTRAV_IS_CULL_FRONT	Ignore front-facing polygons

The bit fields **PFTRAV_IS_PRIM** and **PFTRAV_IS_GSET**, indicate the level at which intersections should be evaluated and discriminator callbacks, if any, invoked. Note that if neither of these level selectors are specified, no intersection testing is done at all. In the pfSegSet, *isectMask* is another bit vector. It is bit-wise AND-ed with the intersection mask of the pfGeoSet. If the result is zero no intersection testing is done.

The *bound* field in a pfSegSet is an optional user provided bounding volume around the set of segments. Currently, the only supported volume is a cylinder. To use a bounding cylinder, bitwise OR **PFTRAV_IS_BCYL** into the *mode* field of the pfSegSet and assign the pointer to the bounding volume to the *bound* field. **pfCylAroundSegs** will construct a cylinder around the segments.

When a bounding volume is supplied, the bounding volume is tested against the pfGeoSet bounding box before examining the individual segments. The largest improvement is for groups of at least several segments which are closely grouped segments. Placing a bounding cylinder around small groups or widely dispersed segments can decrease performance.

The *userData* pointer allows an application to associate other data with the pfSegSet. Upon return and in discriminator callbacks, the pfSegSet's *userData* pointer can be obtained from the returned pfHit with **pfGetUserData**.

discFunc is a user supplied callback function which provides a more powerful means for controlling intersections than the simple mask test. The function acts as a discriminating function which examines information about candidate intersections and judges their validity. When a candidate intersection occurs, the *discFunc* callback is invoked with a pfHit structure containing information about the intersection.

The callback may then return a value which indicates whether and how the intersection should continue.

This value is composed of the following major action specifications with additional modifiers bitwise-OR-ed in as explained below.

PFTRAV_CONT
Indicates that the process should continue traversing the primitive list.

PFTRAV_PRUNE
Stops further testing of the line segment against the current pfGeoSet.

PFTRAV_TERM
Stops further testing of the line segment completely.

To have **PFTRAV_TERM** or **PFTRAV_PRUNE** apply to all segments, **PFTRAV_IS_ALL_SEGS** can be OR-ed into the discriminator return value. This causes the entire traversal to be terminated or pruned.

The callback may OR into the status return value any of:

PFTRAV_IS_IGNORE
Indicates that the current intersection should be ignored, otherwise the intersection is taken as valid.

PFTRAV_IS_CLIP_START
Indicates that for pruned and continued traversals that before proceeding the segment should be clipped to start at the current intersection point.

PFTRAV_IS_CLIP_END
Indicates that for pruned and continued traversals that before proceeding the segment should be clipped to end at the current intersection point.

If *discFunc* is NULL, the behavior is the same as if the discriminator returned (**PFTRAV_CONT | PFTRAV_IS_CLIP_END**), so that the intersection nearest the start of the segment will be returned.

A pfHit object also conveys information to the discriminator callback, if any. The following table lists the information which can be obtained from an pfHit.

Query	Type	Contents
PFQHIT_FLAGS	int	Status flags
PFQHIT_SEGNUM	int	Index of segment in pfSegSet
PFQHIT_SEG	pfSeg	Segment, as clipped
PFQHIT_POINT	pfVec3	Intersection point
PFQHIT_NORM	pfVec3	Normal at intersection point
PFQHIT_VERTS	pfVec3[3]	Vertices of intersected triangle
PFQHIT_TRI	int	Index of triangle in pfGeoSet primitive
PFQHIT_PRIM	int	Index of primitive in pfGeoSet
PFQHIT_GSET	pfGeoSet *	Pointer to intersected pfGeoSet
PFQHIT_NODE	pfNode *	Pointer to pfGeode
PFQHIT_NAME	char *	Name of pfGeode
PFQHIT_XFORM	pfMatrix	Transformation matrix
PFQHIT_PATH	pfPath *	Path within scene graph

pfQueryGSet is a convenience routine for determining the values of implicit pfGeoSet parameters. The *which* argument is a token which selects the parameter from the set **PFQGSET_NUM_TRIS** and **PFQGSET_NUM_VERTS**. The result is written to the address indicated by *dst*. The number of bytes written to *dst* is returned as the value of **pfQueryGSet**. **pfMQueryGSet** is similar but copies a series of items sequentially into the buffer specified by *dst*. The items and their order are defined by a NULL-terminated array of query tokens pointed to by *which*. For both functions, the return value is the number of bytes written to the destination buffer.

pfGetGSetClassType returns the **pfType*** for the class **pfGeoSet**. The **pfType*** returned by **pfGetGSetClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfGeoSet**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

NOTES

The following example shows one way to create a pfGeoSet defining a hexahedron, which is also known as a cube.

```
static pfVec3 coords[] =
{
    {-1.0, -1.0,  1.0},
    { 1.0, -1.0,  1.0},
    { 1.0,  1.0,  1.0},
    {-1.0,  1.0,  1.0},
    {-1.0, -1.0, -1.0},
    { 1.0, -1.0, -1.0},
```

```
    { 1.0,  1.0, -1.0},
    {-1.0,  1.0, -1.0}
};

static ushort cindex[] =
{
    0, 1, 2, 3,    /* front  */
    0, 3, 7, 4,    /* left   */
    4, 7, 6, 5,    /* back   */
    1, 5, 6, 2,    /* right  */
    3, 2, 6, 7,    /* top    */
    0, 4, 5, 1     /* bottom */
};

static pfVec3 norms[] =
{
    { 0.0,  0.0,  1.0},
    { 0.0,  0.0, -1.0},
    { 0.0,  1.0,  0.0},
    { 0.0, -1.0,  0.0},
    { 1.0,  0.0,  0.0},
    {-1.0,  0.0,  0.0}
};

static ushort nindex[] =
{
    0,
    5,
    1,
    4,
    2,
    3
};

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

```
/* Set up an indexed PFGS_QUADS pfGeoSet */
gset = pfNewGSet(NULL);
pfGSetPrimType(gset, PFGS_QUADS);
pfGSetNumPrims(gset, 6);
pfGSetAttr(gset, PFGS_COORD3,  PFGS_PER_VERTEX,
                  memdup(coords, sizeof(coords), NULL),
                  (ushort*)memdup(cindex, sizeof(cindex), NULL));

pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_PRIM,
                  memdup(norms, sizeof(norms), NULL),
                  (ushort*)memdup(nindex, sizeof(nindex), NULL));
```

BUGS

In IRIS GL, **PFGS_POLYS** are rendered as triangle strips for best performance so that in wireframe the edges internal to the polygon are visible. In OpenGL the internal edges will not be visible.

SEE ALSO

pfApplyGState, pfColortable, pfCopy, pfCycleBuffer, pfDelete, pfDisable, pfDispList, pfEnable, pfGSetDrawMode, pfGeoState, pfHit, pfLPointState, pfMalloc, pfMaterial, pfNewHlight, pfObject, pfGSetIsectSegs, pfShadeModel, pfState

NAME

pfNewGState, pfGetGStateClassType, pfApplyGState, pfLoadGState, pfGStateMode, pfGetGStateMode, pfGetGStateCurMode, pfGetGStateCombinedMode, pfGStateVal, pfGetGStateVal, pfGetGStateCurVal, pfGetGStateCombinedVal, pfGStateInherit, pfGetGStateInherit, pfGStateAttr, pfGetGStateAttr, pfGetGStateCurAttr, pfGetGStateCombinedAttr, pfGStateFuncs, pfGetGStateFuncs, pfApplyGStateTable, pfMakeBasicGState, pfGetCurGState, pfGetCurGStateTable, pfGetCurIndexedGState – Create, modify and query geometry state objects

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfGeoState * **pfNewGState**(void *arena);

pfType * **pfGetGStateClassType**(void);

void **pfApplyGState**(pfGeoState *gstate);

void **pfLoadGState**(pfGeoState *gstate);

void **pfGStateMode**(pfGeoState *gstate, int mode, int val);

int **pfGetGStateMode**(const pfGeoState *gstate, int mode);

int **pfGetGStateCurMode**(const pfGeoState *gstate, int mode);

int **pfGetGStateCombinedMode**(const pfGeoState *gstate, int mode,
const pfGeoState *combGState);

void **pfGStateVal**(pfGeoState *gstate, int gsval, float val);

float **pfGetGStateVal**(const pfGeoState *gstate, int gsval);

float **pfGetGStateCurVal**(const pfGeoState *gstate, int gsval);

float **pfGetGStateCombinedVal**(const pfGeoState *gstate, int gsval,
const pfGeoState *combGState);

void **pfGStateInherit**(pfGeoState *gstate, uint mask);

uint **pfGetGStateInherit**(const pfGeoState *gstate);

void **pfGStateAttr**(pfGeoState *gstate, int attr, void *data);

void * **pfGetGStateAttr**(const pfGeoState *gstate, int attr);

void * **pfGetGStateCurAttr**(const pfGeoState *gstate, int attr);

void * **pfGetGStateCombinedAttr**(const pfGeoState *gstate, int attr,
const pfGeoState *combGState);

void **pfGStateFuncs**(pfGeoState* gstate, pfGStateFuncType preFunc,
pfGStateFuncType postFunc, void *data);

void **pfGetGStateFuncs**(const pfGeoState* gstate, pfGStateFuncType *preFunc, pfGStateFuncType *postFunc, void **data);
void **pfApplyGStateTable**(pfList *gstab);
void **pfMakeBasicGState**(pfGeoState *gstate);
pfGeoState * **pfGetCurGState**(void);
pfList* **pfGetCurGStateTable**(void);
pfGeoState* **pfGetCurIndexedGState**(int index);

```
typedef int (*pfGStateFuncType)(pfGeoState *gstate, void *userData);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfGeoState** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfGeoState**. Casting an object of class **pfGeoState** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfGeoState** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

gstate identifies a pfGeoState.

DESCRIPTION

A pfGeoState is an encapsulation of libpr graphics modes and attributes (see **pfState**). For example, a pfGeoState can describe a glass surface by referencing a shiny pfMaterial and enabling transparency. When a pfGeoState is applied by **pfApplyGState** it sets up the graphics state through normal libpr routines such as **pfApplyMtl** and **pfTransparency**.

Most pieces of state that may be manipulated through libpr immediate mode routines may be specified on a per-pfGeoState basis. For customized state management, pfGeoStates provide function callbacks. In addition, pfGeoStates can be indexed through a global table so a single database can have multiple appearances while avoiding database duplication.

The primary use of a pfGeoState is to attach it to a pfGeoSet (**pfGSetGState**) in order to define the appearance of the geometry encapsulated by the pfGeoSet. As discussed below, pfGeoStates have the useful property of order-independence so that paired pfGeoSets and pfGeoStates will be rendered consistently regardless of order.

pfGeoState state may either be locally set or globally inherited. By default, if a state element is not specified on a pfGeoState, then that pfGeoState will inherit that state element from the global state. Global state is set through libpr immediate mode functions, e.g., **pfApplyMtl**, **pfTransparency**, **pfDecal**, **pfApplyFog** or through **pfLoadGState** as described below. Local state is set on a pfGeoState through **pfGStateMode**, **pfGStateAttr**, or **pfGStateVal**.

If all state elements are locally set, then a pfGeoState becomes a full graphics context since all state is defined at the pfGeoState level. While this is useful, it usually makes sense to inherit most state from global default values and explicitly set only those state elements which are expected to change often. Examples of useful global defaults are lighting model (**pfLightModel**), lights (**pfLight**), fog (**pfFog**), and transparency (**pfTransparency**, usually OFF). Highly variable state is likely to be limited to a small set like textures and materials. By default all pfGeoState state is inherited.

State is pushed before, and popped after pfGeoStates are applied so that pfGeoStates do not inherit state from each other. As a result pfGeoStates are order-independent and you need not consider the problem of one pfGeoState corrupting another by state inheritance through the underlying graphics library. The actual pfGeoState pop is a lazy one and does not happen unless a subsequent pfGeoState needs the default state restored. This means that the actual state between pfGeoStates is not necessarily the global state. If a return to global state is required, call **pfFlushState** which will restore the global state.

It is a performance advantage to locally set as little local pfGeoState state as possible. This may be accomplished by setting global defaults which satisfy the majority of pfGeoStates being drawn. For example, if most of your database is textured, you should enable texturing at initialization time (**pfEnable**(-**PFEN_TEXTURE**)) and configure your pfGeoStates to inherit the texture enable mode.

pfNewGState creates and returns a handle to a pfGeoState. *arena* specifies a malloc arena out of which the pfGeoState is allocated or **NULL** for allocation off the process heap. pfGeoStates can be deleted with **pfDelete**. All modes and attributes are inherited by default.

pfGetGStateClassType returns the **pfType*** for the class **pfGeoState**. The **pfType*** returned by **pfGetGStateClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfGeoState**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfGStateMode sets *mode* to *val*. *mode* is a symbolic constant specifying the mode to set. Once set, a mode is no longer inherited but is set to *val*. *mode* is a symbolic token and is one of:

PFSTATE_TRANSPARENCY
PFSTATE_ANTIALIAS
PFSTATE_DECAL
PFSTATE_ALPHAFUNC
PFSTATE_ENLIGHTING
PFSTATE_ENTEXTURE
PFSTATE_ENFOG
PFSTATE_CULLFACE
PFSTATE_ENWIREFRAME
PFSTATE_ENCOLORTABLE
PFSTATE_ENHIGHLIGHTING
PFSTATE_ENLPOINTSTATE
PFSTATE_ENTEXGEN

val specifies the value of *mode* and is a symbolic token appropriate to the type of *mode*. For example when *mode* = **PFSTATE_TRANSPARENCY** then *val* might be **PFTR_ON**. Only modes which differ from the global state should be set. Mode values are not inherited between pfGeoStates. By default all modes are inherited. See the **pfState** manual page for information on global default settings.

pfGStateVal sets the *gsval* value to *val*. *gsval* is a symbolic constant specifying the state value to set. Once set, a value is no longer inherited but is set to *val*. *gsval* is a symbolic token and can be chosen from any of the following list (only one choice at present):

PFSTATE_ALPHAREF

pfGStateAttr sets *attr* state element to *a*. *attr* is a symbolic constant specifying an attribute and is one of:

PFSTATE_FRONTMTL
PFSTATE_BACKMTL
PFSTATE_TEXTURE
PFSTATE_TEXENV
PFSTATE_FOG

PFSTATE_LIGHTMODEL
PFSTATE_LIGHTS
PFSTATE_COLORTABLE
PFSTATE_HIGHLIGHT
PFSTATE_LPOINTSTATE
PFSTATE_TEXGEN

data is a handle to a **libpr** structure that is returned from a **pfNew<*>** routine. If *attr* is **PFSTATE_LIGHTS**, *a* should be an array of pfLight* of length **PF_MAX_LIGHTS** which specifies which pfLights should be used by *gstate*. Empty entries in the light array should be **NULL**.

A pfGeoState ignores the **PFMTL_FRONT** and **PFMTL_BACK** setting of a pfMaterial (see **pfMtlSide**). Instead it uses the attribute value, **PFSTATE_FRONTMTL** or **PFSTATE_BACKMTL** to decide how to apply the material. Consequently, it is legal to use the same material for both front and back sides. However, pfGeoStates do not modify the pfMaterial's side value which is normally set through **pfMtlSide**.

Once set, an attribute is no longer inherited but set to *a*. Only attributes which differ from the global state should be set. Attributes are not inherited between pfGeoStates. By default, all attributes are inherited from the global state.

pfGStateAttr increments the reference count of the supplied attribute and decrements the reference count of the replaced attribute, if there is one. **pfGStateAttr** will *not* delete any pfObject whose reference count reaches 0.

As discussed above, modes, values and attributes may either be locally set on a pfGeoState or inherited from the global state. To help resolve the inheritance characteristics of pfGeoStates, 3 different versions of "get" routines are provided for modes, values and attributes:

1. **pfGetGStateMode**, **pfGetGStateVal**, **pfGetGStateAttr** - The exact mode, value, or attribute of the pfGeoState is returned.
2. **pfGetGStateCurMode**, **pfGetGStateCurVal**, **pfGetGStateCurAttr** - The exact mode, value, or attribute of the pfGeoState is returned if not inherited. Otherwise the mode, value, or attribute of the currently active global pfGeoState is returned. Note that this requires that a pfState be current (see **pfSelectState**).
3. **pfGetGStateCombinedMode**, **pfGetGStateCombinedVal**, **pfGetGStateCombinedAttr** - The exact mode, value, or attribute of the pfGeoState is returned if not inherited. Otherwise the mode, value, or attribute of the *combGState* is returned.

pfGetGStateMode returns the mode value corresponding to *mode*.

pfGetGStateVal returns the pfGeoState value corresponding to *gsval*.

pfGetGStateAttr returns the attribute handle corresponding to *attr*. If *attr* is **PFSTATE_LIGHTS**, the

returned value is the pfLight* array.

pfGStateInherit specifies which state elements should be inherited from the global state. *mask* is a bit-wise OR of tokens listed for **pfGStateMode**, **pfGStateAttr**, and **pfGStateVal**. All of the state elements specified in *mask* will become inherited. All modes and attributes are inherited unless explicitly specified by setting a mode or attribute with **pfGStateAttr**, **pfGStateMode**, or **pfGStateVal**. **pfGetGStateInherit** returns the bitwise OR of the tokens for state which is currently inherited from the global state.

pfApplyGState makes *gstate* the current graphics state. All modes and attributes of *gstate* that are not inherited are applied using libpr immediate mode commands, for example, the **PFSTATE_TEXTURE** attribute is applied with **pfApplyTex**. Inherited modes and attributes that were modified by previous pfGeoStates are reset to their global values. State elements that are overridden (See **pfOverride**) are not changed by **pfApplyGState**.

Another way to apply a pfGeoState is with **pfDrawGSet**. If a pfGeoSet has an attached pfGeoState (see **pfGSetGState**), then **pfDrawGSet** will call **pfApplyGState** with the attached pfGeoState so that graphics state is properly established before the pfGeoSet geometry is rendered.

The following is an example of pfGeoState behavior.

Example 1:

```
/* Configure global default that pfGeoStates can inherit */
pfEnable(PFEN_LIGHTING);
pfApplyLModel(pfNewLModel(NULL));
pfLightOn(pfNewLight(NULL));
pfTransparency(PFTR_OFF);

/* New pfGeoState inherits everything */
gstate = pfNewGState(NULL);

/* Attach 'gstate' to 'gset' */
pfGSetGState(gset, gstate);

/* Configure 'gstate' with a transparent material */
pfGStateAttr(gstate, PFSTATE_FRONTMTL, mtl);
pfGStateMode(gstate, PFSTATE_TRANSPARENCY, PFTR_ON);
```

Method A:

```
/* Draw transparent 'gset' */
pfDrawGSet(gset);
```

Method B:

```
/* Remove 'gstate' from 'gset' */
pfGSetGState(gset, NULL);

/* Apply 'gstate' */
pfApplyGState(gstate);

/* Draw transparent 'gset' */
pfDrawGSet(gset);
```

Method C:

```
/* Remove 'gstate' from 'gset' */
pfGSetGState(gset, NULL);

pfApplyMtl(mtl);
pfTransparency(PFTR_ON);

/* Draw transparent 'gset' */
pfDrawGSet(gset);
```

In the above example, methods A, B, and C are all produce the same visual result. Method A is recommended, however, since the pfGeoState and pfGeoSet pair may be configured at database initialization time and the use of a pfGeoState provides order-independence when rendering.

The following is an example of pfGeoState inheritance:

```
/* Configure global default that pfGeoStates can inherit */
pfEnable(PFEN_LIGHTING);
pfApplyLModel(pfNewLModel(NULL));
pfLightOn(pfNewLight(NULL));

/* Assume 'redMtl' is PFMTL_FRONT */
pfApplyMtl(redMtl);

/* New pfGeoStates inherit everything */
```

```
gstateA = pfNewGState(NULL);
gstateB = pfNewGState(NULL);

/* Attach pfGeoStates to pfGeoSets */
pfGSetGState(gsetA, gstateA);
pfGSetGState(gsetB, gstateB);

/* Configure 'gstateA' with a green material */
pfGStateAttr(gstateA, PFSTATE_FRONTMTL, greenMtl);

/* Draw green 'gset' */
pfDrawGSet(gsetA);

/*
 * The FRONTMTL property of gstateB is not set so it inherits
 * the global default of 'redMtl' which will be restored
 * as the current pfMaterial when gstateB is applied.
 */

/* Draw red 'gset' */
pfDrawGSet(gsetB);

/*
 * Note that gsetA and gsetB could be drawn in the opposite
 * order with the same results.  This is a very important
 * pfGeoState property.
 */
```

pfGetCurGState returns the current pfGeoState that was previously applied directly by **pfApplyGState** or indirectly by **pfDrawGSet**.

pfGStateFuncs sets the callbacks and callback data pointer of *gstate*. The reference count of *data* is incremented and the reference count of the previous data is decremented but no deletion takes place if the reference count reaches 0. Callbacks are invoked by **pfApplyGState** (or indirectly by **pfDrawGSet** as described above) in the following order:

postFunc() of previously-applied pfGeoState
setup state according to current pfGeoState
preFunc() of current pfGeoState

Notice that the post-callback invocation is delayed until a subsequent pfGeoState is applied. However, **pfPushState**, **pfPopState**, and **pfFlushState** will invoke any "leftover" post-callback. It is legal to call **pfPushState** and **pfPopState** in the pre and post callbacks respectively but is not usually necessary because any **libpr** state set inside pfGeoState callbacks is considered to have been set by the pfGeoState. Consequently, the global state is not modified and the normal pfGeoState inheritance rules apply to state set inside the callbacks.

Callbacks are passed a pointer to the parent pfGeoState and the *data* pointer that was previously supplied by **pfGStateFuncs**. The return value from pfGeoState callbacks is currently ignored. **pfGetGStateFuncs** gets back the pre and post pfGeoState callbacks and callback data for *gstate* in *preFunc*, *postFunc*, and *data*, respectively.

A pfGeoSet may either directly reference or indirectly index a pfGeoState with **pfGSetGState** and **pfGSetGStateIndex** respectively. Indexed pfGeoStates use a global table of pfGeoState pointers that is set by **pfApplyGStateTable**. If the global table is **NULL** or the pfGeoState index is out of the range of the global table, no pfGeoState is applied, otherwise the indexed pfGeoState is applied when **pfDrawGSet** is called. Non-indexed pfGeoState references ignore the current pfGeoState table. **pfGetCurGStateTable** returns the current pfGeoState table and **pfGetCurIndexedGState** returns the *index*th pfGeoState* in the current pfGeoState table or **NULL** if the index cannot be properly resolved.

pfLoadGState is similar to **pfApplyGState** except the modes and attributes of *gstate* can be inherited by subsequent pfGeoStates. In other words, *gstate* loads the global state. Overridden state elements are not modified by **pfLoadGState**. If set, the pre-callback of *gstate* is invoked after the graphics state is loaded. As described above, the post-callback is not invoked until a subsequent pfGeoState is applied or **pfPushState**, **pfPopState**, or **pfFlushState** is called.

pfApplyGState, **pfApplyGStateTable**, and **pfLoadGState** are display-listable commands. If a pfDispList has been opened by **pfOpenDList**, these commands will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later executed with **pfDrawDList**. Indexed pfGeoStates are resolved at display list creation time, not at display list execution time. In addition, pfGeoStates are "unwound" into their constituent parts at display list creation time, e.g., a pfGeoState may decompose into **pfApplyMtl** and **pfTransparency** calls. As a result, changes to a pfGeoState which have been captured by a pfDispList will *not* be evident when that pfDispList is executed (**pfDrawDList**). pfGeoState indexing and unwinding at display list creation time is done strictly to improve pfDispList rendering performance.

pfMakeBasicGState configures every state element (value, mode, and attribute) of *gstate* to be identical to the state set with **pfBasicState**. The "basic" state is the initial state of a graphics library window - everything is "off". For example, the PFSTATE_ENLIGHTING mode will be set to PF_OFF, and the PFSTATE_CULLFACE mode will be set to PFCF_OFF. The following code fragment is equivalent to **pfBasicState**:

```
pfGeoState *gstate = pfNewGState(NULL);
pfMakeBasicGState(gstate);
pfLoadGState(gstate);
```

NOTES

In some situations it may appear that pfGeoStates do inherit from each other. This is because IRIS Performer currently does not provide any defaults for the state attributes listed above such as **PFSTATE_TEXTURE** and **PFSTATE_FRONTMTL**. Consequently, if the application does not explicitly set these attributes, it is possible for pfGeoStates which inherit these attributes to inherit them from previous pfGeoStates.

SEE ALSO

pfAlphaFunc, pfAntialias, pfBasicState, pfCullFace, pfDecal, pfDelete, pfDispList, pfDrawGSet, pfEnable, pfFog, pfGeoSet, pfLight, pfList, pfLPointState, pfOverride, pfState, pfTexture, pfTexGen, pfTransparency

NAME

pfNewHlight, pfGetHlightClassType, pfApplyHlight, pfHlightMode, pfGetHlightMode, pfHlightColor, pfGetHlightColor, pfHlightAlpha, pfGetHlightAlpha, pfHlightNormalLength, pfGetHlightNormalLength, pfHlightLineWidth, pfGetHlightLineWidth, pfHlightPntSize, pfGetHlightPntSize, pfHlightLinePat, pfGetHlightLinePat, pfHlightFillPat, pfGetHlightFillPat, pfHlightGState, pfGetHlightGState, pfHlightGStateIndex, pfGetHlightGStateIndex, pfHlightTex, pfGetHlightTex, pfHlightTEnv, pfGetHlightTEnv, pfHlightTGen, pfGetHlightTGen, pfGetCurHlight – Control, create, modify and query highlight state

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfHighlight*	**pfNewHlight**(void *arena);
pfType *	**pfGetHlightClassType**(void);
void	**pfApplyHlight**(pfHighlight * hl);
void	**pfHlightMode**(pfHighlight *hl, uint mode);
uint	**pfGetHlightMode**(const pfHighlight *hl);
void	**pfHlightColor**(pfHighlight *hl, uint which, float r, float g, float b);
void	**pfGetHlightColor**(const pfHighlight *hl, uint which, float *r, float *g, float *b);
void	**pfHlightAlpha**(pfHighlight *hl, float a);
float	**pfGetHlightAlpha**(const pfHighlight *hl);
void	**pfHlightNormalLength**(pfHighlight* hl, float length, float bboxScale);
void	**pfGetHlightNormalLength**(const pfHighlight* hl, float *length, float *bboxScale);
void	**pfHlightLineWidth**(pfHighlight* hl, float width);
float	**pfGetHlightLineWidth**(const pfHighlight* hl);
void	**pfHlightPntSize**(pfHighlight* hl, float size);
float	**pfGetHlightPntSize**(const pfHighlight* hl);
void	**pfHlightLinePat**(pfHighlight* hl, int which, ushort pat);
ushort	**pfGetHlightLinePat**(const pfHighlight* hl, int which);
void	**pfHlightFillPat**(pfHighlight* hl, int which, uint *fillPat);
void	**pfGetHlightFillPat**(const pfHighlight* hl, int which, uint *pat);
void	**pfHlightGState**(pfHighlight* hl, pfGeoState *gstate);
pfGeoState *	**pfGetHlightGState**(const pfHighlight* hl);

void	**pfHlightGStateIndex**(pfHighlight* hl, int id);
int	**pfGetHlightGStateIndex**(const pfHighlight* hl);
void	**pfHlightTex**(pfHighlight* hl, pfTexture *tex);
pfTexture*	**pfGetHlightTex**(const pfHighlight* hl);
void	**pfHlightTEnv**(pfHighlight* hl, pfTexEnv *tev);
pfTexEnv*	**pfGetHlightTEnv**(const pfHighlight* hl);
void	**pfHlightTGen**(pfHighlight* hl, pfTexGen *tgen);
pfTexGen*	**pfGetHlightTGen**(const pfHighlight* hl);
pfHighlight *	**pfGetCurHlight**(void);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfHighlight** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfHighlight**. Casting an object of class **pfHighlight** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void	**pfUserData**(pfObject *obj, void *data);
void*	**pfGetUserData**(pfObject *obj);
int	**pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfHighlight** can also be used with these functions designed for objects of class **pfMemory**.

pfType *	**pfGetType**(const void *ptr);
int	**pfIsOfType**(const void *ptr, pfType *type);
int	**pfIsExactType**(const void *ptr, pfType *type);
const char *	**pfGetTypeName**(const void *ptr);
int	**pfRef**(void *ptr);
int	**pfUnref**(void *ptr);
int	**pfUnrefDelete**(void *ptr);
int	**pfGetRef**(const void *ptr);
int	**pfCopy**(void *dst, void *src);
int	**pfDelete**(void *ptr);
int	**pfCompare**(const void *ptr1, const void *ptr2);
void	**pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void *	**pfGetArena**(void *ptr);

PARAMETERS

hl identifies a pfHighlight.

DESCRIPTION

IRIS Performer supports a mechanism for highlighting individual objects in a scene with a variety of special drawing styles that are activated by applying a pfHighlight state structure. Highlighting makes use of outlining of lines and polygons and of filling polygons with patterned or textured overlays. Highlighted drawing uses a highlighting color, or foreground color, and in some modes, a contrasting, or background, color. Additionally, there are highlighting modes for displaying the bound normals and cached bounding boxes of pfGeoState geometry.

A pfHighlight structure can be applied in immediate mode to the current active pfGeoState with **pfApplyHlight**, and added to a specific pfGeoState with **pfGStateMode**. Highlighting can be enabled and disabled in immediate mode with **pfEnable**(**PFEN_HIGHLIGHTING**) and **pfDisable**(**PFEN_HIGHLIGHTING**), and the override for highlighting can be set with **pfOverride**(**PFSTATE_HIGHLIGHT**). Unlike other types of state, a structure may be applied to a specific pfGeoSet with **pfGSetHlight**. This will cause the pfGeoSet to be drawn as highlighted with the specified highlighting structure, unless highlighting has been overridden as off with **pfOverride**.

This special exception was made because it is assumed that highlighting is to be used dynamically to highlight specific objects for a short period of time and should not impact the rest of the state structure.

Highlighting does have some performance penalty, in part because some of the highlighting modes make use of expensive graphics features, and in part because, to offer this flexibility, highlighted objects go through a slightly slower path in IRIS Performer rendering code.

pfNewHlight creates and returns a handle to a pfHighlight. *arena* specifies a malloc arena out of which the pfHighlight is allocated or **NULL** for allocation off the process heap. pfHighlights can be deleted with **pfDelete**.

pfGetHlightClassType returns the **pfType*** for the class **pfHighlight**. The **pfType*** returned by **pfGetHlightClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfHighlight**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfApplyHlight makes *hl* the current active pfHighlight structure.

pfGetCurHlight returns a pointer to the current active pfHighlight structure.

pfHlightGState sets a highlighting pfGeoState of *hl* to *gstate*. This pfGeoState is made the current pfGeoState for the highlighting phase of the drawing of the highlighted pfGeoSet. Additional highlighting mode changes are applied on top of this pfGeoState. This allows a user to make additional custom state

changes to highlighted objects. **pfGetHlightGState** returns the previously set highlighting pfGeoState of *hl*. **pfHlightGStateIndex** specifies the index into a pfGeoState table to use for the highlighting pfGeoState. **pfGetHlightGStateIndex** returns the previously set highlighting pfGeoState index of *hl*.

pfHlightMode sets the highlighting mode *mode* for *hl*. The mode specifies the drawing style: how the filled region and polygon outlines of an object should be drawn. This mode is a bitmask composed by bitwise OR-ing together the following tokens. The default is **PFHL_FILL**, and a zero mask is ignored.

PFHL_POINTS	Selects the display of object vertices as points using the point size specified by **pfHlightPntSize**.
PFHL_NORMALS	Selects the display of object normals as lines of width determined by **pfHlightLineWidth**, length determined by **pfHlightNormalLength**, and color determined by **pfHlightColor**.
PFHL_BBOX_LINES	Selects the display of the object's cached bounding box in lines of width determined by **pfHlightLineWidth** and color determined by **pfHlightColor**.
PFHL_BBOX_FILL	Selects the display of the object's cached bounding box as a solid filled box of the foreground color of *hl*.
PFHL_LINES	Selects outlining of primitives. The lines are drawn of width determined by **pfHlightLineWidth** and color determined by **pfHlightColor**.
PFHL_LINES_R	Selects outlining of primitives and reverses foreground and background colors for the lines.
PFHL_LINESPAT	Selects outlining of primitives with patterned lines.
PFHL_LINESPAT2	Selects outlining of primitives with 2-pass patterned lines, using both foreground and background highlighting colors.
PFHL_FILL	Selects filling of polygons with the foreground highlighting color. In this mode, the highlighted polygons are filled once. The foreground highlighting color is used as the base color of polygons, and as the material color for lit polygons.
PFHL_FILL_R	Selects outlining of primitives and reverses foreground and background colors for fill highlight modes.
PFHL_FILLTEX	Selects the application of a highlight texture on the object geometry. The default texture may be used, or a texture and associated attributes may be set with **pfHlightTex**, **pfHlightTEnv**, and **pfHlightTGen**.
PFHL_FILLPAT	Selects patterned filling of polygons with the foreground highlighting color. This patterning will be done in addition to the normal filling of the polygons and will be an overlay with the normal base polygons showing through.

PFHL_FILLPAT2	Selects 2-pass patterned filling of polygons using both the foreground and background highlighting colors. This patterning will be done in addition to the normal filling of the polygons and will be an overlay with the normal base polygons showing through.
PFHL_SKIP_BASE	Causes the normal drawing phase of the highlighted pfGeoSet to be skipped. This includes the application of the pfGeoState for that pfGeoSet.

pfGetHlightMode returns the highlighting mode of *hl*.

pfHlightColor sets the specified highlighting color *color*, **PFHL_FGCOLOR** or **PFHL_BGCOLOR**, of *hl*, to *r*, *g*, and *b*. **pfGetHlightColor** copies the specified *color*, **PFHL_FGCOLOR** or **PFHL_BGCOLOR**, of *hl*, into *r*, *g*, and *b*.

pfHlightAlpha sets the alpha of *hl* to *a*. **pfGetHlightAlpha** returns the alpha of *hl*.

pfHlightLineWidth sets the line width to be used for the **PFHL_LINES**, **PFHL_NORMALS**, and **PFHL_BBOX** highlighting modes of *hl* to *width*. If *width* is not greater than zero, the line width will not be set by the highlight structure and will be inherited from the current environment. **pfGetHlightLineWidth** returns the line width of *hl*.

pfHlightNormalLength sets a length and a scale factor for the normals drawn in the **PFHL_NORMALS** highlighting mode. The normals will be drawn of length normalLength + bboxScale*bboxLength. **pfGetHlightNormalLength** will return the normal length and scale values of *hl* in *length* and *bboxScale*, respectively.

pfHlightPntSize sets the point size to be used for the **PFHL_POINTS** highlighting mode of *hl* to *size*. If *size* is not greater than zero, the point size will not be set by the highlight structure and will be inherited from the current environment. **pfGetHlightPntSize** returns the point size of *hl*.

pfHlightLinePat sets the pattern to be used for lines in the **PFHL_LINES** highlighting modes of *hl* to *pat*. **pfGetHlightLinePat** returns the highlighting line pattern of *hl*.

pfHlightFillPat sets the fill pattern to be used in the **PFHL_FILLPAT** highlighting modes of *hl* to *pat*. **pfGetHlightFillPat** returns the highlighting fill pattern of *hl*.

pfHlightTex sets the pfTexture for the **PFHL_TEX** highlighting modes of *hl* to *tex*. **pfGetHlightTex** returns the previously set highlighting texture of *hl*. If a texture is not specified but the **PFHL_TEX** is selected for *hl*, a default two-component texture using the highlighting foreground and background colors will be used.

pfHlightTEnv sets the texture environment (pfTexEnv) for the **PFHL_TEX** highlighting modes of *hl* to *tev*. **pfGetHlightTEnv** returns the previously set highlighting texture environment of *hl*. If a texture

environment is not specified but the **PFHL_TEX** is selected for *hl*, a default texture blend environment will be used.

pfHlightTGen sets the texture coordinate generation attribute (pfTexGen) for the **PFHL_TEX** highlighting modes of *hl* to *tgen*. **pfGetHlightTGen** returns the previously set highlighting pfTexGen of *hl*. If a texture coordinate generation function is not specified and the object to be highlighted has no texture coordinates of its own and the **PFHL_TEX** is selected for *hl*, a default texture coordinate generation function will be used.

EXAMPLES

Example 1: Set up a highlighting structure and apply it in immediate mode to the current pfGeoState.

```
pfHighlight *hl;

/* allocate a new highlight color */
hl = pfNewHlight(NULL);

/* specify highlight modes */
pfHlightMode(hl, PFHL_FILL);
pfHlightColor(hl, PFHL_FGCOLOR, 1.0f,  0.0f,  1.0f);

/* apply highlight */
pfApplyHlight(hl);
```

SEE ALSO

pfDelete, pfDisable, pfDrawHlightedGSet, pfEnable, pfGSetHlight, pfGeoState, pfGetGSetHlight, pfObject, pfOverride, pfState

NAME

pfQueryHit, pfMQueryHit, pfGetHitClassType – Intersection and bounding operations on drawable geometry

FUNCTION SPECIFICATION

#include <Performer/pr.h>

int **pfQueryHit**(pfHit *hit, uint which, void *dst);

int **pfMQueryHit**(pfHit *hit, uint *which, void *dst);

pfType * **pfGetHitClassType**(void);

DESCRIPTION

pfQueryHit and **pfMQueryHit** read out information from the pfHit object. **pfQueryHit** copies an item from the object into the location specified by *dst*. *which* specifies the item to be copied using one of the **PFHIT_** tokens listed above. **pfMQueryHit** copies a series of items sequentially into the buffer specified by *dst*. The items and their order are defined by a NULL-terminated array of query tokens pointed to by *which*. For both functions, the return value is the number of bytes written to the destination buffer.

PFQHIT_FLAGS returns a bit vector indicating the validity of information in the structure. It is formed by a bitwise OR-ing of the **PFHIT_POINT**, **PFHIT_NORM**, **PFHIT_PRIM**, **PFHIT_TRI**, **PFHIT_VERTS** and **PFHIT_XFORM** symbols.

Flags Bit	Validity
PFHIT_POINT	Point of intersection
PFHIT_NORM	Polygon normal
PFHIT_PRIM	Index of primitive in pfGeoSet
PFHIT_TRI	Index of triangle within primitive
PFHIT_VERTS	Triangle vertices
PFHIT_XFORM	Non-identity transformation matrix

Other queried quantities are valid if non-NULL.

PFQHIT_POINT, **PFQHIT_NORM** and **PFQHIT_SEG** query the point of intersection, the normal of the triangle at that point, and the current segment as clipped by the intersection process. All are in local coordinates, i.e. they do not include the transformations of pfSCSes and pfDCSes above them in the scene graph. When intersecting with primitives inside a pfGeoSet, **PFQHIT_PRIM**, **PFQHIT_TRI** and **PFQHIT_VERTS** provide the index of the primitive within the pfGeoSet, the triangle within the primitive, and the vertices of the intersected triangle, respectively. **PFQHIT_GSET** returns the GeoSet. **PFQHIT_NODE** returns the parent pfGeode.

PFQHIT_PATH returns a pfPath* denoting the traversal path. Like the pfHit object it is reused and should not be freed.

pfGetHitClassType returns the **pfType*** for the class **pfHit**. The **pfType*** returned by

pfGetHitClassType is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfHit**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

SEE ALSO

pfCylAroundSegs, pfNodeIsectSegs, pfGeoSet, pfObject, pfSeg

NAME

pfNewLPState, pfGetLPStateClassType, pfLPStateMode, pfGetLPStateMode, pfLPStateVal, pfGetLPStateVal, pfLPStateShape, pfGetLPStateShape, pfLPStateBackColor, pfGetLPStateBackColor, pfApplyLPState, pfMakeLPStateRangeTex, pfMakeLPStateShapeTex, pfGetCurLPState – Set and get pfLPointState size, transparency, directionality, shape, and fog attributes.

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfLPointState * **pfNewLPState**(void *arena);

pfType * **pfGetLPStateClassType**(void);

void **pfLPStateMode**(pfLPointState *lpstate, int mode, int val);

int **pfGetLPStateMode**(const pfLPointState *lpstate, int mode);

void **pfLPStateVal**(pfLPointState *lpstate, int attr, float val);

float **pfGetLPStateVal**(const pfLPointState *lpstate, int attr);

void **pfLPStateShape**(pfLPointState *lpstate, float horiz, float vert, float roll, float falloff, float ambient);

void **pfGetLPStateShape**(const pfLPointState *lpstate, float *horiz, float *vert, float *roll, float *falloff, float *ambient);

void **pfLPStateBackColor**(pfLPointState *lpstate, float r, float g, float b, float a);

void **pfGetLPStateBackColor**(pfLPointState *lpstate, float *r, float *g, float *b, float *a);

void **pfApplyLPState**(pfLPointState *lpstate);

void **pfMakeLPStateRangeTex**(pfLPointState *lpstate, pfTexture *tex, int size, pfFog* fog);

void **pfMakeLPStateShapeTex**(pfLPointState *lpstate, pfTexture *tex, int size);

pfLPointState* **pfGetCurLPState**(void);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfLPointState** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfLPointState**. Casting an object of class **pfLPointState** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfLPointState** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *      pfGetType(const void *ptr);
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

PARAMETERS

lpstate identifies a pfLPointState.

DESCRIPTION

A pfLPointState is a **libpr** data structure which, in conjunction with a pfGeoSet of type **PFGS_POINTS**, supports a sophisticated **light point** primitive type. Examples of light points are stars, beacons, strobes, runway edge and end illumination, taxiway lights, visual approach slope indicators (VASI), precision approach path indicators (PAPI), and street lights when viewed from a great distance.

Light points should not be confused with light sources, such as a pfLight. A light point is visible as one or more self-illuminated small points that do not illuminate surrounding objects. By comparison, a pfLight does illuminate scene contents but is itself not a visible object.

When a pfLPointState is applied with **pfApplyLPState** or through its parent pfGeoState (See **pfDrawGSet** and **pfApplyGState**), any pfGeoSet of type **PFGS_POINTS** will be rendered with the following special light point characteristics (if enabled):

1. Perspective size. Light points can be assigned a real world size and exhibit perspective behavior, e.g., points closer to the eye will be rendered larger than those points farther away.
2. Perspective fading. Once a light point reaches a minimum size, it may be made more transparent in order to enhance the perspective illusion. Fading is often more realistic than simply shrinking the point size to 0.
3. Fog punch-through. Since light points are emissive objects, they must shine through fog more than non-emissive objects.

4. Directionality. Light points can be assigned a direction as well as vertical and horizontal envelopes (or lobes) about this direction vector. Directional light point intensity is then view position-dependent. Light point direction is defined by the normals (**PFGS_NORMAL3**) supplied by **PFGS_POINTS** pfGeoSets.
5. Intensity. Normally, light point color and transparency are defined by the colors (**PFGS_COLOR4**) supplied by **PFGS_POINTS** pfGeoSets. pfLPointStates provide the additional capability of modifying the intensity of all points in a light point pfGeoSet by scaling the alpha of all point colors.

At a minimum, light point usage requires a configuration based on three linked libpr objects: a pfGeoSet, a pfGeoState attached to that pfGeoSet, and a pfLPointState attached to the pfGeoState. Here are the details:

1. A pfGeoSet of type **PFGS_POINTS**. This pfGeoSet must have a **PFGS_COLOR4** attribute binding of **PFGS_PER_VERTEX** in some situations and should have supplied normals (**PFGS_NORMAL3**) if the light points are directional.
2. A pfGeoState which is usually attached to the pfGeoSet and which references a pfLPointState. The pfGeoState should almost always enable transparency since all light point effects except perspective size require transparency.
3. A pfLPointState configured appropriately and attached to the pfGeoState listed in step two.

The following example illustrates how to build a comprehensive light point structure that uses texture mapping to accelerate directionality computations:

```
/*
 * Create pfLPointState and pfGeoState.
 */
pfGeoState     *gst = pfNewGState(arena);
pfLPointState  *lps = pfNewLPState(arena);
pfGStateMode(gst, PFSTATE_ENLPOINTSTATE, 1);
pfGStateAttr(gst, PFSTATE_LPOINTSTATE, lps);

/*
 * Light point projected diameter is computed on CPU. Real world
 * size is 0.07 database units and projected size is clamped be
 * between 0.25 and 4 pixels.
 */
pfLPStateMode(lps, PFLPS_SIZE_MODE, PFLPS_SIZE_MODE_ON);
pfLPStateVal(lps, PFLPS_SIZE_MIN_PIXEL, 0.25f);
pfLPStateVal(lps, PFLPS_SIZE_ACTUAL, 0.07f);
pfLPStateVal(lps, PFLPS_SIZE_MAX_PIXEL, 4.0f);
```

```
/*
 * Light points become transparent when their projected diameter is
 * < 2 pixels. The transparency falloff rate is linear with
 * projected size with a scale factor of 0.6. The transparency
 * multiplier, NOT the light point transparency, is clamped to 0.1.
 */
pfLPStateVal(lps, PFLPS_TRANSP_PIXEL_SIZE, 2.0f);
pfLPStateVal(lps, PFLPS_TRANSP_EXPONENT, 1.0f);
pfLPStateVal(lps, PFLPS_TRANSP_SCALE, 0.6f);
pfLPStateVal(lps, PFLPS_TRANSP_CLAMP, 0.1f);

/*
 * Light points will be fogged as if they were 4 times
 * nearer to the eye than actual to achieve punch-through.
 */
pfLPStateVal(lps, PFLPS_FOG_SCALE, 0.25f);

/* Range to light points computed on CPU is true range */
pfLPStateMode(lps, PFLPS_RANGE_MODE, PFLPS_RANGE_MODE_TRUE);

/*
 * Light points are bidirectional but have different (magenta)
 * back color. Front color is provided by pfGeoSet colors.
 */
pfLPStateMode(lps, PFLPS_SHAPE_MODE, PFLPS_SHAPE_MODE_BI_COLOR);
pfLPStateBackColor(lps, 1.0f, 0.0f, 1.0f, 1.0f);

/*
 * 60 degrees horizontal and 30 degrees vertical envelope.
 * Envelope is rotated -25 degrees about the light point
 * direction. Falloff rate is linear and ambient intensity is 0.1.
 */
pfLPStateShape(lps, 60.0f, 30.0f, -25.0f, 1.0f, 0.1f);

/*
 * Specify that light points should use texturing hardware to simulate
 * directionality and use CPU to compute light point transparency and
 * fog punch-through. Note that if light points are omnidirectional,
 * you should use PFLPS_TRANSP_MODE_TEX and PFLPS_FOG_MODE_TEX instead.
 */
pfLPStateMode(lps, PFLPS_DIR_MODE, PFLPS_DIR_MODE_TEX);
pfLPStateMode(lps, PFLPS_TRANSP_MODE, PFLPS_TRANSP_MODE_ALPHA);
pfLPStateMode(lps, PFLPS_FOG_MODE, PFLPS_FOG_MODE_ALPHA);
```

```
/*
 * Make directionality environment map of size 64 x 64 and attach
 * it to the light point pfGeoState. We assume that a pfTexEnv of
 * type PFTE_MODULATE has been globally applied with pfApplyTEnv.
 */
tex = pfNewTex(arena);
pfMakeLPStateShapeTex(lps, tex, 64);
pfGStateAttr(gst, PFSTATE_TEXTURE, tex);
pfGStateMode(gst, PFSTATE_ENTEXTURE, 1);

/*
 * Make SPHERE_MAP pfTexGen and attach to light point pfGeoState.
 * pfGeoSet normals define the per-light light point direction.
 */
tgen = pfNewTGen(arena);
pfTGenMode(tgen, PF_S, PFTG_SPHERE_MAP);
pfTGenMode(tgen, PF_T, PFTG_SPHERE_MAP);
pfGStateAttr(gst, PFSTATE_TEXGEN, tgen);
pfGStateMode(gst, PFSTATE_ENTEXGEN, 1);

/*
 * Configure light point transparency. Use PFTR_BLEND_ALPHA for high
 * quality transparency. Set pfAlphaFunc so that light points are not
 * drawn unless their alphas exceed 1 when using 8-bit color resolution.
 */
pfGStateMode(gst, PFSTATE_TRANSPARENCY, PFTR_BLEND_ALPHA);
pfGStateVal(gst, PFSTATE_ALPHAREF, 1.0/255.0);
pfGStateMode(gst, PFSTATE_ALPHAFUNC, PFAF_GREATER);

/*
 * Disable pfFog effects since light points are fogged by
 * the pfLPointState.
 */
pfGStateMode(gst, PFSTATE_ENFOG, 0);
/*
 * Disable lighting effects since light points are completely
 * emissive.
 */
pfGStateMode(gst, PFSTATE_ENLIGHTING, 0);

/*
 * Attach the pfGeoState to a pfGeoSet of type PFGS_POINTS and
 * you've got light points!
```

```
 */
pfGSetPrimType(gset, PFGS_POINTS);
pfGSetGState(gset, gst);
```

pfLPointState Modes

Each of the five light point characteristics listed earlier may be achieved through the Graphics Library in different ways depending on the available graphics hardware. **pfLPStateMode/pfLPStateVal** provide control over feature implementation. Modes and their corresponding values accepted by **pfLPStateMode** are:

PFLPS_SIZE_MODE /* Perspective size */

PFLPS_SIZE_MODE_ON - Enable perspective light point size. Perspective size is computed on the CPU.

PFLPS_SIZE_MODE_OFF - Disable perspective light point size.

PFLPS_TRANSP_MODE /* Perspective fading */

PFLPS_TRANSP_MODE_ON - Enable default (CPU-based) light point fading.

PFLPS_TRANSP_MODE_OFF - Disable light point fading.

PFLPS_TRANSP_MODE_ALPHA - Enable light point fading. Compute fade value on CPU and modify light point alpha. This mode requires that pfGeoSets have a **PFGS_COLOR4** binding of **PFGS_PER_VERTEX** and that there be a unique color for each point.

PFLPS_TRANSP_MODE_TEX - Enable light point fading. Use texture mapping to simulate fading.

PFLPS_FOG_MODE /* Fog punch-through */

PFLPS_FOG_MODE_ON - Enable default (CPU-based) fog punch-through.

PFLPS_FOG_MODE_OFF - Disable fog punch-through.

PFLPS_FOG_MODE_ALPHA - Enable fog punch-through. Compute fog value on CPU and modify light point alpha. This mode requires that pfGeoSets have a **PFGS_COLOR4** binding of **PFGS_PER_VERTEX** and that there be a unique color for each point.

PFLPS_FOG_MODE_TEX - Enable fog punch-through. Use texture mapping to simulate fog.

(Normal fogging should be disabled (**pfDisable(PFEN_FOG)** or **pfGStateMode(g, PFSTATE_ENFOG, 0)**) when **PFLPS_FOG_MODE** is not **PFLPS_FOG_MODE_OFF** since the pfLPointState will fog the points)

PFLPS_DIR_MODE /* Directionality enable */
PFLPS_DIR_MODE_ON - Enable default (CPU-based) directional light points.

PFLPS_DIR_MODE_OFF - Disable directional light points.

PFLPS_DIR_MODE_ALPHA - Enable directional light points. Compute directionality on CPU and modify light point alpha. This mode requires that pfGeoSets have a **PFGS_COLOR4** binding of **PFGS_PER_VERTEX** and that there be a unique color for each point.

PFLPS_DIR_MODE_TEX - Enable directional light points. Use texture mapping to simulate directionality.

PFLPS_SHAPE_MODE /* Directionality shape */
PFLPS_SHAPE_MODE_UNI - Directional light points are unidirectional. Light distribution is an elliptical cone specified by **pfLPStateShape**, centered about the light direction vector.

PFLPS_SHAPE_MODE_BI - Directional light points are bidirectional with identical front and back colors. Light distribution is two elliptical cones, specified by **pfLPStateShape**, centered about the positive and negative light direction vectors.

PFLPS_SHAPE_MODE_BI_COLOR - Directional light points are bidirectional with back color specified by **pfLPStateBackColor**. Light distribution is two elliptical cones, specified by **pfLPStateShape**, centered about the positive and negative light direction vectors.

PFLPS_RANGE_MODE
PFLPS_RANGE_MODE_DEPTH - Range to light point is approximated by depth from eye. This may be faster, but less accurate than **PFLPS_RANGE_MODE_TRUE**.

PFLPS_RANGE_MODE_TRUE - Range to light point is true, slanted range to eye. This may be slower, but more accurate than **PFLPS_RANGE_MODE_DEPTH**.

pfLPointState Values

pfLPStateVal sets the attribute of *lpstate* identified by *which* to *val*. **pfGetLPStateVal** returns the attribute of *lpstate* identified by *which*.

Values associated with **PFLPS_SIZE_MODE** and which have effect only when **PFLPS_SIZE_MODE** is **PFLPS_SIZE_MODE_ON** are the following:

PFLPS_SIZE_MIN_PIXEL
val specifies the minimum diameter, in pixels, of light points. Default value is 0.25. Note that actual minimum point size is clamped to the minimum supported by the graphics hardware.

PFLPS_SIZE_MAX_PIXEL
val specifies the maximum diameter, in pixels, of light points. Default value is 4.0. Note that actual maximum point size is clamped to the maximum supported by the graphics hardware.

PFLPS_SIZE_ACTUAL
val specifies light point diameter in eye coordinates. Scales do not affect the actual light point size. Default value is 0.25.

In pseudo-code, the size of a light point is determined as follows:

```
/* NearPixelDistance is described below */
computedSize = PFLPS_SIZE_ACTUAL * NearPixelDistance / rangeToEye;

if (PFLPS_SIZE_MODE == PFLPS _SIZE_MODE_ON)
{
    /* Clamp pixel size of point */
    if (computedSize < PFLPS_SIZE_MIN_PIXEL)
        computedSize = PFLPS_SIZE_MIN_PIXEL;
    else
    if (computedSize > PFLPS_SIZE_MAX_PIXEL)
        computedSize = PFLPS_SIZE_MAX_PIXEL;

    lightPointSize = computedSize;
}
```

Values associated with **PFLPS_TRANSP_MODE** and which have effect only when **PFLPS_TRANSP_MODE** is not **PFLPS_TRANSP_MODE_OFF**.

PFLPS_TRANSP_PIXEL_SIZE
val specifies the threshold diameter, in pixels, at which light point alphas are decreased so that they become more transparent once computed light point size is less than *val*. Default value is 0.25.

PFLPS_TRANSP_EXPONENT
val specifies an exponential falloff for light point fading and should be >= 0.0. Values > 0 and < 1 make the falloff curve flatter while values > 1 make it sharper. Default value is 1.0 for a linear falloff based on projected pixel size.

PFLPS_TRANSP_SCALE
val specifies a scale factor for the light point fade multiplier. Values > 0 and < 1 decrease the falloff rate while values > 1 increase it. Default value is 1.0.

PFLPS_TRANSP_CLAMP - *val* specifies the minimum fade multiplier.

In pseudo-code, the transparency of a light point is determined as follows:

```
if (PFLPS_TRANSP_MODE == PFLPS_TRANSP_MODE_ALPHA &&
    PFLPS_TRANSP_PIXEL_SIZE > computedSize)
{
    float       a;

    a = 1.0f - PFLPS_TRANSP_SCALE *
        powf(PFLPS_TRANSP_PIXEL_SIZE - computedSize,
             PFLPS_TRANSP_EXPONENT);

    /* Clamp alpha multiplier, not alpha */
    if (a < PFLPS_TRANSP_CLAMP)
        a = PFLPS_TRANSP_CLAMP;

    lightPointAlpha *= a;
}
```

PFLPS_FOG_SCALE

val specifies a scale factor that multiplies the range from eye to light point before fogging. Values > 0.0 and < 1.0 cause light points to punch through fog more than non-emissive surfaces. Default value is 0.25.

In pseudo-code, the fog of a light point is determined as follows:

```
/* fogFunction ranges from 0 (no fog) to 1 (completely fogged) */
lightPointAlpha *= 1.0f - fogFunction(rangeToEye * PFLPS_FOG_SCALE);
```

PFLPS_INTENSITY

val multiplies all light point alphas. Default value is 1.0.

PFLPS_SIZE_DIFF_THRESH

val specifies the threshold, in pixels, at which a new point size should be specified to the Graphics Library. It is strictly a tuning parameter which trades off speed for image quality. Default value is 0.1. Higher values improve performance but may degrade light point image quality.

PFLPS_TRANSP_MODE, **PFLPS_FOG_MODE**, and **PFLPS_DIR_MODE** modes each have possible values of **ALPHA** and **TEX** which dictate the mechanism used to simulate the effect. The **ALPHA** mechanism is the default and uses the CPU to compute the effect which is then realized by modifying the alpha of light point colors. pfGeoSets of type **PFGS_POINTS** which use an **ALPHA** mechanism should

have a **PFGS_COLOR4** binding of **PFGS_PER_VERTEX** even if all point colors are the same, since the light point alphas will be different based on **ALPHA** computation by the pfLPointState.

While **ALPHA** mechanisms are graphics hardware-independent, they may be slower than **TEX** mechanisms on machines which provide hardware texture mapping. By supplying an appropriate pfTexture, pfTexGen, and pfTexEnv (usually attached to the pfGeoState to which the pfLPointState is attached), you can use the texture mapping hardware to efficiently simulate directionality or fog punch-through and perspective fading. At this time it is not possible to support all **TEX** mechanisms at once:

1. Only **PFLPS_DIR_MODE_TEX** or,
2. **PFLPS_TRANSP_MODE_TEX** and/or **PFLPS_FOG_MODE_TEX**

It is recommended that directional light points use **PFLPS_DIR_MODE_TEX** since directionality is the most expensive effect to compute on the CPU.

Two convenience routines, **pfMakeLPStateRangeTex** and **pfMakeLPStateShapeTex** are provided to compute a texture image which accurately mimics certain characteristics of *lpstate*.

pfMakeLPStateRangeTex should be used in conjunction with **PFLPS_TRANSP_MODE_TEX** and/or **PFLPS_FOG_MODE_TEX** and will set a computed image on the supplied pfTexture, *tex*. The image will be a 2D array of *size* by *size* if both **PFLPS_TRANSP_MODE_TEX** and **PFLPS_FOG_MODE_TEX** are set on *lpstate* or the image will be a 1D array of length *size* if only 1 of **PFLPS_TRANSP_MODE_TEX** and **PFLPS_FOG_MODE_TEX** is set.

When using **PFLPS_TRANSP_MODE_TEX** and/or **PFLPS_FOG_MODE_TEX**, you must supply a pfTexGen structure which computes the S (and T if both **PFLPS_TRANSP_MODE_TEX** and **PFLPS_FOG_MODE_TEX** are set) texture coordinates as distance from the Z = 0 plane in eye coordinates. For example:

```
tgen = pfNewTGen(arena);
pfTGenPlane(tgen, PF_S, 0.0f, 0.0f, 1.0f, 0.0f);
pfTGenPlane(tgen, PF_T, 0.0f, 0.0f, 1.0f, 0.0f);
pfTGenMode(tgen, PF_S, PFTG_EYE_PLANE);
pfTGenMode(tgen, PF_T, PFTG_EYE_PLANE);
```

pfMakeLPStateRangeTex takes into account only the following values of *lpstate* when building the texture image and should be called again whenever they change:

PFLPS_TRANSP_PIXEL_SIZE
PFLPS_TRANSP_EXPONENT
PFLPS_TRANSP_SCALE
PFLPS_TRANSP_CLAMP

pfMakeLPStateShapeTex computes an environment map which approximates the directional

characteristics of *lpstate*. The computed image is assigned to *tex* and its dimensions are *size* by *size*. When using **PFLPS_DIR_MODE_TEX**, you must supply a pfTexGen structure which uses **PFTG_SPHERE_MAP** to compute both S and T. For example:

```
tgen = pfNewTGen(arena);
pfTGenMode(tgen, PF_S, PFTG_SPHERE_MAP);
pfTGenMode(tgen, PF_T, PFTG_SPHERE_MAP);
```

pfMakeLPStateShapeTex takes into account only the **PFLPS_SHAPE_MODE** modes and those values specified by **pfLPStateShape**. Consequently, **pfMakeLPStateShapeTex** should be called whenever these modes/values change.

fog should represent the desired fog ramp, e.g. **PFFOG_LINEAR**, **PFFOG_SPLINE**, if **PFLPS_FOG_MODE_TEX** is set or NULL if not set. The fog ranges are ignored and *fog* is not modified.

Each of the four main light point features (size, transparency, fog, and directionality) are view-dependent effects. Consequently, knowledge about the viewing and modeling transformations is required in certain situations:

1. When not using **libpf**. Otherwise, **libpf** automatically informs **libpr** of the viewing and modeling transformations.
2. When using an **ALPHA** mechanism, e.g., **PFLPS_DIR_MODE_ALPHA**.
3. When **PFLPS_SIZE_MODE** is **PFLPS_SIZE_MODE_ON**.

Use **pfViewMat** and **pfModelMat** to specify the viewing and modeling matrices respectively. For best performance, these routines should be called only when the corresponding matrix changes. Additionally you may call **pfInvModelMat** to specify the inverse of the modeling matrix if you've already computed it for some other reason. When using **PFLPS_SIZE_MODE_ON**, use **pfNearPixDist** to specify the distance, in pixels, from the eye to the near clip plane. pfLPointState needs this parameter to map world size to pixel size (but only if not using **libpf**). **pfViewMat**, **pfModelMat**, **pfInvModelMat**, and **pfNearPixDist** are all display-listable commands which may be captured by an open pfDispList.

pfNewLPState creates and returns a handle to a pfLPointState. *arena* specifies a malloc arena out of which the pfLPointState is allocated or **NULL** for allocation off the process heap. pfLPointStates can be deleted with **pfDelete**.

pfGetLPStateClassType returns the **pfType*** for the class **pfLPointState**. The **pfType*** returned by **pfGetLPStateClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfLPointState**. When decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfLPointShape specifies the light distribution characteristics of directional light points. Light point directions are specified by pfGeoSet normals after they have been transformed by the current modeling matrix. Note that a **PFGS_NORMAL3** binding of **PFGS_OVERALL** is permitted as well as a binding of **PFGS_PER_VERTEX**. Directional light points require that **PFLPS_DIR_MODE** be **PFLPS_DIR_MODE_ON**, **PFLPS_DIR_MODE_ALPHA**, or **PFLPS_DIR_MODE_TEX**.

horiz and *vert* are total angles (not half-angles) in degrees which specify the horizontal and vertical envelopes about the direction vector. An envelope is a symmetric angular spread in a specific plane about the light direction vector. The default direction is along the positive Y axis so the horizontal envelope is in the X plane and the vertical in the Z plane. The envelopes are twisted about the +Y axis by *roll* degrees, then rotated by the rotation which takes the +Y axis onto the light point direction vector. Default values are:

horiz = 90 degrees
vert = 90 degrees
roll = 0 degrees
falloff = 1
ambient = 0

When the vector from the eyepoint to the light position is outside its envelope, the light point's intensity is *ambient*. If within, the intensity of the light point is computed based on the location of the eye within the elliptical cone. Intensity ranges from 1.0 when the eye lies on the light direction vector to *ambient* on the edge of the cone. *falloff* is an exponent which modifies the intensity. A value of 0 indicates that there is no falloff and values > 0 increase the falloff rate. The default *falloff* is 1. As intensity decreases, the light point's transparency increases.

pfGetLPointShape copies *lpstate*'s shape parameters into *horiz*, *vert*, *roll*, *falloff*, and *ambient*.

pfLPStateBackColor specifies the back color of *lpstate*. If *lpstate*'s shape mode is not **PFLPS_SHAPE_MODE_BI_COLOR**, then the back color has no effect. **pfLPStateBackColor** copies *lpstate*'s back color components into *r*, *g*, *b*, *a*.

pfApplyLPState makes *lpstate* the current pfLPointState which affects all subsequently drawn pfGeoSets of type **PFGS_POINTS**. **pfApplyLPState** is a display-listable command which may be captured by an open pfDispList. A pfLPointState may also be attached to a pfGeoState. **pfGetCurLPState** returns the current pfLPointState or NULL if there is none.

NOTES

Falloff distribution is cosine(incidence angle) ^ falloff.

pfApplyLPState changes, but does not restore the texture matrix if **PFLPS_DIR_MODE_TEX**, **PFLPS_TRANSP_MODE_TEX**, or **PFLPS_FOG_MODE_TEX** is active.

SEE ALSO

pfDelete, pfDispList, pfFog, pfGeoSet, pfGeoState, pfState, pfTexture, pfTexGen, pfuMakeLPStateRangeTex, pfuMakeLPStateShapeTex

NAME

pfNewLight, pfGetLightClassType, pfLightColor, pfGetLightColor, pfLightAtten, pfGetLightAtten, pfLightPos, pfGetLightPos, pfSpotLightDir, pfGetSpotLightDir, pfSpotLightCone, pfGetSpotLightCone, pfLightOn, pfLightOff, pfIsLightOn, pfLightAmbient, pfGetLightAmbient, pfGetCurLights – Create, modify and query lights

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfLight * **pfNewLight**(void *arena);

pfType * **pfGetLightClassType**(void);

void **pfLightColor**(pfLight *lt, int which, float r, float g, float b);

void **pfGetLightColor**(const pfLight *lt, int which, float *r, float *g, float *b);

void **pfLightAtten**(pfLight* light, float constant, float linear, float quadratic);

void **pfGetLightAtten**(pfLight* light, float *constant, float *linear, float *quadratic);

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

void **pfGetLightPos**(const pfLight *lt, float *x, float *y, float *z, float *w);

void **pfSpotLightDir**(pfLight *lt, float x, float y, float z);

void **pfGetSpotLightDir**(const pfLight *lt, float *x, float *y, float *z);

void **pfSpotLightCone**(pfLight *lt, float exponent, float spread);

void **pfGetSpotLightCone**(const pfLight *lt, float *exponent, float *spread);

void **pfLightOn**(pfLight *lt);

void **pfLightOff**(pfLight *lt);

int **pfIsLightOn**(pfLight *lt);

void **pfLightAmbient**(pfLight *lt, float r, float g, float b);

void **pfGetLightAmbient**(const pfLight *lt, float *r, float *g, float *b);

int **pfGetCurLights**(pfLight *lights[PF_MAX_LIGHTS]);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfLight** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfLight**. Casting an object of class **pfLight** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);

void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfLight** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

lt identifies a pfLight.

DESCRIPTION

A pfLight is a light source that illuminates scene geometry, generating realistic shading effects. A pfLight cannot itself be seen but its effect is visible through its illuminative effect on scene geometry. There are some subtle differences between IRIS GL and OpenGL light operation and additional references are recommended and should be noted. See the IRIS GL **lmdef(3g)** or the OpenGL **glLight(3g)** reference page for more details on lights and individual lighting parameters.

pfNewLight creates and returns a handle to a pfLight. *arena* specifies a malloc arena out of which the pfLight is allocated or **NULL** for allocation off the process heap. A NULL pointer is returned to indicate failure. pfLights can be deleted with **pfDelete**.

pfGetLightClassType returns the **pfType*** for the class **pfLight**. The **pfType*** returned by **pfGetLightClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfLight**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfLightColor accepts a token for the color attribute to set (**PFLT_AMBIENT**, **PFLT_DIFFUSE**, or **PFLT_SPECULAR**) and three floating point values (*r*, *g*, and *b*) in the range [0.0 .. 1.0] defining values for the red, green, and blue components of the indicated attribute of the light source. By default, the r, g, and b values are all 1.0. **pfGetLightColor** copies the requested light color values for the given light source

and color attribute into the parameters *r*, *g*, and *b*.

pfLightAtten sets the attenuation parameters of *light*. The light intensity is scaled at each vertex by:

1.0 / (*constant* + *linear* * dist + *quadratic* * dist^2)

where 'dist' is the distance from the light position to the lit vertex. Note that 'dist' is 1.0 for infinite light sources. The default attenuation values are *constant* = 1.0, *linear* = 0.0, *quadratic* = 0.0, i.e., light attenuation is disabled. **pfGetLightAtten** returns the attenuation parameters of *light* in *constant*, *linear*, and *quadratic*. Per-light attenuation is only available in OpenGL. IRIS GL light attenuation is done on the light model; see the **pfLModelAtten** reference page for more information.

pfSpotLightDir specifies the direction in which a spot light source emits its light. It receives three floating point values, *x*, *y*, and *z*, specifying the x, y, and z direction vectors. **pfGetSpotLightDir** copies the x, y, and z direction vectors into the parameters *x*, *y*, and *z*.

pfSpotLightCone specifies the exponent and spread of the spot light cone, and receives two floating point values, *f1* and *f2*, to set the exponent for the intensity, and the spread of the cone, respectively. **pfGetSpotLightCone** copies the current exponent and spread of the cone into the parameters *f1* and *f2*.

pfLightPos receives four floating point values to set the *x*, *y*, *z*, and *w*, coordinates for the position of the light source. Typically, the homogeneous coordinate *w* is 0.0 to indicate that the light position is infinitely far from the origin in the direction (*x*, *y*, *z*). Local light sources are specified by a non-zero value for *w* and usually incur a performance penalty. **pfGetLightPos** copies the x, y, z and w coordinates of the light source into the parameters *x*, *y*, *z* and *w*, respectively.

pfLightOn enables *light* so that its illumination will influence scene geometry if lighting is properly enabled (See below). The maximum number of active lights is determined by the particular graphics library implementation but typically is at least eight.

Modifications made to *light* do not have effect until **pfLightOn** is called.

For geometry to be illuminated, the following must be true:

1. Lighting must be enabled: **pfEnable**(**PFEN_LIGHTING**)
2. A pfLightModel must be applied: **pfApplyLModel**
3. A pfMaterial must be applied: **pfApplyMtl**
4. One or more pfLights must be on: **pfLightOn**
5. Illuminated geometry must have normals: **pfGSetAttr**, **PFGS_NORMAL3**

pfLightOn also affects the position of the light in the scene. When called, the current graphics library ModelView matrix transforms the position of the light set by **pfLightPos**. Calling **pfLightOn** when specific transformations are on the stack will result in different light behaviors, which are outlined in the

following paragraphs.

To simulate a light attached to the viewer (simulating a miner's head-mounted lamp) call **pfLightOn** only once with an identity matrix on the stack:

```
pfLightPos(viewerLight, 0.0, 0.0, 1.0, 0.0);

/*
 * viewerLight always points in direction of view, i.e. - down -Z axis.
 */
pfPushIdentMatrix();
pfLightOn(viewerLight);
pfPopMatrix();

/* Draw scene */
```

To simulate a light "attached" to the world (at a fixed location in world-space coordinates like the sun or moon) call **pfLightOn** every frame with only the viewing transformation on the stack:

```
pfLightPos(sunLight, 0.0, 1.0, 0.0, 0.0);

pfPushIdentMatrix();

/* viewer is at origin looking +30 degrees 'up' */
pfRotate(PF_X, -30.0f);

/* sunLight always points straight down on scene */
pfLightOn(sunLight);

/* Draw scene */

pfPopMatrix();
```

To simulate a light attached to an object like the headlights of a car, call **pfLightOn** every frame with the combined viewing and modeling transformation on the stack:

```
pfLightPos(headLight, 2.0, 0.0, 0.0, 1.0);

pfPushIdentMatrix();

/* Viewer is at origin looking +30 degrees 'up' */
pfRotate(PF_X, -30.0f);
```

```
/* Car is at (100.0f, 100.0f, 100.0f) */
pfTranslate(100.0f, 100.0f, 100.0f);

/*
 * carLight is a point light source at the front of the car
 * provided the car is modeled such that the headlights are
 * 2 units from the center of the car in the +X direction.
 */
pfLightOn(headLight);

/* Draw scene */

pfPopMatrix();
```

pfLightOff disables *light* so that it does not contribute to scene illumination.

pfIsLightOn returns a boolean indicating whether *light* is on or not.

pfGetCurLights returns the number of currently active lights, *n*. The array *lights* is filled with *n* pointers to the pfLight structures of the light sources that are currently ''on''.

The light source state element is identified by the **PFSTATE_LIGHT** token. Use this token with **pfGStateAttr** to set the light array of a pfGeoState and with **pfOverride** to override subsequent light source changes:

pfLightAmbient is provided for compatibility with previous versions of IRIS Performer. It accepts three floating point values in the range from 0.0 through 1.0 to set the *r*, *g*, and *b*, values for the red, green, and blue components of the ambient light. By default, lights have ambient red, green, and blue values of 0.0. **pfGetLightAmbient** copies the ambient light values for the given light source into the parameters *r*, *g*, and *b*. For future compatibility, calls to:

```
pfLightAmbient(lt, r, g, b);
```

should be replaced by

```
pfLightColor(lt, PFLT_AMBIENT, r, g, b);
```

and calls to:

```
pfGetLightAmbient(lt, &r, &g, &b);
```

should be replaced by

```
pfGetLightColor(lt, PFLT_AMBIENT, &r, &g, &b);
```

EXAMPLES

Example 1:

```
pfLight        *lightArray[PF_MAX_LIGHTS];

for (i=0; i<PF_MAX_LIGHTS; i++)
    lightArray[i] = NULL;

lightArray[0] = light0;
lightArray[1] = light1;

/* Set up specially-lit pfGeoState */
pfGStateMode(gstate, PFSTATE_ENLIGHTING, PF_ON);
pfGStateAttr(gstate, PFSTATE_LIGHT, lightArray);

/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/* Set normal array. 'gset' is non-indexed */
pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_VERTEX, norms, NULL);

/* Draw specially-lit gset */
pfDrawGSet(gset);
```

Example 2:

```
pfLightOn(light0);
pfLightOn(light1);

/*
 * Override so that all geometry is lit with light0 and light1
 * if lighting is otherwise properly enabled.
 */
pfOverride(PFSTATE_LIGHT, PF_ON);
```

The array of lights passed to **pfGStateAttr** should be **PF_MAX_LIGHTS** long and should contain references to pfLights that are to be used by the pfGeoState. Empty array elements should be set to **NULL**.

pfLightOn and **pfLightOff** are display-listable commands. If a pfDispList has been opened by **pfOpenDList**, **pfLightOn** and **pfLightOff** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

NOTES

Local lighting results in improper shading of flat-shaded triangle and line strips (-**PFGS_FLAT_TRISTRIPS**, **PFGS_LINE_TRISTRIPS**) which often manifests itself as "faceting" of planar polygons. The only solution is either to use infinite lighting or not use FLAT primitives. Note that when using the IRIS Performer triangle meshing routine, **pfdMeshGSet**, the construction of non-FLAT strips is easily enforced with **pfdMesherMode(PFDMESH_LOCAL_LIGHTING, 1)**.

SEE ALSO

pfDelete, pfDispList, pfGeoState, pfLightModel, pfMaterial, pfObject, pfOverride, pfState, lmbind, lmcolor, lmdef, glLight, glColorMaterial

NAME

pfNewLModel, pfGetLModelClassType, pfApplyLModel, pfLModelAtten, pfGetLModelAtten, pfLModelLocal, pfGetLModelLocal, pfLModelTwoSide, pfGetLModelTwoSide, pfLModelAmbient, pfGetLModelAmbient, pfGetCurLModel – Create, modify and query lighting model

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfLightModel * **pfNewLModel**(void *arena);

pfType * **pfGetLModelClassType**(void);

void **pfApplyLModel**(pfLightModel *lm);

void **pfLModelAtten**(pfLightModel *lm, float a0, float a1, float a2);

void **pfGetLModelAtten**(const pfLightModel *lm, float *a0, float *a1, float *a2);

void **pfLModelLocal**(pfLightModel *lm, int l);

int **pfGetLModelLocal**(const pfLightModel *lm);

void **pfLModelTwoSide**(pfLightModel *lm, int t);

int **pfGetLModelTwoSide**(const pfLightModel *lm);

void **pfLModelAmbient**(pfLightModel *lm, float r, float g, float b);

void **pfGetLModelAmbient**(const pfLightModel *lm, float *r, float *g, float *b);

pfLightModel * **pfGetCurLModel**(void);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfLightModel** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfLightModel**. Casting an object of class **pfLightModel** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfLightModel** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);

```
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

PARAMETERS

lm identifies a pfLightModel.

DESCRIPTION

A pfLightModel defines characteristics of the hardware lighting model used to illuminate geometry. There are some subtle differences between IRIS GL and OpenGL light operation and additional references are recommended and should be noted. See the IRIS GL **lmdef(3g)** or the OpenGL **glLightModel(3g)** reference page for more details on lighting environments and individual parameters.

pfNewLModel creates and returns a handle to a pfLightModel. *arena* specifies a malloc arena out of which the pfLightModel is allocated or **NULL** for allocation off the process heap. A NULL pointer is returned to indicate failure. pfLightModels can be deleted with **pfDelete**.

pfGetLModelClassType returns the **pfType*** for the class **pfLightModel**. The **pfType*** returned by **pfGetLModelClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfLightModel**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfLModelAtten sets the lighting attenuation factors for *lm*. *a1*, *a2*, and *a3* specify the constant, linear, and second-order attenuation factors, respectively. These factors are associated with all non-infinite lights. The default vales for constant, linear, and quadratic attenuation factors are 1.0, 0.0, and 0.0, respectively, effectively disabling each. Attenuation on the light model is done only in IRIS GL operation. OpenGL light attenuation is done per-light. See the **pfLightAtten** reference page for more information.

pfGetLModelAtten copies the lighting attenuation factors for *lm* into the parameters *a1*, *a2*, and *a3*.

pfLModelLocal specifies whether the light reflection calculations are to be done based on a local or infinite viewpoint. The default is **PF_OFF** signifying an infinite viewer for the light model. In general, local lighting is more expensive than infinite lighting.

pfGetLModelLocal returns a boolean value signifying whether or not the effective viewpoint in *lm* is a local viewpoint.

pfLModelTwoSide specifies whether two-sided lighting is to be used in the given light model. The default is **PF_OFF**, disabling two-sided lighting. See the IRIS GL **lmdef(3g)** or the OpenGL **glLightModel** reference page for more details on two-sided lighting.

pfGetLModelTwoSide returns the setting of *lm*'s two-sided lighting mode.

pfLModelAmbient receives three floating point values in the range from 0.0 through 1.0 to set the red, green, and blue, values for the amount of the ambient light associated with the scene for the given light model.

pfGetLModelAmbient copies the red, green, and blue components of the ambient in the given light model into the parameters *r*, *g*, and *b*, respectively. The default value for the ambient red, green, and blue light components is 0.2.

pfApplyLModel causes *lm*, with its current settings, to become the current lighting model. When lighting is enabled (See below), this lighting model will be applied to all geometry drawn after **pfApplyLModel** is called. Modifications to *lm*, such as changing the ambient color, or setting two-sided lighting, will not be applied until **pfApplyLModel** is called with *lm*.

For geometry to be illuminated, the following must be true:

1. Lighting must be enabled: **pfEnable**(**PFEN_LIGHTING**)
2. A pfLightModel must be applied: **pfApplyLModel**
3. A pfMaterial must be applied: **pfApplyMtl**
4. One or more pfLights must be on: **pfLightOn**
5. Illuminated geometry must have normals: **pfGSetAttr**, **PFGS_NORMAL3**

The lighting model state element is identified by the **PFSTATE_LIGHTMODEL** token. Use this token with **pfGStateAttr** to set the lighting model of a pfGeoState and with **pfOverride** to override subsequent lighting model changes:

EXAMPLES

Example 1:

```
pfLModelTwoSide(lmodel, PF_ON);

/* Set up two-sided lighting pfGeoState */
pfGStateMode(gstate, PFSTATE_ENLIGHTING, PF_ON);
pfGStateAttr(gstate, PFSTATE_LIGHTMODEL, lmodel);
pfGStateAttr(gstate, PFSTATE_FRONTMTL, mtl);
pfGStateAttr(gstate, PFSTATE_BACKMTL, mtl);
pfGStateMode(gstate, PFSTATE_CULLFACE, PF_OFF);
```

```
/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/* Set normal array. 'gset' is non-indexed */
pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_VERTEX, norms, NULL);

/* Draw lit, two-sided gset */
pfDrawGSet(gset);
```

Example 2:

```
pfApplyLModel(lmodel);

/* Override so that all geometry is lit with 'lmodel' */
pfOverride(PFSTATE_LIGHTMODEL, PF_ON);
```

pfApplyLModel is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfApplyLModel** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

pfGetCurLModel returns a pointer to the currently active pfLightModel, or NULL if there is no active pfLightModel.

SEE ALSO

pfDelete, pfDispList, pfGeoState, pfLight, glLightModel, pfMaterial, pfObject, pfState, lmbind, lmcolor, lmdef

NAME

pfNewList, pfGetListClassType, pfAdd, pfCombineLists, pfFastRemove, pfFastRemoveIndex, pfGet, pfGetListArray, pfGetListArrayLen, pfGetListEltSize, pfGetNum, pfInsert, pfMove, pfListArrayLen, pfNum, pfRemove, pfRemoveIndex, pfReplace, pfResetList, pfSearch, pfSet – Dynamically-sized list utility

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfList *	**pfNewList**(int eltSize, int listLength, void* arena);
pfType *	**pfGetListClassType**(void);
void	**pfAdd**(pfList* list, void* elt);
void	**pfCombineLists**(pfList* dst, const pfList *a, const pfList *b);
int	**pfFastRemove**(pfList* list, void* elt);
void	**pfFastRemoveIndex**(pfList* list, int index);
void *	**pfGet**(const pfList* list, int index);
const void **	**pfGetListArray**(const pfList* list);
int	**pfGetListArrayLen**(const pfList* len);
int	**pfGetListEltSize**(const pfList* list);
int	**pfGetNum**(const pfList* list);
void	**pfInsert**(pfList* list, int index, void* elt);
int	**pfMove**(pfList* lists, int index, void *elt);
void	**pfListArrayLen**(pfList* list, int len);
void	**pfNum**(pfList *list, int num);
int	**pfRemove**(pfList* list, void* elt);
void	**pfRemoveIndex**(pfList* list, int index);
int	**pfReplace**(pfList* list, void* old, void* new);
void	**pfResetList**(pfList* list);
int	**pfSearch**(const pfList* list, void* elt);
void	**pfSet**(pfList* list, int index, void *elt);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfList** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfList**. Casting an object of class **pfList** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
void*   pfGetUserData(pfObject *obj);
int     pfGetGLHandle(pfObject *obj);
```

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfList** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *      pfGetType(const void *ptr);
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

PARAMETERS

list identifies a pfList.

DESCRIPTION

A pfList is a dynamically-sized array of arbitrary, but homogeneously-sized, elements.

pfNewList creates and returns a handle to a new pfList. *eltSize* specifies the size in bytes of an individual list element. The element size is fixed at creation time and cannot be later changed. *listLength* is the initial length of the list; *listLength* * *eltSize* bytes will be allocated for the list array. The argument *arena* specifies a malloc arena out of which the pfList is to be allocated or **NULL** for allocation from the process heap. pfLists can be deleted with **pfDelete**.

pfGetListClassType returns the **pfType*** for the class **pfList**. The **pfType*** returned by **pfGetListClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfList**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

A pfList dynamically increases its array size by a factor of 2 and zeros the additional memory whenever it runs out of array memory. This way the array size quickly reaches its final size without many reallocations of memory. However, some memory (up to one half of the total allocation) at the end of the array may be wasted. If you know the exact number of elements in the array, you can specify the pfList array length either when creating it (the *listLength* argument to **pfNewList**) or with **pfListArrayLen**.

pfGetListArrayLen returns the current array length of *list*.

Example 1:

```
/* Fit list's array to its current number of elements */
pfListArrayLen(list, pfGetNum(list));
```

pfSet sets the *index*th element of *list* to *elt*. The list is automatically grown if *index* is beyond the current array length.

pfGet returns the element of *list* at index *index* or 0 if *index* is out of bounds.

pfAdd appends *elt* to *list* and automatically grows *list* if necessary.

pfRemove removes *elt* from *list* and shifts the array down over the vacant spot, e.g. - if *elt* had index 0, then index 1 becomes index 0, index 2 becomes index 1 and so on. **pfRemove** returns the index of *elt* if *elt* was actually removed and -1 if it was not found in the list. **pfRemoveIndex** removes the *index*th element of *list*, and like **pfRemove**, shifts the array down over the vacant spot.

pfFastRemove removes *elt* from *list* but does not shift the array; instead it places the last element of the array into the vacated location so it does not preserve the list ordering. **pfFastRemoveIndex** replaces the *index*th element with the last element of *list*.

Note that both **pfRemove** and **pfFastRemove** linearly search the array for *elt* and remove only the first matching element. To remove all occurrences of *elt* do the following:

```
while (pfRemove(list, elt) >= 0)
    /* empty */ ;
```

pfSearch returns the index of *elt* if *elt* was found in *list* and -1 otherwise.

pfInsert inserts *elt* before the array element with index *index*. *index* must be within the range [0 .. **pfGetNum**(*list*)].

pfMove deletes *elt* from its current location and inserts before the array element with index *index*. *index* must be within the range [0 .. **pfGetNum**(*list*)] or else (-1) is returned and no move is executed. If *elt* is not already in *list*, (-1) is returned and *elt* is not inserted into the list. Otherwise, *index* is returned to indicate success.

pfReplace replaces the first instance of *old* with *new* and returns the index of *old* if it was found in *list* and -1 otherwise.

pfGetNum returns the number of elements in *list*. (Actually, *list* may have holes in its array so **pfGetNum** technically should be considered as returning the maximum index of all elements in *list*.)

pfResetList zeros *list*'s array and resets the number of elements to 0. It does not resize the array.

pfCombineLists sets *dst* to *a* appended with *b*. *dst* may be the same as *a* or *b*. Lists must have equal element sizes to be combined.

For quick access to the list array, **pfGetListArray** returns a pointer to the internal array of *list*. Care should be taken with this routine since out of bounds range checking provided by pfList API is bypassed. If you add elements to *list* then use **pfNum** to set the number of elements of *list*.

BUGS

pfLists currently only support an element size of **sizeof(void*)**.

SEE ALSO

pfDelete

NAME

pfNewMStack, pfGetMStackClassType, pfResetMStack, pfPushMStack, pfPopMStack, pfPreMultMStack, pfPostMultMStack, pfLoadMStack, pfGetMStack, pfGetMStackTop, pfGetMStackDepth, pfPreTransMStack, pfPostTransMStack, pfPreRotMStack, pfPostRotMStack, pfPreScaleMStack, pfPostScaleMStack – Create and manipulate a matrix stack.

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfMatStack * **pfNewMStack**(int size, void *arena);

pfType * **pfGetMStackClassType**(void);

void **pfResetMStack**(pfMatStack *stack);

int **pfPushMStack**(pfMatStack *stack);

int **pfPopMStack**(pfMatStack *stack);

void **pfPreMultMStack**(pfMatStack *stack, const pfMatrix m);

void **pfPostMultMStack**(pfMatStack *stack, const pfMatrix m);

void **pfLoadMStack**(pfMatStack *stack, const pfMatrix m);

void **pfGetMStack**(const pfMatStack *stack, pfMatrix m);

pfMatrix * **pfGetMStackTop**(const pfMatStack *stack);

int **pfGetMStackDepth**(const pfMatStack *stack);

void **pfPreTransMStack**(pfMatStack *stack, float x, float y, float z);

void **pfPostTransMStack**(pfMatStack *stack, float x, float y, float z);

void **pfPreRotMStack**(pfMatStack *stack, float degrees, float x, float y, float z);

void **pfPostRotMStack**(pfMatStack *stack, float degrees, float x, float y, float z);

void **pfPreScaleMStack**(pfMatStack *stack, float xs, float ys, float zs);

void **pfPostScaleMStack**(pfMatStack *stack, float xs, float ys, float zs);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfMatStack** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfMatStack**. Casting an object of class **pfMatStack** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);

int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfMatStack** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

DESCRIPTION

These routines allow the creation and manipulation of a stack of 4x4 matrices.

pfNewMStack creates and returns a handle to a pfMatStack. *arena* specifies a malloc arena out of which the pfMatStack is allocated or **NULL** for allocation off the process heap. pfMatStacks can be deleted with **pfDelete**. *size* is the number of pfMatrix's in the matrix stack. The initial depth is 1 and the top of stack is the identity matrix.

pfGetMStackClassType returns the **pfType*** for the class **pfMatStack**. The **pfType*** returned by **pfGetMStackClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfMatStack**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfResetMStack sets the stack depth to 1 and sets the top of stack to the identity matrix.

pfPushMStack pushes down the specified matrix stack duplicating the top. **pfPopMStack** pops the matrix stack. Attempting to pop a matrix stack containing only a single element or pushing past the maximum depth causes a warning and leaves the stack unchanged.

pfPreMultMStack pre-multiplies the top of the stack by the matrix *m* and replaces the top of the stack with the product. Thus if *T* is the top of the stack, the operation replaces *T* with *m*T*. This order corresponds to that used by OpenGL's *glMultMatrix*. **pfPostMultMStack** operates similarly but using post-multiplication, calculating *T*m* instead.

pfLoadMStack replaces the top of the stack with the matrix *m*.

pfGetMStack copies the top of the matrix into the matrix *m*. **pfGetMStackTop** returns a pointer to the top of the matrix stack.

pfGetMStackDepth returns the current depth of the stack. Initially the depth is *1*.

The following transformations pre- and post- multiply the top of the matrix stack:

pfPreTransMStack and **pfPostTransMStack** respectively pre- and post- multiply the top of the matrix stack by the translation matrix generated by the coordinates *x*, *y* and *z*. (See **pfMakeTransMat**).

pfPreRotMStack and **pfPostRotMStack** respectively pre- and post- multiply the top of the matrix stack by the rotation by *degrees* about the axis defined by (*x*, *y*, *z*). (See **pfMakeRotMat**). The results are undefined if the vector (*x*, *y*, *z*) is not normalized.

pfPreScaleMStack and **pfPostScaleMStack** respectively pre- and post- multiply the top of the matrix stack by a scaling matrix. (See **pfMakeScaleMat**). The matrix scales by *x* in the X direction, *y* and the Y direction and *z* in the Z direction.

NOTES

pfPreRotMStack and **pfPostRotMStack** use pfSinCos which is faster than the libm counterpart, but has less resolution.

pfMatStack is not related to the GL matrix stack.

IMPORTANT: The argument order of degrees and axis to the **pfPreRotMStack** are not the same as to the corresponding routine **pfRotMStack** in the IRIS Performer 1.0 and IRIS Performer 1.1 releases. This change was first introduced in the IRIS Performer 1.2 release and is present in subsequent releases.

SEE ALSO

pfDelete, pfMakeRotMat, pfMakeScaleMat, pfMakeTransMat, pfMatrix, pfSinCos, multmatrix

NAME

pfNewMtl, pfGetMtlClassType, pfMtlSide, pfGetMtlSide, pfMtlAlpha, pfGetMtlAlpha, pfMtlShininess, pfGetMtlShininess, pfMtlColor, pfGetMtlColor, pfMtlColorMode, pfGetMtlColorMode, pfApplyMtl, pfGetCurMtl – Create, modify and query a material.

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfMaterial * **pfNewMtl**(void *arena);

pfType * **pfGetMtlClassType**(void);

void **pfMtlSide**(pfMaterial *mtl, int side);

int **pfGetMtlSide**(pfMaterial *mtl);

void **pfMtlAlpha**(pfMaterial *mtl, float alpha);

float **pfGetMtlAlpha**(pfMaterial *mtl);

void **pfMtlShininess**(pfMaterial *mtl, float shininess);

float **pfGetMtlShininess**(pfMaterial *mtl);

void **pfMtlColor**(pfMaterial *mtl, int color, float r, float g, float b);

void **pfGetMtlColor**(pfMaterial *mtl, int color, float *r, float *g, float *b);

void **pfMtlColorMode**(pfMaterial *mtl, int side, int mode);

int **pfGetMtlColorMode**(pfMaterial *mtl, int side);

void **pfApplyMtl**(pfMaterial *mtl);

pfMaterial * **pfGetCurMtl**(int side);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfMaterial** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfMaterial**. Casting an object of class **pfMaterial** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfMaterial** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *      pfGetType(const void *ptr);
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

PARAMETERS

mtl identifies a pfMaterial.

DESCRIPTION

In conjunction with other lighting parameters, a pfMaterial defines the appearance of illuminated geometry. A pfMaterial defines the reflectance characteristics of surfaces such as diffuse color and shininess. There are some subtle differences between IRIS GL and OpenGL light operation and additional references are recommended and should be noted. returned to indicate failure. See the IRIS GL **lmdef(3g)** or OpenGL **glMaterial(3g)** reference page for more details on materials parameters.

pfNewMtl creates and returns a handle to a pfMaterial. *arena* specifies a malloc arena out of which the pfMaterial is allocated or **NULL** for allocation off the process heap. A NULL pointer is returned to indicate failure. pfMaterials can be deleted with **pfDelete**.

pfGetMtlClassType returns the **pfType*** for the class **pfMaterial**. The **pfType*** returned by **pfGetMtlClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfMaterial**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfMtlSide receives a symbolic token, one of **PFMTL_FRONT**, **PFMTL_BACK**, or **PFMTL_BOTH** indicating which side of a polygon the material should affect. If lighting is to affect the back sides of polygons, two-sided lighting must be enabled. Two-sided lighting requires a two-sided pfLightModel (see **pfLModelTwoSide**) and that face culling be disabled (see **pfCullFace**) so that backfacing polygons are not rejected.

pfGetMtlSide returns the side(s) affected by *mtl*.

pfMtlAlpha specifies the alpha of *mtl* in the range 0.0 through 1.0. If transparency is enabled (see **pfTransparency**), a material whose alpha is < 1.0 and whose color mode is **PFMTL_CMODE_OFF** will be

transparent with alpha of 1.0 being completely opaque and 0.0 being completely transparent. The default alpha value is 1.0 or completely opaque. For non-homogeneous transparency, use a color mode other than **PFMTL_CMODE_OFF** and transparency will be taken from geometry colors. In OpenGL, **pfMtlAlpha** sets the alpha of the AMBIENT, DIFFUSE, EMISSIVE, and SPECULAR colors. However, it is the DIFFUSE alpha that determines the resulting alpha value from the lighting calculation.

pfGetMtlAlpha returns the alpha of *mtl*.

pfMtlShininess specifies the specular scattering exponent, or the shininess, of the given material. It receives a floating point value in the range 0.0 to 128.0. The default shininess value is 0.0, which effectively disables specular reflection.

pfGetMtlShininess returns the shininess of *mtl*.

pfMtlColor sets a specific color of *mtl*. *color* indicates which color is to be set by *r*, *g*, and *b* and is one of **PFMTL_AMBIENT**, **PFMTL_DIFFUSE**, **PFMTL_EMISSION**, or **PFMTL_SPECULAR**. The default colors are:

Light Component	Red	Green	Blue
PFMTL_AMBIENT	0.2	0.2	0.2
PFMTL_DIFFUSE	0.8	0.8	0.8
PFMTL_EMISSION	0.0	0.0	0.0
PFMTL_SPECULAR	0.0	0.0	0.0

pfGetMtlColor copies the *color* of *mtl* into *r*, *g*, and *b*. *color* may be one of **PFMTL_AMBIENT**, **PFMTL_DIFFUSE**, **PFMTL_EMISSION**, or **PFMTL_SPECULAR**.

pfMtlColorMode specifies how pfGeoSet and Graphics Library color commands affect *mtl*. *side* is the same symbolic token used for **pfMtlSide** and indicates which side *mode* affects. *mode* is a symbolic token specifying which color property of the material is replaced by color commands:

PFMTL_CMODE_AMBIENT_AND_DIFFUSE, RGB color commands will replace the DIFFUSE and AMBIENT color property of the current material. This is the default pfMaterial color mode.

PFMTL_CMODE_AMBIENT, RGB color commands will replace the AMBIENT color property of the current material.

PFMTL_CMODE_DIFFUSE, RGB color commands will replace the DIFFUSE color property of the current material.

PFMTL_CMODE_EMISSION, RGB color commands will replace the EMISSION color property of the current material.

PFMTL_CMODE_SPECULAR, RGB color commands will replace the SPECULAR color property of the current material.

PFMTL_CMODE_OFF, RGB color commands will be ignored, i.e., overridden by the material colors. Additionally, in IRIS GL, the current GL color will not be changed.

PFMTL_CMODE_COLOR, RGB color commands will replace the current color. In IRIS GL, if a color is the last thing sent before a vertex the vertex will be colored. If a normal is the last thing sent before a vertex the vertex will be lighted. In OpenGL, if lighting is enabled, lit material colors are always used. **PFMTL_CMODE_COLOR** is not available in OpenGL and will be treated as **PFMTL_CMODE_OFF**.

In IRIS GL, the alpha specified in RGBA color commands will replace a material's alpha if its color mode is **PFMTL_CMODE_AMBIENT_AND_DIFFUSE**, **PFMTL_CMODE_AMBIENT**, or **PFMTL_CMODE_DIFFUSE**. In OpenGL, materials do not have a single alpha; rather, the AMBIENT, DIFFUSE, SPECULAR, and EMISSIVE colors have individual alphas which are replaced along with red, green, and blue when the appropriate color mode is enabled.

When enabled, **pfMtlColorMode** can offer substantial performance gains by drastically reducing the number of different pfMaterials required by a database. Instead of using a different pfMaterial for every unique material color, **pfMtlColorMode** can take a color component from the geometry, rather than from *mtl*. For example, if *mode* is **PFMTL_CMODE_DIFFUSE**, then the diffuse color component of *mtl* is ignored. Instead, the color specified by a pfGeoSet or the color specified through the Graphics Library (e.g. **cpack(3g)** in IRIS GL, **glColor(3g)** in OpenGL) becomes the new diffuse color. However, **pfGetMtlColor** will still return the original diffuse color.

The **pfMtlColorMode** of *mtl* must be enabled (other than **PFMTL_CMODE_COLOR** or **PFMTL_CMODE_OFF**) for the colors (**PFGS_COLOR4**) of any pfGeoSets which use *mtl* to have effect. Note that the only way to display per-vertex colors on lit pfGeoSets is to enable **pfMtlColorMode** on the pfMaterial used by the pfGeoSets; specifically, pfGeoSets do not support a different pfMaterial for each vertex.

The default color mode is **PFMTL_CMODE_AMBIENT_AND_DIFFUSE** which causes both diffuse and ambient material colors to be replaced by geometry color commands. Specifically, this setting allows colors specified by pfGeoSets to have effect. When lighting is disabled, the color mode is set to **PFMTL_CMODE_COLOR** in IRIS GL and **PFMTL_CMODE_OFF** in OpenGL.

pfGetMtlColorMode returns the color mode of *mtl* corresponding to *side*.

pfApplyMtl makes *mtl* the current pfMaterial. If lighting is enabled (see below), *mtl* will be applied to all geometry drawn after **pfApplyMtl** is called. Modifications to *mtl*, such as changing the diffuse color, will not be applied until **pfApplyMtl** is called with *mtl*.

For geometry to be illuminated the following must be true:

1. Lighting must be enabled: **pfEnable**(**PFEN_LIGHTING**),
2. A pfLightModel must be applied: **pfApplyLModel**,
3. A pfMaterial must be applied: **pfApplyMtl**,
4. One or more pfLights must be on for diffuse and specular effects: **pfLightOn**,
5. Illuminated geometry must have normals for diffuse and specular effects: **pfGSetAttr**, **PFGS_NORMAL3**. Note that ambient and emissive lighting does not require normals.

The front and back material state elements are identified by the **PFSTATE_FRONTMTL** and **PFSTATE_BACKMTL** tokens. Use these tokens with **pfGStateAttr** to set the materials of a pfGeoState and with **pfOverride** to override subsequent material changes:

Example 1: Define a 50% transparent, shiny red plastic material

```
/* Make it red */
pfMtlColor(redMtl, PFMTL_DIFFUSE, 1.0f, 0.0f, 0.0f);

/* Disable color mode so the PFMTL_DIFFUSE color is not ignored */
pfMtlColorMode(redMtl, PFMTL_FRONT, PFMTL_CMODE_OFF);

/* Make it shiny */
pfMtlColor(redMtl, PFMTL_SPECULAR, 1.0f, 1.0f, 1.0f);
pfMtlShininess(redMtl, 16.0f);

/* Make it 50% transparent */
pfMtlAlpha(redMtl, 0.5f);

/* Set the front material of a pfGeoState */
pfGStateMode(gstate, PFSTATE_ENLIGHTING, PF_ON);
pfGStateMode(gstate, PFSTATE_TRANSPARENCY, PFTR_ON);
pfGStateAttr(gstate, PFSTATE_FRONTMTL, redMaterial);

/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/* Set normal array. 'gset' is non-indexed */
pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_VERTEX, norms, NULL);

/* Draw transparent, shiny red gset */
pfDrawGSet(gset);
```

Example 2:

```
pfMtlSide(mtl, PFMTL_FRONT);
pfApplyMtl(mtl);

/* Override so that all geometry uses 'mtl' as front material */
pfOverride(PFSTATE_FRONTMTL, PF_ON);
```

When setting the pfMaterial(s) of a pfGeoState using **pfGStateAttr**, the side of the material is ignored. Instead, the **PFSTATE** token defines which side the material should be applied to. For example,

```
pfGStateAttr(gstate, PFSTATE_FRONTMTL, mtl)
```

will ensure that *mtl* is always applied to the front side of polygons after *gstate* is applied.

pfApplyMtl is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfApplyMtl** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

pfGetCurMtl receives a symbolic token specifying the side of interest, one of **PFMTL_FRONT** or **PFMTL_BACK**, and returns a pointer to the currently active material for that side, or NULL if there is no active pfMaterial.

BUGS

IRIS GL does not support **lmcolor** for back-sided materials. Consequently, **pfMtlColorMode** has no effect on back-sided materials.

SEE ALSO

lmbind, lmcolor, lmdef, pfCullFace, pfDelete, pfDispList, pfEnable, pfGSetAttr, pfGeoState, pfLight, pfLightModel, pfLightOn, pfLModelTwoSide, pfObject, pfState, pfTransparency

NAME

pfMakeIdentMat, pfMakeTransMat, pfMakeScaleMat, pfMakeRotMat, pfMakeQuatMat, pfMakeEulerMat, pfMakeVecRotVecMat, pfMakeCoordMat, pfGetMatType, pfGetOrthoMatQuat, pfGetOrthoMatCoord, pfSetMat, pfSetMatRowVec3, pfGetMatRowVec3, pfSetMatColVec3, pfGetMatColVec3, pfSetMatRow, pfGetMatRow, pfSetMatCol, pfGetMatCol, pfCopyMat, pfAddMat, pfSubMat, pfScaleMat, pfTransposeMat, pfMultMat, pfPreMultMat, pfPostMultMat, pfPreTransMat, pfPostTransMat, pfPreRotMat, pfPostRotMat, pfPreScaleMat, pfPostScaleMat, pfInvertFullMat, pfInvertAffMat, pfInvertOrthoMat, pfInvertOrthoNMat, pfInvertIdentMat, pfEqualMat, pfAlmostEqualMat – Set and operate on 4x4 matrices.

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfMakeIdentMat**(pfMatrix dst);

void **pfMakeTransMat**(pfMatrix dst, float x, float y, float z);

void **pfMakeScaleMat**(pfMatrix dst, float x, float y, float z);

void **pfMakeRotMat**(pfMatrix dst, float degrees, float x, float y, float z);

void **pfMakeQuatMat**(pfMatrix m, const pfQuat q);

void **pfMakeEulerMat**(pfMatrix dst, float h, float p, float r);

void **pfMakeVecRotVecMat**(pfMatrix dst, const pfVec3 v1, const pfVec3 v2);

void **pfMakeCoordMat**(pfMatrix dst, const pfCoord *c);

int **pfGetMatType**(const pfMatrix mat);

void **pfGetOrthoMatQuat**(const pfMatrix m, pfQuat dst);

void **pfGetOrthoMatCoord**(pfMatrix m, pfCoord* dst);

void **pfSetMat**(const float *m);

void **pfSetMatRowVec3**(pfMatrix dst, int row, const pfVec3 v);

void **pfGetMatRowVec3**(const pfMatrix m, int row, pfVec3 dst);

void **pfSetMatColVec3**(pfMatrix dst, int col, const pfVec3 v);

void **pfGetMatColVec3**(const pfMatrix m, int col, pfVec3 dst);

void **pfSetMatRow**(pfMatrix dst, int row, float x, float y, float z, float w);

void **pfGetMatRow**(const pfMatrix m, int row, float *x, float *y, float *z, float *w);

void **pfSetMatCol**(pfMatrix dst, int col, float x, float y, float z, float w);

void **pfGetMatCol**(const pfMatrix m, int col, float *x, float *y, float *z, float *w);

void **pfCopyMat**(pfMatrix dst, const pfMatrix m);
void **pfAddMat**(pfMatrix dst, const pfMatrix m1, const pfMatrix m2);
void **pfSubMat**(pfMatrix dst, const pfMatrix m1, const pfMatrix m2);
void **pfScaleMat**(pfMatrix dst, float s, pfMatrix m);
void **pfTransposeMat**(pfMatrix dst, pfMatrix m);
void **pfMultMat**(pfMatrix dst, const pfMatrix m1, const pfMatrix m2);
void **pfPreMultMat**(pfMatrix dst, const pfMatrix m);
void **pfPostMultMat**(pfMatrix dst, const pfMatrix m);
void **pfPreTransMat**(pfMatrix dst, float x, float y, float z, pfMatrix m);
void **pfPostTransMat**(pfMatrix dst, const pfMatrix m, float x, float y, float z);
void **pfPreRotMat**(pfMatrix dst, float degrees, float x, float y, float z, pfMatrix m);
void **pfPostRotMat**(pfMatrix dst, const pfMatrix mat, float degrees, float x, float y, float z,);
void **pfPreScaleMat**(pfMatrix dst, float x, float y, float z, pfMatrix m);
void **pfPostScaleMat**(pfMatrix dst, const pfMatrix m, float x, float y, float z);
int **pfInvertFullMat**(pfMatrix dst, const pfMatrix m);
void **pfInvertAffMat**(pfMatrix dst, const pfMatrix m);
void **pfInvertOrthoMat**(pfMatrix dst, const pfMatrix m);
void **pfInvertOrthoNMat**(pfMatrix dst, const pfMatrix m);
int **pfInvertIdentMat**(pfMatrix dst, const pfMatrix m);
void **pfEqualMat**(const pfMatrix m1, const pfMatrix m2);
void **pfAlmostEqualMat**(const pfMatrix m1, const pfMatrix m2, float tol);

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

typedef float pfMatrix[4][4];
```

DESCRIPTION

Routines for pfMatrix, a 4X4 matrix.

pfMakeIdentMat sets *dst* to the identity matrix. **PFMAKE_IDENT_MAT** is an equivalent macro.

The following routines create transformation matrices based on multiplying a row vector by a matrix on the right, i.e. the vector v transformed by m is v * m. Many actions will go considerably faster if the last column is (0,0,0,1).

pfMakeTransMat sets *dst* to the matrix which translates by (*x*, *y*, *z*). Equivalent macro: **PFMAKE_TRANS_MAT**.

pfMakeScaleMat sets *dst* to the matrix which scales by *x* in the X direction, by *y* in the Y direction and by *z* in the Z direction. Equivalent macro: **PFMAKE_SCALE_MAT**

pfMakeRotMat sets *dst* to the matrix which rotates by *degrees* about the axis denoted by the unit vector (*x*, *y*, *z*). If (*x*, *y*, *z*) is not normalized, results are undefined.

pfMakeQuatMat builds a rotation matrix *m* that expresses the rotation defined by the quaternion *q*.

pfMakeEulerMat sets *dst* to a rotation matrix composed of the Euler angles *h*, *p*, *r*: *h* specifies heading, the rotation about the Z axis; *p* specifies pitch, the rotation about the X axis; and, *r* specifies roll, rotation about the Y axis. The matrix created is *dst* = R*P*H, where R is the roll transform, P is the pitch transform and H is the heading transform. All rotations follow the right hand rule. The convention is natural for a model in which +Y is "forward," +Z is "up" and +X is "right". This routine uses **pfSinCos** which is faster than the libm counterpart, but has less resolution (see **pfSinCos**).

pfMakeVecRotVecMat sets *dst* to the rotation matrix which rotates the vector *v1* onto *v2*, i.e. *v2 = v1 * dst*. *v1* and *v2* must be normalized.

pfMakeCoordMat sets *dst* to the matrix which rotates by the Euler transform specified by *c*->hpr and translates by *c*->xyz, i.e. *dst* = R*P*H*T, where R is the roll transform, P is the pitch transform and H is the heading transform, and T is the translation transform.

pfGetOrthoMatQuat constructs a quaternion *dst* equivalent to the rotation expressed by the orthonormal matrix *m*.

pfGetOrthoMatCoord returns in *dst* the translation and rotation of the orthonormal matrix, *m*. The returned pitch ranges from -90 to +90 degrees. Roll and heading range from -180 to +180.

pfDCSMatType allows the specification of information about the type of transformation the matrix represents. This information allows Performer to speed up some operations. The matrix type is specified

as the OR of

PFMAT_TRANS:
matrix includes a translational component in the 4th row.

PFMAT_ROT:
matrix includes a rotational component in the left upper 3X3 submatrix.

PFMAT_SCALE:
matrix includes a uniform scale in the left upper 3X3 submatrix.

PFMAT_NONORTHO:
matrix includes a non-uniform scale in the left upper 3X3 submatrix.

PFMAT_PROJ:
matrix includes projections.

PFMAT_HOM_SCALE:
matrix includes have mat[4][4] != 1.

PFMAT_MIRROR:
matrix includes mirroring transformation that switches between right handed and left handed coordinate systems.

pfGetMatType computes the type of matrix. This information can be useful if a matrix is to be used repeatedly, e.g. to transform many objects, but is somewhat time consuming to compute.

pfSetMatRow. *dst*[*row*][0] = *x*, *dst*[*row*][1] = *y*, *dst*[*row*][2] = *z*, *dst*[*row*][3] = *w*. Use the arguments to set row *row* of *dst*. *row* must be 0, 1, 2, or 3. Equivalent macro: **PFSET_MAT_ROW**.

pfGetMatRow. **x* = *dst*[*row*][0], **y* = *dst*[*row*][1], **z* = *dst*[*row*][2], **w* = *dst*[*row*][3]. Get the arguments to row *row* of *dst*. *row* must be 0, 1, 2, or 3. Equivalent macro: **PFGET_MAT_ROW**.

pfSetMatCol. *dst*[0][*col*] = *x*, *dst*[1][*col*] = *y*, *dst*[2][*col*] = *z*, *dst*[3][*col*] = *w*. Use the arguments to set col *col* of *dst*. *col* must be 0, 1, 2, or 3. Equivalent macro: **PFSET_MAT_COL**.

pfGetMatCol. **x* = *dst*[0][*col*], **y* = *dst*[1][*col*], **z* = *dst*[2][*col*], **w* = *dst*[3][*col*]. Get the arguments to col *col* of *dst*. *col* must be 0, 1, 2, or 3. Equivalent macro: **PFGET_MAT_COL**.

pfSetMatRowVec3. *dst*[*row*][i] = *v*[i], i = 0, 1, 2. Set row *row* of *dst* to the vector *v*. *row* must be 0, 1, 2, or 3. Equivalent macro: **PFSET_MAT_ROWVEC3**.

pfGetMatRowVec3. *dst*[i] = *m*[*row*][i], i = 0, 1, 2. Return row *row* of *m* and in *dst*. *row* must be 0, 1, 2, or 3. Equivalent macro: **PFGET_MAT_ROWVEC3**.

pfSetMatColVec3. *dst*[i][*col*] = *v*[i], i = 0, 1, 2. Set column *col* of *dst* to the vector *v*. *col* must be 0, 1, 2, or 3. Equivalent macro: **PFSET_MAT_COLVEC3**.

pfGetMatColVec3. *dst*[i] = *m*[i][*col*], i = 0, 1, 2. Return column *col* of *m* in *dst*. *col* must be 0, 1, 2, or 3. Equivalent macro: **PFGET_MAT_COLVEC3**.

pfSetMat. *dst*[i][j] = *m*[i*4+j], 0 <= i,j <= 3.

pfCopyMat: *dst* = *m*. Copies *m* into *dst*. Equivalent macro: **PFCOPY_MAT**

pfPreTransMat: *dst* = T(*x*,*y*,*z*) * *m*, where T(*x*,*y*,*z*) is the matrix which translates by (*x*,*y*,*z*).

pfPostTransMat: *dst* = *m* * T(*x*,*y*,*z*), where T(*x*,*y*,*z*) is the matrix which translates by (*x*,*y*,*z*).

pfPreRotMat: *dst* = R(*degrees*, *x*,*y*,*z*) * *m*, where R(*degrees*,*x*,*y*,*z*) is the matrix which rotates by *degrees* about the axis (*x*,*y*,*z*).

pfPostRotMat: *dst* = *m* * R(*degrees*, *x*,*y*,*z*), where R(*degrees*,*x*,*y*,*z*) is the matrix which rotates by *degrees* about the axis (*x*,*y*,*z*).

pfPreScaleMat: *dst* = S(*x*,*y*,*z*) * *m*, where S(*x*,*y*,*z*) is the matrix which scales by (*x*,*y*,*z*).

pfPostScaleMat: *dst* = *m* * S(*x*,*y*,*z*), where S(*x*,*y*,*z*) is the matrix which scales by (*x*,*y*,*z*).

pfAddMat: *dst* = *m1* + *m2*. Sets *dst* to the sum of *m1* and *m2*.

pfSubMat: *dst* = *m1* - *m2*. Sets *dst* to the difference of *m1* and *m2*.

pfScaleMat: *dst* = *s* * *m*. Sets *dst* to the product of the scalar *s* and the matrix *m*. This multiplies the full 4X4 matrix and is not a 3D geometric scale.

pfTransposeMat: *dst* = Transpose(*m*). Sets *dst* to the transpose of *m*.

pfMultMat: *dst* = *m1* * *m2*. Sets *dst* to the product of *m1* and *m2*.

pfPostMultMat: *dst* = *dst* **m*. Postmultiplies *dst* by *m*.

pfPreMultMat: *dst* = *m* * *dst*. Premultiplies *dst* by *m*.

pfInvertFullMat, **pfInvertAffMat**, **pfInvertOrthoMat**, **pfInvertOrthoNMat**, and **pfInvertIdentMat**, set *dst* to the inverse of *m* for general, affine, orthogonal, orthonormal and identity matrices respectively. They are listed here in order of decreasing generality and increasing speed. If the matrix *m* is not of the type specified in the routine name, the result is undefined. **pfInvertFullMat** returns FALSE if the matrix is singular and TRUE otherwise.

pfEqualMat(m1, m2) = (*m1* == *m2*). Tests for strict component-by-element equality of two matrices *m1*

and *m2* and returns FALSE or TRUE. Macro equivalent: **PFEQUAL_MAT**.

pfAlmostEqualMat(m1, m2, tol). Tests for approximate element-by-element equality of two matrices *m1* and *m2*. It returns FALSE or TRUE depending on whether the absolute value of the difference between each pair of elements is less than the tolerance *tol*. Macro equivalent: **PFALMOST_EQUAL_MAT**.

Routines can accept the same matrix as source, destination, or as a repeated operand.

NOTES

Some of these routines use pfSinCos and pfSqrt, which are faster but have less resolution than the libm counterparts. (See **pfSinCos**)

SEE ALSO

pfSinCos, pfSqrt, pfVec3, pfVec4

NAME

pfGetMemoryClassType, pfGetType, pfIsOfType, pfIsExactType, pfGetTypeName, pfGetMemory, pfGetData, pfRef, pfUnref, pfUnrefDelete, pfGetRef, pfCopy, pfDelete, pfCompare, pfPrint, pfMalloc, pfCalloc, pfRealloc, pfFree, pfGetArena, pfGetSize, pfStrdup – Reference, copy, delete, print and query pfMemory

FUNCTION SPECIFICATION

#include <Performer/pf.h>

pfType * **pfGetMemoryClassType**(void);

pfType * **pfGetType**(const void *ptr);

int **pfIsOfType**(const void *ptr, pfType *type);

int **pfIsExactType**(const void *ptr, pfType *type);

const char * **pfGetTypeName**(const void *ptr);

pfMemory * **pfGetMemory**(const void *ptr);

void * **pfGetData**(const void *ptr);

int **pfRef**(void *ptr);

int **pfUnref**(void *ptr);

int **pfUnrefDelete**(void *ptr);

int **pfGetRef**(const void *ptr);

int **pfCopy**(void *dst, void *src);

int **pfDelete**(void *ptr);

int **pfCompare**(const void *ptr1, const void *ptr2);

void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);

void * **pfMalloc**(size_t nbytes, void *arena);

void * **pfCalloc**(size_t numelem, size_t elsize, void *arena);

void * **pfRealloc**(void *ptr, size_t nbytes);

void **pfFree**(void *ptr);

void * **pfGetArena**(void *ptr);

size_t **pfGetSize**(void *ptr);

char * **pfStrdup**(const char *str, void *arena);

DESCRIPTION

pfMemory is the base class from which all major IRIS Performer classes are derived and is also the type used by the IRIS Performer memory allocation routines such as **pfMalloc** and **pfFree**.

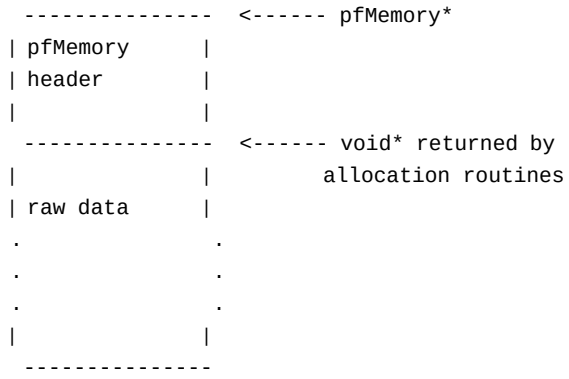
Because most IRIS Performer data structures are derived from pfMemory, they inherit the functionality of the pfMemory routines described here. In practice this means you can use the pfMemory routines listed above with most any IRIS Performer object, such as pfMaterial, pfList, pfFog, pfFrustum, pfChannel, pfGroup, pfGeode or with a data pointer returned by **pfMalloc**.

pfMemory supports the following:

1. Typed data structures.
2. Memory arena allocation.
3. Memory chunks which know their size.
4. Reference counting.

with only a 4 word overhead.

Although the IRIS Performer general memory allocation routines (**pfMalloc**) create pfMemories, they return void* so the application can treat the allocation as raw data. Consequently, all routines that would normally take a pfMemory* take a void* and infer the pfMemory handle so that applications can treat pfMemory as raw memory. However, one caveat is that routines which take raw memory such as **pfGSetAttr** or **pfFree** should not be passed a pointer to static data since the routines may not be able to successfully infer the pfMemory handle from the void*.



Routines which convert between pfMemory* and void* are:

void* -> pfMemory*: **pfGetMemory**

pfMemory* -> void*: **pfGetData**

Note that is it legal to pass either a pfMemory* or a void* to those routines which are prototyped as accepting a void*, e.g., **pfRef**. In this way, a single set of routines supports the same feature set including reference counts, copy, and delete for pfMemories used as IRIS Performer data types like pfGeoSet as well as for pfMemories used as raw data like pfGeoSet attribute arrays.

pfGetMemoryClassType returns the **pfType*** for the class **pfMemory**. The **pfType*** returned by **pfGetMemoryClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfMemory**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

All objects derived from pfMemory have a type identifier (pfType*) that is returned by **pfGetType**

Example 1: API sharing.

```
dcs = pfNewDCS();

/* pfDCS uses pfGroup routine */
pfAddChild(dcs, geode);

/* pfDCS uses pfNode routine */
pfNodeTravMask(dcs, PFTRAV_ISECT, DCS_MASK, PFTRAV_SELF | PFTRAV_DESCEND, PF_SET);
```

Each data type derived from pfMemory has an associated routine for getting a pointer to its corresponding pfType, e.g. **pfGetDCSClassType()** returns the pfType* corresponding to the pfDCS class. The exact type of an object is tested by comparing its pfType* to that returned by one of these **pfGet<*>ClassType** routines or with the **pfIsExactType** test, e.g.

```
if (pfGetType(obj) == pfGetGroupClassType()) ...

if (pfIsExactType(obj, pfGetGroupClassType())) ...
```

But since IRIS Performer allows subclassing and the creation of new types in C++, it's more often desirable to know whether a particular object is of a type derived from a particular type defined by IRIS Performer. In particular, exact type tests makes application code more likely to fail on scene graphs produced by database loaders that use subclassing. **pfIsOfType** performs this test and returns TRUE if

mem's type is derived from *type*:

```
if (pfIsOfType(obj, pfGetGroupClassType())) ...
```

If 'obj' is a pfDCS, then the above conditional would evaluate TRUE since pfDCS is derived from pfGroup.

pfGetTypeName returns a string that identifies the type of *mem*. For example, if *mem* is a pfDCS, the string returned is "pfDCS".

All pfMemories have a reference count which indicates how many times the pfMemory is referenced, either by other pfMemories or by the application. Reference counts are crucial for many database operations, particularly deletion, since it is highly dangerous to delete a pfMemory which is still being used, i.e., its reference count is greater than 0.

Reference counts may be incremented and decremented by **pfRef** and **pfUnref** respectively. **pfGetRef** returns the reference count of *mem*. **pfUnrefDelete** will decrement the reference count of *mem* and delete it if the count is <= 0. Thus it is equivalent to calling **pfUnref** followed by **pfDelete**.

pfDelete frees the memory associated with *mem* if its reference count is <= 0. When an object is freed, it decrements the reference count of all pfMemories that it once referenced and will delete any of these pfMemories with reference counts that are <= 0. Thus, **pfDelete** will follow all reference chains until it encounters a pfMemory which it cannot delete. Note that the reference count of a pfNode is incremented each time it is added as a child to a pfGroup. Thus, a pfNode must be removed from all its parents before it can be deleted.

When multiprocessing in a **libpf** application, pfNodes should be **pfDelete**ed only in the APP or DBASE processes as should **libpr** objects that are referenced directly or indirectly by pfNodes, like pfGeoSets and pfGeoStates. If you wish to delete objects in processes other than the APP or DBASE, use **pfAsyncDelete**.

Example 2: Deletion

```
pfMaterial     *mtl;
pfTexture      *tex;
pfGeoState     *brickStyle, *woodStyle;
pfGeoSet       *brickWall, *woodWall;

mtl = pfNewMtl(arena);

brickStyle = pfNewGState(arena);
tex = pfNewTex(arena);
pfLoadTexFile(tex, "brick.rgb");
pfGStateAttr(brickStyle, PFSTATE_TEXTURE, tex);
```

```
pfGStateAttr(brickStyle, PFSTATE_FRONTMTL, mtl);

woodStyle = pfNewGState(arena);
tex = pfNewTex(arena);
pfLoadTexFile(tex, "wood.rgb");
pfGStateAttr(woodStyle, PFSTATE_TEXTURE, tex);
pfGStateAttr(woodStyle, PFSTATE_FRONTMTL, mtl);

brickWall = pfNewGSet(arena);
pfGSetGState(brickWall, brickStyle);
pfGSetAttr(brickWall, PFGS_COORD3, PFGS_PER_VERTEX, coords);

woodWall = pfNewGSet(arena);
pfGSetAttr(woodWall, PFGS_COORD3, PFGS_PER_VERTEX, coords);
pfGSetGState(woodWall, woodStyle);

pfDelete(woodWall);

/* At this point woodWall, woodStyle, and the wood texture
 * have been deleted. coords and mtl have not been deleted
 * since they are referenced by brickWall and brickStyle respectively.
*/
```

pfDelete returns the following:

FALSE	*mem* was not deleted
TRUE	*mem* was deleted
-1	*mem* is not a pfMemory

pfDelete is implemented for all IRIS Performer objects except the following:

pfPipe
pfChannel
pfEarthSky
pfBuffer
pfPipeWindow
pfTraverser
pfState
pfDataPool

pfCopy copies *src* into *dst*. **pfCopy** is not recursive - it does not follow reference chains but instead copies only the first- level references. The reference counts of objects newly referenced by *dst* are incremented by one while those counts of objects previously reverenced by *dst* are decremented by one. Objects whose reference counts reach 0 during **pfCopy** are *not* deleted.

pfCopy is currently not implemented for any libpf data structures and is not implemented for the following libpr data structures:

pfState
pfDataPool

pfPrint Prints information to a file about the specified pfMemory *mem*. The *file* argument specifies the file. If *file* is **NULL**, the listing is printed to stderr. **pfPrint** takes a verbosity indicator, *verbose*. Valid selections in order of increasing verbosity are:

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 *mem* is a type of pfNode, then *which* specifies whether the print traversal should only traverse the current node (**PFTRAV_SELF**) or print out the entire scene graph rooted by node *mem* by traversing *node* and its descendents in the graph (**PFTRAV_SELF | PFTRAV_DESCEND**). If *mem* is a pfFrameStats, then *which* specifies a bitmask of frame statistics classes that should be printed. If *mem* is a pfGeoSet, then *which* is ignored and information about that pfGeoSet is printed according to the verbosity indicator. The output contains the types, names and bounds of the nodes and pfGeoSets in the hierarchy. This routine is provided for debugging purposes only and the content and format may change in future releases.

Example 3: Print entire contents of a pfGeoSet, *gset*, to stderr.

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

Example 4: Print entire scene graph under node to a file *file* with default verbosity.

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

Example 5: Print select classes of a pfFrameStats structure, *stats*, to stderr.

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

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 **amalloc** function). In addition, these routines provide a reference counting mechanism used by IRIS Performer to efficiently manage memory.

pfMalloc operates identically to the C-library **malloc** function, except that a shared memory arena may be specified to allocate the memory from. If *arena* is NULL, memory is allocated from the heap, otherwise memory is allocated from *arena* which must be a previously configured shared memory arena (see pfSharedMem).

Shared memory arenas can be created using **acreate** and can be found by using **pfGetSharedArena**.

pfCalloc and **pfRealloc** function just as their Unix counterparts, except that they may use shared arenas.

In all cases, a pointer to the allocated memory block is returned or NULL if there is not enough available memory.

The data pointer returned by **pfMalloc**, **pfCalloc**, and **pfRealloc** is actually part of a pfMemory object that, among other things, provides a reference count. Reference counts are used to keep track of how many times each allocated block of memory is referenced or instanced. All IRIS Performer libpr objects (pfMemory) are created with **pfMalloc** so their reference counts are updated by appropriate libpr routines. Examples of references follow:

Example 6:

```
tex = pfNewTex(NULL);

/* Attach 'tex' to gstate0 and gstate1 */
pfGStateAttr(gstate0, PFSTATE_TEXTURE, tex);
pfGStateAttr(gstate1, PFSTATE_TEXTURE, tex);

/* The reference count of 'tex' is now 2 */

/* Remove 'tex' from gstate1 */
pfGStateAttr(gstate1, PFSTATE_TEXTURE, NULL);

/* The reference count of 'tex' is now 1 */
```

Example 7:

```
coords = (pfVec3*) pfMalloc(sizeof(pfVec3) * numVerts, arena);

/* Attach 'coords' to non-indexed pfGeoSet, 'gset' */
pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX, coords, NULL);

/* The reference count of 'coords' is now 1 */
```

Example 8:

```
/* Attach 'gstate0' to 'gset' */
pfGSetGState(gset, gstate0);

/* The reference count of 'gstate0' is now incremented by 1 */
```

pfFree frees the memory associated with *ptr*. It is an error to **pfFree** memory that was not allocated by **pfMalloc**, **pfCalloc**, or **pfRealloc**. It is also an error to use any method other than **pfFree** or **pfDelete** to free memory allocated by **pfMalloc**, **pfCalloc**, or **pfRealloc**.

pfFree does not honor the reference count of *ptr*. This means that you can free a chunk of memory that is still being used (which means that its reference count is > 0) with potentially disastrous results. Typical failure modes are in the form of mysterious memory corruption and segmentation violations.

pfDelete, however, does honor the reference count of *ptr* and will not delete any memory whose reference count is > 0. **pfDelete** returns -1 if *ptr* is not a pfMalloc pointer, TRUE if *ptr* was deleted, and FALSE otherwise. **pfDelete** is recommended if you are not sure of the reference count of a piece of memory. See the pfObject reference page for more details on **pfDelete**.

pfGetArena returns the arena pointer which *ptr* was allocated from or NULL if *ptr* was allocated from the process heap.

pfGetSize returns the size in bytes of the memory referenced by *ptr* or 0 if *ptr* is not a **pfMalloc** pointer.

pfStrdup duplicates the NULL-terminated string *str* by allocating storage in the shared memory arena defined by the *arena* argument.

BUGS

pfPrint is not yet implemented for pfGeoStates and other state structures, and is not implemented for pfPaths or pfLists.

SEE ALSO

acreate, calloc, free, malloc, realloc, pfInitArenas

NAME

pfNotify, pfNotifyLevel, pfGetNotifyLevel, pfNotifyHandler, pfGetNotifyHandler, pfDefaultNotifyHandler – Control error handling, signal errors or log messages

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void	**pfNotify**(int severity, int error, char *format,
void	**pfNotifyLevel**(int severity);
int	**pfGetNotifyLevel**(void);
void	**pfNotifyHandler**(pfNotifyFuncType handler);
pfNotifyFuncType	**pfGetNotifyHandler**(void);
void	**pfDefaultNotifyHandler**(pfNotifyData *data);

```
typedef struct
{
    int severity;
    int pferrno;
    char *emsg;
} pfNotifyData;

typedef void (*pfNotifyFuncType)(pfNotifyData*);
```

DESCRIPTION

These functions provide a general purpose error message and notification handling facility for applications using IRIS Performer. This facility is used internally by IRIS Performer for error, warning, and status notifications and can be used by user developed programs as well.

pfNotifyHandler sets *handler* as the user error handling routine. All errors, warnings and notices will call *handler* with a pointer to a pfNotifyData structure that describes the error or message. The default notification handler **pfDefaultNotifyHandler** prints out a message of the form:

```
PF <LEVEL>/<PFERROR>(<ERRNO>) <MESSAGE>
```

where LEVEL is a string indicating the severity of the error, PFERROR is the type of error detected, ERRNO is the value of the system global **errno** (see perror(3C)), and MESSAGE is the formatted error message given **pfNotify**. The default handler zeros the system global errno. If PFERROR is **PFNFY_MORE**, the message is considered to be a continuation of the previous message and the print format is:

```
PF                      <MESSAGE>
```

The companion function **pfGetNotifyHandler** returns the address of the installed handler function. It is possible to inquire this address and provide it to user installed handlers in order to chain multiple notification handlers to any desired level.

pfNotifyLevel sets the threshold for notification. A notification must have a level less than or equal to the threshold for the default handler to print a message. The notification handler itself is invoked regardless of the notification level. The levels are in decreasing severity:

PFNFY_ALWAYS
PFNFY_FATAL
PFNFY_WARN
PFNFY_NOTICE
PFNFY_INFO
PFNFY_DEBUG
PFNFY_FP_DEBUG.

Call **pfGetNotifyLevel** to query the current notification level. The meaning of these notification levels is as follows:

Error Level	Description
PFNFY_ALWAYS	Always print regardless of notify level
PFNFY_FATAL	Fatal error, the dying gasp of a doomed process
PFNFY_WARN	Serious warning, rarely used for frame-time errors
PFNFY_NOTICE	Warning, may be used for frame time errors
PFNFY_INFO	Information on progress as well as errors
PFNFY_DEBUG	Debug information of significant verbosity
PFNFY_FP_DEBUG	Debug information and floating point exceptions

Setting the notification level to PFNFY_FP_DEBUG also enables floating point exceptions for overflow, underflow and invalid operations. Normally, these floating point errors are handled through kernel exceptions or by the floating point hardware, and may be nearly invisible to an application except from the performance degradation, sometimes very significant, which they can cause. When enabled, pfNotify events are generated for the floating point exceptions mentioned above and messages displayed or passed to the user supplied pfNotify handler.

The environment variable PFNFYLEVEL can be set to override the value specified in pfNotifyLevel. Once the notification level is set via PFNFYLEVEL it can not be changed by an application.

A notification level of PFNFY_FATAL causes the program to exit after notification; less severe levels do

not.

pfNotify generates an error message. *severity* must be one of the above listed constants. *error* may be any integer value, however, IRIS Performer uses the following values internally:

PFNFY_USAGE
PFNFY_RESOURCE
PFNFY_SYSERR
PFNFY_ASSERT
PFNFY_PRINT
PFNFY_INTERNAL
PFNFY_FP_OVERFLOW
PFNFY_FP_DIVZERO
PFNFY_FP_INVALID
PFNFY_FP_UNDERFLOW
PFNFY_MORE does a continuation of the previous message.

The severity must be less than or equal to the severity set in **pfNotifyLevel** for the error message to be output.

NOTES

Notification level is managed on a per process basis. Processes forked off after **pfNotifyLevel** is called inherit the specified level.

BUGS

Enabling floating point exceptions may cause the values returned from exceptions to be different than the system defaults. After an **_INVALID** operation, all subsequent exceptions will generate incorrect return values.

SEE ALSO

errno, handle_sigfpes, perror

NAME

pfGetObjectClassType, pfUserData, pfGetUserData, pfCopyFunc, pfGetCopyFunc, pfDeleteFunc, pfGetDeleteFunc, pfPrintFunc, pfGetPrintFunc, pfGetGLHandle – pfObject, callback and user data operations

FUNCTION SPECIFICATION

#include <Performer/pf.h>

pfType*	**pfGetObjectClassType**(void);
void	**pfUserData**(pfObject *obj, void *data);
void*	**pfGetUserData**(pfObject *obj);
void	**pfCopyFunc**(pfCopyFuncType func);
pfCopyFuncType	**pfGetCopyFunc**(void);
void	**pfDeleteFunc**(pfDeleteFuncType func);
pfDeleteFuncType	**pfGetDeleteFunc**(void);
void	**pfPrintFunc**(pfPrintFuncType func);
pfPrintFuncType	**pfGetPrintFunc**(void);
int	**pfGetGLHandle**(pfObject *obj);

```
typedef void (*pfCopyFuncType)(pfObject *dst, const pfObject *src);
typedef void (*pfDeleteFuncType)(pfObject *obj);
typedef void (*pfPrintFuncType)(const pfObject *obj, uint which, uint verbose, char *, FILE *);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfObject** is derived from the parent class **pfMemory**, so each of these member functions of class **pfMemory** are also directly usable with objects of class **pfObject**. Casting an object of class **pfObject** to an object of class **pfMemory** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfMemory**.

pfType *	**pfGetType**(const void *ptr);
int	**pfIsOfType**(const void *ptr, pfType *type);
int	**pfIsExactType**(const void *ptr, pfType *type);
const char *	**pfGetTypeName**(const void *ptr);
int	**pfRef**(void *ptr);
int	**pfUnref**(void *ptr);
int	**pfUnrefDelete**(void *ptr);

int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

DESCRIPTION

A pfObject is the abstract data type from which the major IRIS Performer data structures are derived. pfObject in turn derives from pfMemory which is the basic memory allocation unit. Although pfObjects cannot be created directly, most IRIS Performer data structures are derived from them and thus inherit the functionality of the pfObject routines described here and those for pfMemory.

pfGetObjectClassType returns the **pfType*** for the class **pfObject**. The **pfType*** returned by **pfGetObjectClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfObject**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfUserData attaches the user-supplied data pointer, *data*, to *obj*. User data provides a mechanism for associating application specific data with IRIS Performer objects.

Example 2: 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);
```

pfGetUserData returns the user-data pointer associated with *obj* or NULL if there is none.

Note that memory from **pfMalloc** is not an considered a pfObject so user-data pointers are not provided for pfMalloc'ed memory.

User data is reference counted if it is a **libpr**-type object like pfTexture, pfGeoSet, or memory allocated from **pfMalloc**. Thus user data is deleted if its reference count reaches 0 when its parent pfObject is deleted.

pfDeleteFunc, **pfCopyFunc**, and **pfPrintFunc** set global function callbacks which are called when deleting, copying, and printing a pfObject with non-NULL user data. These callbacks are provided so you can change the default behavior of user data. If a callback is not specified or is NULL, the default behaviors are:

1. Delete: Call **pfUnrefDelete** on the user data.
2. Copy: Decrement the reference count of the user data attached to the destination pfObject (but do not delete it), increment the reference count of the user data attached to the source pfObject and copy the user data pointer from the source to the destination pfObject. In pseudo-code:

```
pfUnref(dst->userData);
pfRef(src->userData);
dst->userData = src->userData;
```

3. Print: Print the address of the user data.

pfGetDeleteFunc, **pfGetCopyFunc**, and **pfGetPrintFunc** return the global deletion, copy, and print callbacks respectively.

Example 3: How to delete the user data of Example 2.

```
void
myDeleteFunc(pfObject *obj)
{
    myMaterial *mtl = pfGetUserData(obj);

    pfFree(mtl->dataPoints);
    pfFree(mtl);
}
 :
/* allocate a new material */
graniteMtl = pfNewMtl(NULL);
```

```
/* bind user data to material */
pfUserData(graniteMtl, granite);

/* set deletion callback */
pfDeleteFunc(myDeleteFunc);

/*
 * This will trigger callback only if graniteMtl has
 * a reference count <= 0.
 */
pfDelete(graniteMtl);
```

In the above example, the 'dataPoints' array of the 'myMaterial' structure would not have been freed without the deletion callback since pfDelete would have simply deleted the myMaterial structure.

pfGetGLHandle is a back-door mechanism for those who need to tweak the graphics library objects which underly many libpr objects. **pfGetGLHandle** returns the graphics library identifier associated with *obj* or -1 if *obj* has no associated graphics library object.

SEE ALSO

pfDelete, pfMemory

NAME

pfOverride, pfGetOverride, pfGLOverride, pfGetGLOverride – Override state element(s).

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfOverride**(uint mask, int val);

uint **pfGetOverride**(void);

void **pfGLOverride**(int which, float val);

float **pfGetGLOverride**(int which);

PARAMETERS

mask is a bit mask that specifies the state elements that are to be set to the override value *val*. *mask* is a bitwise OR of:

PFSTATE_ENLIGHTING
PFSTATE_ENTEXTURE
PFSTATE_ENFOG
PFSTATE_ENWIREFRAME
PFSTATE_ENHIGHLIGHTING
PFSTATE_ENCOLORTABLE
PFSTATE_FRONTMTL
PFSTATE_BACKMTL
PFSTATE_TEXTURE
PFSTATE_TEXENV
PFSTATE_ALPHAFUNC
PFSTATE_TRANSPARENCY
PFSTATE_ANTIALIAS
PFSTATE_CULLFACE
PFSTATE_DECAL
PFSTATE_ALPHAREF
PFSTATE_COLORTABLE
PFSTATE_HIGHLIGHT
PFSTATE_FOG
PFSTATE_LIGHTS
PFSTATE_LIGHTMODEL

val is a symbolic token and either **PF_ON** or **PF_OFF** indicating whether the state elements in *mask* should be overridden or not.

DESCRIPTION

pfOverride is used to override individual state elements. If a state element is overridden, all subsequent attempts to modify it will be ignored. An overridden state element is locked to the current value at the time **pfOverride** is called.

Example 1:

```
pfTransparency(PFTR_OFF);
pfApplyTex(tex);
pfDisable(PFEN_LIGHTING);
pfOverride(PFSTATE_TRANSPARENCY | PFSTATE_TEXTURE | PFSTATE_ENLIGHTING, PF_ON);
```

This example turns off transparency and lighting and applies *tex* to all subsequent geometry for which texturing is enabled.

pfOverride with a *val* of **PF_OFF** will free the state elements specified by *mask* to be modified. Although state elements will not be restored to their pre-override condition, **pfPushState** and **pfPopState** may be used to do so. The override mask is pushed and popped along with the rest of the state.

pfGetOverride returns the current override mask.

pfOverride is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfOverride** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

pfGLOverride overrides the graphics library mechanism used to achieve a desired effect such as transparency and visually correct coplanar geometry. *which* identifies the mechanism to override and is one of the following:

PFGL_TRANSPARENCY
val may be either **PFGL_TRANSP_MSALPHA** or **PFGL_TRANSP_BLEND**. in which case multisample ("screen-door") or blending will be used when transparency is enabled (see **pfTransparency**).

PFGL_DECAL
val may be either **PFGL_DECAL_STENCIL** or **PFGL_DECAL_DISPLACE** in which case stenciling or displacement will be used for all decals (see **pfDecal**).

pfGetGLOverride returns the override mode corresponding to *which*.

SEE ALSO

pfDispList, pfDrawDList, pfGeoState, pfOpenDList, pfState, pfTransparency, pfDecal

NAME

pfMakePtsPlane, pfMakeNormPtPlane, pfDisplacePlane, pfClosestPtOnPlane, pfPlaneIsectSeg, pfHalfSpaceIsectSeg, pfHalfSpaceContainsPt, pfHalfSpaceContainsBox, pfHalfSpaceContainsSphere, pfHalfSpaceContainsCyl, pfOrthoXformPlane – Set and operate on planes

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfMakePtsPlane**(pfPlane *dst, const pfVec3 pt1, const pfVec3 pt2, const pfVec3 pt3);

void **pfMakeNormPtPlane**(pfPlane *dst, const pfVec3 norm, const pfVec3 pos);

void **pfDisplacePlane**(pfPlane *dst, float d);

void **pfClosestPtOnPlane**(const pfPlane *pln, const pfVec3 pt, pfVec3 dst);

int **pfPlaneIsectSeg**(const pfPlane *pln, const pfSeg *seg, float *d);

int **pfHalfSpaceIsectSeg**(const pfPlane* hs, const pfSeg* seg, float* d1, float* d2);

int **pfHalfSpaceContainsPt**(const pfPlane* p, const pfVec3 pt);

int **pfHalfSpaceContainsBox**(const pfPlane* p, const pfBox *box);

int **pfHalfSpaceContainsSphere**(const pfPlane *hs, const pfSphere *sph);

int **pfHalfSpaceContainsCyl**(const pfPlane *hs, const pfCylinder *cyl);

void **pfOrthoXformPlane**(pfPlane *dst, const pfPlane *pln, const pfMatrix xform);

```
typedef struct
{
    pfVec3      normal;
    float       offset;
} pfPlane;
```

DESCRIPTION

A pfPlane represents an infinite 2D plane as a normal and a distance offset from the origin in the normal direction. A point on the plane satisfies the equation *normal* dot (x, y, z) = offset. pfPlane is a public struct whose data members *normal* and *offset* may be operated on directly.

pfMakePtsPlane sets *dst* to the plane which passes through the three points *pt1*, *pt2* and *pt3*.

pfMakeNormPtPlane sets *dst* to the plane which passes through the point *pt* with normal *norm*.

pfDisplacePlane moves the plane *dst* by a distance *d* in the direction of the plane normal.

pfClosestPtOnPlane returns in *dst* the closest point to *pt* which lies in the plane *pln*. The line segment

connecting *pt* and *dst* is perpendicular to *pln*.

pfHalfSpaceContainsPt returns **TRUE** or **FALSE** depending on whether the point *pt* is in the interior of the specified half-space. The half-space is defined with plane normal pointing to the exterior.

pfHalfSpaceContainsSphere, **pfHalfSpaceContainsBox** and **pfHalfSpaceContainsCyl** test whether the half space specified by pfPlane contains a non-empty portion of the volume specified by the second argument, a sphere, box or cylinder, respectively.

The return value from the these functions is the OR of one or more bit fields. The returned value may be:

PFIS_FALSE:
The intersection of the primitive and the half space is empty.

PFIS_MAYBE:
The intersection of the primitive and the half space might be non-empty.

PFIS_MAYBE | PFIS_TRUE:
The intersection of the primitive and the half space is definitely non-empty.

PFIS_MAYBE | PFIS_TRUE | PFIS_ALL_IN:
The primitive is non-empty and lies entirely inside the half space.

indicating indicate that the second argument is entirely outside, potentially partly inside, partially inside or entirely inside the half space specified by the pfPlane.

pfPlaneIsectSeg tests the line segment *seg* for intersection with the half space *pln*. The possible test results are:

PFIS_FALSE:
seg lies entirely in the exterior.

PFIS_MAYBE | PFIS_TRUE | PFIS_START_IN:
The starting point of *seg* lies in the interior.

PFIS_MAYBE | PFIS_TRUE | PFIS_END_IN:
The ending point of *seg* lies in the interior.

If *d* is non-NULL, on return it contains the position along the line segment (0 <= d <= seg->length) at which the intersection occurred.

pfHalfSpaceIsectSeg intersects the line segment *seg* with the half space *hs* and has return values the same as **pfPlaneIsectSeg** except that it also returns a non-zero value when both points are inside the half-space. In this case it returns:

PFIS_MAYBE | PFIS_TRUE | PFIS_ALL_IN | PFIS_START_IN | PFIS_END_IN:
Both end points of *seg* lie in the interior.

If *d1* and *d2* are non-NULL, on return from **pfHalfSpaceIsectSeg** they contain the starting and ending positions of the line segment (0 <= d1 <= d2 <= seg->length) intersected with the half space.

pfOrthoXformPlane sets *dst* to the plane as transformed by the orthogonal transformation *xform*; *dst = pln * xform*. If *xform* is not an orthogonal transformation the results are undefined.

NOTES

The bit fields returned by the contains functions are structured so that bitwise AND-ing the results of sequential tests can be used to compute composite results, e.g. testing exclusion against a number of half spaces.

SEE ALSO

pfBox, pfMatrix, pfSeg, pfSphere, pfVec3

NAME

pfNewPtope, pfGetPtopeClassType, pfGetPtopeNumFacets, pfPtopeFacet, pfGetPtopeFacet, pfRemovePtopeFacet, pfOrthoXformPtope, pfPtopeContainsPt, pfPtopeContainsSphere, pfPtopeContainsBox, pfPtopeContainsCyl, pfPtopeContainsPtope – Create, configure, transform, and intersect polytopes

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfPolytope* **pfNewPtope**(void *arena);

pfType* **pfGetPtopeClassType**(void);

int **pfGetPtopeNumFacets**(pfPolytope *ptope);

int **pfPtopeFacet**(pfPolytope *ptope, int i, const pfPlane *facet);

int **pfGetPtopeFacet**(pfPolytope *ptope, int i, pfPlane *facet);

int **pfRemovePtopeFacet**(pfPolytope *ptope, int i);

void **pfOrthoXformPtope**(pfPolytope *ptope, const pfPolytope *src, const pfMatrix mat);

int **pfPtopeContainsPt**(const pfPolytope *ptope, const pfVec3 pt);

int **pfPtopeContainsSphere**(const pfPolytope *ptope, const pfSphere *sphere);

int **pfPtopeContainsBox**(const pfPolytope *ptope, const pfBox *box);

int **pfPtopeContainsCyl**(const pfPolytope *ptope, const pfCylinder *cyl);

int **pfPtopeContainsPtope**(const pfPolytope *ptope, const pfPolytope *ptope1);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfPolytope** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfPolytope**. Casting an object of class **pfPolytope** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfPolytope** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);

```
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

PARAMETERS

ptope identifies a pfPolytope

DESCRIPTION

A pfPolytope is a set of half spaces whose intersection defines a convex, possibly semi-infinite, volume which may be used for culling and other intersection testing where a tighter bound than a pfBox, pfSphere, or pfCylinder is of benefit.

pfNewPtope creates and returns a handle to a pfPolytope. *arena* specifies a malloc arena out of which the pfPolytope is allocated or **NULL** for allocation off the process heap.

pfGetPtopeClassType returns the **pfType*** for the class **pfPolytope**. The **pfType*** returned by **pfGetPtopeClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfPolytope**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfPtopeFacet sets the *i*th facet of *ptope* to *facet*. *facet* defines a half space such that the normal of the pfPlane faces "outside". **pfGetPtopeFacet** copies the *i*th facet of *ptope* into *facet*.

pfRemovePtopeFacet removes the *i*th facet of *ptope* from the list. Remaining facets are shifted left over the removed facet.

pfGetPtopeNumFacets returns the number of facets in *ptope*.

pfOrthoXformPtope transforms *src* by *mat* and places the result in *ptope*. *mat* should be an orthonormal matrix or results are undefined.

The **pfPtopeContains<*>** routines compute the intersection of *ptope* with a variety of geometric primitives. **pfPtopeContains<*>** returns one of the following:

PFIS_FALSE:
The intersection of the primitive and the pfPolytope is empty.

PFIS_MAYBE:
The intersection of the primitive and the pfPolytope might be non-empty.

PFIS_MAYBE | PFIS_TRUE:
The intersection of the primitive and the pfPolytope is definitely non-empty.

PFIS_MAYBE | PFIS_TRUE | PFIS_ALL_IN:
The primitive is non-empty and lies entirely inside the pfPolytope.

SEE ALSO

pfBox, pfCylinder, pfDelete, pfFrustum, pfMatrix, pfObject, pfSphere

NAME

pfMakeRotQuat, pfGetQuatRot, pfLengthQuat, pfConjQuat, pfExpQuat, pfLogQuat, pfMultQuat, pfDivQuat, pfInvertQuat, pfSlerpQuat, pfSquadQuat, pfQuatMeanTangent – Set and operate on quaternions

FUNCTION SPECIFICATION

#include <Performer/pr.h>

#include <Performer/prmath.h>

void **pfMakeRotQuat**(pfQuat dst, float angle, float x, float y, float z);

void **pfGetQuatRot**(const pfQuat q, float *angle, float *x, float *y, float *z);

float **pfLengthQuat**(const pfQuat q);

void **pfConjQuat**(pfQuat dst, const pfQuat q);

void **pfExpQuat**(pfQuat dst, const pfQuat q);

void **pfLogQuat**(pfQuat dst, const pfQuat q);

void **pfMultQuat**(pfQuat dst, const pfQuat q1, const pfQuat q2);

void **pfDivQuat**(pfQuat dst, const pfQuat q1, const pfQuat q2);

void **pfInvertQuat**(pfQuat dst, const pfQuat q);

void **pfSlerpQuat**(pfQuat dst, float t, const pfQuat q1, const pfQuat q2);

void **pfSquadQuat**(pfQuat dst, float t, const pfQuat q1, const pfQuat q2, const pfQuat a, const pfQuat b);

extern void **pfQuatMeanTangent**(pfQuat dst, const pfQuat q1, const pfQuat q2, const pfQuat q3);

```
typedef pfVec4 pfQuat;
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfQuat** is derived from the parent class **pfVec4**, so each of these member functions of class **pfVec4** are also directly usable with objects of class **pfQuat**. Casting an object of class **pfQuat** to an object of class **pfVec4** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfVec4**.

void **pfAddScaledVec4**(pfVec3 dst, const pfVec3 v1, float s, const pfVec3 v2);
void **pfAddVec4**(pfVec4 dst, const pfVec4 v1, const pfVec4 v2);
int **pfAlmostEqualVec4**(pfVec4 v1, const pfVec4 v2, float tol);
void **pfCombineVec4**(pfVec4 dst, float s1, const pfVec4 v1, float s2, const pfVec4 v2);

void **pfCopyVec4**(pfVec4 dst, const pfVec4 v);
float **pfDistancePt4**(const pfVec4 pt1, const pfVec4 pt2);
float **pfDotVec4**(const pfVec4 v1, const pfVec4 v2);
int **pfEqualVec4**(const pfVec4 v1, const pfVec4 v2);
float **pfLengthVec4**(const pfVec4 v);
void **pfNegateVec4**(pfVec4 dst, const pfVec4 v);
float **pfNormalizeVec4**(pfVec4 v);
void **pfScaleVec4**(pfVec4 dst, float s, const pfVec4 v);
void **pfSetVec4**(pfVec4 dst, float x, float y, float z, float w);
float **pfSqrDistancePt4**(const pfVec4 pt1, const pfVec4 pt2);
void **pfSubVec4**(pfVec4 dst, const pfVec4 v1, const pfVec4 v2);
void **pfXformVec4**(pfVec4 dst, const pfVec4 v, const pfMatrix m);

DESCRIPTION

pfQuat represents a quaternion as the four floating point values (*x*, *y*, *z*, *w*) of a pfVec4.

pfMakeRotQuat converts an *axis* and *angle* rotation representation to a quaternion. **pfGetQuatRot** is the inverse operation. It produces the axis (as a unit length direction vector) and angle equivalent to the given quaternion. Also see **pfMakeQuatMat** and **pfGetOrthoMatQuat**.

Several monadic quaternion operators are provided. **pfConjQuat** produces the complex conjugate *dst* of *q* by negating only the complex components (*x*, *y*, and *z*) which results in an inverse rotation. **pfExpQuat** and **pfLogQuat** perform complex exponentiation and logarithm functions respectively. The length of a quaternion is computed by **pfLengthQuat** and is defined as the norm of all four quaternion components. Macro equivalents are **PFCONJ_QUAT** and **PFLENGTH_QUAT**. For negation, use the pfVec4 routine, **pfNegateVec4**.

pfMultQuat and **pfDivQuat** are dyadic quaternion operations which provide the product, and quotient of two quaternions. When quaternions are used to represent rotations, multiplication of two quaternions is equivalent, but more efficient, than the multiplication of the two correspondinging rotation matrices.

pfInvertQuat computes the multiplicative inverse of a quaternion. These operations are the basis from which the other quaternion capabilities have been derived. Macro equivalents are **PFMULT_QUAT**, **PFDIV_QUAT**, and **PFINVERT_QUAT**. For addition and scalar multiplication, use the pfVec4 routines **pfAddVec4**, **pfSubVec4**, and **pfScaleVec4**. Comparisons can be made with **pfEqualVec4** and **pfAlmostEqualVec4**.

Interpolation of quaternions (as presented by Ken Shoemake) is an effective technique for rotation interpolation. Spherical linear interpolation is performed with **pfSlerpQuat**, which produces a rotation *dst* that is *t* of the way between *q1* and *q2*.

Spherical quadratic interpolation is provided by **pfSquadQuat** and its helper function,

pfQuatMeanTangent.

NOTES

These functions use a pfVec4 to represent quaternions and store the imaginary part first, thus the array contents q = {x,y,z,w} are a representation of the quaternion w + x*i* + y*j*+ z*k*.

Because both *q* and *-q* represent the same rotation (quaternions have a rotation range of [-360,360] degrees) conversions such as **pfGetOrthoMatQuat** make an arbitrary choice of the sign of the returned quaternion. To prevent the arbitrary sign from introducing large, unintended rotations, **pfSlerpQuat** checks the angle *theta* between *q1* and *q2*. If *theta* exceeds 180 degrees, *q2* is negated changing the interpolations range from [0,*theta*] to [0, *theta*-360 degrees].

When using overloaded operators in C++, assignment operators, e.g. "+=", are somewhat more efficient than the corresponding binary operators, e.g. "+", because the latter construct a temporary intermediate object. Use assignment operators or macros for binary operations where optimal speed is important.

C++ does not support array deletion (i.e. **delete[]**) for arrays of objects allocated new operators that take additional arguments. Hence, the array deletion operator **delete[]** should not be used on arrays of objects created with **new**(arena) pfVec4[n].

For more information on quaternions, see the article by Sir William Rowan Hamilton "*On quaternions; or on a new system of imaginaries in algebra*," in the Philosophical Magazine, xxv, pp. 10-13 (July 1844). More recent references include "*Animating Rotation with Quaternion Curves*," SIGGRAPH Proceedings Vol 19, Number 3, 1985, and "*Quaternion Calculus For Animation*," in "Math for SIGGRAPH", Course Notes, #23, SIGGRAPH 1989, both by Ken Shoemake. An introductory tutorial is available on the Internet at ftp://ftp.cis.upenn.edu/pub/graphics/shoemake/quatut.ps.Z. Note that for consistency with Performer's transformation order, pfQuats are the conjugates of the quaternions described in these references.

SEE ALSO

pfVec4, pfMatrix

NAME

pfQuerySys, pfMQuerySys – Routines for querying parameters of system configuration

FUNCTION SPECIFICATION

#include <Performer/pr.h>

int **pfQuerySys**(int which, int *dst);

int **pfMQuerySys**(int *which, int *dst);

DESCRIPTION

Graphics platforms provide a range of hardware configurations whose parameters may affect the usability of graphics features. These functions provide the ability to query these parameters. IRIS Performer makes use of the following useful routines in determining its information: **getgdesc(3g)**, **XGetVisualInfo(3X11)**, and **glXGetConfig(3g)**.

pfQuerySys takes a **PFQSYS_** token and returns in *dst* the value for the requested parameter. The return value is the number of bytes successfully written.

The **pfQuerySys** query token must be one of:

PFQSYS_GL
returns the GL type currently being used: **PFGL_IRISGL** or **PFGL_OPENGL**.

PFQSYS_NUM_CPUS
returns the number of CPUs on the system.

PFQSYS_NUM_CPUS_AVAILABLE
returns the number of available (non-isolated) CPUs on the system.

PFQSYS_NUM_SCREENS
returns the number of screens, or hardware graphics pipelines.

PFQSYS_SIZE_PIX_X
returns the width of the default screen of the current display in pixels. The current display is that returned by **pfGetCurWSConnection**.

PFQSYS_SIZE_PIX_Y
returns the height of the default screen of the current display in pixels. The current display is that returned by **pfGetCurWSConnection**.

PFQSYS_MAX_SNG_RGBA_BITS
returns the maximum number of configurable single-buffered bits per RGBA color component.

PFQSYS_MAX_DBL_RGBA_BITS
returns the maximum number of configurable double-buffered bits per RGBA color component.

PFQSYS_MAX_SNG_CI_BITS
returns the maximum number of configurable single-buffered bits for colorindex.

PFQSYS_MAX_DBL_CI_BITS
returns the maximum number of configurable double-buffered bits for colorindex.

PFQSYS_MAX_SNG_OVERLAY_CI_BITS
returns the maximum number of configurable bits for colorindex single-buffered overlay buffers.

PFQSYS_MAX_DBL_OVERLAY_CI_BITS
returns the maximum number of configurable bits for colorindex double-buffered overlay buffers.

PFQSYS_MAX_DEPTH_BITS
returns the maximum number of configurable bits for depth buffers.

PFQSYS_MIN_DEPTH_VAL
returns the minimum representable value in the depth buffer. This is IRIS GL only: OpenGL depth buffers range from 0 to 1.

PFQSYS_MAX_DEPTH_VAL
returns the maximum representable value in the depth buffer. This is IRIS GL only: OpenGL depth buffers range from 0 to 1.

PFQSYS_MAX_STENCIL_BITS
returns the maximum number of configurable bits for stencil buffers.

PFQSYS_MAX_MS_SAMPLES
returns the maximum number of configurable subsamples for multisample buffers. Multisample buffers are used for antialiasing. See the **pfWindow** and **pfAntialias** reference pages for more information.

PFQSYS_MAX_MS_DEPTH_BITS
returns the maximum number of configurable bits for multisampled depth buffers.

PFQSYS_MAX_MS_STENCIL_BITS
returns the maximum number of configurable bits for multisampled stencil buffers.

PFQSYS_MAX_LIGHTS
returns the maximum allowable number of GL lights that may be on at one time.

PFQSYS_TEXTURE_MEMORY_BYTES
returns the number of bytes of hardware texture memory.

PFQSYS_MAX_TEXTURE_SIZE
returns the number of bytes in the largest single texture that can can be allocated in hardware texture memory.

pfMQuerySys takes an NULL-terminated array of query tokens and a destination buffer and will do multiple queries. The return value will be the number of bytes successfully written. This routine is more

efficient than **pfQuerySys** if multiple queries are desired.

NOTES

pfQueryWin can be used to query the configuration parameters of a given pfWindow. **pfFeature** can be used to query the availability of specific features on the current hardware configuration.

SEE ALSO

pfFeature, pfGetCurWSConnection, pfWindow, pfAntialias, XGetVisualInfo, glXGetConfig

NAME

pfClipSeg, pfMakePtsSeg, pfMakePolarSeg, pfClosestPtsOnSeg, pfTriIsectSeg – Set and operate on line segments

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfClipSeg**(pfSeg *dst, const pfSeg *seg, float d1, float d2);

void **pfMakePtsSeg**(pfSeg *dst, const pfVec3 p1, const pfVec3 p2);

void **pfMakePolarSeg**(pfSeg *dst, const pfVec3 pos, float azi, float elev, float len);

int **pfClosestPtsOnSeg**(const pfSeg *seg1, const pfSeg *seg2, pfVec3 ptOn1, pfVec3 ptOn2);

int **pfTriIsectSeg**(const pfVec3 v1, const pfVec3 v2, const pfVec3 v3, const pfSeg *seg, float *d);

```
typedef struct
{
    pfVec3      pos;
    pfVec3      dir;
    float       length;
} pfSeg;
```

DESCRIPTION

A **pfSeg** represents a line segment starting at *pos*, extending for a length *length* in the direction *dir*. The routines assume that *dir* is of unit length, otherwise the results are undefined. pfSeg is a public struct whose data members *pos*, *dir* and *length* may be operated on directly.

pfClipSeg is used to select a subset of the input segment *seg*. It sets *dst* to the portion of segment *seg* clipped to start at distance *d1* and end at *d2*. When *d1* = 0 and *d2* = *seg*->length, **pfClipSeg** returns the original segment. Values of *d1* < 0 and *d2* > *seg*->length can be used to extend the segment.

pfMakePtsSeg sets *dst* to the segment which starts at the point *p1* and ends at the point *p2*.

pfMakePolarSeg sets *dst* to the segment which starts at *pos* and has length *length* and points in the direction specified by *azi* and *elev*. *azi* specifies the azimuth (or heading), which is the angle which the projection of the segment in the X-Y plane makes with the +Y axis. *elev* specifies the elevation (or pitch), the angle with respect to the X-Y plane. The positive Y axis is *azi*=0 and *elev*=0. Azimuth follows the right hand rule about the Z axis, i.e. +90 degrees is the -X axis. Similarly, elevation follows the right hand rule about the X axis, i.e. +90 degrees is the +Z axis.

pfClosestPtsOnSeg returns the two closest points on the two segments *seg1* and *seg2*. If the two segments are parallel FALSE is returned and the contents of ptOn1 and ptOn2 are undefined.

pfTriIsectSeg tests the line segment *seg* for intersection with the triangle defined by the three vertices *v1*, *v2*, and *v3*. **pfTriIsectSeg** returns TRUE or FALSE. If *d* is non-null, on return it contains the length position of the intersection between 0 and seg->length.

SEE ALSO

pfNodeIsectSegs, pfGSetIsectSegs, pfVec3

NAME

pfShadeModel, pfGetShadeModel – Set and get the shading model

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfShadeModel**(int model);

int **pfGetShadeModel**(void);

PARAMETERS

model is a symbolic constant and is one of:

PFSM_FLAT	Use flat shading,
PFSM_GOURAUD	Use Gouraud shading.

DESCRIPTION

pfShadeModel sets the shading model to *model*. When flat shading is enabled, the last vertex in a geometric primitive defines the color of the entire geometric primitive. When Gouraud shading, vertex colors are interpolated across the primitive.

The following example shows how data equivalent to OpenGL immediate mode graphics commands would be interpreted in both **PFSM_FLAT** and **PFSM_GOURAUD** shade models.

Example 1:

```
/*
 * Draw a three-primitive triangle strip in OpenGL
 */
glColor3f(0, 0, 0);        /*************************************/
glBegin(GL__TRIANGLE_STRIP);/*   Actual Rendered Triangle Colors    */
glVertex3v(v0);            /* ----------------------------------- */
glVertex3v(v1);            /* Tri  PFSM_FLAT       PFSM_GOURAUD    */
glColor3f(1, 0, 0);        /* ----------------------------------- */
glVertex3v(v2);            /*  0   red             black/black/red */
glColor3f(0, 1, 0);        /*                                      */
glVertex3v(v3);            /*  1   green           black/red/green */
glColor3f(0, 0, 1);        /*                                      */
glVertex3v(v4);            /*  2   blue            red/green/blue  */
glEnd();                   /*************************************/
```

Consequently, strips (triangle **PFGS_TRISTRIPS** or line **PFGS_LINESTRIPS**) which are composed of different colored primitives *must* have flat shading enabled in order to be rendered properly. The pfGeoSet primitive types of **PFGS_FLAT_TRISTRIPS** and **PFGS_FLAT_LINESTRIPS** ensure that flat shading will be enabled when the pfGeoSet is drawn.

Another subtlety of the shading model is related to the current lighting model. If the lighting model is local due to either the pfLightModel (**pfLModelLocal**) or if any pfLights are local (**pfLightPos**), then Gouraud shading must be enabled since lighting effects should be different at each vertex. This means that even if a triangle has a constant color and normal, it should still be drawn with Gouraud shading so the effects of the local lighting can be seen. The exception to this rule are the flat strips discussed above.

The shading model state element is identified by the **PFSTATE_SHADEMODEL** token. Use this token with **pfGStateMode** to set the shading model of a pfGeoState and with **pfOverride** to override subsequent shading model changes.

Example 2:

```
/* Set up flat shaded pfGeoState. */
pfGStateMode(gstate, PFSTATE_SHADEMODEL, PFSM_FLAT);

/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/* Draw flat shaded gset */
pfDrawGSet(gset);
```

Example 3:

```
/* Alternative way to draw flat-shaded pfGeoSet */
pfGSetDrawMode(gset, PFGS_FLATSHADE, PF_ON);
pfDrawGSet(gset);
```

Example 4:

```
/*
 * Draw flat-shaded triangle strip pfGeoSet.  PFGS_FLATSHADE
 * and pfShadeModel are not required.
 */
pfGSetPrimType(gset, PFGS_FLAT_TRISTRIPS);
pfDrawGSet(gset);
```

Example 5:

```
pfShadeModel(PFSM_FLAT);

/* Override shading model to PFSM_FLAT */
pfOverride(PFSTATE_SHADEMODEL, PF_ON);
```

pfShadeModel is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfShadeModel** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

The selection of which shading model a pfGeoSet uses is based upon the following decision hierarchy:

1. Use flat shading if pfGeoSet is **PFGS_FLAT_TRISTRIPS** or **PFGS_FLAT_LINESTRIPS** or if **PFGS_FLATSHADE** is enabled through **pfGSetDrawMode**.
2. Use the shading model set by the attached pfGeoState, if any (see **pfGSetGState**).
3. Use the shading model set by **pfShadeModel**.

The default shading model is Gouraud.

pfGetShadeModel returns the current shading model.

NOTES

Overriding the shading model to **PFSM_FLAT** can be a useful debugging aid since it reveals the facets of a normally smooth surface.

SEE ALSO

pfGSetGState, pfGeoSet, pfGeoState, pfLModelLocal, pfLightPos, pfState

NAME

pfInitArenas, pfFreeArenas, pfGetSharedArena, pfGetSemaArena, pfSharedArenaSize, pfGetSharedArenaSize, pfSharedArenaBase, pfGetSharedArenaBase, pfSemaArenaSize, pfGetSemaArenaSize, pfSemaArenaBase, pfGetSemaArenaBase, pfTmpDir, pfGetTmpDir – Shared Memory Functions

FUNCTION SPECIFICATION

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

int           pfInitArenas(void);
int           pfFreeArenas(void);
void*         pfGetSharedArena(void);
void*         pfGetSemaArena(void);
void          pfSharedArenaSize(size_t size);
size_t        pfGetSharedArenaSize(void);
void          pfSharedArenaBase(void *base);
void *        pfGetSharedArenaBase(void);
void          pfSemaArenaSize(size_t size);
size_t        pfGetSemaArenaSize(void);
void          pfSemaArenaBase(void *base);
void *        pfGetSemaArenaBase(void);
void          pfTmpDir(char *dirname);
const char *  pfGetTmpDir(void);
```

DESCRIPTION

pfInitArenas creates arenas that can be used to allocate shared memory, locks and semaphores from (see **pfMalloc**, **usnewlock**, **usnewsema**). In a libpf application, this function is called by **pfInit** so it is not necessary to call it directly. However, a libpr application that wishes to use Performer's arenas, must call **pfInitArenas** before calling **pfInit** to ensure that the type system is created in shared memory.

In addition to creating a shared memory arena, **pfInitArenas** uses **usinit** to create an arena for locks and semaphores.

pfTmpDir and the environment variable **PFTMPDIR** control where **pfInitArenas** creates its arenas. If neither is specified, the semaphore arena is created in **/usr/tmp** and the shared memory arena is created in /dev/zero (swap space). If a temporary directory is specified, then files are created in that directory for the shared memory and semaphore arenas. Both these files are unlinked so that they are removed from the file system when the process terminates. Once created, these arenas cannot grow beyond the specified size, so IRIS Performer tries to create a large (256MB by default) shared memory arena and a 128KB

semaphore arena. The semaphore arena always uses a memory mapped file.

For the shared memory arena, the default option uses swap space. This is usually preferable to using a memory mapped file in a directory specified by **PFTMPDIR**. Using an actual file is slower at allocation time and requires actual disk space for extending the file length to equal the amount of memory allocated (**pfMalloc**) from the arena. Because the temporary file is unlinked after creation, any memory actually allocated will show up as used under df(1), but not under du(1). The application size in ps(1) will reflect the maximum specified size of the arena, e.g. 256MB. But space is not reserved until accessed, i.e. until required by **pfMalloc**. So the large arena created by **pfInitArenas** does not consume any substantial disk or swap space resources until needed.

pfSharedArenaSize can be used to override the arena size that IRIS Performer uses by default (256MB). *size* specifies the desired size in bytes. Arena size is limited by the largest contiguous possible memory mapping, currently slightly more than 1.7GB in an application linked with DSOs. When attempting large arena mappings, first make sure that the real and virtual memory usage limits set in the shell or with setrlimit() are adequate. **pfSharedArenaSize** must be called before **pfInitArenas** to have effect. **pfGetSharedArenaSize** returns the arena size in bytes.

The comparable calls for the semaphore arena are **pfSemaArenaSize** and **pfGetSemaArenaSize**.

pfSharedArenaBase sets the base address for the mapping of the shared memory arena. Normally, IRIX chooses these base addresses automatically. Direct specification is only useful if the application needs closer control over the layout of virtual address space, e.g. to avoid conflicts with other mappings. **pfGetSharedArenaBase** returns the base address for the arena.

The comparable calls for the semaphore arena are **pfSemaArenaBase** and **pfGetSemaArenaBase**.

pfGetSemaArena returns a handle to the lock arena and **pfGetSharedArena** returns a pointer to the shared memory arena. This pointer cannot be used directly, only as an argument to **pfMalloc**. **pfGetTmpDir** returns the temporary directory set using **pfTmpDir**.

NOTES

These arenas can only be used by related processes. Related in this context means processes that are created by **fork** or **sproc** once **pfInitArenas** has been called. Use **pfDataPool** for sharing memory between unrelated processes. **pfInitArenas** should be called before any **fork** calls are made.

SEE ALSO

acreate, pfFree, pfMalloc, usinit, usnewlock, usnewsema

NAME

pfSinCos, pfTan, pfArcTan2, pfArcSin, pfArcCos, pfSqrt – Fast math routines sin, cos, sqrt

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfSinCos**(float arg, float* sin, float* cos);

float **pfTan**(float arg);

float **pfArcTan2**(float y, float x);

float **pfArcSin**(float arg);

float **pfArcCos**(float arg);

float **pfSqrt**(float arg);

DESCRIPTION

pfSinCos returns sine and cosine of *arg* degrees in the locations pointed to by *sin* and *cos*. The routine is less accurate than that provided in libm, but faster. It also loses precision earlier for large arguments. For accuracy within an error tolerance of 1/1000, the argument must be in the range -7500 to +7500 degrees.

pfTan returns the tangent of *arg* degrees.

pfArcTan2 returns the arc tangent of y/x in the range -180 to +180 degrees using the signs of both arguments to determine the quadrant of the return value.

pfArcSin returns the arc sine of *arg* in the range -90 to +90 degrees.

pfArcCos returns the arc sine of *arg* in the range 0 to 180 degrees.

pfSqrt returns the square root of *arg*. It is faster, but somewhat less accurate than the version in the standard fast math library (libfastm).

SEE ALSO

pfMakeCoordMat, pfNormalizeVec3

NAME

pfMakeEmptySphere, pfSphereExtendByPt, pfSphereExtendBySphere, pfSphereExtendByCyl, pfSphereAroundPts, pfSphereAroundSpheres, pfSphereAroundCyls, pfSphereAroundBoxes, pfSphereContainsPt, pfSphereContainsSphere, pfSphereContainsCyl, pfSphereIsectSeg, pfOrthoXformSphere – Set, transform and extend a sphere

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfMakeEmptySphere**(pfSphere *sph);

void **pfSphereExtendByPt**(pfSphere* dst, const pfVec3 pt);

void **pfSphereExtendBySphere**(pfSphere* dst, const pfSphere* sph);

void **pfSphereExtendByCyl**(pfSphere* dst, const pfCylinder* cyl);

void **pfSphereAroundPts**(pfSphere *dst, const pfVec3 *pts, int npt);

void **pfSphereAroundSpheres**(pfSphere *dst, const pfSphere **sphs, int nsph);

void **pfSphereAroundCyls**(pfSphere *dst, const pfCylinders **sphs, int ncyl);

void **pfSphereAroundBoxes**(pfSphere *dst, const pfBox **boxes, int nbox);

int **pfSphereContainsPt**(const pfSphere* sp, const pfVec3 pt);

int **pfSphereContainsSphere**(const pfSphere *sph1, const pfSphere *sph2);

int **pfSphereContainsCyl**(const pfSphere *sph, const pfCylinder *cyl);

int **pfSphereIsectSeg**(const pfSphere* sph, const pfSeg* seg, float* d1, float* d2);

void **pfOrthoXformSphere**(pfSphere *dst, const pfSphere *sph, const pfMatrix xform);

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

DESCRIPTION

A pfSphere represents a sphere as a *center* and a *radius*. The routines listed here provide means of creating and extending spheres for use as bounding geometry. pfSphere is a public struct whose data members *center* and *radius* may be operated on directly.

pfMakeEmptySphere sets *sph* so that it appears empty to extend and around operations.

pfSphereExtendByPt, **pfSphereExtendBySphere**, and **pfSphereExtendByCyl** set *dst* to a sphere which contains both *dst* and the point *pt*, the sphere *sph* or the cylinder *cyl*, respectively.

pfSphereAroundPts, **pfSphereAroundBoxes**, **pfSphereAroundSpheres** and **pfSphereAroundCyls** set *dst* to a sphere which contains a set of points, boxes, spheres or cylinders, respectively. These routines are passed the address of an array of pointers to the objects being bounded along with the number of objects.

pfSphereContainsPt, returns **TRUE** or **FALSE** depending on whether the point *pt* is in the interior of the specified sphere.

pfSphereContainsSphere and **pfSphereContainsCyl** test whether the sphere specified by the first argument contains a non-empty portion of the volume specified by the secondsecond argument and the sphere is empty.

PFIS_MAYBE:
The intersection of the second argument and the sphere might be non-empty.

PFIS_MAYBE | PFIS_TRUE:
The intersection of the second argument and the sphere is definitely non-empty.

PFIS_MAYBE | PFIS_TRUE | PFIS_ALL_IN:
The second argument is non-empty and lies entirely inside the sphere.

pfSphereIsectSeg intersects the line segment *seg* with the volume of the pfSphere *sphere*. The possible return values are:

PFIS_FALSE:
seg lies entirely in the exterior.

PFIS_MAYBE | PFIS_TRUE | PFIS_START_IN:
The starting point of *seg* lies in the interior.

PFIS_MAYBE | PFIS_TRUE | PFIS_END_IN:
The ending point of *seg* lies in the interior.

PFIS_MAYBE | PFIS_TRUE | PFIS_ALL_IN | PFIS_START_IN | PFIS_END_IN:
Both end points of *seg* lie in the interior.

If *d1* and *d2* are non-NULL, on return from **pfSphereIsectSeg** they contain the starting and ending positions of the line segment (0 <= d1 <= d2 <= seg->length) intersected with the sphere.

pfOrthoXformSphere returns in *dst* the sphere *sph* transformed by the orthogonal transform *xform*.

NOTES

The bit fields returned by the contains functions are structured so that bitwise AND-ing the results of sequential tests can be used to compute composite results, e.g. testing exclusion against a number of half spaces.

Some of the extend and around operations are time consuming and should be used sparingly. In general, the quality of a bound generated by a series of extend operations will be no better, and sometimes much worse, than a bound generated by a single around operation.

SEE ALSO

pfBox, pfCylinder, pfSeg, pfVec3

NAME

pfNewSprite, pfGetSpriteClassType, pfSpriteMode, pfGetSpriteMode, pfSpriteAxis, pfGetSpriteAxis, pfBeginSprite, pfEndSprite, pfPositionSprite, pfGetCurSprite – Create and update sprite transformation primitive.

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfSprite * **pfNewSprite**(void *arena);

pfType * **pfGetSpriteClassType**(void);

void **pfSpriteMode**(pfSprite *sprite, int which, int val);

int **pfGetSpriteMode**(const pfSprite *sprite, int which);

void **pfSpriteAxis**(pfSprite *sprite, float x, float y, float z);

void **pfGetSpriteAxis**(pfSprite *sprite, float *x, float *y, float *z);

void **pfBeginSprite**(pfSprite *sprite);

void **pfEndSprite**(void);

void **pfPositionSprite**(float x, float y, float z);

pfSprite * **pfGetCurSprite**(void);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfSprite** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfSprite**. Casting an object of class **pfSprite** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfSprite** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);

```
int         pfUnrefDelete(void *ptr);
int         pfGetRef(const void *ptr);
int         pfCopy(void *dst, void *src);
int         pfDelete(void *ptr);
int         pfCompare(const void *ptr1, const void *ptr2);
void        pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *      pfGetArena(void *ptr);
```

PARAMETERS

sprite identifies a pfSprite.

DESCRIPTION

A "sprite" is a term borrowed from the video game industry and refers to a movable graphical object that always appears orthogonal to the viewer. In 2D, a sprite can be implemented by simply drawing an image that is aligned with the screen, a technique called "bit-blitting". pfSprite extends this support to 3D geometry by rotating the geometry appropriately based on the viewer and object locations to achieve a consistent screen alignment. In this respect, pfSprite is not really a graphical object itself, rather it is an intelligent transformation and is logically grouped with other libpr transformation primitives like **pfMultMatrix**.

Sprite transformations (subsequently referred to as sprites) are useful for complex objects that are roughly symmetrical about an axis or a point. By rotating the model about the axis or point, the viewer only sees the "front" of the model so complexity is saved in the model by omitting the "back" geometry. A further performance enhancement is to incorporate visual complexity in a texture map rather than in geometry. Thus, on machines with fast texture mapping, sprites can present very complex images with very little geometry. Classic examples of textured sprites use a single quadrilateral that when rotated about a vertical axis simulate trees and when rotated about a point simulate clouds or puffs of smoke.

pfNewSprite creates and returns a handle to a pfSprite. *arena* specifies a malloc arena out of which the pfSprite is allocated or **NULL** for allocation off the process heap. pfSprites can be deleted with **pfDelete**.

pfGetSpriteClassType returns the **pfType*** for the class **pfSprite**. The **pfType*** returned by **pfGetSpriteClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfSprite**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

Sprite transformations are simply rotations which are based on:

1. The viewer location.
2. The sprite location.

3. The sprite mode/axis.

The viewer's coordinate system is specified with **pfViewMat** whose last row is the viewer's location. The sprite location in object coordinates is specified with **pfPositionSprite**. This location is transformed by the current modeling matrix (**pfModelMat**) into "world" coordinates before computing the rotation based on the sprite mode.

pfSpriteMode sets the *which* mode of *sprite* to *val*. *which* identifies a particular mode and is one of:

PFSPRITE_ROT *val* specifies the rotation constraints of *sprite* and is one of:

PFSPRITE_AXIAL_ROT
sprite rotation is constrained to rotate about the sprite axis defined by **pfSpriteAxis**.

PFSPRITE_POINT_ROT_EYE
sprite rotation is constrained to rotate about the sprite location specified by **pfPositionSprite**. The sprite axis is ignored but the rotation is constrained so that the +Z object coordinate axis maps to the +Y window coordinate axis, i.e., the object +Z axis stays upright on the screen.

PFSPRITE_POINT_ROT_WORLD
sprite rotation is constrained to rotate about the sprite location specified by **pfPositionSprite**. The rotation is further constrained so that the +Z object coordinate axis maps to the sprite axis.

The default **PFSPRITE_ROT** mode is **PFSPRITE_AXIAL_ROT**.

PFSPRITE_MATRIX_THRESHOLD
pfGeoSets which contain a number of vertices less than *val* will be transformed on the CPU, rather than through the Graphics Library transformation stack. If *sprite* is to affect non-pfGeoSet geometry, then *val* should be <= 0 in which case the Graphics Library transformation stack will always be used. Specifically, **pfBeginSprite** will push the matrix stack, **pfPositionSprite** will modify the top of stack with the current sprite rotation, and **pfEndSprite** will pop the matrix stack. It is not necessary to push/pop the matrix stack within **pfBeginSprite**/**pfEndSprite**. The default threshold value is 10.

pfGetSpriteMode returns the value of the mode identified by *which*.

Sprite rotations are based on the object coordinate system as follows:

1. The -Y object coordinate axis of geometry is rotated to point to the viewer.
2. The +Z object coordinate axis of geometry is the axis of rotation for axial sprites, i.e. the +Z object axis is rotated onto the sprite axis, then the transformed -Y axis is rotated about the sprite axis to face the viewer.

pfSpriteAxis sets *sprite*'s axis to (*x*, *y*, *z*). **pfGetSpriteAxis** returns the axis of *sprite* in *x*, *y*, *z*.

pfBeginSprite makes *sprite* the current pfSprite and applies its effects to subsequently drawn pfGeoSets and non-pfGeoSet geometry if *sprite*'s matrix threshold value is <= 0. **pfPositionSprite** specifies the sprite origin and **pfEndSprite** sets the current pfSprite to **NULL** and exits "sprite mode". **pfPositionSprite** may be called outside **pfBeginSprite/pfEndSprite** and any number of times within **pfBeginSprite/pfEndSprite** to render geometry with many different origins but which share the characteristics of *sprite*.

Example 1: Draw trees as axial sprites which rotate about +Z.

```
pfVec3         treeOrg[NUMTREES];
pfGeoSet       *trees[NUMTREES];

sprite = pfNewSprite(arena);
pfBeginSprite(sprite);

for (i=0; i<NUMTREES; i++)
{
    pfPositionSprite(treeOrg[i][0], treeOrg[i][1], treeOrg[i][1]);
    pfDrawGSet(trees[i]);
}

pfEndSprite();
```

pfBeginSprite, **pfEndSprite**, and **pfPositionSprite** are all display-listable commands. If a pfDispList has been opened by **pfOpenDList**, these commands will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

NOTES

Both **PFSPRITE_AXIAL_ROT** and **PFSPRITE_POINT_ROT_WORLD** sprites may "spin" about the Y axis of the pfGeoSet when viewed along the rotation or alignment axis.

SEE ALSO

pfDelete, pfDispList, pfGeoSet, pfModelMat, pfState, pfViewMat

NAME

pfNewState, pfGetStateClassType, pfSelectState, pfLoadState, pfAttachState, pfInitState, pfGetCurState, pfPushState, pfPopState, pfGetState, pfFlushState, pfBasicState – Create, modify and query graphics state

FUNCTION SPECIFICATION

```
#include <ulocks.h>
#include <Performer/pr.h>
pfState *   pfNewState(void);
pfType *    pfGetStateClassType(void);
void        pfSelectState(pfState *state);
void        pfLoadState(pfState *state);
void        pfAttachState(pfState *state, pfState *state1);
void        pfInitState(usptr_t* arena);
pfState *   pfGetCurState(void);
void        pfPushState(void);
void        pfPopState(void);
void        pfGetState(pfGeoState *gstate);
void        pfFlushState(void);
void        pfBasicState(void);
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfState** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfState**. Casting an object of class **pfState** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

```
void    pfUserData(pfObject *obj, void *data);
void*   pfGetUserData(pfObject *obj);
int     pfGetGLHandle(pfObject *obj);
```

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfState** can also be used with these functions designed for objects of class **pfMemory**.

```
pfType *    pfGetType(const void *ptr);
```

```
int           pfIsOfType(const void *ptr, pfType *type);
int           pfIsExactType(const void *ptr, pfType *type);
const char *  pfGetTypeName(const void *ptr);
int           pfRef(void *ptr);
int           pfUnref(void *ptr);
int           pfUnrefDelete(void *ptr);
int           pfGetRef(const void *ptr);
int           pfCopy(void *dst, void *src);
int           pfDelete(void *ptr);
int           pfCompare(const void *ptr1, const void *ptr2);
void          pfPrint(const void *ptr, uint which, uint verbose, FILE *file);
void *        pfGetArena(void *ptr);
```

DESCRIPTION

IRIS Performer manages a subset of the graphics library state for convenience and improved performance. Further, this state is conceptually partitioned into two divisions: modes and attributes. A mode is a simple "setting" usually represented by a single value while an attribute is a larger collection of related modes that is encapsulated in an IRIS Performer structure, such as **pfFog**.

Modes usually have two routines to set and get them while attributes have many routines for accessing their parameters and a **pfApply<*>** routine which "applies" the attribute's characteristics to the graphics system via graphics library state commands. Modes are represented by basic data types like 'int' and 'float' while attributes are pointers to opaque IRIS Performer structures whose contents are accessible only through function calls.

An example of a mode is the shading model set by **pfShadeModel** and an attribute is exemplified by a pfMaterial which is applied with **pfApplyMtl**. Each mode and attribute is identified by a **PFSTATE_** token. These tokens are used in **pfGStateMode** and **pfGStateAttr** when initializing a pfGeoState and in **pfOverride** to override mode and attribute settings.

The following table lists the state components that are modes.

Mode	PFSTATE_ Token	Routine(s)	Default
Transparency	TRANSPARENCY	pfTransparency	PFTR_OFF
Antialiasing	ANTIALIAS	pfAntialias	PFAA_OFF
Decal	DECAL	pfDecal	PFDECAL_OFF
Face culling	CULLFACE	pfCullFace	PFCF_OFF
Alpha function	ALPHAFUNC	pfAlphaFunc	PFAF_ALWAYS
Alpha reference	ALPHAREF	pfAlphaFunc	0
Lighting enable	ENLIGHTING	pfEnable/pfDisable	PF_OFF
Texturing enable	ENTEXTURE	pfEnable/pfDisable	PF_OFF
Fogging enable	ENFOG	pfEnable/pfDisable	PF_OFF
Wireframe enable	ENWIREFRAME	pfEnable/pfDisable	PF_OFF
Colortable enable	ENCOLORTABLE	pfEnable/pfDisable	PF_OFF
Highlighting enable	ENHIGHLIGHTING	pfEnable/pfDisable	PF_OFF
Light Point enable	ENLPOINTSTATE	pfEnable/pfDisable	PF_OFF
TexGen enable	ENTEXGEN	pfEnable/pfDisable	PF_OFF

The following table lists the state components that are attributes.

Attribute	PFSTATE_ Token	Routine	Default
pfLightModel	LIGHTMODEL	pfApplyLModel	NULL
pfLights	LIGHTS	pfLightOn	all NULL
front pfMaterial	FRONTMTL	pfApplyMtl	NULL
back pfMaterial	BACKMTL	pfApplyMtl	NULL
pfTexEnv	TEXENV	pfApplyTEnv	NULL
pfTexture	TEXTURE	pfApplyTex	NULL
pfFog	FOG	pfApplyFog	NULL
pfColortable	COLORTABLE	pfApplyCtab	NULL
pfHighlight	HIGHLIGHT	pfApplyHlight	NULL
pfLPointState	LPOINTSTATE	pfApplyLPState	NULL
pfTexGen	TEXGEN	pfApplyTGen	NULL

State values may be established within libpr in one of three ways:

1. Immediate mode
2. Display list mode
3. pfGeoState mode

Like the graphics library itself, IRIS Performer has two command execution modes: immediate mode and display list mode. In immediate mode, the setting of a mode or the application of an attribute is carried out immediately. Any geometry rendered afterwards will be drawn with that mode or attribute

characteristics. In display list mode, the command will be "captured" by the open display list (pfDispList) and will not have effect until the display list is closed and later drawn with **pfDrawDList**.

All the routines listed in Table 1 are display-listable, which is to say that they will be captured by an open pfDispList. In immediate mode, most of the above routines send command tokens to the graphics pipeline. Thus, the process invoking these commands must have a graphics context open to accept the command tokens, otherwise a segmentation violation or similar severe exception will result. In addition to an open graphics context, a global pfState must have been selected by **pfSelectState**. Note that neither a graphics context nor pfState is required when drawing in display list mode because the commands will be captured by the display list.

Example 1:

```
/* Enable wireframe in immediate mode */
pfEnable(PFEN_WIREFRAME);

/* Draw 'gset' in wireframe */
pfDrawGSet(gset);
```

Example 2:

```
/* Enter display list mode by opening 'dlist' for appending */
pfOpenDList(dlist);
pfEnable(PFEN_WIREFRAME);
pfDrawGSet(gset);
pfCloseDList();

/* Draw 'gset' in wireframe */
pfDrawDList(dlist);
```

It is important to realize that IRIS Performer display lists (pfDispLists) are different from graphics library display lists. A pfDispList captures only **libpr** commands and does not contain low-level geometric information like vertex coordinates and colors.

The pfGeoState encapsulates all of libpr state, i.e. it has all mode settings and a pointer to a definition for each attribute type. Through **pfGStateMode** and **pfGStateAttr** it is possible to set every state element of a pfGeoState. When the pfGeoState is applied through **pfApplyGState** all the state settings encapsulated by the pfGeoState become active. pfGeoStates also have useful inheritance properties that are discussed in the pfGeoState man page. Typical use of a pfGeoState is to "build" it at database initialization time and attach it to a pfGeoSet (**pfGSetGState**). In this way the pfGeoState defines the graphics state of the geometry encapsulated by the pfGeoSet (pfDrawGSet will call **pfApplyGState** if the pfGeoSet has an

attached pfGeoState).

Example 3:

```
/* Set up wireframe pfGeoState */
gstate = pfNewGState(NULL);
pfGStateMode(gstate, PFSTATE_ENWIREFRAME, PF_ON);

/* Draw wireframe pfGeoSet in "pfGeoState" mode */
pfApplyGState(gstate);          /* Apply 'gstate' */
pfDrawGSet(gset);               /* Draw 'gset' in wireframe */

/* Preferred method for drawing wireframe pfGeoSet */
pfGSetGState(gset, gstate);     /* Attach 'gstate' to 'gset' */
pfDrawGSet(gset);               /* Draw 'gset' in wireframe */
```

pfInitState initializes internal IRIS Performer state. *arena* specifies a shared semaphore arena created by **usinit** for multiprocess operation of IRIS Performer or **NULL** for single process operation. For proper multiprocess operation, **pfInitState** should be called by a single process before calls to **sproc** or **fork** that will generate other processes using IRIS Performer state, with an arena which is shared by all application processes. In either single or multi-process operation, **pfInitState** must be called before any state attributes such as pfTextures are created and should only be called once.

pfNewState creates and returns a handle to a pfState. *arena* specifies a malloc arena out of which the pfState is allocated or **NULL** for allocation off the process heap. pfStates can be deleted with **pfDelete**. Specifically, a pfState has a stack of state structures that shadow IRIS Performer and graphics library state. This stack may be manipulated by routines described below.

pfGetStateClassType returns the **pfType*** for the class **pfState**. The **pfType*** returned by **pfGetStateClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfState**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

A pfState should be created for each graphics context that the process draws to with libpr routines. When a process switches graphics contexts it should also switch to the corresponding pfState with **pfSelectState**.

pfSelectState makes *state* the current state. pfState is a global value so it is shared by all share group processes(See **sproc**). **pfSelectState** should be used when switching between different graphics contexts. It does not configure the graphics context with its state settings. **pfGetCurState** returns a pointer to the current pfState or NULL if there is no active pfState.

Each pfState structure maintains a 64-deep stack of pfGeoStates. A pfGeoState shadows all libpr modes and attributes. Changes to the current state made through any of the 3 methods listed above are recorded in the top of the pfGeoState stack. **pfGetState** copies the top of the pfGeoState stack into *gstate*.

pfPushState pushes the pfGeoState stack of the current pfState. When pushed, the configuration of the current state is recorded so that when popped, that state will be restored, overwriting any state changes made between push and pop. The bit vector which represents state elements that are overridden by **pfOverride** is also pushed.

pfPopState compares the current pfGeoState with that of the previously pushed pfGeoState and calls graphics library routines to restore the previously pushed state. The override bit vector is popped before popping any state elements. State changes made to a graphics context must be made using the IRIS Performer for **pfPushState** and **pfPopState** to work correctly. Calls made by the application directly to the graphics library will circumvent IRIS Performer state management which may or may not be desired.

pfFlushState is only useful for applications which use pfGeoStates. pfGeoStates do not inherit state from each other so state is pushed and popped when drawing them. For performance, state is not actually popped unless a subsequent pfGeoState requires it. This means that in-between pfGeoStates, the state may not be what the application expects. **pfFlushState** will return the state to the global default.(See pfGeoState for more on state flushing). **pfPushState** calls **pfFlushState**.

pfBasicState is a convenience routine for disabling all modes and is useful for drawing things like text which usually should not be lit or fogged. Specifically, **pfBasicState** is equivalent to the following:

```
/* return graphics pipeline to basic state */
pfDisable(PFEN_FOG);
pfDisable(PFEN_LIGHTING);
pfDisable(PFEN_TEXTURE);
pfDisable(PFEN_WIREFRAME);
pfDisable(PFEN_COLORTABLE);
pfDisable(PFEN_HIGHLIGHTING);
pfDisable(PFEN_LPOINTSTATE);
pfDisable(PFEN_TEXGEN);

pfShadeModel(PFSM_GOURAUD);
pfAlphaFunc(0, PFAF_OFF);
pfCullFace(PFCF_OFF);
pfTransparency(PFTR_OFF);

if (multisampling-type antialiasing is not enabled)
    pfAntialias(PFAA_OFF);

pfDecal(PFDECAL_OFF);
```

Use **pfMakeBasicGState** to configure every state element (value, mode, and attribute) of *gstate* to be identical to the state set with **pfBasicState**. The following code fragment is equivalent to **pfBasicState**:

```
pfGeoState *gstate = pfNewGState(NULL);
pfMakeBasicGState(gstate);
pfLoadGState(gstate);
```

Each of **pfSelectState**, **pfPushState**, **pfPopState**, **pfFlushState**, and **pfBasicState** are display-listable commands.

SEE ALSO

pfAlphaFunc, pfAntialias, pfColortable, pfCullFace, pfDecal, pfDelete, pfEnable, pfFog, pfGeoSet, pfGeoState, pfHighlight, pfLight, pfLightModel, pfLPointState, pfMaterial, pfOverride, pfShadeModel, pfTexEnv, pfTexGen, pfTexture, pfTransparency

NAME

pfNewStats, pfGetStatsClassType, pfStatsClass, pfGetStatsClass, pfStatsClassMode, pfGetStatsClassMode, pfOpenStats, pfCloseStats, pfGetOpenStats, pfStatsAttr, pfGetStatsAttr, pfStatsHwAttr, pfGetStatsHwAttr, pfEnableStatsHw, pfDisableStatsHw, pfGetStatsHwEnable, pfCopyStats, pfResetStats, pfClearStats, pfStatsCountGSet, pfAccumulateStats, pfAverageStats, pfQueryStats, pfMQueryStats, pfGetCurStats – Maintain statistics on IRIS Performer operations and system usage

FUNCTION SPECIFICATION

```
#include <Performer/pr.h>
#include <Performer/prstats.h>
pfStats *  pfNewStats(void *arena);
pfType *   pfGetStatsClassType(void);
uint       pfStatsClass(pfStats *stats, uint enmask, int val);
uint       pfGetStatsClass(pfStats *stats, uint enmask);
uint       pfStatsClassMode(pfStats *stats, int class, uint mask, int val);
uint       pfGetStatsClassMode(pfStats *stats, int class);
uint       pfOpenStats(pfStats *stats, uint enmask);
uint       pfCloseStats(uint enmask);
uint       pfGetOpenStats(pfStats *stats, uint enmask);
void       pfStatsAttr(pfStats *fstats, int attr, float val);
float      pfGetStatsAttr(pfStats *fstats, int attr);
void       pfStatsHwAttr(int attr, float val);
float      pfGetStatsHwAttr(int attr);
void       pfEnableStatsHw(uint which);
void       pfDisableStatsHw(uint which);
uint       pfGetStatsHwEnable(uint which);
void       pfCopyStats(pfStats *dst, pfStats *src, uint which);
void       pfResetStats(pfStats *stats);
void       pfClearStats(pfStats *stats, uint which);
void       pfStatsCountGSet(pfStats *stats, pfGeoSet *gset);
void       pfAccumulateStats(pfStats *dst, pfStats *src, uint which);
```

void **pfAverageStats**(pfStats *dst, pfStats *src, uint which, int num);

int **pfQueryStats**(pfStats *stats, uint which, void *dst, int size);

int **pfMQueryStats**(pfStats *stats, uint *which, void *dst, int size);

pfStats * **pfGetCurStats**(void);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfStats** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfStats**. Casting an object of class **pfStats** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfStats** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

DESCRIPTION

These functions are used to collect, manipulate, print, and query statistics on state operations, geometry, and graphics and system operations.

Since some statistics can be expensive to gather, and so might possibly influence other statistics, statistics are divided into different classes based on the tasks that they monitor and one may select the specific statistics classes of interest with **pfStatsClass**.

Statistics classes also have different modes of collection so that expensive modes of a class can be disabled with **pfStatsClassMode**. The statistics class enables may be used for directing operations on statistics

structures, including statistics collection, specified via **pfOpenStats**, and also printing, copying, clearing, accumulation, and averaging. These enables and disables are specified with bitmasks. Each statistics class has an enable token: a **PFSTATS_EN*** token that can be OR-ed with other statistics enable tokens and the result passed in to enable and disable statistics operations.

Statistics classes that require special hardware support have token names that start with **PFSTATSHW_**. These tokens are used as class enable tokens in the usual statistics routines, and also to enable and disable the hardware statistics gathering via **pfEnableStatsHw** and **pfDisableStatsHw**.

Statistics classes also have modes that select different elements of a class for collection. These modes are set through **pfStatsClassMode**. Each statistics class starts with a default mode setting.

The following tables provide details of the statistics class structure. The first table lists the statistics classes, their naming tokens, and their enable tokens for forming bitmasks.

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**
Memory	**PFSTATS_MEM**	**PFSTATS_ENMEM**

This second table defines the statistics classes and their naming token and enable tokens for forming bitmasks.

Statistics Mode Table

Class	PFSTATS_ Token	Modes
Graphics Rendered	**PFSTATS_GFX**	**PFSTATS_GFX_GEOM** **PFSTATS_GFX_TSTRIP_LENGTHS** **PFSTATS_GFX_ATTR_COUNTS** **PFSTATS_GFX_STATE** **PFSTATS_GFX_XFORM**
Pixel Fill	**PFSTATSHW_GFXPIPE_FILL**	**PFSTATSHW_GFXPIPE_FILL_DEPTHCMP** **PFSTATSHW_GFXPIPE_FILL_TRANSPARENT**
CPU	**PFSTATSHW_CPU**	**PFSTATSHW_CPU_SYS** **PFSTATSHW_CPU_IND**

The individual stats classes and modes are discussed in more detail in the explanation of the statistics routines.

pfNewStats creates and returns a handle to a pfStats. *arena* specifies a malloc arena out of which the pfStats is allocated or **NULL** for allocation off the process heap. pfStats can be deleted with **pfDelete**.

pfGetStatsClassType returns the **pfType*** for the class **pfStats**. The **pfType*** returned by **pfGetStatsClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfStats**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfResetStats takes a pointer to a statistics structure, *stats*, and will reset that entire statistics structure to its initial state.

pfStatsClass takes a pointer to a statistics structure, *stats*, and will set the classes specified in the bitmask, *enmask*, according to the *val*, which must be set to one of the following:

PFSTATS_ON	Enables the specified classes.
PFSTATS_OFF	Disables the specified classes.
PFSTATS_DEFAULT	Resets the specified classes to default values.
PFSTATS_SET	Sets the entire class enable mask to *enmask*.

All stats collection can be set at once to on, off, or the default by using **PFSTATS_ALL** for the bitmask and the appropriate value for the enable flag. For example, the following example enables all stats classes with their current class mode settings.

```
pfStatsClass(stats, PFSTATS_ALL, PFSTATS_ON);
```

pfGetStatsClass takes a pointer to a stats structure, *stats*, and the statistics classes of interest specified in the bitmask, *enmask*. If any of the statistics classes specified in *enmask* are enabled, then **pfGetStatsClass** will return the bitmask of those classes, and otherwise, will return zero. If classes of an open pfStats structure are disabled, then collection of those classes stop immediately and those classes are considered closed.

pfStatsClassMode takes a pointer to a stats structure, *stats*, the name of the class to set, *class*, a mask of the modes to set, *mask*, and the value for the modes, *val*. The class modes offer further control over the statistics that are to be accumulated for a given class. For each statistics class, a set of modes is enabled by default. Some modes of a statistics class may be somewhat expensive, and therefore they are not enabled by default. *val* must be one of:

PFSTATS_ON	Enable the modes specified in *mask*.
PFSTATS_OFF	Disable the modes specified in *mask*.
PFSTATS_DEFAULT	Set modes specified in *mask* to default values.
PFSTATS_SET	Set class mode mask to the specified *mask*.

As a convenience, all classes may have all of their modes set to on, off, or their default values by specifying a *class* of **PFSTATS_CLASSES**, and a mask of **PFSTATS_ALL**. These defaults may differ between machines and may change in the future; so, code should not assume the current defaults but query the mode values for a given class where needed. No statistics for a given mode are accumulated unless the corresponding class has been enabled with **pfStatsClass**. If modes of an open pfStats structure are disabled, then collection of those modes stop immediately.

Graphics Statistics Modes

PFSTATS_GFX_GEOM
This counts geometry that is drawn via **pfDrawGSet**. Statistics include the number of pfGeoSets drawn, and the numbers of pfGeoSets that have each binding of each attribute, colors, normals, texture coordinates. The number of each type of pfGeoSet primitive is counted, as well as the number of base primitives drawn: total triangles, lines, and points. Statistics are also kept on the number of triangle strips drawn and the number of the total triangles that were actually in a triangle strip. This mode is enabled by default.

PFSTATS_GFX_TSTRIP_LENGTHS
The number of triangles in strips whose length in terms of triangle count is shorter than **PFSTATS_TSTRIP_LENGTHS_MAX** is recorded. Triangles in strips whose triangle count is greater than or equal to **PFSTATS_TSTRIP_LENGTHS_MAX** are all counted together. Quads are counted as strips of length two and independent triangles are counted as strips of length one. An average triangle strip length (that uses all of the actual lengths) is also maintained. Keeping these triangle strip statistics is expensive for the drawing operation and so this mode is not enabled by default, but must be enabled with **pfStatsClassMode**.

PFSTATS_GFX_ATTR_COUNTS
The number of each of the different types of geometry attributes (colors, normals, and texture coordinates) that are drawn is counted. Keeping attribute statistics is expensive for the drawing operation and so this mode is not enabled by default, but must be enabled with **pfStatsClassMode**.

PFSTATS_GFX_STATE
This mode enables the counting of calls to state changes, as well as the number of actual state changes themselves. Such state changes include the immediate mode routines such as **pfAntialias**, and the application of the state structures, such as **pfApplyTex**. Also counted is the number of pfGeoStates encountered and the number of state stack operations, such as **pfLoadGState**, **pfApplyGState**, **pfPushState**, and **pfPopState**. This mode is enabled by default.

PFSTATS_GFX_XFORM
This mode enables the counting of calls to transformations, such as **pfTranslate**, **pfScale**, and **pfRotate**, and graphics matrix stack operations, such as **pfLoadMatrix**, etc. the number of actual state changes themselves, as well as the number of pfGeoStates encountered. This mode is enabled by default.

Graphics Pipe Fill Statistics

These modes enable the accumulation of fill depth-complexity statistics and require the corresponding hardware statistics to be enabled:

```
pfEnableStatsHw(PFSTATSHW_GFXPIPE_FILL_DEPTHCMP);
```

PFSTATSHW_GFXPIPE_FILL_DCPAINT
This mode causes **pfCloseStats** to paint the screen according to the number of times each pixel is touched. This mode is enabled by default.

PFSTATSHW_GFXPIPE_FILL_DCCOUNT
This mode causes **pfCloseStats** to read back the framebuffer for the calculation of fill depth-complexity statistics. This mode is enabled by default.

PFSTATSHW_GFXPIPE_FILL_DEPTHCMP
By default, only actual pixel writes are counted with depth complexity stats. This mode enables counting of Z compares as well. This mode is not enabled by default.

PFSTATSHW_GFXPIPE_FILL_TRANSP
This mode enables counting of fully transparent pixels. This mode is not enabled by default.

CPU Statistics

The CPU statistics keep track of system usage and requires that the corresponding hardware statistics be enabled:

```
pfEnableStatsHw(PFSTATSHW_ENCPU);
```

The percentage of time CPUs spend idle, busy, in user code, and waiting on the Graphics Pipeline, or on the swapping of memory is calculated. Counted is the number of context switches (process and graphics), the number of system calls, the number of times the graphics FIFO is found to be full, the number of times a CPU went to sleep waiting on a full graphics FIFO, the number of graphics pipeline IOCTLs issued (by the system), and the number of swapbuffers seen. All of these statistics are computed over an elapsed period of time, and using an elapsed interval of at least one second is recommended.

PFSTATSHW_CPU_SYS
This mode enables computation of the above CPU statistics for the entire system. This includes statistics on system usage, cpu-graphics interactions, and memory. CPU usage statistics are summed over all CPUs. This mode is enabled by default.

PFSTATSHW_CPU_IND
This mode enables tracking of CPU statistics for each individual CPU and is much more expensive than using just the summed statistics. It is not enabled by default.

pfGetStatsClassMode takes a pointer to a stats structure, *stats*, and the name of the class to query, *class*. The return value is the mode of *class*.

pfOpenStats takes a pointer to a stats structure, *stats*, and a bitmask specifying the statistics classes that are to be opened for collection in *enmask*. This statistics structure will become the one and only statistics structure open for collection. The return value will be the bitmask for all currently open statistics classes. If another pfStats structure is already open, then this call to **pfOpenStats** will be ignored and the return value will be 0. When statistics classes that use statistics hardware are open for collection, **pfOpenStats** will access that hardware for initialization. Therefore, for graphics pipe statistics, it is imperative that the statistics hardware only be enabled for the drawing process. Furthermore, only one processes should be using statistics hardware at a time since it is a shared global resource. Finally, for statistics that are actually accumulated in statistics hardware, it is best to let some time elapse before the statistics are collected (in **pfCloseStats**). Refer the examples at the end of this manual page.

pfCloseStats takes a bitmask *which* specifying the classes whose collected statistics are to be accumulated into the current **pfStats** structure. Further collection of these statistics are then disabled and they will have to be re-opened with **pfOpenStats** for further collection. A pfStats structure is considered to be open until all opened statistics classes have been closed with **pfCloseStats**. The return value for **pfCloseStats** is the bitmask of the remaining open classes. If *stats* has no open classes, a value of 0 will be returned. When statistics classes that use statistics hardware are open for collection, **pfCloseStats** will access that hardware to collect the specified statistics. Therefore, for graphics pipe statistics, it is imperative that the statistics hardware only be enabled for the drawing process. Furthermore, only one processes should be using statistics hardware at a time since it is a shared global resource.

pfGetOpenStats takes a pointer to a stats structure, *stats*, a bitmask *enmask* specifying the statistics classes that are being queried. If any of the statistics classes specified in *enmask* are open for collection, then the bitmask of those statistics classes is returned, and otherwise, zero.

pfGetCurStats returns the currently open statistics structure, or NULL if there is no statistics structure open for accumulation.

pfStatsAttr takes a pointer to a stats structure, *stats*, the name of the attribute to set, *attr*, and the attribute value, *val*. Currently, there are no pfStats attributes. **pfGetStatsAttr** takes a pointer to a stats structure, *stats*, and the name of the attribute to query, *attr*. The return value is that of attribute *attr*.

pfEnableStatsHw takes a bitmask *which* specifying the hardware statistics that should be enabled. These bitmasks are the statistics class enable bitmasks that have start with **PFSTATSHW_***. Statistics hardware must be enabled for the corresponding classes of statistics to be accumulated. Having statistics hardware enabled will have some cost to performance; however, in most cases, it pays to leave this hardware enabled if corresponding statistics classes are being frequently enabled and disabled. When statistics classes that use statistics hardware are open for collection, **pfOpenStats** and **pfCloseStats** will access that hardware. For graphics pipe statistics, it is therefore imperative that the statistics hardware only be enabled for a process that is connected to the graphics pipeline. Furthermore, only one processes should be using statistics hardware at a time since it is a shared global resource.

Graphics Statistics Hardware Enables

PFSTATSHW_ENGFXPIPE_FILL
Enables hardware to support tracking of depth complexity statistics. When this mode is enabled, the framebuffer keeps track of the number of times each pixel is touched. This may require a framebuffer reconfiguration which can be quite expensive, and which may not be possible in GLX windows.

PFSTATSHW_ENCPU
This mode enable gathering of CPU statistics by the system. This mode should only be enabled by one process at a time.

pfDisableStatsHw takes a bitmask *which* specifying the hardware statistics that should be disabled.

pfGetStatsHwEnable takes a bitmask *which* specifying the hardware statistics that are being queried. If any of the hardware statistics classes specified in *which* have their corresponding hardware enabled, then the bitmask of those statistics classes is returned, and otherwise, zero is returned.

pfStatsHwAttr takes the name of the attribute to set, *attr*, and the attribute value, *val*. There is currently one stats hardware attribute: **PFSTATSHW_FILL_DCBITS**. Its value must be and integer value in the range of 1 to 4. The default value is 3. This attribute sets the maximum number of stencil bits used for tracking fill depth complexity. See the GL manual page for **stensize(3g)** for more information on stencil bitplanes.

pfGetStatsHwAttr Returns the value of attribute *attr*.

Collected statistics can be printed to stderr or a file with **pfPrint**, and can be queried at run-time with **pfQueryStats** for a single-value query, and **pfMQueryStats** for getting back a collection of statistics.

pfQueryStats takes a pointer to a statistics structure, *stats*, a query token in *which*, and a destination buffer *dst*. The size of the expected return data is specified by *size* and if non-zero, will prevent **pfQueryStats** from writing beyond a buffer that is too small. The return value is the number of bytes written to the destination buffer. There are tokens for getting back all of the statistics, entire sub-structures, and individual values. The exposed query structure types and query tokens are all defined in

<*Performer/prstats.h*>. Every structure and field is commented with its corresponding query token. For example, the exposed structure type pfStatsValues can be used to hold the entire contents of a pfStats structure and has corresponding query token **PFSTATSVAL_ALL**.

```
typedef struct pfStatsValues
{   /* to get back all stats:       PFSTATSVAL_ALL                */
    /* PFSTATS_GFX class:             PFSTATSVAL_GFX              */
    pfStatsValGeom      geom;        /* PFSTATSVAL_GFX_GEOM        */
    pfStatsValModes     modeChanges; /* PFSTATSVAL_GFX_MODECHANGES */
    pfStatsValModes     modeCalls;   /* PFSTATSVAL_GFX_MODECALLS   */
    pfStatsValState     state;       /* PFSTATSVAL_GFX_STATE       */
    pfStatsValXforms    xform;       /* PFSTATSVAL_GFX_XFORM       */
    /* PFSTATSHW_GFXPIPE_FILL class: PFSTATSVAL_GFXPIPE_FILL       */
    pfStatsValFill      fill;        /* PFSTATSVAL_GFXPIPE_FILL    */
    /* the PFSTATSHW_CPU class:      PFSTATSVAL_CPU                */
    pfStatsValCPU       cpu;         /* PFSTATSVAL_CPU             */
} pfStatsValues;
```

The following example will return all of the contents of a pfStats structure into the contents of a structure of the exposed type pfStatsValues.

```
pfStats *stats;
pfStatsValues val;
pfQueryStats(stats, PFSTATSVAL_ALL, (float *) val);
```

pfMQueryStats takes a pointer to a statistics structure, *stats*, a pointer to the start of an array of query tokens in *which*, and a destination buffer *dst*. The size of the expected return data is specified by *size* and if non-zero, will prevent **pfMQueryStats** from writing beyond a buffer that is too small. The return value is the number of bytes written to the destination buffer.

pfCopyStats takes pointers to statistics structures, *dst* and *src*, and a bitmask *which* specifying the statistics that are to be copied from *src* to *dst*. This function is provided to enable more control over the default pfObject function **pfCopy**. Note that only statistics data is copied and not any enable/disable settings or modes.

pfClearStats takes a pointer to a statistics structure, *stats*, and a bitmask *which* specifying the statistics that are to be cleared to zeroes.

pfStatsCountGSet takes a pointer to a statistics structure, *stats*, and a pointer to a pfGeoSet, *gset*, whose geometry statistics are to be accumulated into *stats*.

pfAccumulateStats takes pointers to statistics structures, *dst* and *src*, and a bitmask *which* specifying the statistics that are to be accumulated from *src* to *dst*.

pfAverageStats takes pointers to statistics structures, *dst* and *src*, and a bitmask *which* specifying the statistics classes that are to be averaged from *src* and the result placed in *dst*, and *num*, the number of elements over which the specified statistics in *src* are to be averaged.

For a class of statistics to be collected, the following must be true:

1. A statistics structure must be created.
2. The corresponding statistics class must be enabled with **pfStatsClass**. No statistics classes are enabled by default.
3. The corresponding statistics class mode must be enabled with **pfStatsClassMode**. However, each statistics class has a popular set of statistics modes enabled by default.
4. Any relevant hardware must be enabled **pfEnableStatsHw**.
5. The statistics class must be opened for collection with **pfOpenStats**.

EXAMPLES

This example creates a statistics structure and enabling the graphics statistics class with the triangle-strip statistics enabled.

```
pfStats *stats = NULL;
stats = pfNewStats(NULL);
pfStatsClass(stats, PFSTATS_ENGFX, PFSTATS_ON);
pfStatsClassMode(stats, PFSTATS_GFX, PFSTATS_GFX_TSTRIP_LENGTHS, PFSTATS_ON);
```

This is an example of collecting CPU statistics over an elapsed period of time.

```
pfStats *stats = NULL;
double lastTime = 0;
stats = pfNewStats(NULL);

/* enable the CPU stats class - using the default summed CPU statistics */
pfStatsClass(stats, PFSTATSHW, PFSTATS_ON);

/* enable CPU stats hardware */
pfEnableStatsHw(PFSTATSHW_ENCPU);
 :
/* snap CPU stats every 2 seconds */
if (pfGetTime() - lastTime > 2.0)
{
```

```
    if (pfGetOpenStats(stats, PFSTATSHW_ENCPU))
    {
        /*
         * final snap of CPU stats is done here and difference
         * between this and the initial snap is calculated.
         */
        pfCloseStats(PFSTATSHW_ENCPU);
    }
    else
    {
        /* initial snap of CPU stats is done here */
        pfOpenStats(stats, PFSTATSHW_ENCPU);
    }
}
```

This example shows the enabling and disabling of fill statistics.

```
pfStats *stats = NULL;
stats = pfNewStats(NULL);

/* enable fill statistics collection */
pfStatsClass(stats, PFSTATSHW_ENGFXPIPE_FILL, PFSTATS_ON);

/* enable fill stats hardware - put framebuffer in correct configuration */
pfEnableStatsHW(PFSTATSHW_ENGFXPIPE_FILL);

/* open fill statistics collection and initialize hardware */
pfOpenStats(stats, PFSTATSHW_ENGFXPIPE_FILL);

/* draw geometry */
 :
/*
 * paint window by number of times each pixel was touched in the
 * previous drawing and read back the framebuffer and
 * examine the counts
 */
pfCloseStats(stats, PFSTATSHW_ENGFXPIPE_FILL);
```

NOTES

Fill stats are currently calculated by using stencil tests and therefore require stencil bitplanes to be allocated. Furthermore, the **PFSTATSHW_GFXPIPE_FILL_TRANSP** mode currently disables modes that reject fully transparent pixels, such as **pfAlphaFunc**, which will alter what pixels get written into the zbuffer and therefore should be used in conjunction with **PFSTATSHW_GFXPIPE_FILL_DEPTHCMP**.

The pfStats routines, structures, and constants are defined in the <Performer/prstats.h> header file.

The Indy graphics platforms do not offer stencil under IRIS GL operation. Allocation of stencil bits may affect other attributes of your framebuffer configuration, such as depth buffer resolution and number of samples for multisample.

BUGS

The checking of *size* in **pfQueryStats** and **pfMQueryStats** is not yet implemented.

SEE ALSO

pfDelete, pfPrint

NAME

pfNewString, pfGetStringClassType, pfStringMat, pfGetStringMat, pfStringMode, pfGetStringMode, pfStringFont, pfGetStringFont, pfStringString, pfGetStringString, pfStringBBox, pfGetStringBBox, pfStringSpacingScale, pfGetStringSpacingScale, pfStringColor, pfGetStringColor, pfGetStringCharGSet, pfGetStringCharPos, pfGetStringStringLength, pfStringGState, pfGetStringGState, pfDrawString, pfFlattenString, pfStringIsectMask, pfGetStringIsectMask, pfStringIsectSegs – String facility using pfFont

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfString*	**pfNewString**(void *arena);
pfType*	**pfGetStringClassType**(void);
void	**pfStringMat**(pfString *string, const pfMatrix *mat);
void	**pfGetStringMat**(const pfString *string, pfMatrix at);
void	**pfStringMode**(pfString *string, int mode, int val);
int	**pfGetStringMode**(const pfString *string, int mode);
void	**pfStringFont**(pfString *string, pfFont* font);
pfFont*	**pfGetStringFont**(const pfString *string);
void	**pfStringString**(pfString *string, const char* cstr);
const char*	**pfGetStringString**(const pfString *string);
void	**pfStringBBox**(pfString *string, const pfBox *newBBox);
const pfBox*	**pfGetStringBBox**(pfString *string);
void	**pfStringSpacingScale**(pfString *string, float horiz, float vert, float depth);
void	**pfGetStringSpacingScale**(const pfString *string, float *horiz, float *vert, float *depth);
void	**pfStringColor**(pfString *string, float r, float g, float b, float a);
void	**pfGetStringColor**(const pfString *string, float *r, float *g, float *b, float *a);
const pfGeoSet*	**pfGetStringCharGSet**(pfString *string, int index);
const pfVec3*	**pfGetStringCharPos**(pfString *string, int index);
int	**pfGetStringStringLength**(pfString *string);
void	**pfStringGState**(pfString *string, pfGeoState* gstate);
pfGeoState*	**pfGetStringGState**(const pfString *string);
void	**pfDrawString**(pfString *string);

void **pfFlattenString**(pfString *string);

void **pfStringIsectMask**(pfString *string, uint mask, int setMode, int bitOp);

uint **pfGetStringIsectMask**(pfString *string);

int **pfStringIsectSegs**(pfString *string, pfSegSet *segSet, pfHit **hits[]);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfString** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfString**. Casting an object of class **pfString** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfString** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

string identifies a pointer to a pfString

font identifies a pointer to a pfFont

DESCRIPTION

pfNewString creates and returns a handle to a pfString. *arena* specifies a malloc arena out of which the pfString is allocated or **NULL** for allocation off the process heap. pfStrings can be deleted with **pfDelete**.

pfGetStringClassType returns the **pfType*** for the class **pfString**. The **pfType*** returned by **pfGetStringClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfString**. Because IRIS Performer allows subclassing of built-in types, when decisions are made

based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfStringMat sets the transformation matrix for *string* to *mat* or to the identity matrix if *mat* is NULL. Call **pfGetStringMat** to retrieve the matrix.

Use **pfStringMode** and **pfGetStringMode** to set and get modes of the *string*. Use the **PFSTR_JUSTIFY** mode to set the justification of the string to one of the following:

PFSTR_FIRST or **PFSTR_LEFT**
Set the string to be left-justified; the first character will be immediately to the right of the origin.

PFSTR_MIDDLE or **PFSTR_CENTER**
Set the string to be center-justified; the string will be centered at the origin.

PFSTR_LAST or **PFSTR_RIGHT**
Set the string to be right-justified; the last character will be immediately to the left of the origin.

Use the **PFSTR_CHAR_SIZE** to specify the size, in bytes, of each character in the string. *val* is one of **PFSTR_CHAR**, **PFSTR_SHORT**, **PFSTR_INT** indicating that character sizes are sizeof(char), sizeof(short), and sizeof(int) respectively.

Call **pfGetStringBBox** to retrieve the bounding box around *string* as if it were drawn using the assigned justification, drawing style, font, and character string. Call **pfStringBBox** to set this bounding box explicitly.

pfStringSpacingScale and **pfGetStringSpacingScale** set and get the spacing scales of *string*. Normally after rendering a character, the rendering position is translated in x, y, and z by the character's spacing which is supplied by the pfFont associated with the pfString. This spacing is scaled by the supplied scale factors enabling greater/lesser distances between characters. Also, when the **PFSTR_AUTO_SPACING** mode is enabled, the spacing scales can be used to change the direction of the rendered string, for example, spacing scales of (0, -1, 0) will render the string vertically, suitable for labeling the vertical axis of a 2D graph.

of the steps used when drawing each succeeding character from *string*. The default spacing scales are all 1.0 An interesting example is horizontal scale 0 and vertical scale -1, in which case the string is drawn downward like a neon sign outside a downtown bar and grill.

pfStringGState and **pfGetStringGState** set and get the **pfGeoState** attached to *string*. If no **pfGeoState** is attached, then *string* will be drawn with whatever graphics state is active at draw time.

Call **pfStringFont** and **pfGetStringFont** to set or get the pfFont used by *string*. Call **pfStringString** and

pfGetStringString to set or get the character string for *string*. **pfGetStringStringLength** will return the length of a **pfString**'s current character string. **pfStringColor** and **pfGetStringColor** get and set *string's* color.

pfGetStringCharGSet returns a pointer to the GeoSet currently being used to draw the character at place *index* in the pfString's string. **pfGetStringCharPos** likewise returns a pointer to a pfVec3 that specifies the exact location relative to the current transform where the character in position *index* in the string array will be drawn. Both of the above functions will return NULL if *index* is greater than or equal to the pfString's length - which may be obtained through **pfGetStringStringLength**.

Call **pfDrawString** to draw the string, including applying the **pfGeoState** if available, the color, and the texture if this is a **PFSTR_TEXTURED** string.

A **pfString** is normally composed of a **pfGeoSet** per character in the string. Call **pfFlattenString** to flatten the character spacings into the individual GeoSets. A flattened primitive is faster, but flattening has overhead which should be avoided if the string will be changing quickly, e.g. in every frame. **pfFlattenString** Must be called every time you change the contents of the string geometry; e.g. after calling **pfStringString**.

pfStringIsectMask enables intersections and sets the intersection mask for *string*. *mask* is a 32-bit mask used to determine whether a particular pfString should be examined during a particular intersection request. A non-zero bit-wise AND of the pfString's mask with the mask of the intersection request (-**pfStringIsectSegs**) indicates that the pfString should be tested. The default mask is all 1's, i.e. 0xffffffff.

pfGetStringIsectMask returns the intersection mask of the specified pfString.

pfStringIsectSegs tests for intersection between the pfString *string* and the group of line segments specified in *string*. This is done by testing against each of the pfGeoSets in the pfString. The resulting intersections (if any) are returned in *hits*. The return value of **pfStringIsectSegs** is the number of segments that intersected the pfString. See **pfGSetIsectSegs** for intersection information returned from tests against the geometric primitives inside pfGeoSets.

NOTES

See **pfText** for sample code demonstrating **pfString**.

SEE ALSO

pfFont, pfDelete, pfGeoSet, pfText

NAME

pfNewTEnv, pfGetTEnvClassType, pfTEnvMode, pfGetTEnvMode, pfTEnvComponent, pfGetTEnvComponent, pfTEnvBlendColor, pfGetTEnvBlendColor, pfApplyTEnv, pfGetCurTEnv – Create, modify and query texture environment

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfTexEnv * **pfNewTEnv**(void *arena);

pfType * **pfGetTEnvClassType**(void);

void **pfTEnvMode**(pfTexEnv *tenv, int mode);

int **pfGetTEnvMode**(const pfTexEnv *tenv);

void **pfTEnvComponent**(pfTexEnv *tenv, int comp);

int **pfGetTEnvComponent**(const pfTexEnv *tenv);

void **pfTEnvBlendColor**(pfTexEnv *tenv, float r, float g, float b, float a);

void **pfGetTEnvBlendColor**(pfTexEnv *tenv, float* r, float* g, float* b, float* a);

void **pfApplyTEnv**(pfTexEnv *tenv);

pfTexEnv * **pfGetCurTEnv**(void);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfTexEnv** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfTexEnv**. Casting an object of class **pfTexEnv** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfTexEnv** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);

int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

tenv identifies a pfTexEnv.

DESCRIPTION

pfNewTEnv creates and returns a handle to a pfTexEnv. *arena* specifies a malloc arena out of which the pfTexEnv is allocated or **NULL** for allocation off the process heap. pfTexEnvs can be deleted with **pfDelete**. See the IRIS GL **tevdef** or OpenGL **glTexEnv(3g)** reference pages for more details on texture environments.

pfGetTEnvClassType returns the **pfType*** for the class **pfTexEnv**. The **pfType*** returned by **pfGetTEnvClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfTexEnv**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfTEnvMode sets the texture environment mode. *mode* is a symbolic token that specifies a texture environment mode and is one of the following:

PFTE_MODULATE
PFTE_BLEND
PFTE_DECAL
PFTE_ALPHA

The default mode is **PFTE_MODULATE**. **pfGetTEnvMode** returns the mode of *tenv*.

n**pfTEnvBlendColor** sets the texture environment blend color. This color is used only when the texture environment mode is **PFTE_BLEND**. See the IRIS GL **tevdef(3g)** or OpenGL **glTexEnv(3g)** reference pages for more details on texture environment blending. The default IRIS Performer texture environment blend color is [1,1,1,1]. This matches the default blend color for IRIS GL. However, it is different from the default OpenGL blend color of [0,0,0,0]. This was done to maintain compatibility with previous IRIS Performer releases. **pfGetTEnvBlendColor** copies the texture environment blend color into *r*, *g*, *b*, *a*.

The *comp* argument of **pfTEnvComponent** selects one or more components of the currently active pfTexture and treats the selection as a smaller 1 or 2 component texture. Thus you can use a 16-bit per texel RGBA texture as 4 independent 4-bit intensity textures. *comp* is one of:

PFTE_COMP_OFF
disables component select.

PFTE_COMP_I_GETS_R
Uses the red component of a 4 component texture as a 1 component texture.

PFTE_COMP_I_GETS_G
Uses the green component of a 4 component texture as a 1 component texture.

PFTE_COMP_I_GETS_B
Uses the blue component of a 4 component texture as a 1 component texture.

PFTE_COMP_I_GETS_A
Uses the alpha component of a 4 or 2 component texture as a 1 component texture.

PFTE_COMP_IA_GETS_RG
Uses the red and green components from a 4 component texture as a 2 component texture.

PFTE_COMP_IA_GETS_BA
Uses the blue and alpha components from a 4 component texture as a 2 component texture.

PFTE_COMP_I_GETS_I
Uses the intensity component from a 2 component texture as a 1 component texture.

The selected component or components obey the texture environment mode set by **pfTEnvMode**. See the **tevdef(3g)** man page for more details. **pfGetTEnvComponent** returns the component select value of *tenv*.

pfApplyTEnv makes *tenv* the current texture environment. When texturing is enabled (see below), this texture environment will be applied to all geometry drawn after **pfApplyTEnv** is called. Only one pfTexEnv may be active at a time although many may be defined. Modifications to *tenv*, such as changing the blend color, will not have effect until **pfApplyTEnv** is called with *tenv*.

For geometry to be textured, the following must be true:

1. Texturing must be enabled: **pfEnable**(**PFEN_TEXTURE**)
2. A pfTexEnv must be applied: **pfApplyTEnv**
3. A pfTexture must be applied: **pfApplyTex**
4. Geometry must have texture coordinates: **pfGSetAttr**, **PFGS_TEXCOORD2**

The texture environment state element is identified by the **PFSTATE_TEXENV** token. Use this token with **pfGStateAttr** to set the texture environment of a pfGeoState and with **pfOverride** to override subsequent texture environment changes.

Example 1:

```
/* Set up blue 'blend' texture environment */
tev = pfNewTEnv(NULL);
pfTEnvMode(tev, PFTE_BLEND);
pfTEnvBlendColor(tev, 0.0f, 0.0f, 1.0f);

/* Set up textured/blended pfGeoState */
pfGStateMode(gstate, PFSTATE_ENTEXTURE, PF_ON);
pfGStateAttr(gstate, PFSTATE_TEXENV, tev);
pfGStateAttr(gstate, PFSTATE_TEXTURE, tex);

/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/* Set texture coordinate array. 'gset' is non-indexed */
pfGSetAttr(gset, PFGS_TEXCOORD2, PFGS_PER_VERTEX, tcoords, NULL);

/* Draw textured gset */
pfDrawGSet(gset);
```

Example 2:

```
pfApplyTEnv(tev);

/* Override so that all textured geometry uses 'tev' */
pfOverride(PFSTATE_TEXENV, PF_ON);
```

pfApplyTEnv is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfApplyTEnv** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

pfGetCurTEnv returns the currently active pfTexEnv or NULL if there is no active pfTexEnv.

NOTES

pfTEnvComponent and **PFTE_ALPHA** are supported only on RealityEngine graphics systems.

SEE ALSO

pfDelete, pfDispList, pfEnable, pfGeoState, pfObject, pfState, pfTexture, tevbind, tevdef, texbind, texdef, glTexEnv, glTexImage2D.

NAME

pfNewTGen, pfGetTGenClassType, pfTGenMode, pfGetTGenMode, pfTGenPlane, pfGetTGen-Plane, pfApplyTGen, pfGetCurTGen – Create, modify and query texture coordinate generator

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfTexGen * **pfNewTGen**(void *arena);

pfType * **pfGetTGenClassType**(void);

void **pfTGenMode**(pfTexGen *tgen, int texCoord, int mode);

int **pfGetTGenMode**(const pfTexGen *tgen, int texCoord);

void **pfTGenPlane**(pfTexGen *tgen, int texCoord, float x, float y, float z, float d);

void **pfGetTGenPlane**(pfTexGen *tgen, int texCoord, float* x, float* y, float* z, float* d);

void **pfApplyTGen**(pfTexGen *tgen);

pfTexGen* **pfGetCurTGen**(void);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfTexGen** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfTexGen**. Casting an object of class **pfTexGen** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfTexGen** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);

int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

tgen identifies a pfTexGen

texCoord identifies a texture coordinate and is one of:

PF_S, **PF_T**, **PF_R**, **PF_Q**

DESCRIPTION

The **pfTexGen** capability is used to automatically generate texture coordinates for geometry, typically for special effects like projected texture, reflection mapping, and for light points (**pfLPointState**). **pfNewTGen** creates and returns a handle to a pfTexGen. *arena* specifies a malloc arena out of which the pfTexGen is allocated or **NULL** for allocation off the process heap. pfTexGens can be deleted with **pfDelete**.

pfGetTGenClassType returns the **pfType*** for the class **pfTexGen**. The **pfType*** returned by **pfGetTGenClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfTexGen**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfTGenMode sets the mode of *tgen* corresponding to texture coordinate *texCoord* to *mode*. *mode* must be one of the following:

PFTG_OFF
Disables texture coordinate generation

PFTG_OBJECT_LINEAR
Generate texture coordinate as distance from plane in object space.

PFTG_EYE_LINEAR
Generate texture coordinate as distance from plane in eye space. The plane is transformed by the inverse of the ModelView matrix when *tgen* is applied.

PFTG_EYE_LINEAR_IDENT
Generate texture coordinate as distance from plane in eye space. The plane is not transformed by the inverse of the ModelView matrix.

PFTG_SPHERE_MAP
Generate texture coordinate based on the view vector reflected about the vertex normal in eye space.

See the IRIS GL **texgen(3g)** or OpenGL **glTexGen(3g)** man pages for the specific mathematics of the

texture coordinate generation modes.

The default texture generation mode for all texture coordinates is **PFTG_OFF**. **pfGetTGenMode** returns the mode of *tgen*.

pfTGenPlane sets the plane equation used for generating coordinates for texture coordinate *texCoord* to $aX + bY + cZ + d = 0$. This plane equation is used when *tgen*'s mode is **PFTG_EYE_LINEAR**, **PFTG_EYE_LINEAR_IDENT**, or **PFTG_OBJECT_LINEAR** but is ignored when its mode is **PFTG_SPHERE_MAP**. The default plane equations are:

PF_S: (1, 0, 0, 0)

PF_T: (0, 1, 0, 0)

PF_R: (0, 0, 1, 0)

PF_Q: (0, 0, 0, 1)

pfGetTGenPlane will return the plane equation parameters for texture coordinate *texCoord* of *tgen* in *x*, *y*, *z*, *d*.

pfApplyTGen configures the graphics hardware with the texture coordinate generating parameters encapsulated by *tgen*. Only the most recently applied pfTexGen is active although any number of pfTexGen definitions may be created. Texture coordinate generation must be enabled (**pfEnable**(-**PFEN_TEXGEN**)) for *tgen* to have effect and modifications made to *tgen* do not have effect until the next time **pfApplyTGen** is called. **pfGetCurTGen** returns the currently active pfTexGen.

The pfTexGen state element is identified by the **PFSTATE_TEXGEN** token. Use this token with **pfGStateAttr** to set the pfTexGen attribute of a pfGeoState and with **pfOverride** to override subsequent pfTexGen changes:

Example 1:

```
/* Set up pfGeoState */
pfGStateMode(gstate, PFSTATE_ENTEXGEN, PF_ON);
pfGStateAttr(gstate, PFSTATE_TEXGEN, pfNewTGen(arena));

/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/* Draw pfGeoSet whose texture coordinates are generated */
pfDrawGSet(gset);
```

Example 2:

```
/* Override so that all geometry is affected by 'tgen' */
pfEnable(PFEN_TEXGEN);
pfApplyTGen(tgen);
pfOverride(PFSTATE_TEXGEN | PFSTATE_ENTEXGEN, PF_ON);
```

pfApplyTGen is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfApplyTGen** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

SEE ALSO

pfDelete, pfDispList, pfGeoState, pfLPointState, pfMemory, pfObject, pfOverride, texgen, glTexGen.

NAME

pfNewTex, pfGetTexClassType, pfTexName, pfGetTexName, pfTexImage, pfGetTexImage, pfTexFormat, pfGetTexFormat, pfTexFilter, pfGetTexFilter, pfTexRepeat, pfGetTexRepeat, pfTexBorderColor, pfGetTexBorderColor, pfTexBorderType, pfGetTexBorderType, pfTexSpline, pfGetTexSpline, pfTexDetail, pfGetTexDetail, pfGetTexDetailTex, pfDetailTexTile, pfGetDetailTexTile, pfTexList, pfGetTexList, pfTexFrame, pfGetTexFrame, pfTexLoadMode, pfGetTexLoadMode, pfTexLevel, pfGetTexLevel, pfTexLoadOrigin, pfGetTexLoadOrigin, pfTexLoadSize, pfGetTexLoadSize, pfApplyTex, pfFormatTex, pfLoadTex, pfTexLoadImage, pfGetTexLoadImage, pfLoadTexLevel, pfLoadTexFile, pfSubloadTex, pfSubloadTexLevel, pfFreeTexImage, pfIdleTex, pfIsTexLoaded, pfIsTexFormatted, pfGetCurTex – Create, modify and query texture

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfTexture * **pfNewTex**(void *arena);

pfType * **pfGetTexClassType**(void);

void **pfTexName**(pfTexture *tex, const char *name);

const char * **pfGetTexName**(const pfTexture *tex);

void **pfTexImage**(pfTexture *tex, uint* image, int comp, int ns, int nt, int nr);

void **pfGetTexImage**(const pfTexture *tex, uint **image, int *comp, int *ns, int *nt, int *nr);

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

int **pfGetTexFormat**(const pfTexture *tex, int format);

void **pfTexFilter**(pfTexture *tex, int filt, int type);

int **pfGetTexFilter**(const pfTexture *tex, int filt);

void **pfTexRepeat**(pfTexture *tex, int wrap, int type);

int **pfGetTexRepeat**(const pfTexture *tex, int wrap);

void **pfTexBorderColor**(pfTexture* tex, pfVec4 clr);

void **pfGetTexBorderColor**(pfTexture* tex, pfVec4 *clr);

void **pfTexBorderType**(pfTexture* tex, int type);

int **pfGetTexBorderType**(pfTexture* tex);

void **pfTexSpline**(pfTexture *tex, int type, pfVec2 *pts, float clamp);

void **pfGetTexSpline**(const pfTexture *tex, int type, pfVec2 *pts, float *clamp);

void **pfTexDetail**(pfTexture *tex, int level, pfTexture *detail);

void **pfGetTexDetail**(const pfTexture *tex, int *level, pfTexture **detail);

pfTexture * **pfGetTexDetailTex**(const pfTexture *tex);
void **pfDetailTexTile**(pfTexture *tex, int j, int k, int m, int n, int scram);
void **pfGetDetailTexTile**(const pfTexture *tex, int *j, int *k, int *m, int *n, int *scram);
void **pfTexList**(pfTexture *tex, pfList *list);
pfList * **pfGetTexList**(const pfTexture *tex);
void **pfTexFrame**(pfTexture *tex, float frame);
float **pfGetTexFrame**(const pfTexture *tex);
void **pfTexLoadMode**(pfTexture *tex, int mode, int val);
int **pfGetTexLoadMode**(const pfTexture *tex, int mode);
void **pfTexLevel**(pfTexture *tex, int level, pfTexture *ltex);
pfTexture * **pfGetTexLevel**(pfTexture *tex, int level);
void **pfTexLoadOrigin**(pfTexture *tex, int which, int xo, int yo);
void **pfGetTexLoadOrigin**(pfTexture *tex, int which, int *xo, int *yo);
void **pfTexLoadSize**(pfTexture *tex, int xs, int ys);
void **pfGetTexLoadSize**(const pfTexture *tex, int *xs, int *ys);
void **pfApplyTex**(pfTexture *tex);
void **pfFormatTex**(pfTexture *tex);
void **pfLoadTex**(pfTexture *tex);
void **pfTexLoadImage**(pfTexture* tex, uint *image);
uint * **pfGetTexLoadImage**(const pfTexture* tex);
void **pfLoadTexLevel**(pfTexture *tex, int level);
int **pfLoadTexFile**(pfTexture *tex, char *fname);
void **pfSubloadTex**(pfTexture* tex, int source, uint *image, int xsrc, int ysrc, int xdst, int ydst, int xsize, int ysize);
void **pfSubloadTexLevel**(pfTexture* tex, int source, uint *image, int xsrc, int ysrc, int xdst, int ydst, int xsize, int ysize, int level);
void **pfFreeTexImage**(pfTexture *tex);
void **pfIdleTex**(pfTexture *tex);
int **pfIsTexLoaded**(const pfTexture *tex);

int **pfIsTexFormatted**(const pfTexture *tex);

pfTexture* **pfGetCurTex**(void);

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfTexture** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfTexture**. Casting an object of class **pfTexture** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfTexture** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

tex identifies a pfTexture.

DESCRIPTION

pfNewTex creates and returns a handle to a pfTexture. *arena* specifies a malloc arena out of which the pfTexture is allocated or **NULL** for allocation off the process heap. pfTextures can be deleted with **pfDelete**.

pfGetTexClassType returns the **pfType*** for the class **pfTexture**. The **pfType*** returned by **pfGetTexClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfTexture**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfLoadTexFile opens and loads the IRIS image file specified by *file*, using the global search paths set up by **pfFilePath** and the environment variable **PFPATH** to find *file*. The loaded image is then formatted and used as the image for *tex*. Image memory is allocated out of the malloc arena in which the pfTexture was created. A return of 0 indicates failure, 1 success.

pfLoadTexFile also sets the name of *tex* to the pathname of the file that was loaded (see **pfTexName**).

pfTexImage sets the image used by *tex*. *image* is an array of 4-byte words containing the texel data. **pfTexImage** only copies the pointer to *image* and does not make a separate copy of the texture data. *image* should be a pointer returned from **pfMalloc** so the image may be properly reference counted (see **pfMalloc** and **pfObject**).

The texture image is loaded from left to right, bottom to top. Texels must be aligned on long word boundaries, so rows must be byte-padded to the end of each row to get the proper alignment. *comp* is the number of 8-bit components per image pixel. 1, 2, 3, and 4 component textures are supported. *ns*, *nt*, *nr* are the number of texels in the s, t, and r dimensions of *image*. See the IRIS GL **texdef(3g)** or the OpenGL **glTexImage(3g)** man pages for a more detailed description of texture formats. Note that the ordering of color components in the image array is reversed between IRIS GL and OpenGL. **pfGetTexImage** returns the texture image parameters of *tex*.

pfFreeTexImage frees the texture image memory associated with *tex* after the next **pfApplyTex** is called if the image's reference count is 0. The texture image memory should be allocated by **pfMalloc** and may be shared between multiple pfTextures. However, one should take care with **pfFreeTexImage** if *tex* is to be used in multiple IRIS GL or OpenGL windows. If the image is freed on the first **pfApplyTex** then it will not be around when *tex* is applied in a second window. In this case the image should be freed only when *tex* has been applied in all windows.

pfTexName assigns the character string *name* to *tex*. Names are useful for sharing textures in order to optimize texture memory usage. A pfTexture's name is also set by **pfLoadTexFile** which sets the name to the pathname of the file that was loaded. **pfGetTexName** returns the name of *tex*.

pfTexFormat specifies how the texture image memory associated with *tex* is formatted by **pfFormatTex**. *format* is a symbolic token specifying which format to set and is **PFTEX_INTERNAL_FORMAT**, **PFTEX_EXTERNAL_FORMAT**, or **PFTEX_SUBLOAD_FORMAT**. *type* is a symbolic token specifying the format type appropriate to *format*. The tokens for **PFTEX_EXTERNAL_FORMAT** describe how the data in the image array is packed and may be one of **PFTEX_PACK_8** for 8 bit components in packed bytes (the default) or **PFTEX_PACK_16** for images presented as 16bit components. The tokens for **PFTEX_INTERNAL_FORMAT** include the component names use in the format and number of bits per component in the format. Internal formats with fewer bits per texel can have faster performance. The internal formats, and the number of bits per texel for each, are:

PFTEX_IA_8	*16-bit* texels
PFTEX_RGB_5	*16-bit* texels
PFTEX_RGB_4	*16-bit* texels
PFTEX_RGBA_4	*16-bit* texels
PFTEX_I_12A_4	*16-bit* texels
PFTEX_IA_12	*24-bit* texels
PFTEX_RGBA_8	*32-bit* texels
PFTEX_RGB_12	*48-bit* texels
PFTEX_RGBA_12	*48-bit* texels
PFTEX_I_16	*16-bit* texels

There are additional boolean format options:

PFTEX_DETAIL_TEXTURE
specifies that the texture is to be used as a detail texture. Once a texture has been specified to be a detail texture, it can no longer be used as a base texture. Calling **pfApplyTex** on a detail texture will bind the texture to the detail target, not the base target.

PFTEX_SUBLOAD_FORMAT
is a boolean mode that specifies that texture is to be able to be loaded in pieces, if supported on the current machine. *type* should be either **PF_ON** or **PF_OFF**. If an image has been assigned to the pfTexture, it will be automatically downloaded upon formatting. This type of texture may be formatted with a NULL image, in which case, no image is automatically downloaded upon formatting. If the texture is swapped out of hardware texture memory, the image will not be automatically restored upon pfApplyTex unless the **PFTEX_BASE_AUTO_SUBLOAD** mode has been specified for the **pfTexLoadMode**. and can always be explicitly reloaded with **pfLoadTex**. This format also specifies that all loads of the texture will use the origin, size and image specified by **pfTexLoadOrigin**, **pfTexLoadSize**, and **pfTexLoadImage**. If the current graphics hardware configuration cannot support texture subloading, this mode will be ignored. In IRIS GL, this requires a non-MIPmapped texture and support for **subtexload**. See the **subtexload(3g)** IRIS GL man page for more information. In OpenGL, this requires the **EXT_subtexture** extension; see the **EXT_subtexture** section of the OpenGL **glIntro(3g)** reference page for more information. If **PFTEX_SUBLOAD_FORMAT** is enabled, **pfFreeTexImage** should not be used as long as the texture is in use.

PFTEX_FAST_DEFINE
is a boolean IRIS GL mode to share user image data with GL - no GL copy is made. *type* should be either **PF_ON** or **PF_OFF**. In IRIS GL, this requires a non-MIPmapped texture. If **PFTEX_FAST_DEFINE** is enabled, **pfFreeTexImage** should not be used as

long as the texture is in use. This mode is now redundant with **PFTEX_SUBLOAD_FORMAT**.

See the IRIS GL **texdef(3g)** and **subtexload(3g)** man pages and the OpenGL **glTexImage2D** and **glTexSubImage** man pages for a description of texture formats and corresponding texture loading behavior. **pfGetTexFormat** returns the format mode corresponding to *format*.

pfTexRepeat specifies a texture coordinate repeat function. *wrap* is a symbolic token that specifies which texture coordinate(s) are affected by the repeat function and is one of **PFTEX_WRAP**, **PFTEX_WRAP_R**, **PFTEX_WRAP_S**, or **PFTEX_WRAP_T**. *mode* is a symbolic token that specifies how texture coordinates outside the range 0.0 through 1.0 are handled. *mode* token values may be one of: **PFTEX_REPEAT** or **PFTEX_CLAMP**. The default texture repeat function is **PFTEX_REPEAT** for all texture coordinates. **pfGetTexRepeat** returns the texture coordinate repeat function corresponding to *wrap*. See the IRIS GL **texdef(3g)** man page for a description of texture repeat.

pfTexFilter sets a filter function used by *tex*. *type* may be one of **PFTEX_MINFILTER**, **PFTEX_MAGFILTER**, **PFTEX_MAGFILTER_ALPHA**, or **PFTEX_MAGFILTER_COLOR**. *filter* is a combination of bitmask tokens which specify a particular minification or magnification filter. Filters may be partially specified, in which case, IRIS Performer will use defaults based on performance considerations for the current graphics platform. **PFTEX_FAST** can be included in a texture filter and may cause a slightly different filter to be substituted in texture application for fast performance on the current graphics platform. All filters may include basic interpolation tokens:

PFTEX_POINT
PFTEX_LINEAR
PFTEX_BILINEAR
PFTEX_TRILINEAR
PFTEX_QUADLINEAR - for 3D texture only.
PFTEX_LEQUAL, **PFTEX_GEQUAL** - currently only supported in IRIS GL.

Additionally, filters may specify additional minification or magnification functions.

Texture Filter Table

Filter	PFTEX_ tokens
PFTEX_MINFILTER	**PFTEX_MIPMAP** **PFTEX_BICUBIC** (IRIS GL only) **PFTEX_BICUBIC_LEQUAL** (IRIS GL only) **PFTEX_BICUBIC_GEQUAL** (IRIS GL only)
PFTEX_MAGFILTER	**PFTEX_DETAIL_LINEAR** **PFTEX_MODULATE**

	PFTEX_ADD
	PFTEX_DETAIL_COLOR
	PFTEX_DETAIL_ALPHA
PFTEX_MAGFILTER_ALPHA	**PFTEX_DETAIL**
	PFTEX_MODULATE
	PFTEX_ADD
	PFTEX_DETAIL_ALPHA
	PFTEX_SHARPEN
	PFTEX_SHARPEN_ALPHA
PFTEX_MAGFILTER_COLOR	**PFTEX_DETAIL**
	PFTEX_MODULATE
	PFTEX_ADD
	PFTEX_DETAIL_COLOR
	PFTEX_SHARPEN_COLOR
	PFTEX_SHARPEN

For convenience, there are compound tokens that include the most of usual filter combinations:

PFTEX_MIPMAP_POINT
PFTEX_MIPMAP_LINEAR
PFTEX_MIPMAP_BILINEAR
PFTEX_MIPMAP_TRILINEAR
PFTEX_MIPMAP_QUADLINEAR
PFTEX_MAGFILTER_COLOR
PFTEX_MAGFILTER_ALPHA
PFTEX_SHARPEN_COLOR
PFTEX_SHARPEN_ALPHA
PFTEX_DETAIL_COLOR
PFTEX_DETAIL_ALPHA
PFTEX_DETAIL_LINEAR
PFTEX_MODULATE_DETAIL
PFTEX_ADD_DETAIL
PFTEX_BICUBIC_LEQUAL
PFTEX_BICUBIC_GEQUAL
PFTEX_BICUBIC_LEQUAL
PFTEX_BICUBIC_GEQUAL
PFTEX_BILINEAR_LEQUAL
PFTEX_BILINEAR_GEQUAL

If the desired filter requires support that is not present on the current graphics platform, that part of the specified filter will be ignored. See the IRIS GL **texdef(3g)** or OpenGL **glTexImage(3g)** mans page for

descriptions of texture filter types. The default filter types are: magfilter = **PFTEX_BILINEAR**, minfilter = **PFTEX_MIPMAP_TRILINEAR** for Reality Engine and **PFTEX_MIPMAP_LINEAR** otherwise. **pfGetTexFilter** returns the filter type of *filter*.

Textures are permitted to have explicit borders. By default, these borders are not enabled. Texture borders can be enabled by setting a texture border type with **pfTexBorderType** where *type* is one of the following tokens: **PFTEX_BORDER_NONE**, **PFTEX_BORDER_COLOR**, or **PFTEX_BORDER_TEXEL**. The default texture border type is **PFTEX_BORDER_NONE**. If **PFTEX_BORDER_COLOR** is specified, the corresponding texture border color may be set with **pfTexBorderColor**. The default texture border color is black. If **PFTEX_BORDER_TEXEL** borders are enabled on a pfTexture, it is assumed that the corresponding image for that pfTexture includes the border texels and the corresponding size of the pfTexture also includes the border texels. **pfGetTexBorderType** will return the texture border type and **pfGetTexBorderColor** will return the texture border color. If the current graphics hardware configuration does not support the selected border type, it will be ignored. Texel borders are only available in OpenGL operation. Texture borders should be used only when there is strong motivation and with extreme caution. Texel borders can add considerable texture storage requirements to the pfTexture and cause subsequent performance degradations.

pfTexDetail sets *detail* as the detail texture of *tex* or disables detailing of *tex* if *detail* is **NULL**. The level of magnification between the base texture and detail texture is a non-positive number specified by *level* and may be **PFTEX_DEFAULT**. The detail texture is replicated as necessary to create a resulting texture that is 2^**level** times the size of the base (level 0) texture. The default tex level is four which creates a 16:1 mapping from detail texels to base texels. **pfTexDetail** will also set the magnification filter of *tex* to **PFTEX_MODULATE_DETAIL** or **PFTEX_BILINEAR** if detailing is enabled or disabled respectively. See the IRIS GL **texdef** and OpenGL **glDetailTexFuncSGIS** man pages for a description of detail texture filters and splines. For use with OpenGL, the OpenGL **GL_SGIS_detail_texture** extension is required. Once a texture has been specified to be a detail texture, it can no longer be used as a base texture. Calling **pfApplyTex** on a detail texture will bind the texture to the detail target, not the base target. Please also see the additional notes on using detail textures below.

pfGetTexDetail returns the detail texture of *tex* in *detail* or **NULL** if the texture is not detailed, and the detail texture level in *level*. The level of magnification at which detail is actually applied is controlled by **pfTexSpline**. **pfGetTexDetailTex** returns the detail texture of *tex*, or **NULL** if the texture is not detailed.

pfDetailTexTile is provided for compatibility with previous versions of IRIS GL IRIS Performer. It sets the tiling parameters of detail texture *detail*. *detail* is interpreted as a collection of *j* X *k* subimages which are applied to an *m* X *n* block of texels from the base texture. *scram* is a flag which enables or disables detail texture scrambling, but is currently ignored. Default values are $j = k = m = n = 4$, $scram = 0$. There corresponding detail texture level for pfTexDetail can be calculated as follows:

> IRIS GL detail textures are required to be of size 256x256 and the *i* and *j* are fixed at 4, which implies that the subtiles are of size 64x64 texels.

The magnification level *L* must satisfy *m* = *n* = 64 / 2^*L*, which implies that the maximum IRIS GL allowable detail level is 6. If you have an IRIS GL *m*, you can compute the detail level *L*:

```
L = Log base 2 (64 / m) = 6 - (log(m) / M_LN2)
```

See the IRIS GL **texdef(3g)** man page and IRIS GL Graphics Library Programming Guide for a description of IRIS GL detail texture tiling. **pfGetDetailTexTile** returns the tiling parameters of *detail*, and again, is only provided for compatibility with previous versions of IRIS GL IRIS Performer.

pfTexLevel sets a minification or magnification texture *ltex* for the level *level* for the base texture *tex*. If *level* is positive, it is taken to be a minification level and *ltex* is made the *level*th MIPmap level for the base texture. If *level* is zero or negative, *ltex* is taken to be a detail texture whose corresponding magnification level will be -*level*. **pfGetTexLevel** will return the texture for the specified *level*.

pfTexSpline sets the parameters of a cubic interpolating spline used in certain magnification filters. *type* is a symbolic token specifying a particular filter spline and is either **PFTEX_SHARPEN_SPLINE** or **PFTEX_DETAIL_SPLINE** which correspond to magfilters for sharpen (**PFTEX_SHARPEN**) and detail texture (**PFTEX_MODULATE_DETAIL** or **PFTEX_ADD_DETAIL**) respectively. *pts* is an array of pfVec2 of length 4 which specifies the control points of the filter spline. A control point is of the form (-LOD, scale). The specified LOD is negative since it indicates a magnification LOD. The spline is clamped to *clamp*. If *clamp* is **PFTEX_DEFAULT**, the spline will be automatically clamped to its maximum value. If *clamp* is zero, no clamping will be done. See the notes below on compatibility of texture splines with previous versions of IRIS GL IRIS Performer. See the IRIS GL **texdef** and OpenGL **glDetailTexFuncSGIS** and **glSharpenTexFuncSGIS** man pages for a description of filter splines. **pfGetTexSpline** copies the spline parameters of the filter designated by *type* into *pts* and *clamp*. Please also see the additional notes on using detail textures below.

pfTexList sets a pfList of pfTexture*, *list*, on the pfTexture, *tex*. **pfGetTexList** returns the texture list. **pfTexFrame** selects a frame from the texture list of *tex* upon **pfApplyTex**. The default frame value is (-1) which selects the base texture. **pfGetTexFrame** returns the current pfTexture frame. **pfTexList** and **pfTexFrame** together provide a mechanism for doing texture animations or managing multiple textures on geometry. The base pfTexture is applied to the geometry, but different pfTextures from the set in the list are selected based on the frame value in the base texture. The **PFTEX_LOAD_LIST** mode to **pfTexLoadMode** controls how textures replace previous texture from the same list for efficient hardware texture memory management.

pfFormatTex creates a GL/hardware ready texture for the specified pfTexture, *tex*. All pfTexture parameters are take into account. Changing any parameter on a pfTexture will cause it to need to be reformatted. **pfIsTexFormatted** will return the formatted state of the texture. **pfGetGLHandle** will return the handle to the resulting GL texture, valid only for the current GL context.

pfLoadTex downloads the specified texture source to graphics hardware texture memory allocated to *tex*. Repeated calls to **pfLoadTex** will reload the image specified by **pfTexLoadImage** or else the main texture image set by **pfTexImage** or **pfLoadTexFile** down into the graphics subsystem, allowing the contents of the texture image to be changed dynamically. If a reformatting of the texture is required, **pfFormatTex** will be called automatically. In IRIS GL operation, only **PFTEX_SUBLOAD_FORMAT** textures can have their images changed without expensive formatting. If the texture is of format **PFTEX_SUBLOAD_FORMAT** and the current graphics hardware configuration supports texture subloading, then the origin and size specified with **pfTexLoadOrigin** and **pfTexLoadSize** will be used. **pfIsTexLoaded** will return whether or not the specified pfTexture is currently resident in hardware texture memory. However, **pfIsTexLoaded** will not reflect whether or not additional changes made to the contents of the texture image have been downloaded. **pfLoadTexLevel** will load a specific level of the pfTexture and is available in OpenGL only. **pfLoadTex** and **pfLoadTexLevel** change the current pfTexture to be undefined.

pfSubloadTex downloads a specified texture source to graphics hardware memory allocated to *tex* according to the specified source, image, origin, destination, and size which may be different than the load parameters in *tex*. This routine will not change any of these load parameters for future texture loads. If the current graphics hardware configuration cannot support texture subloading, this command will have no effect. **pfSubloadTexLevel** will load a specific level of the pfTexture and is available in OpenGL only. **pfSubloadTex** and **pfSubloadTexLevel** change the current pfTexture to be undefined.

pfTexLoadImage can be used to update a location for texels to be downloaded from without causing a reformatting of the pfTexture and without loosing the main image pointer on the pfTexture. This specified *image* pointer will then be used for texture downloads triggered by **pfApplyTex** and **pfLoadTex**. If *image* is NULL, then texture downloads will go back to using the main image pointer on the pfTexture, set through **pfTexImage** or **pfLoadTexFile**. **pfGetTexLoadImage** will return the previously set load image of *tex*.

Portions of a texture images may be updated by specifying a **pfTexLoadOrigin** and **pfTexLoadSize**. **pfTexLoadOrigin** sets the origin of the texture image source or destination, according to *which*. **pfTexLoadSize** sets the texture area size, in texels, that is to be downloaded. These settings will affect future texture loads that are triggered by **pfApplyTex** and **pfLoadTex**. **pfGetTexLoadSize** will return the previously set load size of *tex*. **pfGetTexLoadOrigin** will return the source or destination load origin, as specified by *which* of *tex*.

pfTexLoadMode sets parameters that configure texture downloading, specified by *mode* to *val*. *mode* may be one of **PFTEX_LOAD_SOURCE**, **PFTEX_LOAD_BASE**, or **PFTEX_LOAD_LIST**. Values for the **PFTEX_LOAD_SOURCE** select the source for the texture image data and may be one of **PFTEX_IMAGE** to select the image specified by **pfTexImage**, **PFTEX_VIDEO** for video texture, or **PFTEX_SOURCE_FRAMEBUFFER**. The default texture load source is **PFTEX_SOURCE_IMAGE**. Texture sources of **PFTEX_SOURCE_VIDEO** and **PFTEX_SOURCE_FRAMEBUFFER** also require the **PFTEX_SUBLOAD_FORMAT** and set it automatically. The **PFTEX_LOAD_BASE** mode controls how

base textures are loaded. The default, **PFTEX_BASE_APPLY** will do a load of the specified texture source when the texture is dirty upon a **pfApplyTex** of the pfTexture. The **PFTEX_BASE_AUTO_SUBLOAD** will automatically replace the texture from the specified texture source upon every call to **pfApplyTex**. The **PFTEX_LOAD_LIST** mode controls how textures from the texture list are loaded. New selections from the texture list may be loaded in the following ways: **PFTEX_LIST_APPLY**, the default, will apply the selected pfTexture from the texture list; **PFTEX_LIST_AUTO_IDLE** will idle the previous pfTexture selected from the list so that its graphics texture memory will be freed. If the base texture is formatted with **PFTEX_SUBLOAD_FORMAT**, **PFTEX_LIST_SUBLOAD** will replace the texture image of the base texture with the texture image from the selected pfTexture of the texture list. **PFTEX_LIST_AUTO_SUBLOAD** re-uses the hardware texture memory of the base texture and so is the most efficient means of sharing graphics memory amongst pfTextures. However, **PFTEX_LIST_SUBLOAD** is not available on all graphics platforms (see notes below) and it requires that the pfTextures be identical in number of components and formats. **PFTEX_LIST_AUTO_SUBLOAD** will obey the origin and size set by **pfTexLoadOrigin** and **pfTexLoadSize**. **pfGetTexLoadMode** will return the value of *mode* for *tex*.

pfApplyTex makes *tex* the current texture. When texturing is enabled (see below), this texture will be applied to all geometry drawn after **pfApplyTex** is called. Only one pfTexture may be active at a time although many may be defined. If formatting or downloading of the texture is required at the time of the call to **pfApplyTex**, **pfFormatTex** and **pfLoadTex** will be called automatically. Modifications to *tex*, such as changing the filter type, will not have effect until **pfApplyTex** is called with *tex*. **pfGetCurTex** returns the currently active pfTexture.

pfApplyTex will automatically apply the detail texture associated with *tex* and will disable detail texturing if *tex* has no associated detail texture.

For geometry to be textured, the following must be true:

1. Texturing must be enabled: **pfEnable**(**PFEN_TEXTURE**)
2. A pfTexEnv must be applied: **pfApplyTEnv**
3. A pfTexture must be applied: **pfApplyTex**
4. Geometry must have texture coordinates: **pfGSetAttr**, **PFGS_TEXCOORD2**

The texture state element is identified by the **PFSTATE_TEXTURE** token. Use this token with **pfGStateAttr** to set the texture of a pfGeoState and with **pfOverride** to override subsequent texture changes:

Example 1:

```
/* Apply texture environment to be used by textured geometry */
pfApplyTEnv(tev);

/* Set up textured pfGeoState */
```

```
pfGStateMode(gstate, PFSTATE_ENTEXTURE, PF_ON);
pfGStateAttr(gstate, PFSTATE_TEXTURE, tex);

/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/* Set texture coordinate array. 'gset' is non-indexed */
pfGSetAttr(gset, PFGS_TEXCOORD2, PFGS_PER_VERTEX, tcoords,NULL);

/* Draw textured gset */
pfDrawGSet(gset);
```

Example 2:

```
pfApplyTex(tex);

/* Override so that all textured geometry uses 'tex' */
pfOverride(PFSTATE_TEXTURE, PF_ON);
```

pfGetCurTex returns the current pfTexture or **NULL** if no pfTexture is active.

pfIdleTex and **pfIsTexLoaded** can help you efficiently manage hardware texture memory. **pfIdleTex** signifies that *tex* is no longer needed in texture memory and may be replaced by new textures. **pfIsTexLoaded** returns TRUE or FALSE depending on whether *tex* is already loaded in texture memory or not. With these two commands it is possible to implement a rudimentary texture paging mechanism.

pfApplyTex and **pfIdleTex** are display-listable commands. If a pfDispList has been opened by **pfOpenDList**, **pfApplyTex** and **pfIdleTex** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

NOTES

Since textures are an expensive hardware resource, the sharing of textures is highly recommended. For best performance on machines which support hardware texturing, all textures should fit in hardware texture memory. Otherwise, the GL must page textures from main memory into the graphics pipeline with a corresponding performance hit. For best performance and use of memory:

Use the **PFTEX_INTERNAL_FORMATS** that have 16bit texels.

Keep textures of size an even power of two since they are always rounded up to the next power of two for storage in hardware texture memory.

Share pfTextures across pfGeoStates and share pfGeoStates across pfGeoSets whenever possible.

Minimize the number of distinct detail and sharpen splines.

Check the graphics state statistics (see the **pfStats** man page) to see if hardware texture memory is being swapped. As an additional diagnostic, **pfIsTexLoaded** can be used in a pfGeoState call-back before applying a pfTexture to see if a load or swap will be required.

Detail texturing, filter splines, sharpen filtering, **pfIdleTex**, and **pfIsTexLoaded** are supported only on RealityEngine graphics systems.

For OpenGL operation, detail texturing requires the **GL_SGIS_detail_texture** OpenGL extension and the sharpen filter requires the **GL_SGIS_sharpen_texture** extension. **pfIsTexLoaded** requires the **EXT_texture_object** OpenGL extension. See the **EXT_texture_object** section of the OpenGL **glIntro(3g)** reference page for more information.

The spline representation for detailed and sharpened textures in IRIS GL IRIS Performer 2.0 differs from previous IRIS GL IRIS Performer releases. Previously, the LOD component of a spline point was positive and as of IRIS Performer 2.0, it should be negative. Additionally, there is a difference in the effect of control point values for IRIS GL and OpenGL detail texture splines. In IRIS GL, detail texels are mapped to the range of [-0.5,0.5] and in OpenGL, they are mapped to the range [-1.0,1.0]. IRIS Performer handles this difference internally by rescaling the specified detail texture spline for IRIS GL. This means that detail texture splines for previous IRIS GL versions of IRIS Performer need to have their control points scaled by 0.5. Sharpen texture splines should not be scaled. For compatibility, old IRIS GL spline representations are accepted - if the LOD components of the spline are non-negative, the spline is taken to be the old representations. New development should use the new representation. See the OpenGL **glDetailTexFuncSGIS(3g)** and **glSharpenTexFuncSGIS(3g)** man pages and specs for more information on these splines. If IRIS Performer detects an IRIS GL format spline, it will print the usage message:

IRIS GL spline specification is obsolete - use OpenGL style

Clamping of detail and sharpen splines on the RealityEngine is supported only in IRIS GL.

Using a single detail pfTexture on multiple base pfTextures with different detail levels will have a severe performance penalty in IRIS GL operation. Therefore, for IRIS GL operation, a detail pfTexture should always be used with the same detail level.

A texture source of **PFTEX_VIDEO** is supported only on RealityEngine graphics systems.

For IRIS GL operation, downloading of sub-textures in only supported on RealityEngine and VGX(T) graphics systems.

For IRIS GL operation, only **PFTEX_SUBLOAD_FORMAT** textures can have their images changed

without formatting. Such textures may not be MIPmapped.

BUGS

Specification of a non-zero source and destination for PFTEX_SOURCE_IMAGE textures is not supported under IRIS GL operation.

SEE ALSO

pfDelete, pfDispList, pfEnable, pfFilePath, pfGeoState, pfGetGLHandle, pfMalloc, pfObject, pfOverride, pfState, pfStats, pfTexEnv, tevbind, tevdef, texbind, texdef, glTexImage, glDetailTexFuncSGIS, glSharpenTexFuncSGIS

NAME

pfInitClock, pfGetTime, pfClockMode, pfGetClockMode, pfClockName, pfGetClockName, pfWrapClock – Initialize and query high resolution clock

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pid_t	**pfInitClock**(double time);
double	**pfGetTime**(void);
void	**pfClockMode**(int mode);
int	**pfGetClockMode**(void);
void	**pfClockName**(char *name);
const char *	**pfGetClockName**(char *name);
void	**pfWrapClock**(void);

DESCRIPTION

Most SGI platforms have high resolution hardware timers. The routines described in this man page provide a simple interface to these timers.

On the first call to **pfInitClock**, IRIS Performer finds the highest resolution clock available and initializes it, setting the initial time to *time* seconds. Subsequent calls simply reset the initial time.

pfGetTime returns a high resolution clock time in seconds that is relative to the initial time set by **pfInitClock**. It determines the highest resolution clock available and uses that clock in all subsequent calls. On Indy, Indigo, Indigo2, 4D/35, Power Series systems with IO3 boards, and Onyx, the resolution of the clock is submicrosecond. If the hardware does not support a high resolution counter, the time of day clock is used which typically has 10ms resolution (see below).

By default, processes forked after the first call to **pfInitClock** share the same clock. All such related processes receive the same time from **pfGetTime** and see the effects of any subsequent calls to **pfInitClock**.

Unrelated processes can similarly share a single clock by invoking **pfClockName** before the first call to **pfInitClock**. **pfClockName** allows a character string name to be associated with a clock before it is initialized. **pfClockName** has no effect after the clock has been initialized. **pfGetClockName** returns the name of the clock.

IRIS Performer periodically checks to see if the underlying hardware counter has wrapped by having an interval timer (see setitimer(2)) regularly invoke **pfWrapClock**. By default, this interval timer and the associated alarm signal are handled in a separate process created by **pfInitClock**. The wrapping behavior can be changed by calling **pfClockMode** before the first call to **pfInitClock**. A *mode* of **PFCLOCK_APPWRAP** causes wrapping to be handled in the process that called **pfInitClock**. In this

case, the SIGALRM signals generated by the interval timer are generated and handled in that process. Wrap checking can be disabled altogether with a *mode* of **PFCLOCK_NOWRAP**. In this case, the application should invoke **pfWrapClock** with sufficient frequency to avoid missing a cycle of the counter (typically the hardware counters wrap every 60-120 secs, but the R3000 Indigo's clock wraps every 500ms). **pfGetClockMode** returns the clock mode.

When wrapping is done in a separate process, only one process is spawned, regardless of the number of processes using the same clock. If the calling process has super-user privileges, both the spawned process and the kernel clock functions are forced to run on CPU 0. **pfInitClock** returns the process ID of the process handling clock wrapping and 0 otherwise. A value of -1 indicates failure.

NOTES

If none of the hardware clocks are available, the time of day clock is used, but with only 10ms resolution it is insufficient for many applications such as frame rate control. In this case, it may be useful to enable the fast clock (see ftimer(1)).

The pfDataPool file for clock data is created in the directory */usr/tmp* or in the directory indicated by the **PFTMPDIR** environment variable, if it is specified.

Processes specifying a clock of the same name, share that clock through a pfDataPool. With an unnamed clock, the pfDataPool is deleted on exit, but when using a named clock, the application must explicitly call **pfReleaseDPool** with the clock name to remove the pfDataPool.

SEE ALSO

fork, ftimer, setitimer, lboot

NAME

pfTransparency, pfGetTransparency – Set/get the transparency mode

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfTransparency**(int type);

int **pfGetTransparency**(void);

PARAMETERS

type is a symbolic constant and is one of:

PFTR_OFF
Disable transparency

PFTR_ON
IRIS Performer will use a default transparency mechanism depending on the machine being used.

PFTR_HIGH_QUALITY
IRIS Performer will use a transparency mechanism that provides the highest quality, not necessarily the fastest, transparency.

PFTR_FAST
IRIS Performer will use a transparency mechanism that provides the fastest, not necessarily the highest quality, transparency.

PFTR_BLEND_ALPHA
IRIS Performer will use the IRIS GL **blendfunction(3g)** or OpenGL **glBlendFunc(3g)** method of transparency.

PFTR_MS_ALPHA
IRIS Performer will use the IRIS GL **msalpha(3g)** or the OpenGL **glEnable(GL_SAMPLE_ALPHA_TO_ONE_SGIS)** method of transparency when multisampling is available and enabled. Source alpha values will be converted to 1.0 (full opacity) before writing to the framebuffer.

PFTR_MS_ALPHA_MASK
IRIS Performer will use the IRIS GL **msalpha(3g)** OpenGL **glEnable(GL_SAMPLE_ALPHA_TO_MASK_SGIS)** method of transparency when multisampling is enabled. Source alpha values will not be modified.

type may be OR-ed with **PFTR_NO_OCCLUDE** if transparent geometry is not to occlude other geometry.

DESCRIPTION

pfTransparency sets the transparency mode to *mode*. Enabling transparency is not enough to render transparent geometry. Geometry colors must have alpha values that are less than the maximum (alpha < 1 for **c4f** and alpha < 255 for **cpack**) in order to be transparent. When alpha is less than maximum, it defines the blending of geometry color with framebuffer color according to the following equation.

finalColor = alpha*geometryColor + (1–alpha)*colorInFramebuffer

In other words, alpha is the "inverse" of transparency.

The transparency mode value may be OR-ed with **PFTR_NO_OCCLUDE**. **PFTR_NO_OCCLUDE** disables writes to the depth buffer so that any geometry rendered after **pfTransparency** is called with this value will not modify the depth buffer and so will not be able to occlude any other geometry. Since you can "see-through" transparent geometry, this is a useful option when using **PFTR_BLEND_ALPHA** type transparency and you are unable to render transparent geometry back to front.

pfGetTransparency returns the current transparency mode.

The transparency mode state element is identified by the **PFSTATE_TRANSPARENCY** token. Use this token with **pfGStateMode** to set the transparency mode of a pfGeoState and with **pfOverride** to override subsequent transparency mode changes:

Example 1:

```
/* Set up transparent pfGeoState */
pfGStateMode(gstate, PFSTATE_TRANSPARENCY, PFTR_HIGH_QUALITY);

/* Attach gstate to gset */
pfGSetGState(gset, gstate);

/*
 * Draw transparent gset. 'gset' must have alpha values
 * that are < 1.0f for transparency to have effect.
 */
pfDrawGSet(gset);
```

Example 2:

```
/* Override transparency mode to PFTR_OFF */
pfTransparency(PFTR_OFF);
pfOverride(PFSTATE_TRANSPARENCY, PF_ON);
```

The **MS_ALPHA** transparency methods only work when the window is configured for multisampling. In this case alpha values are converted to a multisample mask, a "screen door" if you will, that allows the geometry color to only partially influence each pixel. This kind of transparency is most efficient when multisampling and has the important benefit of *not* requiring sorting of transparent geometry. **PFTR_BLEND_ALPHA** on the other hand actually blends the geometry color with what is already in the

framebuffer. Thus it requires the following for proper results:

1. Transparent geometry be rendered after opaque geometry.
2. Transparent geometry be rendered back to front.

pfTransparency is a display-listable command. If a pfDispList has been opened by **pfOpenDList**, **pfTransparency** will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

BUGS

pfTransparency modifies the **zwritemask** but does not restore it.

SEE ALSO

blendfunction, msalpha, msmask, glEnable, glSampleMaskSGIS, pfDispList, pfGeoState, pfState, zwritemask

NAME

pfNewType, pfGetTypeName, pfGetTypeParent, pfIsDerivedFrom, pfMaxTypes – pfType, data typing system

FUNCTION SPECIFICATION

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

pfType*      pfNewType(pfType *parent, char *name);
const char*  pfGetTypeName(void* type);
pfType*      pfGetTypeParent(pfType* type);
int          pfIsDerivedFrom(pfType* type, pfType *ancestor);
void         pfMaxTypes(int n);
```

DESCRIPTION

Objects derived from pfMemory have an associated pfType. An object's pfType* is returned by calling pfGetType.

pfGetTypeParent returns the parent class of the pfType. Note that Performer only uses single inheritance and the type system only maintains a single inheritance chain.

pfGetTypeName returns the name of the class.

pfIsDerivedFrom tests whether the pfType has *ancestor* somewhere in its inheritance ancestry. It returns TRUE if the ancestor was found, and FALSE otherwise.

The type system must be initialized in shared memory before other processes that will share the type system are created. When the type system is initialized by **pfInit**, each Performer class creates a type for itself. **pfNewType** allows an application to add additional types to the type system. *parent* specifies the parent class, or NULL if the class has no parents. *name* is the name of the class.

pfMaxTypes allows an application to increase the number of allowed types. This must be called before **pfInit** to have effect.

pfMemory also has two convenience functions **pfIsOfType**, **pfGetTypeName** that allow access to type information with a single call.

Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsDerivedFrom** or **pfIsOfType** to test the type of an object rather than to test for the strict equality of the **pfType***'s.

NOTES

pfTypes cannot be deleted.

SEE ALSO

pfMemory

NAME

pfStartVClock, pfStopVClock, pfInitVClock, pfGetVClockOffset, pfVClockOffset, pfGetVClock, pfVClockSync – Initialize and query vertical retrace clock

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfStartVClock**(void);

void **pfStopVClock**(void);

void **pfInitVClock**(int ticks);

int **pfGetVClockOffset**(void);

void **pfVClockOffset**(int offset);

int **pfGetVClock**(void);

int **pfVClockSync**(int rate, int offset);

DESCRIPTION

A pfVClock (Video Clock) is a clock which runs at the video retrace rate. There is one clock for each hardware graphics pipeline which runs at the video rate of that pipeline. A pipeline's video rate is defined by its video format (see **setmon**). To access the video clock a Graphics Library window must be opened and made current although the window does not need to be mapped to the display surface (**noport** in IRIS GL). The screen of the current window determines which pipe's video clock is accessed on multipipe machines.

pfStartVClock starts the video clock by enabling CPU interrupts from the graphics pipeline while **pfStopVClock** disables CPU interrupts. At this time, the video clock can be started/stopped only on VGX and VGXT graphics hardware. On other hardware such as RealityEngine, CPU interrupts are always enabled.

pfInitVClock sets the initial value of the video clock to *ticks*. Note that **pfInitVClock** does not set the video clock, rather it computes an offset which is added to the real video clock's value. This offset is unique to a given address space - **fork**ed processes each have their own offset while **sproc**ed processes share the offset. The offset computed by **pfInitVClock** is returned by **pfGetVClockOffset** and may be set directly with **pfVClockOffset**, simplifying clock synchronization across processes.

pfGetVClock returns the current video retrace count relative to the initial value set by **pfInitVClock**.

pfVClockSync puts the calling process to sleep until *count* modulo *rate* is equal to *offset*. For instance, if the count is 0, the *rate* is 10 and *offset* is 4, then **pfVClockSync** will return when the count is 4. Subsequent calls with the same values will return when the count is 14, 24, etc. *offset* must be a non-negative number less than *rate* and *rate* must be positive.

If the retrace count modulo *rate* is equal to *offset* at the time **pfVClockSync** is called, the caller will not

return immediately but will go to sleep until *rate* ticks later.

The following code fragment illustrates a true video clock which will wake up each vertical retrace period (subject to process priorities):

```
while (1)
{
    /* wait for next vertical retrace */
    pfVClockSync(1, 0);

    /* perform per-interval actions */
     :
}
```

NOTES

A GL window (which may be a **noport** window) must be open before any pfVClock routines are called.

The video clock count is *not* the **swapbuffers** count.

NAME

pfAddScaledVec2, pfAddVec2, pfAlmostEqualVec2, pfCombineVec2, pfCopyVec2, pfDistancePt2, pfDotVec2, pfEqualVec2, pfLengthVec2, pfNegateVec2, pfNormalizeVec2, pfScaleVec2, pfSetVec2, pfSqrDistancePt2, pfSubVec2 – Set and operate on 2-component vectors

FUNCTION SPECIFICATION

#include <Performer/pr.h>

#include <Performer/prmath.h>

void **pfAddScaledVec2**(pfVec2 dst, const pfVec2 v1, float s, const pfVec2 v2);

void **pfAddVec2**(pfVec2 dst, const pfVec2 v1, const pfVec2 v2);

int **pfAlmostEqualVec2**(const pfVec2 v1, const pfVec2 v2, float tol);

void **pfCombineVec2**(pfVec2 dst, float s1, const pfVec2 v1, float s2, const pfVec2 v2);

void **pfCopyVec2**(pfVec2 dst, const pfVec2 v);

float **pfDistancePt2**(const pfVec2 pt1, const pfVec2 pt2);

float **pfDotVec2**(const pfVec2 v1, const pfVec2 v2);

int **pfEqualVec2**(const pfVec2 v1, const pfVec2 v2);

float **pfLengthVec2**(const pfVec2 v);

void **pfNegateVec2**(pfVec2 dst, const pfVec2 v);

float **pfNormalizeVec2**(pfVec2 v);

void **pfScaleVec2**(pfVec2 dst, float s, const pfVec2 v);

void **pfSetVec2**(pfVec2 dst, float x, float y);

float **pfSqrDistancePt2**(const pfVec2 pt1, const pfVec2 pt2);

void **pfSubVec2**(pfVec2 dst, const pfVec2 v1, const pfVec2 v2);

```
typedef float pfVec2[2];
```

DESCRIPTION

Math functions for 2-component vectors. Most of these routines have macro equivalents.

pfSetVec2(dst, x, y): *dst*[0] = *x*, *dst*[1] = *y*. Macro equivalent is **PFSET_VEC2**.

pfCopyVec2(dst, v): *dst* = *v*. Macro equivalent is **PFCOPY_VEC2**.

pfNegateVec2(dst, v): *dst* = -*v*. Macro equivalent is **PFNEGATE_VEC2**.

pfAddVec2(dst, v1, v2): *dst = v1 + v2*. Sets *dst* to the sum of vectors *v1* and *v2*. Macro equivalent is **PFADD_VEC2**.

pfSubVec2(dst, v1, v2): *dst = v1 - v2*. Sets *dst* to the difference of *v1* and *v2*. Macro equivalent is **PFSUB_VEC2**.

pfAddScaledVec2(dst, v1, s, v2): *dst = v1 + s * v2*. Sets *dst* to the vector *v1* plus the vector *v2* scaled by *s*. Macro equivalent is **PFADD_SCALED_VEC2**.

pfScaleVec2(dst, s, v): *dst = s * v1*. Sets *dst* to the vector *v* scaled by *s*. Macro equivalent is **PFSCALE_VEC2**.

pfCombineVec2(dst, s1, v1, s2, v2): *dst = s1 * v1 + s2 * v2*. Sets *dst* to be the linear combination of *v1* and *v2* with scales *s1* and *s2*, respectively. Macro equivalent: **PFCOMBINE_VEC2**.

pfNormalizeVec2(v): *v = v* / length(*v*). Normalizes the vector *v* to have unit length and returns the original length of the vector.

pfDotVec2(v1, v2) = *v1* dot *v2* = *v1*[0] * *v2*[0] + *v1*[1] * *v2*[1] . Returns dot product of the vectors *v1* and *v2*. Macro equivalent is **PFDOT_VEC2**.

pfLengthVec2(v) = |v| = sqrt(*v* dot *v*). Returns length of the vector *v*. Macro equivalent is **PFLENGTH_VEC2**.

pfSqrDistancePt2(v1, v2) = (*v1* - *v2*) dot (*v1* - *v2*). Returns square of distance between two points *v1* and *v2*. Macro equivalent is **PFSQR_DISTANCE_PT2**.

pfDistancePt2(v1, v2) = sqrt((*v1* - *v2*) dot (*v1* - *v2*)). Returns distance between two points *v1* and *v2*. Macro equivalent is **PFDISTANCE_PT2**.

pfEqualVec2(v1, v2) = (*v1* == *v2*). Tests for strict component-wise equality of two vectors *v1* and *v2* and returns FALSE or TRUE. Macro equivalent is **PFEQUAL_VEC2**.

pfAlmostEqualVec2(v1, v2, tol). Tests for approximate component-wise equality of two vectors *v1* and *v2*. It returns FALSE or TRUE depending on whether the absolute value of the difference between each pair of components is less than the tolerance *tol*. Macro equivalent is **PFALMOST_EQUAL_VEC2**.

Routines can accept the same vector as source, destination, or as a repeated operand.

SEE ALSO

pfMatrix, pfVec3, pfVec4

NAME

pfAddScaledVec3, pfAddVec3, pfEqualVec3, pfAlmostEqualVec3, pfCombineVec3, pfCopyVec3, pfCrossVec3, pfDistancePt3, pfDotVec3, pfLengthVec3, pfNegateVec3, pfNormalizeVec3, pfScaleVec3, pfSetVec3, pfSqrDistancePt3, pfSubVec3, pfXformVec3, pfXformPt3, pfFullXformPt3 – Set and operate on 3-component vectors

FUNCTION SPECIFICATION

#include <Performer/pr.h>

#include <Performer/prmath.h>

void **pfAddScaledVec3**(pfVec3 dst, const pfVec3 v1, float s, const pfVec3 v2);

void **pfAddVec3**(pfVec3 dst, const pfVec3 v1, const pfVec3 v2);

int **pfEqualVec3**(const pfVec3 v1, const pfVec3 v2);

int **pfAlmostEqualVec3**(const pfVec3 v1, const pfVec3 v2, float tol);

void **pfCombineVec3**(pfVec3 dst, float s1, const pfVec3 v1, float s2, const pfVec3 v2);

void **pfCopyVec3**(pfVec3 dst, const pfVec3 v);

void **pfCrossVec3**(pfVec3 dst, const pfVec3 v1, const pfVec3 v2);

float **pfDistancePt3**(const pfVec3 pt1, const pfVec3 pt2);

float **pfDotVec3**(const pfVec3 v1, const pfVec3 v2);

float **pfLengthVec3**(const pfVec3 v);

void **pfNegateVec3**(pfVec3 dst, const pfVec3 v);

float **pfNormalizeVec3**(pfVec3 v);

void **pfScaleVec3**(pfVec3 dst, float s, const pfVec3 v);

void **pfSetVec3**(pfVec3 dst, float x, float y, float z);

float **pfSqrDistancePt3**(const pfVec3 pt1, const pfVec3 pt2);

void **pfSubVec3**(pfVec3 dst, const pfVec3 v1, const pfVec3 v2);

void **pfXformVec3**(pfVec3 dst, const pfVec3 v, const pfMatrix m);

void **pfXformPt3**(pfVec3 dst, const pfVec3 v, const pfMatrix m);

void **pfFullXformPt3**(pfVec3 dst, const pfVec3 v, const pfMatrix m);

```
typedef float pfVec3[3];
```

DESCRIPTION

Math functions for 3-component vectors. Most of these routines have macro equivalents.

pfSetVec3(dst, x, y, z): *dst*[0] = *x*, *dst*[1] = *y*, *dst*[2] = *z*. Macro equivalent is **PFSET_VEC3**.

pfCopyVec3(dst, v): *dst* = *v*. Macro equivalent is **PFCOPY_VEC3**.

pfNegateVec3(dst, v): *dst* = -*v*. Macro equivalent is **PFNEGATE_VEC3**.

pfAddVec3(dst, v1, v2): *dst* = *v1* + *v2*. Sets *dst* to the sum of vectors *v1* and *v2*. Macro equivalent is **PFADD_VEC3**.

pfSubVec3(dst, v1, v2): *dst* = *v1* - *v2*. Sets *dst* to the difference of *v1* and *v2*. Macro equivalent is **PFSUB_VEC3**.

pfScaleVec3(dst, s, v): *dst* = *s* * *v1*. Sets *dst* to the vector *v* scaled by *s*. Macro equivalent is **PFSCALE_VEC3**.

pfAddScaledVec3(dst, v1, s, v2): *dst* = *v1* + *s* * *v2*. Sets *dst* to the vector *v1* plus the vector *v2* scaled by *s*. Macro equivalent is **PFADD_SCALED_VEC3**.

pfCombineVec3(dst, s1, v1, s2, v2): *dst* = *s1* * *v1* + *s2* * *v2*. Sets *dst* to be the linear combination of *v1* and *v2* with scales *s1* and *s2*, respectively. Macro equivalent: **PFCOMBINE_VEC3**.

pfNormalizeVec3(v): *v* = *v* / length(*v*). Normalizes the vector *v* to have unit length and returns the original length of the vector.

pfCrossVec3(dst, v1, v2): *dst* = *v1* X *v2*. Sets *dst* to the cross-product of two vectors *v1* and *v2*.

pfXformVec3(dst, v, m): *dst* = *v4* * *m* (*v4*[i]=*v[i]* i=0, 1, 2; *v4*[3] = 0). Transforms *v* as a vector by the matrix *m*.

pfXformPt3(dst, v, m): *dst* = *v4* * *m* (*v4*[i]=*v[i]* i=0, 1, 2; *v4*[3] = 1). Transforms *v* as a point by the matrix *m* using the 4X3 submatrix.

pfFullXformPt3(dst, v, m): *dst* = *v4* * *m* (*v4*[i]=*v[i]* i=0, 1, 2; *v4*[3] = 1). Transforms *v* as a point by the matrix *m* using the full 4X4 matrix and scaling *dst* by the resulting w coordinate.

pfDotVec3(v1, v2) = *v1* dot *v2* = *v1*[0] * *v2*[0] + *v1*[1] * *v2*[1] + *v1*[2] * *v2*[2]. Returns dot product of the vectors *v1* and *v2*. Macro equivalent is **PFDOT_VEC3**.

pfLengthVec3(v) = |v| = sqrt(*v* dot *v*). Returns length of the vector *v*. Macro equivalent is

PFLENGTH_VEC3.

pfSqrDistancePt3(v1, v2) = (*v1* - *v2*) dot (*v1* - *v2*). Returns square of distance between two points *v1* and *v2*. Macro equivalent is **PFSQR_DISTANCE_PT3**.

pfDistancePt3(v1, v2) = sqrt((*v1* - *v2*) dot (*v1* - *v2*)). Returns distance between two points *v1* and *v2*. Macro equivalent is **PFDISTANCE_PT3**.

pfEqualVec3(v1, v2) = (*v1* == *v2*). Tests for strict component-wise equality of two vectors *v1* and *v2* and returns FALSE or TRUE. Macro equivalent is **PFEQUAL_VEC3**.

pfAlmostEqualVec3(v1, v2, tol). Tests for approximate component-wise equality of two vectors *v1* and *v2*. It returns FALSE or TRUE depending on whether the absolute value of the difference between each pair of components is less than the tolerance *tol*. Macro equivalent is **PFALMOST_EQUAL_VEC3**.

Routines can accept the same vector as source, destination, or as a repeated operand.

SEE ALSO

pfMatrix, pfVec2, pfVec4

NAME

pfAddScaledVec4, pfAddVec4, pfAlmostEqualVec4, pfCombineVec4, pfCopyVec4, pfDistancePt4, pfDotVec4, pfEqualVec4, pfLengthVec4, pfNegateVec4, pfNormalizeVec4, pfScaleVec4, pfSetVec4, pfSqrDistancePt4, pfSubVec4, pfXformVec4 – Set and operate on 4-component vectors

FUNCTION SPECIFICATION

#include <Performer/pr.h>

#include <Performer/prmath.h>

void **pfAddScaledVec4**(pfVec3 dst, const pfVec3 v1, float s, const pfVec3 v2);

void **pfAddVec4**(pfVec4 dst, const pfVec4 v1, const pfVec4 v2);

int **pfAlmostEqualVec4**(pfVec4 v1, const pfVec4 v2, float tol);

void **pfCombineVec4**(pfVec4 dst, float s1, const pfVec4 v1, float s2, const pfVec4 v2);

void **pfCopyVec4**(pfVec4 dst, const pfVec4 v);

float **pfDistancePt4**(const pfVec4 pt1, const pfVec4 pt2);

float **pfDotVec4**(const pfVec4 v1, const pfVec4 v2);

int **pfEqualVec4**(const pfVec4 v1, const pfVec4 v2);

float **pfLengthVec4**(const pfVec4 v);

void **pfNegateVec4**(pfVec4 dst, const pfVec4 v);

float **pfNormalizeVec4**(pfVec4 v);

void **pfScaleVec4**(pfVec4 dst, float s, const pfVec4 v);

void **pfSetVec4**(pfVec4 dst, float x, float y, float z, float w);

float **pfSqrDistancePt4**(const pfVec4 pt1, const pfVec4 pt2);

void **pfSubVec4**(pfVec4 dst, const pfVec4 v1, const pfVec4 v2);

void **pfXformVec4**(pfVec4 dst, const pfVec4 v, const pfMatrix m);

```
typedef float pfVec4[4];
```

DESCRIPTION

Math functions for 4-component vectors.

pfSetVec4(dst, x, y, z, w): $dst[0] = x$, $dst[1] = y$, $dst[2] = z$, $dst[3] = w$. Macro equivalent is **PFSET_VEC4**.

pfCopyVec4(dst, v): $dst = v$. Macro equivalent is **PFCOPY_VEC4**.

pfNegateVec4(dst, v): *dst* = -*v*. Macro equivalent is **PFNEGATE_VEC4**.

pfAddVec4(dst, v1, v2): *dst* = *v1* + *v2*. Sets *dst* to the sum of vectors *v1* and *v2*. Macro equivalent is **PFADD_VEC4**.

pfSubVec4(dst, v1, v2): *dst* = *v1* - *v2*. Sets *dst* to the difference of *v1* and *v2*. Macro equivalent is **PFSUB_VEC4**.

pfScaleVec4(dst, s, v): *dst* = *s* * *v1*. Sets *dst* to the vector *v* scaled by *s*. Macro equivalent is **PFSCALE_VEC4**.

pfAddScaledVec4(dst, v1, s, v2): *dst* = *v1* + *s* * *v2*. Sets *dst* to the vector *v1* plus the vector *v2* scaled by *s*. Macro equivalent is **PFADD_SCALED_VEC4**.

pfCombineVec4(dst, s1, v1, s2, v2): *dst* = *s1* * *v1* + *s2* * *v2*. Sets *dst* to be the linear combination of *v1* and *v2* with scales *s1* and *s2*, respectively. Macro equivalent is **PFCOMBINE_VEC4**.

pfNormalizeVec4(v): *v* = *v* / length(*v*). Normalizes the vector *v* to have unit length and returns the original length of the vector.

pfXformVec4(dst, v, m): *dst* = *v* * *m*. Transforms *v* by the matrix *m*.

pfDotVec4(v1, v2) = *v1* dot *v2* = *v1*[0] * *v2*[0] + *v1*[1] * *v2*[1] + *v1*[2] * *v2*[2] + *v1*[3] * *v2*[3]. Returns dot product of the vectors *v1* and *v2*. Macro equivalent is **PFDOT_VEC4**.

pfLengthVec4(v) = |v| = sqrt(*v* dot *v*). Returns length of the vector *v*. Macro equivalent is **PFLENGTH_VEC4**.

pfSqrDistancePt4(v1, v2) = (*v1* - *v2*) dot (*v1* - *v2*). Returns square of distance between two points *v1* and *v2*. Macro equivalent is **PFSQR_DISTANCE_PT4**.

pfDistancePt4(v1, v2) = sqrt((*v1* - *v2*) dot (*v1* - *v2*)). Returns distance between two points *v1* and *v2*. Macro equivalent is **PFDISTANCE_PT4**.

pfEqualVec4(v1, v2) = (*v1* == *v2*). Tests for strict component-wise equality of two vectors *v1* and *v2* and returns FALSE or TRUE. Macro equivalent is **PFEQUAL_VEC4**.

pfAlmostEqualVec4(v1, v2, tol). Tests for approximate component-wise equality of two vectors *v1* and *v2*. It returns FALSE or TRUE depending on whether the absolute value of the difference between each pair of components is less than the tolerance *tol*. Macro equivalent is **PFALMOST_EQUAL_VEC4**.

Routines can accept the same vector as source, destination, or as a repeated operand.

SEE ALSO

pfMatrix, pfVec2, pfVec3

NAME

pfModelMat, pfGetModelMat, pfViewMat, pfGetViewMat, pfInvModelMat, pfGetInvModelMat, pfNearPixDist, pfGetNearPixDist, pfTexMat, pfGetTexMat – Set/get shadows of viewing, modeling, texturing, and inverse modeling matrices; set/get near plane pixel distance

FUNCTION SPECIFICATION

#include <Performer/pr.h>

void **pfModelMat**(pfMatrix mat);

void **pfGetModelMat**(pfMatrix mat);

void **pfViewMat**(pfMatrix mat);

void **pfGetViewMat**(pfMatrix mat);

void **pfInvModelMat**(pfMatrix mat);

void **pfGetInvModelMat**(pfMatrix mat);

void **pfNearPixDist**(float npd);

float **pfGetNearPixDist**(void);

void **pfTexMat**(pfMatrix mat);

void **pfGetTexMat**(pfMatrix mat);

DESCRIPTION

Certain eyepoint-dependent features require knowledge about the viewing and modeling matrices as well as knowledge about the mapping of eye to window coordinates. Current eyepoint-dependent features include pfLPointState, which renders **PFGS_POINTS** pfGeoSets as "light points" and pfSprite, which rotates geometry to face the viewer.

pfViewMat, **pfModelMat**, **pfTexMat** and **pfNearPixDist** set the current viewing matrix, modeling matrix, texture matrix, and distance to the near (view) plane in pixels. These routines in no way affect the graphics library or its transformation stack. They only specify values used internally by **libpr**. **libpf** applications do not need to call these routines unless they modify associated values through the graphics library instead of through **libpr** matrix routines, e.g., if the viewing transform is set with **pfLoadMatrix** instead of **pfChanViewMat**. **pfTexMat** is provided so **libpr** can change, then restore the current texture matrix without incurring the cost of querying the graphics library for the matrix value.

pfLPointState and pfSprite both require the viewing and modeling matrices. The inverse modeling matrix is automatically computed by IRIS Performer when necessary unless it is specified by the application with **pfInvModelMat**. pfLPointState also requires the distance, measured in pixels, from the eyepoint to the near plane of the viewing frustum. This value is required in order to render light points at their proper screen size. Use **pfNearPixDist** to set the "near pixel distance" and **pfGetNearPixDist** to get it.

pfViewMat makes *mat* the current viewing matrix. *mat* is a 4x4 homogeneous matrix which defines the

view coordinate system such that the upper 3x3 submatrix defines the coordinate system axes and the bottom vector defines the coordinate system origin. IRIS Performer defines the view direction to be along the positive Y axis and the up direction to be the positive Z direction, e.g., the second row of *mat* defines the viewing direction and the third row defines the up direction in world coordinates. *mat* must be orthonormal or results are undefined. **pfGetViewMat** copies the current viewing matrix into *mat*.

pfModelMat makes *mat* the current modeling matrix. **pfGetModelMat** copies the current modeling matrix into *mat*.

pfViewMat, **pfModelMat**, **pfTexMat**, **pfInvModelMat**, and **pfNearPixDist** are all display-listable commands. If a pfDispList has been opened by **pfOpenDList**, these commands will not have immediate effect but will be captured by the pfDispList and will only have effect when that pfDispList is later drawn with **pfDrawDList**.

SEE ALSO

pfLPointState, pfSprite, pfDispList

NAME

pfGetCurWSConnection, pfOpenWSConnection, pfSelectWSConnection, pfOpenScreen, pfCloseWSConnection, pfGetWSConnectionName, pfGetScreenSize, pfChooseFBConfig, pfChooseFBConfigData – Window system utility routines

FUNCTION SPECIFICATION

#include <Performer/pr.h>

pfWSConnection **pfGetCurWSConnection**(void);

pfWSConnection **pfOpenWSConnection**(const char *str, int shared);

void **pfSelectWSConnection**(pfWSConnection ws);

pfWSConnection **pfOpenScreen**(int screen, int shared);

void **pfCloseWSConnection**(pfWSConnection ws);

const char* **pfGetWSConnectionName**(pfWSConnection ws);

void **pfGetScreenSize**(int screen, int *x, int *y);

pfFBConfig **pfChooseFBConfig**(pfWSConnection ws, int screen, int *attr);

pfFBConfig **pfChooseFBConfigData**(void **dst, pfWSConnection ws, int screen, int *attr, void *arena);

```
/* typedef of X-based Performer Types */
typedef Display         *pfWSConnection;
typedef XVisualInfo      pfFBConfig;
typedef Window           pfWSWindow;
typedef Drawable         pfWSDrawable;

/* typedef of GL-based Performer Types */
#ifdef IRISGL
typedef int              pfGLContext;
#else /* OPENGL */
typedef GLXContext       pfGLContext;
#endif
```

PARAMETERS

ws identifies a pfWSConnection.

DESCRIPTION

These functions provide a single API for communicating with the window system that works across the IRIS GL, IRIS GL mixed model (GLX), and OpenGL/X environments. Window system independent types have been provided to match the X Window System types to provide complete portability between the IRIS GL and OpenGL/X windowing environments.

These routines communicate with the window system via a **pfWSConnection** to a window server. This connection is *per-process*. If a process tries to use the pfWSConnection of another process, bad things are likely to result. A process may use its pfWSConnection to open and communication with all windows on all screens managed by that window server. Typically, a given machine will have a single window server, even if they have multiple screens -- all screens are managed by the single server. Exceptions are the multiple-keyboard machines. A pfWSConnection has a default screen on which windows will be opened if no other screen is explicitly named (such as through **pfWinScreen**). This default screen is determined by the **DISPLAY** environment variable, and is screen 0 if the **DISPLAY** variable is unset.

A **pfWSWindow** is a window-server window. In OpenGL or IRIS GL mixed model (GLX), this will be an X window. In pure IRIS GL, this will be an IRIS GL window. By contrast, a **pfWSDrawable** is a window system primitive that may be connected to a GL drawing context. In IRIS GL, this is simple a window; however, in OpenGL, **pfWSDrawable** includes Pixmaps. **pfGLContext** is the GL context that is attached to the **pfWSDrawable**.

Windows have framebuffer resources associated with them, such as a zbuffer, stencil planes, and possibly multisample buffers. A framebuffer configuration can be created by **pfChooseFBConfig**, or **pfChooseWinFBConfig**, and is returned in a **pfFBConfig**.

pfGetCurWSConnection returns the current connection to the window system. If there is no current open connection, one is opened on the default display (using the **DISPLAY** environment variable).

pfOpenWSConnection opens the window server connection named by *str* and if *shared* is true, this connection will be shared with IRIS Performer and made the current libpr window system connection via a call to **pfSelectWSConnection**.

pfOpenScreen opens a local window server connection with the default screen of *screen*. If *screen* is (-1), the default screen as set by the **DISPLAY** environment variable will be used, or screen 0 if this variable is unset. Note that all window system connections can communication with all screens managed by that window server, regardless of the value of the default screen.

pfSelectWSConnection sets the current IRIS Performer libpr window system connection to be *ws*. The **DISPLAY** environment variable for that process will be set to the connection name for *ws* using the **putenv(3C)** command. This window system connection will be that returned by **pfGetCurWSConnection**.

pfCloseWSConnection will close the specified window server connection via XCloseDisplay. If *ws* was the currently selected libpr, window server connection, the current connection will be reset to NULL.

pfGetWSConnectionName will return the string name for the window system connection. In the X window system, this corresponds to the string returned by the **XDisplayString** call.

pfGetScreenSize returns the X and Y size of *screen* for the server of the current window system

connection in *x* and *y*.

pfChooseFBConfig takes an array of **PFFB_** attribute tokens in *attr* and returns a matching pfFBConfig describing a framebuffer configuration for the given window system connection, *ws*, and screen *screen*. If *ws* is null, the current libpr window system connection will be used. If *screen* is (-1), the default screen for the window system connection will be used. The attribute tokens and their values match the OpenGL GLX tokens (see **glXChooseVisual** for additional information). These tokens and their values are (booleans are true if present and should NOT be followed by any values):

PFFB_USE_GL
Boolean, true if present. Use GLX rendering (the default).

PFFB_BUFFER_SIZE
Depth of the color buffer

PFFB_LEVEL
Level in plane stacking. 0 is the main window. negative numbers are underlay planes, positive numbers are overlay planes.

PFFB_RGBA
Boolean, does RGBA mode if present

PFFB_DOUBLEBUFFER
Boolean, does double buffering if present

PFFB_STEREO
Boolean, does stereo buffering if present

PFFB_AUX_BUFFERS
Number of auxiliary buffers

PFFB_RED_SIZE
Number of red component bits

PFFB_GREEN_SIZE
Number of green component bits

PFFB_BLUE_SIZE
Number of blue component bits

PFFB_ALPHA_SIZE
Number of alpha component bits

PFFB_DEPTH_SIZE
Number of depth bits

PFFB_STENCIL_SIZE
Number of stencil bits

PFFB_ACCUM_RED_SIZE
Number of red accumulation bits

PFFB_ACCUM_GREEN_SIZE
Number of green accumulation bits

PFFB_ACCUM_BLUE_SIZE
Number of blue accumulation bits

PFFB_ACCUM_ALPHA_SIZE
Number of alpha accumulation bits

PFFB_SAMPLES_SGIS
Number of samples per pixel

PFFB_SAMPLE_BUFFERS_SGIS
The number of multisample buffers

The list must be terminated with a **NULL** or **None**. IRIS Performer will try to use a multisample buffer with 8 samples per pixel unless the number of samples or number of multisample buffers has been explicitly set to 0 in the attribute array. The default Performer framebuffer configuration looks like:

```
static int FBAttrs[] =
{
    PFFB_RGBA,
    PFFB_DOUBLEBUFFER,
    PFFB_DEPTH_SIZE, 24,
    PFFB_RED_SIZE, 8,
    PFFB_SAMPLES, 8,
    PFFB_STENCIL_SIZE, 4,
    None
};
```

pfChooseFBConfigData is similar to **pfChooseFBConfig** but it also returns any extra window system dependent data in *data*. This is useful for IRIS GL mixed model programs where *data* will be a complete GLX attribute array appropriate for **GLXlink**. See the **GLXlink** man page for more information.

SEE ALSO

pfWindow, glXChooseVisual, GLXgetconfig, GLXlink, XOpenDisplay, XCloseDisplay

NAME

pfNewWin, pfGetWinClassType, pfWinAspect, pfWinFBConfig, pfWinFBConfigAttrs, pfWinFBConfigData, pfWinFBConfigId, pfWinFullScreen, pfWinGLCxt, pfWinIndex, pfWinList, pfWinMode, pfWinName, pfWinOrigin, pfWinOriginSize, pfWinOverlayWin, pfWinScreen, pfWinShare, pfWinSize, pfWinStatsWin, pfWinType, pfWinWSConnectionName, pfWinWSDrawable, pfWinWSWindow, pfGetWinAspect, pfGetWinCurOriginSize, pfGetWinCurScreenOriginSize, pfGetWinCurState, pfGetWinCurWSDrawable, pfGetWinFBConfig, pfGetWinFBConfigAttrs, pfGetWinFBConfigData, pfGetWinFBConfigId, pfGetWinGLCxt, pfGetWinIndex, pfGetWinList, pfGetWinMode, pfGetWinName, pfGetWinOrigin, pfGetWinOverlayWin, pfGetWinScreen, pfGetWinSelect, pfGetWinShare, pfGetWinSize, pfGetWinStatsWin, pfGetWinType, pfGetWinWSConnectionName, pfGetWinWSDrawable, pfGetWinWSWindow, pfAttachWin, pfDetachWin, pfChooseWinFBConfig, pfCloseWin, pfCloseWinGL, pfIsWinOpen, pfMQueryWin, pfOpenNewNoPortWin, pfOpenWin, pfQueryWin, pfSelectWin, pfSwapWinBuffers, pfGetCurWin, pfInitGfx – GL-independent window creation/management routines

FUNCTION SPECIFICATION

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

pfWindow*       pfNewWin(void *arena);
pfType*         pfGetWinClassType(void);
void            pfWinAspect(pfWindow *win, int x, int y);
void            pfWinFBConfig(pfWindow *win, pfFBConfig vInfo);
void            pfWinFBConfigAttrs(pfWindow *win, int *attr);
void            pfWinFBConfigData(pfWindow *win, void *data);
void            pfWinFBConfigId(pfWindow *win, int id);
void            pfWinFullScreen(pfWindow *win);
void            pfWinGLCxt(pfWindow *win, pfGLContext gCxt);
void            pfWinIndex(pfWindow *win, int index);
void            pfWinList(pfWindow *win, pfList *wlist);
void            pfWinMode(pfWindow *win, int mode, int val);
void            pfWinName(pfWindow *win, const char *name);
void            pfWinOrigin(pfWindow *win, int xo, int yo);
void            pfWinOriginSize(pfWindow *win, int xo, int yo, int xs, int ys);
void            pfWinOverlayWin(pfWindow *win, pfWindow *ow);
void            pfWinScreen(pfWindow *win, int s);
```

void	**pfWinShare**(pfWindow *win, uint mode);
void	**pfWinSize**(pfWindow *win, int xs, int ys);
void	**pfWinStatsWin**(pfWindow *win, pfWindow *ow);
void	**pfWinType**(pfWindow *win, uint type);
void	**pfWinWSConnectionName**(const pfWindow *win, const char *name);
void	**pfWinWSDrawable**(pfWindow *win, pfWSConnection dsp, pfWSDrawable xWin);
void	**pfWinWSWindow**(pfWindow *win, pfWSConnection dsp, pfWSWindow xWin);
void	**pfGetWinAspect**(const pfWindow *win, int *x, int *y);
void	**pfGetWinCurOriginSize**(pfWindow *win, int *xo, int *yo, int *xs, int *ys);
void	**pfGetWinCurScreenOriginSize**(pfWindow *win, int *xo, int *yo, int *xs, int *ys);
pfState*	**pfGetWinCurState**(const pfWindow *win);
pfWSDrawable	**pfGetWinCurWSDrawable**(const pfWindow *win);
pfFBConfig	**pfGetWinFBConfig**(const pfWindow *win);
int*	**pfGetWinFBConfigAttrs**(const pfWindow *win);
void*	**pfGetWinFBConfigData**(pfWindow *win);
int	**pfGetWinFBConfigId**(const pfWindow *win);
pfGLContext	**pfGetWinGLCxt**(const pfWindow *win);
int	**pfGetWinIndex**(const pfWindow *win);
pfList*	**pfGetWinList**(const pfWindow *win);
int	**pfGetWinMode**(const pfWindow *win, int mode);
const char*	**pfGetWinName**(const pfWindow *win);
void	**pfGetWinOrigin**(const pfWindow *win, int *xo, int *yo);
pfWindow*	**pfGetWinOverlayWin**(const pfWindow *win);
int	**pfGetWinScreen**(const pfWindow *win);
pfWindow*	**pfGetWinSelect**(pfWindow *win);
uint	**pfGetWinShare**(const pfWindow *win);
void	**pfGetWinSize**(const pfWindow *win, int *xs, int *ys);
pfWindow*	**pfGetWinStatsWin**(const pfWindow *win);
uint	**pfGetWinType**(const pfWindow *win);

const char*	**pfGetWinWSConnectionName**(const pfWindow *win);
pfWSDrawable	**pfGetWinWSDrawable**(const pfWindow *win);
pfWSWindow	**pfGetWinWSWindow**(const pfWindow *win);
int	**pfAttachWin**(pfWindow *win0, pfWindow *win1);
int	**pfDetachWin**(pfWindow *win0, pfWindow *win1);
pfFBConfig	**pfChooseWinFBConfig**(pfWindow *win, int *attr);
void	**pfCloseWin**(pfWindow *win);
void	**pfCloseWinGL**(pfWindow *win);
int	**pfIsWinOpen**(pfWindow *win);
int	**pfMQueryWin**(const pfWindow *win, int *which, int *dst);
pfWindow*	**pfOpenNewNoPortWin**(const char *name, int screen);
void	**pfOpenWin**(pfWindow *win);
int	**pfQueryWin**(const pfWindow *win, int which, int *dst);
pfWindow*	**pfSelectWin**(pfWindow *win);
void	**pfSwapWinBuffers**(pfWindow *win);
extern pfWindow *	**pfGetCurWin**(void);
extern void	**pfInitGfx**(void);

```
/* typedef of X-based Performer Types */
typedef Display         *pfWSConnection;
typedef XVisualInfo     pfFBConfig;
typedef Window          pfWSWindow;
typedef Drawable        pfWSDrawable;

#ifdef IRISGL
typedef int             pfGLContext;
#else /* OPENGL */
typedef GLXContext      pfGLContext;
#endif
```

PARENT CLASS FUNCTIONS

The IRIS Performer class **pfWindow** is derived from the parent class **pfObject**, so each of these member functions of class **pfObject** are also directly usable with objects of class **pfWindow**. Casting an object of class **pfWindow** to an object of class **pfObject** is taken care of automatically. This is also true for casts to objects of ancestor classes of class **pfObject**.

void **pfUserData**(pfObject *obj, void *data);
void* **pfGetUserData**(pfObject *obj);
int **pfGetGLHandle**(pfObject *obj);

Since the class **pfObject** is itself derived from the parent class **pfMemory**, objects of class **pfWindow** can also be used with these functions designed for objects of class **pfMemory**.

pfType * **pfGetType**(const void *ptr);
int **pfIsOfType**(const void *ptr, pfType *type);
int **pfIsExactType**(const void *ptr, pfType *type);
const char * **pfGetTypeName**(const void *ptr);
int **pfRef**(void *ptr);
int **pfUnref**(void *ptr);
int **pfUnrefDelete**(void *ptr);
int **pfGetRef**(const void *ptr);
int **pfCopy**(void *dst, void *src);
int **pfDelete**(void *ptr);
int **pfCompare**(const void *ptr1, const void *ptr2);
void **pfPrint**(const void *ptr, uint which, uint verbose, FILE *file);
void * **pfGetArena**(void *ptr);

PARAMETERS

win identifies a pfWindow.

DESCRIPTION

These functions provide a single API for creating and managing windows that works across the IRIS GL, IRIS GL mixed model (GLX), and OpenGL/X environments. Window system independent types have been provided to match the X Window System types to provide complete portability between the IRIS GL and OpenGL/X windowing environments.

pfNewWin creates and returns a handle to a pfWindow. *arena* specifies a malloc arena out of which the pfWindow is allocated or **NULL** for allocation off the process heap.

pfGetWinClassType returns the **pfType*** for the class **pfWindow**. The **pfType*** returned by **pfGetWinClassType** is the same as the **pfType*** returned by invoking **pfGetType** on any instance of class **pfWindow**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfWinAspect sets the aspect ratio of *win* to be *x*:*y*. **pfGetWinAspect** returns the aspect X and Y components of *win* in *x* and *y*.

pfWinFullScreen will cause the window to be a full screen window and change its size appropriately.

Future queries of size and origin will reflect this new full screen size.

pfWinFBConfig sets the framebuffer configuration for *win* to be that specified by the XVisualInfo*, *vInfo*. This will determine the framebuffer configuration used to create the graphics context. **pfGetWinFBConfig** will return the XVisualInfo* of *win*.

pfWinFBConfigAttrs provides a window system independent list of attribute tokens, *attr*, to describe the desired framebuffer configuration of *win*. The attribute list format is the same as the SGI GLX attribute format for OpenGL, but with matching **PFFB_*** tokens that can be used with either IRIS GL or OpenGL in place of the **GLX_*** tokens. See the **glXChooseVisual** reference pages for more information. **pfGetWinFBConfigAttrs** will return the attribute array corresponding to the visual of *win*.

pfWinFBConfigData can be used to provide GL dependent data directly to the IRIS GL or OpenGL framebuffer configuration routine. **pfGetWinFBConfigData** can be used to get back GL dependent data resulting from these calls. In the case of IRIS GL, this routine is only useful for X windows and the resulting data will not be the same data that was passed in, but will be that data returned by IRIS GL routine **GLXgetconfig(3g)** and expected by the IRIS GL routine **GLXlink(3)**. This data is useful for IRIS GL X windows because IRIS GL X query routines and Motif routines require this data. See the IRIS **GLXgetconfig(3g)** and **GLXlink(3g)** reference pages for more information.

pfWinFBConfigId allows you to directly set the OpenGL X visual id to be used in configuring the resulting OpenGL/X window. **pfGetWinFBConfigId** will return the current OpenGL visual id of the window (or -1 if the id is not known, or if running under IRIS GL). See the **XVisualIDFromVisual(3X11)** and **XGetVisualInfo(3X11)** reference pages for more information about X visuals. This functionality is not supported under IRIS GL operation.

pfWinGLCxt sets the graphics context of *win* to be *gCxt*. If the graphics context window of *win* has been set, **pfOpenWin** on *win* will use that context and not create another. **pfGetWinGLCxt** will return the graphics context of *win*.

pfWinIndex sets the alternate configuration window list index of *win* to be *index*. If *index* is greater than or equal to zero, it will select an alternate configuration window from the pfWindow **pfWinList**. *index* may also select one of the standard windows: **PFWIN_STATS_WIN**, **PFWIN_OVERLAY_WIN**, and the default, **PFWIN_GFX_WIN**. The window indexing is only one level deep - if the selected pfWindow has a window list and index, it is ignored and the graphics window and context of that pfWindow is used to determine the drawing area. **pfGetWinIndex** will return the current index for *win*. **pfGetWinSelect** will return the pointer to the currently selected pfWindow for *win*.

pfWinList sets a pfWindow list of alternate configuration windows for *win* to be the pfList of pfWindow*s, *wlist*. **pfGetWinList** will return the current window list. These alternate configuration windows are assumed to have the same pfWSWindow parent window as the base pfWindow. Additionally, for pure IRIS GL windows, they must have the same graphics window and graphics context as the base pfWindow. These alternate configuration windows allow you to provide specify multiple framebuffer

configurations for the same drawing area on the screen for tasks such as overlay drawing or single-buffered drawing.

pfWinMode sets the pfWindow mode specified by *mode* of *win* to *val*. *mode* may be one of:

PFWIN_ORIGIN_LL	will cause placement of *win* to be relative to the lower left corner of it and its parent window.
PFWIN_NOBORDER	will cause the window to not have the window system border around the outside of its drawing area. To have a drawing area that is truly full screen, this mode should be set.
PFWIN_AUTO_RESIZE	will cause sub-pfWindows of *win* who also have this mode set to be automatically reconfigured to match the size and position of *win*. This includes the **PFWIN_OVERLAY_WIN**, the **PFWIN_STATS_WIN**, and the current selected window from the pfWindow list. The selection of a new window from the pfWindow list will also be automatically sized and positioned if it is using the **PFWIN_AUTO_RESIZE** mode. This mode is only useful for windows of type **PFWIN_TYPE_X** and will have no effect on IRIS GL windows. This mode is on by default.
PFWIN_HAS_OVERLAY	will cause **pfOpenWin** to automatically create and open (if necessary) an overlay window for the main pfWindow. If the window is of type **PFWIN_AUTO_RESIZE**, the overlay window will be automatically configured to keep the same size/position as the main window.
PFWIN_EXIT	Causes the window to receive special ClientMessage X events when the user selects "Quit" or "Exit" from the window manager border menu on the window. The XEvent.xclient.message_type field will be set to point to the X atom for "WM_PROTOCOLS" and XEvent.xclient.data.l[0] will be set to point to the X atom, "WM_DELETE_WINDOW". See the examples below and the **XClientMessageEvent** reference page for more information. This mode is only useful for windows of type **PFWIN_TYPE_X** and will have no effect on IRIS GL windows.

pfGetWinMode will return the value of the requested mode in *mode*.

pfWinName sets the name of a pfWindow. By default, pfWindows have no name. **pfGetWinName** returns the name of a pfWindow.

pfWinOrigin sets the origin of the pfWindow *win* to be (*xo*,*yo*), relative to its parent window. **pfGetWinOrigin** returns the origin set by **pfWinOrigin**. If the pfWindow mode is **PFWIN_ORIGIN_LL**, the origin of the window is considered to be the lower-left corner. Otherwise, the origin of the window is considered to be the X-style upper-left corner. **pfWinSize** sets the size of the pfWindow *win* to be *x* by *y*.

pfGetWinSize returns the size of the pfWindow set by **pfWinSize**, by **pfOpenWin**, or by **pfGetWinCurOriginSize**. **pfWinOriginSize** sets both the origin and size of *win*.

pfGetWinCurOriginSize returns the current origin and size of the pfWindow *win*, if open, otherwise it returns the origin and size set by **pfWinOrigin**, **pfWinSize**, or **pfWinOriginSize**. The internal origin and size of *win* will be updated. This routine accesses the graphics context and/or the X server and is slow on a window of type **PFWIN_TYPE_X**.

pfGetWinCurScreenOriginSize returns the current screen-relative origin and size of the pfWindow *win*, if open, otherwise it returns the origin and size set by **pfWinOrigin**, **pfWinSize**, or **pfWinOriginSize**. This routine accesses the graphics context. For windows of type **PFWIN_TYPE_X**, it must make expensive queries to the X server and can be very slow.

pfWinOverlayWin sets the pfWindow *overlay* to be the associated **PFWIN_OVERLAY_WIN** window for the main drawing pfWindow, *win*. This pfWindow is selected on *win* with **pfWinIndex**(*win*, **PFWIN_OVERLAY_WIN**). *overlay* should have the same parent X window as *win*. **pfGetWinOverlayWin** will return the **PFWIN_OVERLAY_WIN** pfWindow.

pfWinScreen sets the screen of *win* to be *screen*. The screen selection takes effect upon **pfOpenWin**. A screen will be set by **pfOpenWin** if on was not previously set. The screen of a pfWindow cannot be changed once set. **pfGetWinScreen** will return the screen of a pfWindow.

pfWinShare sets the attributes that are to be shared by pfWindows of the share group of *win* to be the bitmask specified by *share*. Some share attributes, such as **PFWIN_SHARE_TYPE** and **PFWIN_SHARE_GFX_OBJS**, must be specified before windows in the share group are opened. The following tokens specify the attributes that may be shared among pfWindows and may be or-ed together to from *share*:

PFWIN_SHARE_MODE

PFWIN_SHARE_FBCONFIG

PFWIN_SHARE_GL_CXT

PFWIN_SHARE_GL_OBJS

PFWIN_SHARE_STATE

PFWIN_SHARE_OVERLAY_WIN

PFWIN_SHARE_STATS_WIN

PFWIN_SHARE_TYPE

PFWIN_SHARE_WSDRAWABLE

PFWIN_SHARE_WSWINDOW

pfGetWinShare will return the share bitmask of *win*.

pfWinStatsWin sets the pfWindow *statsWin* to be the associated **PFWIN_OVERLAY_WIN** window for the main drawing pfWindow, *win*. This pfWindow is selected on *win* with **pfWinIndex**(*win*, **PFWIN_STATS_WIN**). *statsWin* should have the same parent X window as *win*. **pfGetWinStatsWin** will return the **PFWIN_STATS_WIN** pfWindow.

pfWinType sets the type of a pfWindow where *type* is an or-ed bitmask that may contain the type constants listed below. **pfGetWinType** returns the type of a pfWindow. The type of a pfWindow only takes effect by the call of **pfOpenWin**. The type of an open pfWindow cannot be changed. The pfWindow type attributes all start with **PFWIN_TYPE_** and are:

PFWIN_TYPE_NOPORT

The resulting window will have a graphics context but will not be mapped onto the screen. Windows of type **PFWIN_TYPE_NOPORT** are useful for queries about the graphics resources of the system and are needed for accessing the Video Sync Clock. To facilitate this, there is the special utility routine, **pfOpenNewNoPortWin**. Also, see the reference pages for **pfInitVClock**, and **pfQuerySys**. If this token is specified, all other type tokens are ignored.

PFWIN_TYPE_X

The window opened will be an X window. OpenGL windows are always of type **PFWIN_TYPE_X** so this mode only has effect in IRIS GL and will cause the resulting window to be a IRIS GL mixed model (GLX) window. Windows of this type have as their pfWSWindow an X window (**pfWinWSWindow**). The pfWSDrawable that is attached to the graphics context is by default an X window (but can be set as an X Pixmap -- **pfWinWSWindow**) and has a framebuffer configuration matching that specified by the pfWindow. If the pfWSDrawable of the pfWindow is a separate X window from the parent pfWSWindow X window (as is created by default), the pfWSDrawable window can actually be changed for one with a different framebuffer configuration without ugly flashing. To facilitate this, pfWindows may also have a list of windows (**pfWinList**) that may have different framebuffer configuration types and pfWSDrawable X windows but all share the same parent X pfWSWindow.

PFWIN_TYPE_OVERLAY

The pfWindow *win* will be given an appropriate framebuffer configuration, if not already set, that will support the standard overlay draw configuration at the time of the call to **pfOpenWin**. For X windows, an X colormap will also be created and attached to the corresponding X window. See the reference pages for **XCreateWindow**, and **XCreateColormap** for more information.

PFWIN_TYPE_STATS
The pfWindow *win* will be given a framebuffer configuration, if one has not already been specified through **pfWinFBConfig** or **pfChooseWinFBConfig**, that will support the current **pfStatsHwAttr** configuration at the time of the call to **pfOpenWin**. See the **pfStatsHwAttr** reference page for information on different statistics attributes and their framebuffer requirements.

pfWinWSWindow sets the main window system window of *win* to be *xWin* This routine is only relevant for pfWindows of type **PFWIN_TYPE_X**. The WSWindow of a pfWindow, if not NULL, is used to manage the size and position of the pfWindow. The WSWindow should also be the parent window of the WSDrawable window of *Win*. The WSWindow should be shared amongst all sub-pfWindows, such as the overlay window, the stats window, and any windows n the pfWindow list. If the WSWindow of *win* has been set, **pfOpenWin** on *win* will use that X window and not create another. **pfGetWinWSWindow** will return the X Window of *win*. Pure IRIS GL windows require that the WSWindow, WSDrawable, and GL Context are all the same and can be set with **pfWinGLCxt**.

pfWinWSDrawable sets the graphics X drawable of *win* to be *drawable*. The drawable of a pfWindow is attached to the graphics context and may be an X Pixmap or X Window. If provided as an X Window, it should be a child of the WSWindow of the pfWindow. If a drawable has not been provided by the time of **pfOpenWin**, an X Window will be created by default. If the graphics drawable of *win* has been set, **pfOpenWin** on *win* will use that X window and not create another. **pfGetWinWSDrawable** will return the drawable of *win*. **pfGetWinCurWSDrawable** will return the drawable of the currently selected window of the window list. Pure IRIS GL windows require that the WSWindow, WSDrawable, and GL Context are all the same and can be set with **pfWinGLCxt**.

pfWinWSConnectionName allows you to specify the exact window server and default screen for the successive opening of the window. This can be used for specifying remote displays or on machines running more than one window server. **pfGetWinWSConnectionName** will return the name specifying the current window server target.

pfAttachWin puts *win1*, and its current share group, in the pfWindow share group of *win0*. The attributes of *win0* will be copied to *win1* and all of the windows in *win1*'s previous share group. pfWindows cannot be removed from a share group. **pfDetachWin** removes *win1* from the share group of *win0*.

pfChooseWinFBConfig will select a framebuffer configuration for *win*, constrained by current settings, such as type and the framebuffer configuration attributes, on *win*. Additionally, the selection of the framebuffer configuration will be relative to the screen of *win*. If the screen has not been set, it will be determined from the default screen of the current pfWSConnection or DISPLAY environment variable. The return value is the resulting pfFBConfig, or NULL, indicating failure. For X windows, **pfChooseWinFBConfig** and **pfGetWinFBConfig** returns the resulting XVisualInfo*.

pfOpenWin creates a graphics context and window, constrained by the settings of *win* on the current selected display via **pfGetCurWSConnection**. Attributes of *win* that are not set are created and set as

necessary. If the graphics window and context are not set, they will be created.

If the pfWindow framebuffer configuration is not set (**pfWinFBConfig** or **pfChooseWinFBConfig**), the graphics window will get the default rendering framebuffer configuration for its current type (-**pfWinType**). For a rendering graphics window of type **PFWIN_TYPE_X**, if the graphics drawable has been set via **pfWinWSDrawable**, the framebuffer configuration of that window is used for the graphics context. Otherwise, a default rendering framebuffer configuration for the current machine will be chosen via (**pfChooseFBConfig**). There is a key difference in the default configurations for X windows and pure IRIS GL windows on machines that offer hardware multisample antialiasing buffers. Because an X window cannot have its configuration changed, X windows will have multisample buffers by default. However, pure IRIS GL buffers can be reconfigured if antialiasing is requested and so do not have multisample buffers by default. All pfWindows are automatically initialized with **pfInitGfx** upon opening with **pfOpenWin**.

If the x or y size of the pfWindow is <= 0, then a rubber-band window will be created for the user to determine the origin and size of the window, constrained by the pfWindow aspect if set (**pfWinAspect**). If the size of the pfWindow is <= 0 but the origin is < 0, then the graphics window will be opened with fixed size but allow the user to place the window. The pfWindow origin and size may both be internally set by **pfOpenWin**. If the **PFWIN_HAS_OVERLAY** mode has been set, a **PFWIN_OVERLAY_WIN** will be automatically created (if not already set, **pfWinOverlayWin**) and opened. If *win* has a pfWindow list (**pfWinList**) and the current pfWindow index is not **PFWIN_GFX_WIN**, then the selected pfWindow from the list will be opened. A pfState is automatically created for the pfWindow and the pfWindow is made the current libpr pfWindow. **pfGetWinCurState** will return the current pfState of *win*. **pfGetCurWin** will return the pointer to the current libpr pfWindow.

pfCloseWin will destroy the graphics context of the open *win*. If *win* is of type **PFWIN_TYPE_X**, its X windows will be unmapped. **pfCloseWinGL** will destroy the current graphics context and graphics X window but leaves the top level X window in tact.

pfIsWinOpen returns the open status of *win*.

pfQueryWin takes a window configuration query token *which* and writes into *dst* the value for the corresponding configuration of the the opened pfWindow *win*. The pfWindow query token may be one of:

PFQWIN_RGB_BITS	returns the number of bits per R_G_B color component allocated in the main color buffer.
PFQWIN_ALPHA_BITS	returns the number of bits allocated for alpha in the main color buffer.
PFQWIN_CI_BITS	returns the number of bits for colorindex indices.

PFQWIN_DEPTH_BITS returns the number of framebuffer bits allocated for Z.

PFQWIN_MIN_DEPTH_VAL
returns the minimum representable Z depth value.

PFQWIN_MAX_DEPTH_VAL
returns the maximum representable Z depth value.

PFQWIN_MS_SAMPLES returns the number of multisample samples.

PFQWIN_STENCIL_BITS returns the number of bits in the stencil buffer.

PFQWIN_NUM_STEREO returns whether the window has stereo buffers allocated.

PFQWIN_NUM_STEREO returns whether the window has double-buffered color buffers allocated.

PFQWIN_NUM_AUX_BUFFERS
returns the number of auxiliary color buffers allocated for the window.

PFQWIN_LEVEL returns the level of window planes (0 is normal drawing level, negative for underlay and positive for overlay).

PFQWIN_ACCUM_RGB_BITS
returns the number of bits per R_G_B color component allocated in the accumulation buffer.

PFQWIN_ACCUM_ALPHA_BITS
returns the number of bits of alpha allocated in the accumulation buffer.

pfMQueryWin takes an NULL-terminated array of query tokens and a destination buffer and will do multiple queries. The return value will be the number of bytes successfully written. This routine is more efficient than **pfQueryWin** if multiple queries are desired.

pfSwapWinBuffers causes the currently selected front and back buffers of the normal framebuffer of an open window to be exchanged during the next vertical retrace period.

pfGetCurWin will return the pointer current IRIS Performer pfWindow. This a window is made the current window when it is opened with **pfOpenWin** or selected with **pfSelectWin**.

pfInitGfx will configure the current graphics context correctly for IRIS Performer rendering operation and is called automatically when pfWindows are opened. It will enable z-buffer depth testing, viewport clipping, and subpixel vertex accuracy mode. The Viewing projection will be a two-dimensional one-to-one orthographic mapping from eye coordinates to window coordinates with distances to near and far clipping planes -1 and 1, respectively. The model matrix will be the current matrix and will be initialized to the identity matrix. It is highly recommended that a libpr application managing its own windows call **pfInitGfx** for its normal drawing.

For pure IRIS GL windows, if no framebuffer configuration attributes have been specified for any existing current IRIS Performer pfWindow via **pfWinFBConfigAttrs**, **pfInitGfx** will put the window in RGB and double-buffer mode and multisample buffers as appropriate to the system, and will configure hardware sizes for color, stencil, and Z. The default configuration, if available will be double buffered with eight bits per component of RGBA color, 24 bits of depth buffer, and four bits of stencil. If this is not be available, one bit of stencil will be used. Some graphics hardware may support fewer bits of depth and RGBA as well. If a multisample buffer has previously been allocated, such as by a call to **pfAntialias**, the default multisample configuration will be restored. See the **pfAntialias** reference page for detail on the multisample antialiasing framebuffer configuration. The **pfQuerySys** command can be used to query these parameters for the current system. **pfQueryWin** can be used to query the configuration of a specific window after it has been opened.

NOTES

X Window origin and size: There are some subtle issues in the management of origin and size of X windows. With X windows, it can be very expensive to obtain the current screen relative origin of a window, particularly if the window is in a hierarchy. Therefore, the origin of a window is defined to be that relative to it's parent window (which can be the screen). Additionally, there are a bevy of routines for getting the window origin. User code should not rely on knowning the screen relative origin but should be window-coordinate relative to be efficient and reliable.

X Framebuffer Configuration: The selection of framebuffer configurations for X windows uses the default GL selection utilities: **GLXgetconfig(3g)** for IRIS GLX and **glXChooseVisual(3X11)** for OpenGL/X. These utilities return the maximum possible framebuffer configuration matching the requested attributes. However, this may not be the optimal configuration for performance. One such example occurs with OpenGL on the RealityEngine: when requesting four multisample subsamples and a depth buffer of 24 bits, a depth buffer of 32 bits will be returned which has measurably slower fill rate than a 24 bit depth buffer. If the default utilities are not returning the desired framebuffer configuration, you can do your own X visual selection and set the visual id or the visual itself on the pfWindow with **pfWinFBConfigId** and **pfWinFBConfig** respectively. Additionally, libpfutil provides an OpenGL visual chooser, **pfuChooseFBConfig**, that limits the performance critical attributes: multisamples, depth, RGB color, and stencil.

A special case for framebuffer configuration exists for the Extreme graphics platforms. On these platforms, the default framebuffer configuration has NO allocated stencil bits because stencil bits will reduce depth buffer resolution. The user may explicitly request stencil bits if desired. The Indy graphics platforms do not offer stencil under IRIS GL operation.

EXAMPLES

This example creates a pfWindow structure and opens and initializes the window for Performer drawing. This window will be an OpenGL/X window if linked with the OpenGL Performer libraries and will be a pure IRIS GL window if linked with the IRIS GL IRIS Performer libraries.

```
{
        pfWindow *win;

        pfInitState();
        win = pfNewWin(NULL);
        pfWinName(win, "Performer");
        pfOpenWin(win); /* create window and rendering context */
        ......
}
```

This example is a more detailed example for creating a window of pre-defined size and position. It also uses IRIS GL mixed model (GLX) when linked with the IRIS GL libraries and will have an overlay window created automatically when the window is opened.

```
{
        pfWindow *win;
        pfWindow *overlay;
        pfWSConnection dsp;

        pfInitState();
        win = pfNewWin(NULL);
        pfWinName(win, "Performer");
        pfWinOriginSize(win, 0, 0, 500, 500);
        pfWinType(win, PFWIN_TYPE_X);
        pfWinMode(win, PFWIN_HAS_OVERLAY, PF_TRUE);
        pfOpenWin(win); /* create window, overlay, and rendering context */

        /* get back some useful things created by Performer */
        overlay = pfGetWinOverlayWin(win);
        /* get back Performer's internal shared display
           connection to use for event handling */
        dsp = pfGetCurWSConnection();
}
```

This example demonstrates how to catch the X ClientMessage event when a window of a pfWindow of mode PFWIN_EXIT is killed via the Quit or Exit option in the window manager menu on the window

border.

```
{
    WSConnection theDisplay = pfGetCurWSConnection();
    Atom WMProtocols = XInternAtom(theDisplay, "WM_PROTOCOLS", 1);
    ATOM WMDeleteWindow = XInternAtom(theDisplay, "WM_DELETE_WINDOW", 1);
    ....
    /* in X event handling loop */
    {
        XEvent event;
        XNextEvent(theDisplay, &event);
        /* in X event handling switch */
        case ClientMessage:
            if ((event.xclient.message_type == wm_protocols) &&
                (event.xclient.data.l[0] == wm_delete_window)) {
                    /* handle window exit */
                }
                break;
        .....
    }
}
```

SEE ALSO

pfStatsHwAttr, pfNewState, pfSelectWSConnection, pfGetCurWSConnection, pfuChooseFBConfig, GLXgetconfig, GLXlink, XCreateWindow, XGetWindowAttributes, XGetVisualInfo, XVisualIDFrom-Visual

libpfdu

libpfdu is a database utility library, with functions for reading and writing scene graphs.

This library contains key computational geometry and data organization tools for tasks including polygon decomposition, triangle meshing, attribute sharing, pfGeoSet construction, and structure optimization.

NAME

pfdBreakup – Create an artificial hierarchy from unstructured input geometry.

FUNCTION SPECIFICATION

#include <Performer/pfdu.h>

pfNode * **pfdBreakup**(pfGeode *geode, float geodeSize, int stripLength, int geodeChild);

DESCRIPTION

pfdBreakup accepts a pfGeode that contains pfGeoSets of type **PFGS_TRISTRIPS** and builds a new scene graph, with the same geometric content but with a spatial subdivision structure designed for efficient processing. When the subdivision process is successful **pfdBreakup** returns the root of the new scene graph. **NULL** is returned on failure. Failure is often due to providing non-**PFGS_TRISTRIPS** **pfGeoSets** to **pfdBreakup**.

The **pfGeoSets** in *geode* are subdivided based on their geometrical centers using an octree. The degree of recursive partitioning desired is specified in the function arguments. The resulting scene graph will have **pfGeodes** of a size no larger than *geodeSize* and triangle strips no longer than *stripLength*. Each **pfGeode** created will have no more than *geodeChild* **pfGeoSet**s.

NOTES

The input geode is not deleted.

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

BUGS

pfdBreakup does not work for indexed **pfGeoSets**.

SEE ALSO

pfGeoSet, pfGeode, pfNode

NAME

pfdInitBldr, pfdExitBldr, pfdNewBldr, pfdDelBldr, pfdSelectBldr, pfdGetCurBldr, pfdAddBldrGeom, pfdAddIndexedBldrGeom, pfdBldrStateVal, pfdGetBldrStateVal, pfdBldrStateMode, pfdGetBldrStateMode, pfdBldrStateAttr, pfdGetBldrStateAttr, pfdBldrStateInherit, pfdGetBldrStateInherit, pfdCaptureDefaultBldrState, pfdResetBldrState, pfdPushBldrState, pfdPopBldrState, pfdSaveBldrState, pfdLoadBldrState, pfdBldrGState, pfdGetBldrGState, pfdBuild, pfdBuildNode, pfdSelectBldrName, pfdGetCurBldrName, pfdGetTemplateObject, pfdResetAllTemplateObjects, pfdResetObject, pfdMakeDefaultObject, pfdResetBldrGeometry, pfdResetBldrShare, pfdCleanBldrShare, pfdDefaultGState, pfdGetDefaultGState, pfdMakeSceneGState, pfdOptimizeGStateList, pfdBldrMode, pfdGetBldrMode, pfdBldrAttr, pfdGetBldrAttr, pfdBldrDeleteNode – Provides a simple interface between model input code and internal Performer model representations.

FUNCTION SPECIFICATION

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

void            pfdInitBldr(void);
void            pfdExitBldr(void);
pfdBuilder *    pfdNewBldr(void);
void            pfdDelBldr(pfdBuilder *bldr);
void            pfdSelectBldr(pfdBuilder *bldr);
pfdBuilder *    pfdGetCurBldr(void);
void            pfdAddBldrGeom(pfdGeom *prims, int count);
void            pfdAddIndexedBldrGeom(pfdGeom *prims, int count);
void            pfdBldrStateVal(int which, float val);
float           pfdGetBldrStateVal(int which);
void            pfdBldrStateMode(int mode, int val);
int             pfdGetBldrStateMode(int mode);
void            pfdBldrStateAttr(int which, const void *attr);
const void *    pfdGetBldrStateAttr(int attr);
void            pfdBldrStateInherit(uint mask);
uint            pfdGetBldrStateInherit(void);
void            pfdCaptureDefaultBldrState(void);
void            pfdResetBldrState(void);
void            pfdPushBldrState(void);
```

void	**pfdPopBldrState**(void);
void	**pfdSaveBldrState**(void *name);
void	**pfdLoadBldrState**(void *name);
void	**pfdBldrGState**(const pfGeoState *gstate);
const pfGeoState *	**pfdGetBldrGState**(void);
pfNode *	**pfdBuild**(void);
pfNode *	**pfdBuildNode**(void *name);
void	**pfdSelectBldrName**(void *name);
void *	**pfdGetCurBldrName**(void);
pfObject *	**pfdGetTemplateObject**(pfType *type);
void	**pfdResetAllTemplateObjects**(void);
void	**pfdResetObject**(pfObject *obj);
void	**pfdMakeDefaultObject**(pfObject *obj);
void	**pfdResetBldrGeometry**(void);
void	**pfdResetBldrShare**(void);
void	**pfdCleanBldrShare**(void);
void	**pfdDefaultGState**(pfGeoState *def);
const pfGeoState *	**pfdGetDefaultGState**(void);
pfGeoState *	**pfdMakeSceneGState**(pfList *gstateList, pfGeoState *previousGlobalState);
void	**pfdOptimizeGStateList**(pfList *gstateList, pfGeoState *globalGState);
void	**pfdBldrMode**(int mode, int val);
int	**pfdGetBldrMode**(int mode);
void	**pfdBldrAttr**(int which, void *attr);
void *	**pfdGetBldrAttr**(int which);
void	**pfdBldrDeleteNode**(pfNode *node);

DESCRIPTION

Converting a model database into Performer runtime structures is a very common task which almost all Performer applications must do. However, the form in which databases are modeled does not necessarily correspond to Performer runtime structures. Therefore, the task of reading database information can be tedious.

The Performer *Builder* is meant to manage most of the details of constructing efficient runtime structures. It provides a simple and convenient interface for bringing scene data into the application without the

need for considering how best to structure that data for efficient rendering in Performer. The Builder provides a comprehensive interface between model input code (such as database file parsers) and the internal mechanisms of scene representation in Performer.

The operational state of the Builder is encapsulated in a **pfdBuilder** object. During execution, there is always a current pfdBuilder defined. **pfdInitBldr** initializes the current (default) pfdBuilder and **pfdExitBldr** terminates its use. For those who wish to explicitly manage the use of several builders, **pfdNewBldr** creates a new pfdBuilder object and **pfdDelBldr** destroys a pfdBuilder object. **pfdSelectBldr** and **pfdGetCurBldr** set and query the current pfdBuilder, respectively.

The Builder is used in an immediate mode fashion. A typical client will feed a series of geometric primitives to the Builder, occasionally setting certain graphics states, and finally request that the Builder convert all the input data into a Performer scene graph.

pfdAddBldrGeom is the means by which a new geometric primitive is entered into the Builder. It accepts a pointer to a single **pfdGeom** structure *prims*. If the indicated **pfdGeom** is a line-strip, then the *count* argument provides the number of line segments in the strip. **pfdAddIndexedBldrGeom** provides the same functions for indexed primitives. See the documentation for the Performer geometry builder (-**pfdGeoBuilder**) for documentation on the structure and use of the **pfdGeom** structure.

In addition to geometry, we need to specify various graphics states. There is always a current graphics state being applied to input geometric primitives. **pfdBldrStateVal, pfdGetBldrStateVal, pfdBldrStateMode, pfdGetBldrStateMode, pfdBldrStateAttr, pfdGetBldrStateAttr, pfdBldrStateInherit**, and **pfdGetBldrStateInherit** are used to access and alter this state. These functions are an exact duplication of the interface to **pfGeoState**s; refer to the **pfGeoState** documentation for details on graphics states and their use.

The current graphics state can also be manipulated as a whole. **pfdPushBldrState** and **pfdPopBldrState** manipulate a stack of graphics states, the top of which is the current state. **pfdDefaultGState** records the state defined by pfGeoState *def* as the Builder's default state. **pfdResetBldrState** resets the current graphics state to the default state, and **pfdCaptureDefaultBldrState** makes the current state the default state. Finally, **pfdSaveBldrState** saves the current state with the given name, and **pfdLoadBldrState** loads the named state into the current state.

For convenience, the current graphics state can be set directly from a pfGeoState using **pfdBldrGState**. The given pfGeoState will be used as a template to set the current Builder graphics state. **pfdGetBldrGState** returns the current Builder state as a pfGeoState. The returned pfGeoState *should not* be modified in any way.

pfdBuild takes all the geometry data and graphics state information that has been input into the Builder and constructs a Performer scene graph for it. The function returns a pfNode which is the root of this graph.

While transferring database information to the Builder, it is possible to partition the database into logical units. Each unit is given a name; in normal operation, all data is tagged with the same default name. Every named partition will correspond to a disjoint subgraph in the scene graph generated by **pfdBuild**. The function **pfdBuildNode** can be used to build the graph for a single specific partition, rather than the entire input database. **pfdSelectBldrName** sets the name being applied to input data and **pfdGetCurBldrName** queries the current name being used.

It is also possible to remove all accumulated data in the Builder. **pfdResetBldrGeometry** clears away all geometry information stored in the Builder. **pfdResetBldrShare** deletes all shared graphics state in the Builder. **pfdCleanBldrShare** deletes all shared graphics state that is only referenced through the current pfdBuilder's share structure (this is accomplished through **pfdCleanShare** which in turn uses **pfUnrefDelete** to delete individual state elements that are only referenced by the pfdBuilder).

pfdBldrDeleteNode gets a list of all state attributes attached to leaf nodes under *node*, activates **pfDelete** on *node*, then proceeds to remove each state attribute from the current pfdBuilder's share structure via **pfdRemoveSharedObject**.

The Builder also tries to alleviate the need for the user to manipulate Performer objects while building the database. **pfdGetTemplateObject** returns a standard object of the given type. The user can then fill in the appropriate slots of this object and pass this object to the Builder. This allows the Builder to deal with some of the details of Performer data structures as well as accept the burden of managing memory for database objects. **pfdResetObject** fills in a template object with the default values for that object type. **pfdMakeDefaultObject** sets the default values using the given object as a template. Use **pfdResetAllTemplateObjects** to restore all of the template objects to the values defined by the Builder's default pfGeoState.

As another form of sharing, Performer databases can also utilize a default global pfGeoState. For example, **pfSceneGState** sets the global pfGeoState for an entire scene. All pfGeoStates in the relevant scene graph would inherit state from the global default. See the manual pages for **pfGeoState** for further details on state inheritance.

The construction of this default state is also incorporated into the Builder. While processing incoming data, the Builder remembers the default pfGeoState. **pfdDefaultGState** sets this default state and **pfdGetDefaultGState** can be used to query it. All pfGeoStates constructed by the Builder will inherit the attributes of this default pfGeoState. If a series of pfGeoStates have already been constructed, **pfdMakeSceneGState** can be used to extract a default pfGeoState which maximizes sharing through inheritance. In order to do this, **pfdMakeSceneGState** requires the list of pfGeoStates to be optimized and the default pfGeoState in effect when they were originally constructed. Finally, **pfdOptimizeGStateList** takes a list of pfGeoStates and forces them to inherit state wherever that state agrees with the specified default.

The behavior of the Builder can be controlled by changing various Builder modes and attributes. **pfdBldrMode** and **pfdGetBldrMode** set and query the Builder modes, respectively. **pfdBldrAttr** and

pfdGetBldrAttr set and query Builder attributes.

The supported Builder modes are:

PFDBLDR_MESH_ENABLE
PFDBLDR_AUTO_COLORS
PFDBLDR_AUTO_ORIENT
PFDBLDR_AUTO_NORMALS
These modes are used to control the geometry builder. See the documentation for **pfdGeoBldrMode** for details.

PFDBLDR_MESH_SHOW_TSTRIPS
PFDBLDR_MESH_INDEXED
PFDBLDR_MESH_MAX_TRIS
PFDBLDR_MESH_RETESSELLATE
These modes are used to control the triangle strip mesher. Refer to the documentation for **pfdMesherMode**.

PFDBLDR_BREAKUP
When this mode is enabled (its value is non-NULL), input geometry will be fed through **pfdBreakup**. This utility takes the unstructured input geometry and creates an output graph reflecting a spatial subdivision of the input.

PFDBLDR_BREAKUP_SIZE
PFDBLDR_BREAKUP_BRANCH
PFDBLDR_BREAKUP_STRIP_LENGTH
If **pfdBreakup** is called, the values of these modes determine the argument values passed to it.

PFDBLDR_SHARE_MASK
This is a bitwise-OR of the attributes that should be shared in the Performer scene graph constructed by the Builder (see **pfdShare**).

PFDBLDR_ATTACH_NODE_NAMES
If this mode is enabled, every pfNode constructed by the Builder will have its name set to the name assigned to it during input to the Builder; that is, the name set using **pfdSelectBldrName** will become the name of the new nodes.

PFDBLDR_DESTROY_DATA_UPON_BUILD
If this mode is set, all the data in the builder along with associated state information (except for information about attributes that should be shared) will be destroyed when **pfdBuild** returns.

The available attributes are:

PFDBLDR_NODE_NAME_COMPARE
The value of this attribute is a function to compare two node names. If its value is NULL, the == operator is used for comparisons.

PFDBLDR_STATE_NAME_COMPARE
The value of this attribute is a function to compare two state names. If its value is NULL, the == operator is used for comparisons.

The following code sample illustrates the use of the Builder. It is designed to read a database file which is simply a list of vertex coordinates. Every vertex triple defines a triangle.

```
pfNode * pfdLoadFile_tri (char *fileName)
{
    FILE       *triFile       = NULL;
    pfNode     *node          = NULL;
    pfdGeom    *prim          = NULL;
    int         v             = 0;

    /* open ".tri" file */
    if ((triFile = pfdOpenFile(fileName)) == NULL)
        return NULL;

    /* allocate primitive buffer */
    prim = pfdNewGeom(3);

    /* discard any lingering geometry in the builders */
    pfdResetBldrGeometry();

    /* pick a random not-too-dark color */
    pfuRandomColor(prim->colors[0], 0.4f, 0.8f);

    /* specify control data */
    prim->numVerts = 3;
    prim->primtype = PFGS_POLYS;
    prim->nbind = PFGS_PER_VERTEX;
    prim->cbind = PFGS_PER_PRIM;
    prim->tbind = PFGS_OFF;

    /* read triangles from ".tri" file */
    while (!feof(triFile))
    {
        /* read vertices from ".tri" file */
        for (v = 0; v < 3; v++)
        {
            /* read vertex data */
```

```
            fscanf(triFile, "%f %f %f",
                   &prim->coords[v][PF_X],
                   &prim->coords[v][PF_Y],
                   &prim->coords[v][PF_Z]);
            fscanf(triFile, "%f %f %f",
                   &prim->norms[v][PF_X],
                   &prim->norms[v][PF_Y],
                   &prim->norms[v][PF_Z]);
        }

        /* add this line to builder */
        pfdAddBldrGeom(prim, 1);
    }

    /* close ".tri" file */
    fclose(triFile);

    /* release primitive buffer */
    pfdDelGeom(prim);

    /* get a complete scene graph representing file's primitives */
    node = pfdBuild();

    return node;
}
```

NOTES

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfGeoSet, pfGeoState, pfObject, pfdBreakup, pfdGeoBuilder, pfdMeshGSet, pfdMesherMode, pfSceneGState

NAME

pfdPreDrawTexgenExt, pfdPostDrawTexgenExt, pfdPreDrawReflMap, pfdPostDrawReflMap, pfdPreDrawContourMap, pfdPostDrawContourMap, pfdPreDrawLinearMap, pfdPostDrawLinearMap, pfdTexgenParams – Node callbacks examples for special effects.

FUNCTION SPECIFICATION

#include <Performer/pfdu.h>

int **pfdPreDrawTexgenExt**(pfTraverser *trav, void *data);

int **pfdPostDrawTexgenExt**(pfTraverser *trav, void *data);

int **pfdPreDrawReflMap**(pfTraverser *trav, void *data);

int **pfdPostDrawReflMap**(pfTraverser *trav, void *data);

int **pfdPreDrawContourMap**(pfTraverser *trav, void *data);

int **pfdPostDrawContourMap**(pfTraverser *trav, void *data);

int **pfdPreDrawLinearMap**(pfTraverser *trav, void *data);

int **pfdPostDrawLinearMap**(pfTraverser *trav, void *data);

void **pfdTexgenParams**(const float *newParamsX, const float *newParamsY);

DESCRIPTION

These callback functions provide a useful generic callback prototype for **pfNode**s. Use these callbacks to create effects that IRIS Performer does not support directly, or to perform graphics library tasks differently than the built-in mechanisms. The currently implemented callbacks perform reflection mapping using **texgen**.

To write a database loader that loads reflective materials (as in **pfdLoadFile_obj**), just group reflective materials under different geodes from non-reflective materials and call **pfNodeTravFuncs** using **pfdPreDrawReflMap** as the pre-draw callback and **pfdPostDrawReflMap** as the post-draw callback as follows.

```
pfNodeTravFuncs(MyReflMappedGeode, PFTRAV_DRAW,
    pfdPreDrawReflMap, pfdPostDrawReflMap);
```

This will tell the loader to set **pfdPreDrawReflMap** as the pre-draw callback and **pfdPostDrawReflMap** as the post-draw callback for the geode containing the reflective materials.

At one time the **pfdLoadFile_obj** database loader used these mechanisms to implement reflection mapping. Now that the pfTexGen attribute has been added to libpr's pfGeoState this is no longer necessary and the OBJ loader has been updated. The techniques of the reflection map callback utilities is provided as an example of how similar functions might be implemented.

NOTES

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfNode, pfNodeTravFuncs, pfdLoadFile_obj, texgen

NAME

pfdFreezeTransforms, pfdCleanTree, pfdReplaceNode, pfdInsertGroup, pfdRemoveGroup – Scene graph optimizations

FUNCTION SPECIFICATION

#include <Performer/pfdu.h>

pfNode * **pfdFreezeTransforms**(pfNode *node, pfuTravFuncType func);

pfNode * **pfdCleanTree**(pfNode *node, pfuTravFuncType func);

void **pfdReplaceNode**(pfNode *oldn, pfNode *newn);

void **pfdInsertGroup**(pfNode *oldn, pfGroup *grp);

void **pfdRemoveGroup**(pfGroup *oldn);

DESCRIPTION

pfdCleanTree traverses the scene graph rooted at *node* and removes redundant and empty pfGroups from the scene graph. It also converts any pfSCS with an identity transformation into a pfGroup.

By default, pfGroups are eliminated if they have one or fewer children. Any children are reparented to the parent of the pfGroup being eliminated. The one exception is a pfSwitch node with one child, which is not eliminated. An optional function *func* can be provided to alter this determination during the traversal. At each candidate pfGroup, *func* is passed a pfuTraverser which includes the current node. If *func* returns TRUE, the current node is eliminated. If *func* returns FALSE, the current node is retained.

pfdFreezeTransforms traverses the scene graph rooted at *node* and converts pfDCSes to pfSCSes. Usually in preparation for a subsequent call to **pfFlatten**. By default the conversion occurs for those pfDCSes that don't have any callbacks and do not have the string "dcs" or "DCS" embedded in the node name. An optional function *func* can be provided to alter this determination during the traversal. At each candidate pfGroup, *func* is passed a pfuTraverser which includes the current node. If *func* returns TRUE, the current node is converted. If *func* returns FALSE, the current node remains a pfDCS.

pfdReplaceNode, **pfdInsertGroup** and **pfdRemoveGroup** are helper routines. **pfdReplaceNode** replaces the node *oldn* with the node *newn* in the scene graph, including reparenting *newn* to the *oldn*'s parents and reparenting *oldn*'s children to *newn*. *oldn* is not deleted. Any callbacks, traversal masks and the node name are also copied. **pfdInsertGroup** inserts the group *grp* above *oldn* in the the scene graph. *grp* replaces *oldn* as a child of all *grp*'s parents. *oldn* becomes a child of *grp*. **pfdRemoveGroup** removes *oldn* from the scene graph, reparenting all of *oldn*'s children to the parents of *oldn*. *oldn* is not deleted.

SEE ALSO

pfFlatten

NAME

pfdCombineBillboards – Merge pfBillboard nodes together.

FUNCTION SPECIFICATION

#include <Performer/pfdu.h>

void **pfdCombineBillboards**(pfNode *node, int sizeLimit);

DESCRIPTION

pfdCombineBillboards() gathers sibling billboard nodes together into as few large pfBillboard nodes as possible. This can often increase the efficiency of billboard processing in databases with large numbers of single-pfGeoSet pfBillboard nodes, such as the IRIS Performer Town database. The *sizeLimit* argument defines an upper limit to the merging process. No pfBillboard node will be made to have more than this number of pfGeoSets, and no pfBillboard node already having more than this number will be modified during the traversal.

There are a few requirements and restrictions to the pfBillboard combination process. In order to be mergable, pfBillboard nodes must be both *similar* and *combinable*.

For a pair of pfBillboard nodes to be *similar* they must each have the same values for their pfBillboard attributes. The attributes of pfBillboard nodes are:

Mode — Both pfBillboard nodes must have the same rotation mode. The modes are PFBB_AXIAL_ROT, PFBB_POINT_ROT_EYE, and PFBB_POINT_ROT_WORLD.

Axis — When the rotation mode is PFBB_AXIAL_ROT or PFBB_POINT_ROT_WORLD then both pfBillboard nodes must have the same axis of rotation. No such restriction is required of PFBB_POINT_ROT_EYE pfBillboards. The axis is compared using **pfAlmostEqualVec3()** with a tolerance of 0.0001.

In order for two pfBillboard nodes to be considered *combinable*, two additional conditions must be met:

Size — The destination pfBillboard node must not be too big. The argument *sizeLimit* defines the maximum number if pfGeoSets that any combined node may contain, so nodes that already have at least *sizeLimit* pfGeoSets will not receive additional pfGeoSets.

Safe — Both pfBillboard nodes being combined--the one giving up pfGeoSets and the one receiving them--must be uninstanced nodes. This means that each node has only one parent. This situation can be forced if desired by calling **pfFlatten()** on a scene graph before passing that scene graph to **pfdCombineBillboards()**.

NOTES

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfFlatten, pfAlmostEqualVec3

NAME

pfdCombineLayers – Merge layer nodes together.

FUNCTION SPECIFICATION

#include <Performer/pfdu.h>

void **pfdCombineLayers**(pfNode *node);

DESCRIPTION

pfdCombineLayers gathers sibling layers (**pfLayer**) together into a single super-layer. All bases are grouped together followed by all the layers.

All layer nodes, both existing and newly created, are set to use the **PFDECAL_LAYER_DISPLACE** style for rendering coplanar geometry.

NOTES

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfLayer

NAME

pfdAddState, pfdStateCallback, pfdGetStateCallback, pfdGetStateToken, pfdGetUniqueStateToken, pfdNewExtensor, pfdNewExtensorType, pfdCompareExtensor, pfdCompareExtraStates, pfdCopyExtraStates, pfdGetExtensor, pfdGetExtensorType, pfdUniqifyData – Flexible callback extension mechanism

FUNCTION SPECIFICATION

#include <Performer/pfdu.h>

int	**pfdAddState**(void *name, long dataSize, void (*initialize)(void *data), void (*deletor)(void *data), int (*compare)(void *data1, void *data2), long (*copy)(void *dst, void *src), int token);
void	**pfdStateCallback**(int stateToken, int whichCBack, pfNodeTravFuncType callback);
pfNodeTravFuncType	**pfdGetStateCallback**(int stateToken, int which);
int	**pfdGetStateToken**(void *name);
int	**pfdGetUniqueStateToken**(void);
pfdExtensor *	**pfdNewExtensor**(int which);
pfdExtensorType *	**pfdNewExtensorType**(int token);
int	**pfdCompareExtensor**(void *a, void *b);
int	**pfdCompareExtraStates**(void *lista, void *listb);
void	**pfdCopyExtraStates**(pfList *dst, pfList *src);
pfdExtensor *	**pfdGetExtensor**(int token);
pfdExtensorType *	**pfdGetExtensorType**(int token);
void *	**pfdUniqifyData**(pfList *dataList, const void *data, long dataSize, void *(*newData)(long), int (*compare)(void *, void *), long (*copy)(void *, void *), int *compareResult);

DESCRIPTION

```
/* Define minimal Extensor as state in the builder */
/* Note a token will be created for you and passed back if the token */
/* arg is NULL */
extern int            pfdAddState(void *name,
                          long dataSize,
                          void (*initialize)(void *data),
                          void (*deletor)(void *data),
                          int (*compare)(void *data1, void *data2),
                          long (*copy)(void *dst, void *src),
                          int token);
```

```
/* Extend Extensor Definition through callbacks listed above */
/* Specify function callbacks for an Extensor */
extern void pfdStateCallback(int stateToken, int whichCBack, pfNodeTravFuncType callback);
extern pfNodeTravFuncType pfdGetStateCallback(int stateToken, int which);

/* Look up the builder state token to use for a registered name */
extern int pfdGetStateToken(void *name);

/* Get the next Unique State token that can be used as a valid */
/* token for user state in the builder */
extern int pfdGetUniqueStateToken(void);

/* Create Generic Extensors and Extensor Definitions */
/* Note the builder creates these automatically based on user */
/* definition of extensors through pfdAddState and appropriately */
/* creates instances of an Extensor based on the current user state */
extern pfdExtensor* pfdNewExtensor(int which);
extern pfdExtensorType* pfdNewExtensorType(int token);

/* Needed to share Extensors and do internal extensor caching */
extern int pfdCompareExtensor(void *a, void *b);

/* Compare a list of Extensors */
extern int pfdCompareExtraStates(void *lista, void *listb);
extern void pfdCopyExtraStates(pfList *dst, pfList *src);

/* Find an instance of an Extensor in the Builder's current User State */
extern pfdExtensor* pfdGetExtensor(int token);

/* Find a Extensor Definitions in the Builder */
extern pfdExtensorType* pfdGetExtensorType(int token);

/* Share Arbitrary data */
extern void *pfdUniqifyData(pfList *dataList, const void *data,
        long dataSize, void *(*newData)(long),
        int (*compare)(void *,void *),
        long (*copy)(void *, void *),
        int *compareResult);
```

NOTES

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

NAME

pfdNewCube, pfdNewSphere, pfdNewCylinder, pfdNewPipe, pfdNewCone, pfdNewPyramid, pfdNewArrow, pfdNewDoubleArrow, pfdNewCircle, pfdNewRing, pfdXformGSet, pfdGSetColor – Construct simple pfGeoSets and perform simple pfGeoSet manipulations.

FUNCTION SPECIFICATION

#include <Performer/pfdu.h>

pfGeoSet * **pfdNewCube**(void *arena);

pfGeoSet * **pfdNewSphere**(int ntris, void *arena);

pfGeoSet * **pfdNewCylinder**(int ntris, void *arena);

pfGeoSet * **pfdNewPipe**(float botRadius, float topRadius, int ntris, void *arena);

pfGeoSet * **pfdNewCone**(int ntris, void *arena);

pfGeoSet * **pfdNewPyramid**(void *arena);

pfGeoSet * **pfdNewArrow**(int ntris, void *arena);

pfGeoSet * **pfdNewDoubleArrow**(int ntris, void *arena);

pfGeoSet * **pfdNewCircle**(int ntris, void *arena);

pfGeoSet * **pfdNewRing**(int ntris, void *arena);

void **pfdXformGSet**(pfGeoSet *gset, pfMatrix mat);

void **pfdGSetColor**(pfGeoSet *gset, float r, float g, float b, float a);

DESCRIPTION

These routines are provided to conveniently construct pfGeoSets for various geometric objects. The resulting objects are always positioned and sized in canonical ways. The user can then apply a transformation to these pfGeoSets to achieve the desired shape and position.

Some of these routines (such as **pfdNewSphere**) polygonalize smooth surfaces. These functions take an argument *ntris* which specifies how many triangular faces to use when polygonizing the surface. All the constructor routines allocate storage in the shared memory arena *arena*.

pfdNewCube creates a new pfGeoSet describing a unit cube.

pfdNewSphere creates a unit sphere centered at the origin.

pfdNewCylinder creates a new cylinder along the Z axis from -1 to 1 with a radius of 1.

pfdNewPipe creates a pipe (a cylinder without caps) extending along the Z axis from -1 to 1. The radius of the pipe at Z=-1 is given by *botRadius* while *topRadius* determines the pipe's radius at Z=1.

pfdNewCone creates a cone extending along the Z axis. The base of the cone is at Z=0 and has radius 1.

The cone extends to Z=1. The base of the cone is capped with a circle centered at the origin.

pfdNewPyramid creates a pyramid with a unit square base cap centered at the origin and extending along the Z axis from Z=0 to Z=1.

pfdNewArrow constructs an arrow extending along the Z axis from Z=0 to Z=1.

pfdNewDoubleArrow constructs a double-headed arrow. The arrows extend along the Z axis from Z=0 to Z=1 and Z=-1.

pfdNewCircle creates a filled unit circle centered at the origin in the XY plane. The circle is oriented so as to face in the direction of the positive Z axis.

pfdNewRing also creates a circle, but it is unfilled. The perimeter of the circle is made up of connected lines.

pfdXformGSet transforms the coordinates in the given pfGeoSet by the matrix *mat*.

pfdGSetColor is a simple convenience routine for setting the global color of a pfGeoSet. This function only works if the color binding of *gset* is **PFGS_OVERALL**.

NOTES

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfGeoSet

NAME

pfdNewGeom, pfdResizeGeom, pfdDelGeom, pfdReverseGeom, pfdNewGeoBldr, pfdDelGeoBldr, pfdGeoBldrMode, pfdGetGeoBldrMode, pfdTriangulatePoly, pfdAddGeom, pfdAddPoint, pfdAddLine, pfdAddTri, pfdAddPoly, pfdAddPoints, pfdAddLines, pfdAddLineStrips, pfdAddIndexedPoints, pfdAddIndexedLines, pfdAddIndexedLineStrips, pfdAddIndexedTri, pfdAddIndexedPoly, pfdGetNumTris, pfdBuildGSets, pfdPrintGSet – Create optimized pfGeoSets from independent geometry.

FUNCTION SPECIFICATION

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

pfdGeom *         pfdNewGeom(int numV);
void              pfdResizeGeom(pfdGeom *geom, int numV);
void              pfdDelGeom(pfdGeom *geom);
int               pfdReverseGeom(pfdGeom *geom);
pfdGeoBuilder *   pfdNewGeoBldr(void);
void              pfdDelGeoBldr(pfdGeoBuilder* bldr);
void              pfdGeoBldrMode(pfdGeoBuilder* bldr, int mode, int val);
int               pfdGetGeoBldrMode(pfdGeoBuilder* bldr, int mode);
int               pfdTriangulatePoly(pfdGeom *pgon, pfdPrim *triList);
void              pfdAddGeom(pfdGeoBuilder *bldr, pfdGeom *Geom, int num);
void              pfdAddPoint(pfdGeoBuilder *bldr, pfdPrim *Point);
void              pfdAddLine(pfdGeoBuilder *bldr, pfdPrim *line);
void              pfdAddTri(pfdGeoBuilder *bldr, pfdPrim *tri);
void              pfdAddPoly(pfdGeoBuilder *bldr, pfdGeom *poly);
void              pfdAddPoints(pfdGeoBuilder *bldr, pfdGeom *points);
void              pfdAddLines(pfdGeoBuilder *bldr, pfdGeom *lines);
void              pfdAddLineStrips(pfdGeoBuilder *bldr, pfdGeom *lineStrips, int num);
void              pfdAddIndexedPoints(pfdGeoBuilder *bldr, pfdGeom *points);
void              pfdAddIndexedLines(pfdGeoBuilder *bldr, pfdGeom *lines);
void              pfdAddIndexedLineStrips(pfdGeoBuilder *bldr, pfdGeom *lines, int num);
void              pfdAddIndexedTri(pfdGeoBuilder *bldr, pfdPrim *tri);
void              pfdAddIndexedPoly(pfdGeoBuilder *bldr, pfdGeom *poly);
```

int **pfdGetNumTris**(pfdGeoBuilder *bldr);

pfList * **pfdBuildGSets**(pfdGeoBuilder *bldr);

void **pfdPrintGSet**(pfGeoSet *gset);

```
typedef struct _pfdPrim
{
    int         flags;
    int         nbind, cbind, tbind;

    float       pixelsize;

    pfVec3      coords[3];
    pfVec3      norms[3];
    pfVec4      colors[3];
    pfVec2      texCoords[3];

    pfVec3      *coordList;
    pfVec3      *normList;
    pfVec4      *colorList;
    pfVec2      *texCoordList;

    ushort      icoords[3];
    ushort      inorms[3];
    ushort      icolors[3];
    ushort      itexCoords[3];

    struct _pfdPrim         *next;
} pfdPrim;

typedef struct _pfdGeom
{
    int         flags;
    int         nbind, cbind, tbind;

    int         numVerts;
    short       primtype;
    float       pixelsize;

    pfVec3      *coords;
    pfVec3      *norms;
    pfVec4      *colors;
    pfVec2      *texCoords;
```

```
    pfVec3      *coordList;
    pfVec3      *normList;
    pfVec4      *colorList;
    pfVec2      *texCoordList;

    ushort      *icoords;
    ushort      *inorms;
    ushort      *icolors;
    ushort      *itexCoords;

    struct _pfdGeom         *next;
} pfdGeom;

typedef pfdGeom pfdPoly;
typedef pfdPrim pfdTri;
```

DESCRIPTION

The **pfdGeoBuilder** tools greatly simplify the task of creating IRIS Performer geometry structures (-**pfGeoSets**). More importantly, the **pfdGeoBuilder** utility creates optimized line-strip and triangle-strip **pfGeoSets** that can significantly improve rendering performance and decrease memory usage.

The **pfdGeoBuilder** only manages geometry. For managing geometry and state (appearance attributes such as texture and material) there is a higher level **pfdBuilder** tool that itself uses the **pfdGeoBuilder**.

Typically the higher-level **pfdBuilder** (rather than the low-level **pfdGeoBuilder**) is used when writing a database importer for IRIS Performer. In either case, the loaders take external data in popular database file formats and convert them into IRIS Performer runtime scene-graph structures. There are many examples of file loaders based on the pfdBuilder and pfdGeoBuilder facilities in the **libpfdb** database loader library.

The pfdGeoBuilder is used to build pfGeoSets from arbitrary input geometry in the following way:

1. Create a **pfdGeoBuilder** data structure by calling **pfdNewGeoBldr**.
2. Create a pfdGeom data structure by calling **pfdNewGeom** with the maximum number of vertices required. This size can be changed later via calls to **pfdResizeGeom**.
3. Add geometric objects one at a time to the builder created in Step 1 via calls to **pfdAddGeom**, **pfdAddPoint**, **pfdAddLine**, **pfdAddTri**, **pfdAddPoly**, **pfdAddPoints**, **pfdAddLines**, **pfdAddLineStrips**, **pfdAddIndexedPoints**, **pfdAddIndexedLines**, **pfdAddIndexedLineStrips**, **pfdAddIndexedTri**, or **pfdAddIndexedPoly**.

4. Once all geometry has been added to the builder, call **pfdBuildGSets** to obtain a list of **pfGeoSet**s representing the geometry in line strips and triangle strips wherever possible.

pfdNewGeom allocates a **pfdGeom** structure capable of containing a single geometric object with up to *numV* vertices. This object contains a number of internal arrays whose sizes scale with *numV*, so allocating a large **pfdGeom** can require a considerable amount of storage.

pfdResizeGeom is used to change the vertex limit of the **pfdGeom** *geom*. The old vertex values are retained in this reallocation, so that loaders can simply invoke **pfdResizeGeom** to enlarge a **pfdGeom** without special attention to the existing data. When the new size (*numV*) is smaller than the previous size, only the first *numV* old vertices are kept.

pfdDelGeom frees the storage allocated for the pfdGeom *geom*.

pfdReverseGeom reverses the order of the vertices in *geom*. Use this to generate polygonal objects with a consistent vertex ordering (clockwise or counterclockwise) when viewed from the outside. When this is the case, backface culling can be enabled for improved graphics performance.

pfdNewGeoBldr allocates a new **pfdGeoBuilder** structure and initializes it to accept geometry. The **pfdGeoBuilder**'s internal data is self-sizing and will grow as needed when points, lines, and polygons are added to it.

pfdDelGeoBldr frees the storage allocated to the **pfdGeoBuilder** *bldr*.

pfdGeoBldrMode specifies modes to be used by the **pfdGeoBuilder** *bldr* as it processes input geometry and constructs **pfGeoSets**. The supported modes are:

PFDGBLDR_AUTO_COLORS
Generate random colors for geometric objects. There are four options for this mode, and they are:

PFDGBLDR_COLORS_PRESERVE
Leave color definitions as they are. This is the default mode.

PFDGBLDR_COLORS_MISSING
Generate colors for those primitives that do not provide them. This mode only replaces missing colors, it does not override any colors that have been defined.

PFDGBLDR_COLORS_GENERATE
Generate a new random color for each primitive.

PFDGBLDR_COLORS_DISCARD
Discard existing color definitions and do not generate any replacement colors.

PFDGBLDR_AUTO_NORMALS

Generate normals for geometric objects. There are four options for this mode, and they are:

PFDGBLDR_NORMALS_PRESERVE
: Leave normal definitions as they are. This is the default mode.

PFDGBLDR_NORMALS_MISSING
: Generate normals for those primitives that do not provide them. This mode only replaces missing normals, it does not override any normals that have been defined.

PFDGBLDR_NORMALS_GENERATE
: Generate a new normal for each primitive.

PFDGBLDR_NORMALS_DISCARD
: Discard existing normal definitions and do not generate any replacement normals.

PFDGBLDR_AUTO_TEXTURE

Generate texture coordinates for geometric objects. There are four options for this mode, and they are:

PFDGBLDR_TEXTURE_PRESERVE
: Leave texture coordinate definitions as they are. This is the default mode.

PFDGBLDR_TEXTURE_MISSING
: Generate texture coordinates for those primitives that do not provide them. This mode only replaces missing texture coordinates, it does not override any texture coordinates that have been defined. *This option is provided for future expansion. It is not currently implemented.*

PFDGBLDR_TEXTURE_GENERATE
: Generate texture coordinates for each primitive. *This option is provided for future expansion. It is not currently implemented.*

PFDGBLDR_TEXTURE_DISCARD
: Discard existing texture coordinate definitions and do not generate any replacement texture coordinates.

PFDGBLDR_AUTO_ORIENT

Automatically reverse normal vector direction or vertex order for polygons that have a supplied overall normal or per-vertex normals if the internally computed normal value indicates that the input vertices had clockwise rather than counterclockwise orientation.

PFDGBLDR_ORIENT_PRESERVE
: Do not modify vertex orientation or normal direction.

PFDGBLDR_ORIENT_NORMALS
Reverse direction (by negating normals) to make the sidedness of polygons consistent with the standard orientation, which is counterclockwise when viewed from the outside of the surface. The outside is defined as the direction in which the normal points.

PFDGBLDR_ORIENT_VERTICES
Reverse direction (by reversing the order of vertices) to make the sidedness of polygons consistent with the standard orientation, which is counterclockwise when viewed from the outside of the surface. The outside is defined as the direction in which the normal points. This is the default method, since people who provide a normal usually know which way they want it to point.

PFDGBLDR_MESH_ENABLE
Generate triangle meshes from input geometry. This task is actually performed using the **pfdMeshGSet** function. See the **pfdTMesher** man page for further details. The default is **TRUE**.

pfdGetBldrMode returns the current value of **pfdGeoBuilder** *bldr*'s internal processing mode, *mode*. The valid *mode* arguments are those listed for **pfdGeoBldrMode** above.

pfdTriangulatePoly triangulates a polygon defined by pfdGeom *pgon* and appends the resulting triangles to the list of triangles in *triList*. If the input polygon is concave, **pfdTriangulatePoly** will OR the **PFDPOLY_CONCAVE** bit into the *flags* member of the *poly* structure. The return value is **TRUE** if *poly* is concave and **FALSE** otherwise. Note that **pfdTriangulatePoly** will not "fan out" convex polygons but will "zigzag" them so the resultant triangles can be easily formed into a single triangle strip (see **pfdMeshGSet**).

Geometric objects are added to a **pfdGeoBuilder** using the general **pfdAddGeom** function or via the related functions described below. **pfdAddGeom** adds one a pfdGeom object to the designated **pfdGeoBuilder** *bldr*. If the pfdGeom is a line strip, then the argument *num* specifies the number of lines in the line strip **pfdAddGeom** is the general way to add geometry to a **pfdGeoBuilder**.

Four distinct types of geometric objects can be defined in a pfdGeom: points, lines, line strips, and polygons, and there is a lower-level primitive adding function for each: **pfdAddPoints**, **pfdAddLines**, **pfdAddLineStrips**, and **pfdAddPoly**. These functions are invoked by **pfdAddGeom** to process input geometry and are not usually called directly by users.

The **pfdGeoBuilder** also supports the optimization of indexed geometric data. It is only necessary to specify index list information in the pfdGeom structure and then call one of the indexed versions of the geometry adding functions: **pfdAddIndexedPoints**, **pfdAddIndexedLines**, **pfdAddIndexedLineStrips**, or **pfdAddIndexedPoly**.

There are four remaining geometry adding functions. These accept low-level **pfdPrim** geometry definitions rather than the higher-level **pfdGeom** definitions. Use of these functions is discouraged. **pfdAddPoint** adds the point **pfdPrim** *Point* to **pfdGeoBuilder** *bldr*, **pfdAddLine** adds the line *line*, **pfdAddTri** adds the triangle *tri*, and **pfdAddIndexedTri** adds the indexed triangle *tri*.

In all cases, these geometry processing functions copy the geometric definition into internal **pfdGeoBuilder** memory so that the application need not manage multiple **pfdGeom** or **pfdPrim** data structures. The fields of the **pfdGeom** or **pfdPrim** structure should be set as follows:

nbind, *cbind*, *tbind* specify the normal, color, and texture coordinate binding respectively. They may be one of the following values:

PFGS_PER_VERTEX
An attribute is specified for each vertex of the point, line, or polygon.

PFGS_PER_PRIM, **PFGS_OVERALL**
The first element of the attribute array specifies the attribute for the point, line, or polygon, e.g. norms[0] is the normal for the entire object.

PFGS_OFF
No attribute value is specified.

pixelsize defines the width in pixels to be used when drawing the indicated point or line. *pixelsize* is ignored for polygon data.

coords, *norms*, *colors*, *texCoords* specify the coordinates, normals, colors, and texture coordinates of the point, line, or polygon according to the binding types described above, e.g. *coords*[0], *coords*[1], *coords*[2] define the coordinates of a pfdTri.

Example 1:

```
pfdGeoBuilder *bldr;
pfdGeom       *geom;
pfList        *gsetList;

/* allocate pfdGeoBuilder and pfdGeom storage */
bldr = pfdNewGeoBldr();
geom = pfdNewGeom(4);

/* feed polygons to pfdGeoBuilder */
while (!done)
{
        :
    geom->flags = 0;
```

```
    geom->nbind = PFGS_PER_PRIM;
    geom->cbind = PFGS_OFF;
    geom->tbind = PFGS_OFF;

    geom->numVerts = 4;
    pfCopyVec3(geom->coords[0], myCoords[i]);
    pfCopyVec3(geom->coords[1], myCoords[i+1]);
    pfCopyVec3(geom->coords[2], myCoords[i+2]);
    pfCopyVec3(geom->coords[3], myCoords[i+3]);

    pfCopyVec3(geom->norms[0], myNorms[j]);

    pfdAddGeom(bldr, geom, 1);
        :
}

/* generate optimized triangle mesh GeoSet */
gsetList = pfdBuildGSets(bldr);

/* add returned pfGeoSets to geode */
for (i=0; i<pfGetNum(gsetList); i++)
    pfAddGSet(geode, pfGet(gsetList, i));

/* release pfdGeoBuilder and pfdGeom storage */
pfdDelGeoBldr(bldr);
pfdDelGeom(geom);
```

pfdGetNumTris returns the number of triangles currently contained in the **pfdGeoBuilder** structure *bldr*.

pfdBuildGSets converts all accumulated points, lines, and polygons into point, line, line strip, triangle, and triangle-strip pfGeoSets and returns a **pfList** referencing these pfGeoSets. **pfdBuildGSets** also resets *bldr*, removing the geometric definitions therein. The pfGeoSets created by the builder are meshed by **pfdMeshGSet**, subject to the meshing mode set by **pfdMesherMode**.

pfdPrintGSet prints a representation of pfGeoSet *gset* using the **pfNotify** mechanism.

NOTES

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfGeoSet, pfGeode, pfList, pfdMeshGSet, pfdMesherMode, pfdTMesher

NAME

pfdOpenFile, pfdLoadFile, pfdStoreFile, pfdConvertFrom, pfdConvertTo, pfdAddExtAlias, pfdInitConverter, pfdExitConverter, pfdConverterMode, pfdGetConverterMode, pfdConverterVal, pfdGetConverterVal, pfdConverterAttr, pfdGetConverterAttr, pfdPrintSceneGraphStats – Utilities for loading object databases into Performer applications.

FUNCTION SPECIFICATION

#include <Performer/pfdu.h>

FILE* **pfdOpenFile**(const char *file);

pfNode* **pfdLoadFile**(const char *fileName);

int **pfdStoreFile**(const char *fileName, pfNode *root);

pfNode* **pfdConvertFrom**(const char *ext, void *root);

void* **pfdConvertTo**(const char *ext, pfNode *root);

void **pfdAddExtAlias**(const char *ext, const char *alias);

int **pfdInitConverter**(const char *ext);

int **pfdExitConverter**(const char *ext);

void **pfdConverterMode**(const char *ext, int mode, int value);

int **pfdGetConverterMode**(const char *ext, int mode);

void **pfdConverterVal**(const char *ext, int which, float val);

float **pfdGetConverterVal**(const char *ext, int which);

void **pfdConverterAttr**(const char *ext, int which, void *attr);

void* **pfdGetConverterAttr**(const char *ext, int which);

void **pfdPrintSceneGraphStats**(pfNode *node, double elapsedTime);

DESCRIPTION

pfdOpenFile searches through the IRIS Performer search path for the named file and opens it using fopen(). It is a convenience function used by several of the functions described here.

pfdLoadFile builds in-memory data structures from an external database file. The filename's extension is used to determine the file format. If no path to the file is given, the directories in the active Performer file search path (see **pfFilePath**) are scanned for the given filename. **pfdLoadFile** may only be called after **pfConfig**.

pfdStoreFile writes a subgraph of a Performer scene rooted at *root* to a file in the format specified by *ext*.n

pfdConvertFrom converts the in-memory data structure *root* of the format specified by *ext* into an in-memory Performer scene. **pfdConvertTo** reverses the process taking an in-memory Performer scene and converting into the specified in-memory format.

pfdAddExtAlias registers an alias for the given file name extension. Whenever a file with extension *alias* is encountered, the loader for type *ext* will be used.

pfdInitConverter dynamically links the converter corresponding to the extension *ext* into the current executable. This routine should be called before **pfConfig** for all extensions that an executable will use to ensure that any routines and static data required at run-time are available in all Performer processes. If the corresponding loader is already in the executable, e.g. as a statically linked object, **pfdInitConverter** takes no action. **pfdInitConverter** returns TRUE if the loader is available or FALSE if it could not be found or loaded.

The search for the converter DSO proceeds through the following locations in order:

1. In the current directory.
2. In the directory indicated by the environment variable PFLD_LIBRARY_PATH, if it is defined.
3. In the directory indicated by the rld environment variable LD_LIBRARY_PATH, if it is defined.
4. In the directory "$PFHOME/usr/lib{,32,64}/libpfdb", if the environment variable PFHOME is defined. The empty, "32" and "64" lib suffix strings correspond to O32, N32 and N64 modes of compilation and execution.
5. In the directory "$PFHOME/usr/share/Performer/lib/libpfdb", if the environment variable PFHOME is defined.

The loader DSO name is created as "libpfEXT{_igl,_ogl,}{-g}.so" where "_igl" is the IRIS GL version, "_ogl" is the OpenGL version, and the "-g" suffix is for a full symbol table debug version. **pfdInitConverter** will only load the debug version of the converter DSO if it is unable to find the optimized version of the DSO in any of paths mentioned above. **pfdLoadFile** also requires that the DSO version number match that of libpfdu. When PFLD_LIBRARY__PATH is set, **pfdLoadFile** prints diagnostic information about the DSO search using **pfNotify** at the PFNFY_DEBUG notification level.

pfdExitConverter discards the database loader for the extension *ext* and unlinks any dynamically linked objects from the current executable. The only reason to call this function is to save swap space. If *ext* is NULL, all converters are unlinked.

pfdConverterMode, **pfdGetConverterMode**, **pfdConverterAttr**, **pfdGetConverterAttr**, **pfdConverterVal** and **pfdGetConverterVal** allow the user to access and alter the modes, attributes and values of specific loaders. These modes, attributes and values are defined inside each individual loader. These functions are provided as a means for the user to communicate with the loaders which are usually loaded as Dynamic Shared Objects at run-time.

pfdPrintSceneGraphStats uses **pfNotify** to print some simple statistics about the primitives in the scene graph. The *elapsedTime* argument is provided by the caller and indicates the time it took for the scene

graph to be loaded. It is used by **pfdPrintSceneGraphStats** to print primitive loading rate statistics. When the value is zero, none of the loading rate statistics are printed

The routines that take an extension as an argument may be passed a full file name, in which case the extension is extracted and used.

NOTES

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

Very few of the database DSOs provided in the current release support **pfdStoreFile**, none of them support **pfdConvertTo**, and only the Open Inventor loader supports **pfdConvertFrom**. More pervasive support of these operations is planned for future releases, however, and developers of new database conversion tools are encouraged to provide the full set of conversion functions in the database tools they develop.

When statically linking a loader library into an executable that calls **pfdLoadFile**, you can use the '-u' option to ld to force the inclusion of the loader object even though it is never referenced in the executable, e.g. "cc -o myapp myapp.o -u pfdLoadFile_iv libpfiv_igl.a ..."

SEE ALSO

pfFilePath, pfNotify, ld

NAME

pfdLoadFont, pfdLoadFont_type1 – Utilities for loading fonts into Performer applications

FUNCTION SPECIFICATION

#include <Performer/pfdu.h>

pfFont * **pfdLoadFont**(char *ext, char *fontFileName, int style);

pfFont * **pfdLoadFont_type1**(char *fontName, int style);

DESCRIPTION

pfdLoadFont tries to load a font of specific type using the *ext* to find a routine that is capable of building this type of font. The routine name is of the form pfdLoadFont_ext. This routine will then use *fontFileName* to find a file containing the font description and try to create the pfFont using this description and the *style* token. Current font style tokens include: **PFDFONT_TEXTURED**, **PFDFONT_OUTLINED**, **PFDFONT_FILLED**, **PFDFONT_EXTRUDED**, and **PFDFONT_VECTOR**. Although only some of these styles may be available for any particular font.

pfdLoadFont_type1 is one instance of a font loader. It loads Haeberli font definitions into Performer structures. Valid styles are all of the above except **PFDFONT_VECTOR**.

SEE ALSO

pfFont, pfString, pfText

NAME

pfdOpenFile – Search for and open a file.

FUNCTION SPECIFICATION

#include <Performer/pfdu.h>

FILE * **pfdOpenFile**(char *fileName);

DESCRIPTION

pfdOpenFile opens the specified file (much like **fopen**). However, rather than simply looking in the current directory, **pfdOpenFile** will search the Performer file path (see **pfFilePath**) for the given filename.

NOTES

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

fopen, pfFilePath

NAME

pfdNewShare, pfdDelShare, pfdPrintShare, pfdCountShare, pfdGetSharedList, pfdFindSharedObject, pfdAddSharedObject, pfdRemoveSharedObject, pfdNewSharedObject, pfdMakeShared, pfdMakeSharedScene, pfdCleanShare, pfdGetNodeGStateList – Facilitate the sharing of graphics state objects in a Performer scene graph.

FUNCTION SPECIFICATION

#include <Performer/pfdu.h>

pfdShare * **pfdNewShare**(void);

void **pfdDelShare**(pfdShare *share, int deepDelete);

void **pfdPrintShare**(pfdShare *share);

int **pfdCountShare**(pfdShare *share);

pfList * **pfdGetSharedList**(pfdShare *share, pfType *type);

pfObject * **pfdFindSharedObject**(pfdShare *share, pfObject *object);

int **pfdAddSharedObject**(pfdShare *share, pfObject *object);

int **pfdRemoveSharedObject**(pfdShare *share, pfObject *object);

pfObject * **pfdNewSharedObject**(pfdShare *share, pfObject *object);

void **pfdMakeShared**(pfNode *node);

void **pfdMakeSharedScene**(pfScene *scene);

int **pfdCleanShare**(pfdShare *share);

pfList * **pfdGetNodeGStateList**(pfNode *node);

DESCRIPTION

It is obviously desirable to share state between database objects in IRIS Performer whenever possible. The notion of pervasive state sharing underpins the entire **pfGeoState** mechanism. Common data such as texture, materials, and lighting models are often duplicated in many different objects throughout a database. This collection of functions provides the means necessary to easily achieve sharing among these objects by automatically producing a non-redundant set of states.

pfdNewShare constructs a new **pfdShare** structure in shared memory. This structure is the object used to track shared state objects.

pfdDelShare deletes a **pfdShare** structure. If *deepDelete* is non-NULL, all the data referenced by the **pfdShare** will also be deleted.

pfdPrintShare will print statistics about how many object references are being held in the given sharing structure.

pfdCountShare returns the total number of shared objects referenced by the given sharing structure.

Each **pfdShare** structure maintains lists of shared objects with distinct types of objects stored in distinct lists. **pfdGetSharedList** returns the list of shared objects of the given type.

pfdNewSharedObject returns a shared object matching *object*. If a matching object was already present in the sharing structure, that object is returned. If no such object exists, a new object is created using *object* as a template. This new object is entered into the sharing structure and is returned to the caller.

pfdFindSharedObject looks through the given pfdShare structure for an object matching *object*. A NULL is returned if no matching object is found.

pfdAddSharedObject adds *object* to the given sharing structure, if it is not already present. The object's index in the sharing list is returned.

pfdRemoveSharedObject removes *object* from the given sharing structure, if it is present. The object's index in the sharing list is returned. The object is deleted if its reference count reaches zero via **pfUnrefDelete**.

pfdMakeShared can be used to force sharing of state within an already existing scene graph. It will traverse the graph rooted at *node* looking for duplicate state objects. Any references to such objects will be made to point to a single shared copy, and the duplicates will be freed.

pfdMakeSharedScene is similar to pfdMakeShared except that it works on a scene and computes an optimal pfScene pfGeoState based on all of the pfGeoStates referenced in *scene* (see **pfdMakeSceneGState** for further information about the scene pfGeoState computation itself). The pfGeoStates in *scene* are then optimized based on the new scene GeoState (see **pfdOptimizeGStateList**). Lastly, the optimized pfGeoState is assigned as the scene's pfGeoState so that the inheritance for the newly optimized states will be effective (see **pfSceneGState**).

pfdCleanShare removes all shared objects that are referenced only by the shared structure itself. It is useful to call **pfdCleanShare** after deleting parts of database that were created using this share structure to release the memory allocated for currently unused state elements. **pfdCleanShare** returns the number of elements actually removed.

pfdGetNodeGStateList creates and returns a pfList of unique pfGeoStates that are referenced by geometry under *node*.

NOTES

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfGeoSet, pfGeoState, pfList

NAME

pfdSpatialize, pfdTravGetGSets – Collect and partition pfGeoSets in scene graphs.

FUNCTION SPECIFICATION

#include <Performer/pfdu.h>

pfGroup * **pfdSpatialize**(pfGroup *group, float maxGeodeSize, int maxGeoSets);

pfList * **pfdTravGetGSets**(pfNode *node);

DESCRIPTION

pfdSpatialize gathers together all the **pfGeoSets** referenced in the graph rooted at *group*. It constructs a new subgraph where all the **pfGeoSets** are grouped together by their spatial location. An octree is used to control the grouping. It is ensured that no more than *maxGeoSets* **pfGeoSets** will be grouped together in one **pfGeode** and that the spatial width of every individual **pfGeode** constructed will never exceed *maxGeodeSize*. The new graph is returned to the caller.

pfdTravGetGSets traverses the graph rooted at *node* to find all the referenced pfGeoSets. A list containing every pfGeoSet found by this traversal is returned to the caller.

NOTES

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfGeoSet

NAME

pfdMeshGSet, pfdMesherMode, pfdGetMesherMode, pfdShowStrips – Create triangle meshes from pfGeoSets.

FUNCTION SPECIFICATION

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

pfGeoSet *  pfdMeshGSet(pfGeoSet *gset);
void        pfdMesherMode(int mode, int val);
int         pfdGetMesherMode(int mode);
void        pfdShowStrips(pfGeoSet *gset);
```

DESCRIPTION

Forming independent triangles into triangle strips (or meshes) can significantly improve rendering performance on IRIS systems. Strips reduce the amount of work required by the CPU, bus, and graphics subsystem. The IRIS Performer utility mesher is adapted from the triangle mesh code supplied in "/usr/people/4Dgifts". It is modified to work with Performer **pfGeoSet** data structures and is optimized for optimal performance.

pfdMeshGSet takes as input a **pfGeoSet** consisting of independent triangles, (**PFGS_TRIS**). The input may be either indexed or non-indexed. This routine outputs the input **pfGeoSet** if it cannot strip the input or a single **pfGeoSet** that is a collection of triangle strips (PFGS_**TRISTRIPS**) if it is successful. The output **pfGeoSet** is non-indexed and the input **pfGeoSet** is not deleted although the application generally should do so to avoid wasted memory.

The mesher attempts to maximize the average length of triangle strips inside *gset*. The code is complex but works well and can significantly improve drawing performance if the average number of triangles in the triangle strips in the output GeoSets is at least four. The length of triangle strips necessary to achieve peak drawing performance is dependent on the exact hardware configuration.

pfdMesherMode sets the mode that the mesher will use when forming triangle strips. Currently two modes are supported:

PFDMESH_SHOW_TSTRIPS
Generate a random color for each triangle strip generated. This is a diagnostic mode which is extremely useful in understanding the structure and efficiency of databases.

PFDMESH_RETESSELLATE
With this mode enabled the mesher will retessellate planar quads to achieve longer strips.

pfdGetMesherMode returns the current setting of *mode*.

NOTES

If **pfdMeshGSet** generates a set of triangle strips whose lengths are all three or four i.e. they are all independent triangles or quads, the output **pfGeoSet** will be of type **PFGS_TRIS** or **PFGS_QUADS**, respectively.

pfdMeshGSet also calls **pfuHashGSetVerts** which may delete the attribute and index arrays of the input **pfGeoSet**. Thus you may wish to **pfRef** your arrays to avoid their deletion.

pfdShowStrips will assign each triangle strip a random color. The first triangle in each strip is distinguished by being slightly darker than the rest of the strip

The libpfdu source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfGeoSet, pfRef, pfuHashGSetVerts

libpfui

libpfui is a user interface management facility that manages keyboard, mouse, and window system events as well as motion models to support scene-graph manipulation.

The library contains a user interface management facility that distributes and handles keyboard, mouse, and window system events as well as direct-manipulation trackball and vehicle simulation motion models to support viewpoint and scene-graph manipulation.

NAME

pfiGetCollideClassType, pfiNewCollide, pfiEnableCollide, pfiDisableCollide, pfiGetCollideEnable, pfiCollideMode, pfiGetCollideMode, pfiCollideStatus, pfiGetCollideStatus, pfiCollideDist, pfiGetCollideDist, pfiCollideHeightAboveGrnd, pfiGetCollideHeightAboveGrnd, pfiCollideGroundNode, pfiGetCollideGroundNode, pfiCollideObjNode, pfiGetCollideObjNode, pfiCollideCurMotionParams, pfiGetCollideCurMotionParams, pfiGetCollideMotionCoord, pfiCollideFunc, pfiGetCollisionFunc, pfiUpdateCollide – pfiCollide functions.

FUNCTION SPECIFICATION

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

pfType *        pfiGetCollideClassType(void);
pfiCollide *    pfiNewCollide(void *arena);
void            pfiEnableCollide(pfiCollide *collide);
void            pfiDisableCollide(pfiCollide *collide);
int             pfiGetCollideEnable(pfiCollide *collide);
void            pfiCollideMode(pfiCollide *collide, int mode, int val);
int             pfiGetCollideMode(pfiCollide *collide, int mode);
void            pfiCollideStatus(pfiCollide *collide, int status);
int             pfiGetCollideStatus(pfiCollide *collide);
void            pfiCollideDist(pfiCollide *collide, float dist);
float           pfiGetCollideDist(pfiCollide *collide);
void            pfiCollideHeightAboveGrnd(pfiCollide *collide, float dist);
float           pfiGetCollideHeightAboveGrnd(pfiCollide *collide);
void            pfiCollideGroundNode(pfiCollide *collide, pfNode* ground);
pfNode *        pfiGetCollideGroundNode(pfiCollide *collide);
void            pfiCollideObjNode(pfiCollide *collide, pfNode* db);
pfNode *        pfiGetCollideObjNode(pfiCollide *collide);
void            pfiCollideCurMotionParams(pfiCollide *collide, pfCoord* pos, pfCoord* prevPos,
                    float speed);
void            pfiGetCollideCurMotionParams(pfiCollide *collide, pfCoord* pos, pfCoord* prevPos,
                    float *speed);
void            pfiGetCollideMotionCoord(pfiCollide *collide, pfiMotionCoord* xcoord);
```

void **pfiCollideFunc**(pfiCollide *collide, pfiCollideFuncType func, void *data);

void **pfiGetCollisionFunc**(pfiCollide *collide, pfiCollideFuncType *func, void **data);

int **pfiUpdateCollide**(pfiCollide *collide);

```
typedef int (*pfiCollideFuncType)(pfiCollide *, void *);
```

DESCRIPTION

pfiCollide functions.

The pfiCollide has a complete C++ API following the conventions of general IRIS Performer C++ API; C++ methods are declared in /usr/include/Performer/pfui/pfiCollide.h.

pfiInit should be called once before any pfiCollide routines and before the forked creation of any additional processes that will be calling pfiCollide routines.

NOTES

The libpfui source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

NAME

pfiGetInputCoordClassType, pfiNewInputCoord, pfiInputCoordVec, pfiGetInputCoordVec – pfiInputCoord functions.

FUNCTION SPECIFICATION

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

pfType*          pfiGetInputCoordClassType(void);
pfiInputCoord *  pfiNewInputCoord(void *arena);
void             pfiInputCoordVec(pfiInputCoord *ic, float *vec);
void             pfiGetInputCoordVec(pfiInputCoord *ic, float *vec);
```

DESCRIPTION

pfiInputCoord functions.

NOTES

The libpfui source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

NAME

pfiNewIXform, pfiNewInputCoord, pfiNewMotionCoord, pfiNewInput, pfiGetIXformClassType, pfiGetInputClassType, pfiGetInputCoordClassType, pfiGetMotionCoordClassType, pfiIsIXformInMotion, pfiInputName, pfiIsIXGetName, pfiInputFocus, pfiGetInputFocus, pfiInputEventMask, pfiGetInputEventMask, pfiInputEventStreamCollector, pfiGetInputEventStreamCollector, pfiInputEventStreamProcessor, pfiGetInputEventStreamProcessor, pfiInputEventHandler, pfiGetInputEventHandler, pfiIXformMat, pfiGetIXformMat, pfiIXformInput, pfiGetIXformInput, pfiIXformInputCoordPtr, pfiGetIXformInputCoordPtr, pfiIXformMotionCoord, pfiGetIXformMotionCoord, pfiIXformResetCoord, pfiGetIXformResetCoord, pfiIXformCoord, pfiGetIXformCoord, pfiIXformStartMotion, pfiGetIXformStartMotion, pfiIXformMotionLimits, pfiGetIXformMotionLimits, pfiIXformDBLimits, pfiGetIXformDBLimits, pfiIXformBSphere, pfiGetIXformBSphere, pfiIXformUpdateFunc, pfiGetIXformUpdateFunc, pfiIXformMotionFuncs, pfiGetIXformMotionFuncs, pfiResetInput, pfiCollectInputEvents, pfiProcessInputEvents, pfiStopIXform, pfiResetIXform, pfiUpdateIXform, pfiResetIXformPosition, pfiHaveFastMouseClick – pfiInputXform functions.

FUNCTION SPECIFICATION

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

pfiInputXform *     pfiNewIXform(void *arena);
pfiInputCoord *     pfiNewInputCoord(void *arena);
pfiMotionCoord *    pfiNewMotionCoord(void *arena);
pfiInput *          pfiNewInput(void *arena);
pfType *            pfiGetIXformClassType(void);
pfType *            pfiGetInputClassType(void);
pfType *            pfiGetInputCoordClassType(void);
pfType *            pfiGetMotionCoordClassType(void);
int                 pfiIsIXformInMotion(pfiInputXform *ix);
void                pfiInputName(pfiInput *in, const char *name);
const char *        pfiIsIXGetName(pfiInput *in);
void                pfiInputFocus(pfiInput *in, int focus);
int                 pfiGetInputFocus(pfiInput *in);
void                pfiInputEventMask(pfiInput *in, int emask);
int                 pfiGetInputEventMask(pfiInput *in);
void                pfiInputEventStreamCollector(pfiInput *in, pfiEventStreamHandlerType func,
                        void *data);
```

void	**pfiGetInputEventStreamCollector**(pfiInput *in, pfuEventHandlerFuncType *func, void **data);
void	**pfiInputEventStreamProcessor**(pfiInput *in, pfiEventStreamHandlerType func, void *data);
void	**pfiGetInputEventStreamProcessor**(pfiInput *in, pfuEventHandlerFuncType *func, void **data);
void	**pfiInputEventHandler**(pfiInput *in, pfuEventHandlerFuncType func, void *data);
void	**pfiGetInputEventHandler**(pfiInput *in, pfuEventHandlerFuncType *func void **data);
void	**pfiIXformMat**(pfiInputXform *ix, PFMATRIX mat);
void	**pfiGetIXformMat**(pfiInputXform *ix, PFMATRIX mat);
void	**pfiIXformInput**(pfiInputXform *_ix, pfiInput *_in);
pfiInput *	**pfiGetIXformInput**(pfiInputXform *_ix);
void	**pfiIXformInputCoordPtr**(pfiInputXform *ix, pfiInputCoord *icoord);
pfiInputCoord *	**pfiGetIXformInputCoordPtr**(pfiInputXform *ix);
void	**pfiIXformMotionCoord**(pfiInputXform *ix, pfiMotionCoord *xcoord);
void	**pfiGetIXformMotionCoord**(pfiInputXform *ix, pfiMotionCoord *xcoord);
void	**pfiIXformResetCoord**(pfiInputXform *ix, pfCoord *resetPos);
void	**pfiGetIXformResetCoord**(pfiInputXform *ix, pfCoord *resetPos);
void	**pfiIXformCoord**(pfiInputXform *ix, pfCoord *coord);
void	**pfiGetIXformCoord**(pfiInputXform *ix, pfCoord *coord);
void	**pfiIXformStartMotion**(pfiInputXform *ix, float startSpeed, float startAccel);
void	**pfiGetIXformStartMotion**(pfiInputXform *ix, float *startSpeed, float *startAccel);
void	**pfiIXformMotionLimits**(pfiInputXform *ix, float maxSpeed, float angularVel, float maxAccel);
void	**pfiGetIXformMotionLimits**(pfiInputXform *ix, float *maxSpeed, float *angularVel, float *maxAccel);
void	**pfiIXformDBLimits**(pfiInputXform *ix, pfBox *dbLimits);
void	**pfiGetIXformDBLimits**(pfiInputXform *ix, pfBox *dbLimits);
void	**pfiIXformBSphere**(pfiInputXform *ix, pfSphere *_sphere);

void **pfiGetIXformBSphere**(pfiInputXform *ix, pfSphere *_sphere);

void **pfiIXformUpdateFunc**(pfiInputXform *ix, pfiInputXformUpdateFuncType func, void *data);

void **pfiGetIXformUpdateFunc**(pfiInputXform *ix, pfiInputXformUpdateFuncType *func, void **data);

void **pfiIXformMotionFuncs**(pfiInputXform *ix, pfiInputXformFuncType start, pfiInputXformFuncType stop, void *data);

void **pfiGetIXformMotionFuncs**(pfiInputXform *ix, pfiInputXformFuncType *start, pfiInputXformFuncType *stop, void **data);

void **pfiResetInput**(pfiInput *in);

void **pfiCollectInputEvents**(pfiInput *in);

void **pfiProcessInputEvents**(pfiInput *in);

void **pfiStopIXform**(pfiInputXform *ix);

void **pfiResetIXform**(pfiInputXform *ix);

void **pfiUpdateIXform**(pfiInputXform *ix);

void **pfiResetIXformPosition**(pfiInputXform *ix);

int **pfiHaveFastMouseClick**(pfuMouse *mouse, int button, float msecs);

```
typedef int (*pfiEventStreamHandlerType)(pfiInput *, pfuEventStream *);
typedef int (*pfiInputXformFuncType)(pfiInputXform *, void *);
typedef int (*pfiInputXformUpdateFuncType)(pfiInputXform *, pfiInputCoord *, void *);
```

DESCRIPTION

pfiInputXform is a basic facility for tying together routines to get and process user input, invoke a model for computing transformations based on that input, and applying those transformations to the viewing position or transformation matrix to be applied to a database.

The pfiInputXform has a complete C++ API following the conventions of general IRIS Performer C++ API; C++ methods are declared in /usr/include/Performer/pfui/pfiInputXform.h.

pfiInit should be called once before any pfiInputXform routines and before the forked creation of any additional processes that will be calling pfiInputXform routines.

pfiGetIXformClassType returns the **pfType*** for the class **pfiXformer**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfiNewIXform creates a new **pfiInputXform** data structure as described above and returns a pointer to that structure. *arena* should specify the shared arena handle returned by **pfGetSharedArena** or from the process dynamic memory area.

pfiIXformInput makes *in* the pfiInput of the pfiInputXform *ix*. A pfiInput structure make be created with **pfiNewInput**. By default, a pfiInputXform has a NULL pfiInput*. **pfiGetIXformInput** will return the pfiInput* of the pfiInputXform *ix*.

pfiIXformInputCoordPtr makes *icoord* the pfiInputCoord of the pfiInputXform *ix*. A pfiInput structure make be created with **pfiNewInputCoord**. By default, a pfiInputXform has a NULL pfiInputCoord*. **pfiGetIXformInputCoordPtr** will return the pfiInputCoord* of the pfiInputXform *ix*.

pfiIXformMotionCoord will copy the contents of *mcoord* to the pfiMotionCoord of *ix*. A pfiMotionCoord structure make be created with **pfiNewMotionCoord**. A pfiInputXform has a fixed pfiMotionCoord structure.

pfiIXformMat sets the current transformation matrix of *ix* to be *mat*. This will also cause the current position of *ix* to be recomputed. **pfiGetIXformMat** will return the current transformation matrix of *ix*.

pfiIXformCoord sets the current position (XYZ and HPR) of *ix* to be *coord*. This will also cause the current transformation matrix of *ix* to be recomputed. **pfiGetIXformCoord** will return the current position pfCoord of *ix*.

pfiIXformResetCoord specifies the position that *ix* should reset to upon a call to **pfiResetIXformPosition**. **pfiGetIXformResetCoord** will return the current reset position. This position is by default xyz=(0.0, 0.0, 0.0) and hpr=(0.0, 0.0, 0.0).

pfiIXformDBLimits will set the database bounding box of *ix* to be *dbLimits*. A bounding sphere and database center (if not previously set with **pfiIXformBSphere**) will be automatically recalculated. The default bounding box is of size PFI_BIGDB and is centered at (0.0, 0.0, 0.0). **pfiGetIXformDBLimits** will return the current bounding box of *ix* in *dblimits*. The database size, center, and limits are often used in setting and constraining the speed and position of motion models.

pfiIXformBSphere will fix a bounding sphere *sphere* for *ix*. **pfiGetIXformBSphere** will return the current bounding sphere.

pfiIXformStartMotion specifies the starting speed and acceleration to be used by *ix* when starting motion. These parameters may be queried with **pfiGetIXformStartMotion**.

pfiIXformMotionLimits specifies the maximum speed, angular velocity, and acceleration for *ix*. These parameters may be queried with **pfiGetIXformMotionLimits**.

pfiHaveFastMouseClick will determine if a mouse button *button* of *mouse* was clicked for less than *msecs*.

pfiIXformUpdateFunc specifies the callback function and associated data that will be called upon a call to **pfiUpdateIXform** on *ix*. **pfiGetIXformUpdateFunc** will return the update callback function pointer and associated data. By default, the update callback function is NULL.

pfiInputEventStreamCollector sets the callback function and associated data for the collection of events for the event stream of the pfiInput *in*. This routine will be called upon a call to **pfiCollectInputEvents** on *in*. *pfiGetInputEventStreamCollector* **will return the current callback function and data.**

pfiInputEventStreamProcessor **sets the callback function and associated data for processing of the events stored in the event stream of the pfiInput** *in. This routine will be called upon a call to pfiProcessInputEvents* **on** *in*. **pfiGetInputEventStreamProcessor will return the current callback function and data.**

pfiIXformMode sets the specified *mode* **of** *ix* **to have value** *val*. **pfiGetIXformMode returns the value of specified** *mode* **of** *ix*. **pfiInputXform modes and corresponding values are:**

PFIX_MODE_MOTION
sets the basic motion mode for the motion model. **PFIX_MOTION_STOP** is the basic motion mode that all pfiInputXforms must understand.

PFIX_MODE_MOTION_MOD
sets a bitmask of special motion modifiers. A bitmask of 0x0 implies that motion does not undergo any special modifications beyond that expected by the current motion mode.

PFIX_MODE_ACCEL
sets the current acceleration mode for *ix*. **PFIX_ACCEL_NONE** is the basic acceleration mode that all pfiInputXforms must understand.

PFIX_MODE_AUTO
sets the automatic motion condition *val*.

PFIX_MODE_LIMIT_POS
sets a bitmask indicating the types of position limits that should be enforced by *ix*. The bitmask *val* may contain one or more of **PFIX_LIMIT_POS_HORIZ**, **PFIX_LIMIT_POS_BOTTOM**, and **PFIX_LIMIT_POS_TOP**. A value of **PFIX_LIMIT_POS_NONE** will prohibit limits on position.

PFIX_MODE_LIMIT_SPEED
sets a bitmask indicating what types of speed limits that *ix* should enforce. The bitmask *val* may contain one or more of **PFIX_LIMIT_SPEED_MAX** and **PFIX_LIMIT_SPEED_DB**. A value of **PFIX_LIMIT_SPEED_NONE** will prohibit limits on speed.

PFIX_MODE_LIMIT_ACCEL
sets a bitmask indicating what types of speed limits that *ix* should enforce. The bitmask *val* may contain one or more of **PFIX_LIMIT_ACCEL_MAX** and **PFIX_LIMIT_ACCEL_DB**. A value of **PFIX_LIMIT_ACCEL_NONE** will prohibit

limits on acceleration.

pfiStopIXform halts motion of *ix* by setting the current speed and acceleration to zero.

pfiResetIXform resets the motion and positional parameters to their initial values.

pfiResetIXformPosition sets current position of *ix* to be the current reset position, which can be set with **pfiIXformResetCoord**.

pfiUpdateIXform copies the current position of the pfiMotionCoord of *ix* to the stored previous position of the pfiMotionCoord and calls the current update callback function with its associated data.

NOTES

The libpfui source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

NAME

pfiGetIXformDriveClassType, pfiNewIXformDrive, pfiIXformDriveMode, pfiGetIXformDriveMode, pfiIXformDriveHeight, pfiGetIXformDriveHeight, pfiCreate2DIXformDrive, pfiUpdate2DIXformDrive – pfiInputXformDrive functions.

FUNCTION SPECIFICATION

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

pfType *                pfiGetIXformDriveClassType(void);
pfiInputXformDrive *    pfiNewIXformDrive(void *arena);
void                    pfiIXformDriveMode(pfiInputXformDrive *drive, int mode, int val);
int                     pfiGetIXformDriveMode(pfiInputXformDrive *drive, int mode);
void                    pfiIXformDriveHeight(pfiInputXformDrive* drive, float height);
float                   pfiGetIXformDriveHeight(pfiInputXformDrive* drive);
pfiInputXformDrive *    pfiCreate2DIXformDrive(void *arena);
int                     pfiUpdate2DIXformDrive(pfiInputXform *drive, pfiInputCoord *icoord,
                            void *data);
```

DESCRIPTION

pfiInputXformDrive functions.

A pfiInputXformDrive is a child of the pfiInputXform class and so pfiInputXform routines may be called with a pfiInputXformDrive. See the pfiInputXform reference page for information on pfiInputXform functionality. The pfiInputXformDrive has a complete C++ API following the conventions of general IRIS Performer C++ API; C++ methods are declared in /usr/include/Performer/pfui/pfiInputXformDrive.h.

pfiInit should be called once before any pfiInputXformDrive routines and before the forked creation of any additional processes that will be calling pfiInputXformDrive routines.

NOTES

The libpfui source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

NAME

pfiGetIXformFlyClassType, pfiNewIXFly, pfiIXformFlyMode, pfiGetIXformFlyMode, pfiCreate2DIXformFly, pfiUpdate2DIXformFly – pfiInputXformFly functions.

FUNCTION SPECIFICATION

#include <Performer/pfui.h>

pfType *	**pfiGetIXformFlyClassType**(void);
pfiInputXformFly *	**pfiNewIXFly**(void *arena);
void	**pfiIXformFlyMode**(pfiInputXformFly *fly, int mode, int val);
int	**pfiGetIXformFlyMode**(pfiInputXformFly *fly, int mode);
pfiInputXformFly *	**pfiCreate2DIXformFly**(void *arnea);
int	**pfiUpdate2DIXformFly**(pfiInputXform *fly, pfiInputCoord *icoord, void *data);

DESCRIPTION

pfiInputXformFly functions.

A pfiInputXformFly is a child of the pfiInputXform class and so pfiInputXform routines may be called with a pfiInputXformFly. See the pfiInputXform reference page for information on pfiInputXform functionality. The pfiInputXformFly has a complete C++ API following the conventions of general IRIS Performer C++ API; C++ methods are declared in /usr/include/Performer/pfui/pfiInputXformFly.h.

pfiInit should be called once before any pfiInputXformFly routines and before the forked creation of any additional processes that will be calling pfiInputXformFly routines.

NOTES

The libpfui source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

NAME

pfiGetIXformTrackballClassType, pfiNewIXformTrackball, pfiIXformTrackballMode, pfiGetIXformTrackballMode, pfiCreate2DIXformTrackball, pfiUpdate2DIXformTrackball – pfiInputXformTrackball functions.

FUNCTION SPECIFICATION

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

pfType *                  pfiGetIXformTrackballClassType(void);
pfiInputXformTrackball *  pfiNewIXformTrackball(void *arena);
void                      pfiIXformTrackballMode(pfiInputXformTrackball *tb, int mode, int val);
int                       pfiGetIXformTrackballMode(pfiInputXformTrackball *tb, int mode);
pfiInputXformTrackball *  pfiCreate2DIXformTrackball(void *arena);
int                       pfiUpdate2DIXformTrackball(pfiInputXform *tb, pfiInputCoord *icoord,
                              void *data);
```

DESCRIPTION

pfiInputXformTrackball functions.

A pfiInputXformTrackball is a child of the pfiInputXform class and so pfiInputXform routines may be called with a pfiInputXformTrackball. See the pfiInputXform reference page for information on pfiInputXform functionality. The pfiInputXformTrackball has a complete C++ API following the conventions of general IRIS Performer C++ API; C++ methods are declared in /usr/include/Performer/pfui/pfiTrackball.h.

pfiInit should be called once before any pfiInputXformTrackball routines and before the forked creation of any additional processes that will be calling pfiInputXformTrackball routines.

NOTES

The libpfui source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

NAME

pfiGetMotionCoordClassType, pfiNewMotionCoord – pfiMotionCoord functions.

FUNCTION SPECIFICATION

#include <Performer/pfui.h>

pfType * **pfiGetMotionCoordClassType**(void);

pfiMotionCoord * **pfiNewMotionCoord**(void *arena);

DESCRIPTION

pfiMotionCoord functions.

NOTES

The libpfui source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

NAME

pfiGetPickClassType, pfiNewPick, pfiPickMode, pfiGetPickMode, pfiPickHitFunc, pfiGetPicktHitFunc, pfiAddPickChan, pfiInsertPickChan, pfiRemovePickChan, pfiGetPickNumHits, pfiGetPickNode, pfiGetPickGSet, pfiSetupPickChans, pfiDoPick, pfiResetPick – pfiPick functions.

FUNCTION SPECIFICATION

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

pfType *    pfiGetPickClassType(void);
pfiPick *   pfiNewPick(void *arena);
void        pfiPickMode(pfiPick *pick, int mode, int val);
int         pfiGetPickMode(pfiPick *pick, int mode);
void        pfiPickHitFunc(pfiPick *pick, pfiPickFuncType func, void *data);
void        pfiGetPicktHitFunc(pfiPick *pick, pfiPickFuncType *func, void **data);
void        pfiAddPickChan(pfiPick *pick, pfChannel *chan);
void        pfiInsertPickChan(pfiPick *pick, int index, pfChannel *chan);
void        pfiRemovePickChan(pfiPick *pick, pfChannel *chan);
int         pfiGetPickNumHits(pfiPick *pick);
pfNode *    pfiGetPickNode(pfiPick *pick);
pfGeoSet *  pfiGetPickGSet(pfiPick *pick);
void        pfiSetupPickChans(pfiPick *pick);
int         pfiDoPick(pfiPick *pick, int x, int y);
void        pfiResetPick(pfiPick *pick);

   typedef int (*pfiPickFuncType)(pfiPick *, void *);
```

DESCRIPTION

pfiPick functions.

The pfiPick has a complete C++ API following the conventions of general IRIS Performer C++ API; C++ methods are declared in /usr/include/Performer/pfui/pfiPick.h.

pfiInit should be called once before any pfiPick routines and before the forked creation of any additional processes that will be calling pfiPick routines.

NOTES

The libpfui source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

NAME

pfiNewTDFXformer, pfiGetTDFXformerClassType, pfiCreateTDFXformer, pfiTDFXformerStartMotion, pfiGetTDFXformerStartMotion, pfiTDFXformerFastClickTime, pfiGetTDFXformerFastClickTime, pfiTDFXformerTrackball, pfiTDFXformerDrive, pfiTDFXformerFly, pfiGetTDFXformerTrackball, pfiGetTDFXformerDrive, pfiGetTDFXformerFly, pfiProcessTDFXformerMouseEvents, pfiProcessTDFXformerMouse, pfiProcessTDFTrackballMouse, pfiProcessTDFTravelMouse – Performer utility module used by perfly for managing a collection of motion models with a default user interface.

FUNCTION SPECIFICATION

#include <Performer/pfui.h>

pfiTDFXformer *	**pfiNewTDFXformer**(void* arena);
pfType*	**pfiGetTDFXformerClassType**(void);
pfiXformer *	**pfiCreateTDFXformer**(pfiInputXformTrackball *tb, pfiInputXformDrive *drive, pfiInputXformFly *fly, void *arena);
void	**pfiTDFXformerStartMotion**(pfiTDFXformer* xf, float startSpeed, float startAccel, float accelMult);
void	**pfiGetTDFXformerStartMotion**(pfiTDFXformer* xf, float *startSpeed, float *startAccel, float *accelMult);
void	**pfiTDFXformerFastClickTime**(pfiTDFXformer* xf, float msecs);
float	**pfiGetTDFXformerFastClickTime**(pfiXformer* xf);
void	**pfiTDFXformerTrackball**(pfiTDFXformer *xf, pfiInputXformTrackball *tb);
void	**pfiTDFXformerDrive**(pfiTDFXformer *xf, pfiInputXformDrive *tb);
void	**pfiTDFXformerFly**(pfiTDFXformer *xf, pfiInputXformFly *tb);
pfiInputXformTrackball *	**pfiGetTDFXformerTrackball**(pfiTDFXformer *xf);
pfiInputXformDrive *	**pfiGetTDFXformerDrive**(pfiTDFXformer *xf);
pfiInputXformFly *	**pfiGetTDFXformerFly**(pfiTDFXformer *xf);
int	**pfiProcessTDFXformerMouseEvents**(pfiInput *, pfuEventStream *, void *data);
void	**pfiProcessTDFXformerMouse**(pfiTDFXformer *xf, pfuMouse *mouse, pfChannel *inputChan);
void	**pfiProcessTDFTrackballMouse**(pfiTDFXformer *xf, pfiInputXformTrackball *trackball, pfuMouse *mouse);
void	**pfiProcessTDFTravelMouse**(pfiTDFXformer *xf, pfiInputXformTravel *tr, pfuMouse *mouse);

DESCRIPTION

pfiTDFXformer is a facility developed using the pfiXformer to provide a convenient utility for managing motion models derived from pfiInputXformTrackball, pfiInputXformDrive, and pfiInputXformFly and providing a default user interface for such motion models based on input received through a pfuMouse structure. This utility is used by IRIS Performer sample programs, such as perfly, in conjunction with the libpfutil input collection utilities (see the **pfuInitInput** reference page for more information).

A pfiTDFXformer is a child of the pfiXformer class and so pfiXformer routines may be called with a pfiTDFXformer. See the pfiXformer and pfiInputXform reference pages for information on other general functionality. Functionality specific to the pfiXformer are discussed here The pfiTDFXformer has a complete C++ API following the conventions of general IRIS Performer C++ API; C++ methods are declared in /usr/include/Performer/pfui/pfiXformer.h. This reference page only discusses the C API.

pfiInit should be called once before any pfiTDFXformer routines and before the forked creation of any additional processes that will be calling pfiXformer routines.

pfiGetTDFXformerClassType returns the **pfType*** for the class **pfiXformer**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfiNewTDFXformer creates a new **pfiTDFXformer** data structure as described above and returns a pointer to that structure. *arena* should specify the shared arena handle returned by **pfGetSharedArena** or from the process dynamic memory area.

pfiProcessTDFXformerMouse is the default update function used by the pfiTDFXformer and implements a interface based on input from pfuMouse and pfChannel specified with **pfiXformerAutoInput**. The type of the currently selected motion model (selected with **pfiSelectXformerModel**) is to see if it is derived from one of pfiInputXformTrackball, or pfiInputXformTravel, causes the pfiTDFXformer to invoke the mouse handling routine pfiProcessTDFTrackballMouse or pfiProcessTDFTravelMouse for the corresponding motion model type. The trackball, drive, and fly motion models have the following interpretations of mouse events:

TRACKBALL

Motion models derived from **pfiInputXformTrackball** will get the **TRACKBALL** mouse mapping. This mode causes the transformation matrix to be transformed as if the user was using the mouse pointer to spin a virtual trackball that surrounds the scene. The center of the scene is the computed center of the database supplied by **pfiXformerNode**, or the center of the supplied database bounding box via **pfiXformerLimits** or **pfiIXformDBLimits** in world-space coordinates. Trackball motion is intended to applied to the matrix of a pfDCS transforming the database.. This can be done automatically if the pfDCS was supplied with **pfiXformerAutoPosition**. The trackball transformation matrix can be requested from the pfiTDFXformer with **pfiGetXformerModelMat**. Collision

detection is disabled in trackball mode. Two kinds of motion are possible in this mode.

ROTATION

The user can "spin" the virtual trackball by holding down the middle mouse button and moving the cursor relative to where it was at the time the middle button was pressed. Thus, when the middle mouse button is down, moving the cursor horizontally on the screen will cause rotation around Performer's world-space Z-axis, while moving the cursor vertically will cause rotation around Performer's world-space X-axis. Releasing the middle mouse button while moving the cursor will cause the **pfiXformer** to continue to rotate the transformation matrix at the current rate. Holding down both the middle and right mouse buttons will cause rotation around Performer's world-space Y-axis.

TRANSLATION

The user can "move" or translate the virtual trackball in the XZ plane in Performer's world-space (the plane of the screen when the view into the **pfChannel** is directed down the Y-axis - the default in **perfly**) by holding down the left mouse button and moving the cursor relative to its position at the time the left mouse button was pressed. The user can "zoom" or translate the virtual trackball on the Y axis in Performer's world-space by holding down the right (or both left and right) mouse buttons while moving the cursor vertically on the screen.

Mouse Action	Motion Effect
Left mouse down	Translation in X and Z
Middle mouse down	Rotation around the X and Z azis
Middle+Right mouse down	Rotation about Y axis
Right mouse down: Translation along Y axis (zoom)	
Middle+Right mouse down	Translation in along Y axis (zoom)
No mouse down	mouse position is ignored.

If the motion model is changed from the trackball model to a moving viewpoint model, pfiInputXformDrive or pfiInputXformFly, the corresponding final transformations on the scene pfDCS are then transformed to the viewpoint.

DRIVE

Motion models derived from **pfiInputXformDrive** will get the **DRIVE** mouse mapping. This mode causes the transformation matrix to be transformed as if the user were using the mouse to control a car or other land-based vehicle. In this mode the cursor's position relative to the center of the screen will continue to cause relative turning of the transformation matrix. Moving the cursor to the right causes a right turn. Moving the cursor to the left causes a left turn. Keeping the cursor in the middle keeps the transformation "facing" the same direction. The left and right mouse buttons control acceleration and deceleration, respectively. If moving forward, the right mouse button will decelerate until eventually you start moving backward. The left and right buttons together will set your current speed to zero but allow you to control viewing direction. So, if you are moving forwards and you desire to be going in reverse immediately, hit the right and left mouse buttons together, then release the left mouse button. The middle mouse button allows control of viewing direction while maintaining a constant speed, or will maintain the current position if the viewer was

stopped when the middle mouse button was pushed. Additionally, when the middle mouse button is pressed, the driving height may be altered by holding down a ctrl-key and moving the mouse up and down. A single fast middle-mouse click anywhere on the screen will cause motion to stop. Additionally, if all three mouse buttons are down, motion will stop. When no mouse buttons are down, the mouse position is ignored; motion in progress when mouse buttons were pressed will continue at a constant speed.

The mouse buttons are the same as are interpreted as follows:

Mouse Action	Motion Effect
Left mouse down	Accelerate forward motion and steer
Right mouse down	Decelerate motion direction and steer
Middle mouse down	Maintain current motion and and steer
Left and Right down	Halt current motion and steer
Fast middle click	Halt all motion
No mouse down	mouse position is ignored.

FLY Motion models derived from **pfiInputXformFly** will get the **FLY** mouse mapping. This mode causes the transformation matrix to be transformed as if the user were using the mouse to direct flight in 3D space. The viewer position will follow the mouse: vertical motion of the mouse will direct motion up and down. The behavior in this mode is different from that of classic flight models where moving the mouse up pushes the nose of the aircraft up and moving the mouse down pushes the nose down. Here the pitch of the aircraft follows the mouse. Motion controls are analogous to the pfiInputXformDrive model. The left and right buttons do acceleration and deceleration, middle mouse directs heading and maintains current motion at a constant speed. Keeping the cursor in the middle of the screen will maintain current direction. The mouse buttons are interpreted this way.

Mouse Action	Motion Effect
Left mouse down	Accelerate forward motion and steer
Right mouse down	Decelerate motion direction and steer
Middle mouse down	Maintain current motion and and steer
Left and Right down	Halt current motion and steer
Fast middle click	Halt all motion
No mouse down	mouse position is ignored.

pfiTDFXformerFastClickTime will set the maximum time for a mouse button to be down and still to qualify as a "fast click" for *xf* to be *msecs*. If *msecs* is less than 0, fast click checking will be disabled and no such clicks will be recognized. The default fast click time is 300msecs. **pfiGetTDFXformerFastClickTime** returns the fast click time for *xf*.

NOTES

The libpfui source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

NAME

pfiInit, pfiGetXformerClassType, pfiNewXformer, pfiXformerModel, pfiSelectXformerModel, pfiGetXformerCurModel, pfiGetXformerCurModelIndex, pfiRemoveXformerModel, pfiRemoveXformerModelIndex, pfiStopXformer, pfiResetXformer, pfiResetXformerPosition, pfiCenterXformer, pfiXformerAutoInput, pfiXformerMat, pfiGetXformerMat, pfiXformerModelMat, pfiGetXformerModelMat, pfiXformerCoord, pfiGetXformerCoord, pfiXformerResetCoord, pfiGetXformerResetCoord, pfiXformerNode, pfiGetXformerNode, pfiXformerAutoPosition, pfiGetXformerAutoPosition, pfiXformerLimits, pfiGetXformerLimits, pfiEnableXformerCollision, pfiDisableXformerCollision, pfiGetXformerCollisionEnable, pfiXformerCollision, pfiGetXformerCollisionStatus, pfiUpdateXformer, pfiCollideXformer – Performer utility module for managing a collection of motion models.

FUNCTION SPECIFICATION

#include <Performer/pfui.h>

void	**pfiInit**(void);
extern pfType*	**pfiGetXformerClassType**(void);
pfiXformer *	**pfiNewXformer**(void* arena);
void	**pfiXformerModel**(pfiXformer* xf, int index, pfiInputXform* model);
void	**pfiSelectXformerModel**(pfiXformer* xf, int which);
pfiInputXform *	**pfiGetXformerCurModel**(pfiXformer* xf);
int	**pfiGetXformerCurModelIndex**(pfiXformer* xf);
int	**pfiRemoveXformerModel**(pfiXformer* xf, int index);
int	**pfiRemoveXformerModelIndex**(pfiXformer* xf, pfiInputXform* model);
void	**pfiStopXformer**(pfiXformer* xf);
void	**pfiResetXformer**(pfiXformer* xf);
void	**pfiResetXformerPosition**(pfiXformer* xf);
void	**pfiCenterXformer**(pfiXformer* xf);
void	**pfiXformerAutoInput**(pfiXformer* xf, pfChannel* chan, pfuMouse* mouse, pfuEventStream* events);
void	**pfiXformerMat**(pfiXformer* xf, PFMATRIX mat);
void	**pfiGetXformerMat**(pfiXformer* xf, PFMATRIX mat);
void	**pfiXformerModelMat**(pfiXformer* xf, PFMATRIX mat);
void	**pfiGetXformerModelMat**(pfiXformer* xf, PFMATRIX mat);

void	**pfiXformerCoord**(pfiXformer* xf, pfCoord *coord);
void	**pfiGetXformerCoord**(pfiXformer* xf, pfCoord *coord);
void	**pfiXformerResetCoord**(pfiXformer* xf, pfCoord *resetPos);
void	**pfiGetXformerResetCoord**(pfiXformer* xf, pfCoord *resetPos);
void	**pfiXformerNode**(pfiXformer* xf, pfNode *node);
pfNode *	**pfiGetXformerNode**(pfiXformer* xf);
void	**pfiXformerAutoPosition**(pfiXformer* xf, pfChannel *chan, pfDCS *dcs);
void	**pfiGetXformerAutoPosition**(pfiXformer* xf, pfChannel **chan, pfDCS **dcs);
void	**pfiXformerLimits**(pfiXformer* xf, float maxSpeed, float angularVel, float maxAccel, pfBox* dbLimits);
void	**pfiGetXformerLimits**(pfiXformer* xf, float *maxSpeed, float *angularVel, float *maxAccel, pfBox* dbLimits);
void	**pfiEnableXformerCollision**(pfiXformer* xf);
void	**pfiDisableXformerCollision**(pfiXformer* xf);
int	**pfiGetXformerCollisionEnable**(pfiXformer* xf);
void	**pfiXformerCollision**(pfiXformer* xf, int mode, float val, pfNode* node);
int	**pfiGetXformerCollisionStatus**(pfiXformer* xf);
void	**pfiUpdateXformer**(pfiXformer* xf);
int	**pfiCollideXformer**(pfiXformer* xf);

```
typedef struct _pfiXformer pfiXformer;
```

PARAMETERS

xf identifies a pfiXformer.

DESCRIPTION

pfiXformer is a facility developed to address the common task of updating a transformation matrix based on a vehicle motion model and control input from a user. A transformation matrix is computed using a motion model selected from a list of possible motion models maintained by the pfiXformer. This transformation matrix can then be used to update a **pfChannel's** view specification (**pfChanViewMat**) or a **pfDCS** node's matrix, or the pfiXformer can update these elements automatically if specified with **pfiXformerAutoPosition**. **pfiXformerAutoInput** can be used to set up mouse-based input to the pfiXformer.

A pfiXformer is a child of the pfiInputXform class and so pfiInputXform routines may be called with a pfiXformer. See the pfiInputXform reference page for information on pfiInputXform functionality.

Functionality specific to the pfiXformer are discussed here. The pfiXformer has a complete C++ API (more complete than the C API) following the conventions of general IRIS Performer C++ API; C++ methods are declared in /usr/include/Performer/pfui/pfiXformer.h. This reference page only discusses the C API.

pfiInit should be called once before any pfiXformer routines and before the forked creation of any additional processes that will be calling pfiXformer routines.

pfiGetXformerClassType returns the **pfType*** for the class **pfiXformer**. Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType** to test if an object is of a type derived from a Performer type rather than to test for strict equality of the **pfType***'s.

pfiNewXformer creates a new **pfiXformer** data structure as described above and returns a pointer to that structure. *arena* should specify the shared arena handle returned by **pfGetSharedArena** or from the process dynamic memory area.

pfiXformerAutoInput causes the pfiXformer to use the specified mouse and channel when collecting user input. These structures will be examined every frame, typically *xf* when **pfiUpdateXformer** is called. **pfiUpdateXformer** should be called every frame and will "update" *xf* based on the input structures specified in **pfiXformerAutoInput** and the currently selected motion model.

pfiXformerAutoPosition will cause the pfiXformer to automatically update the view and position matrices of *chan* and *dcs* automatically. One or both of *chan* or *dcs* may be NULL to prevent channel or dcs updates.

pfiXformerCollision specifies with what types of objects **xf** should collide. *mode* is a bit field that will be bitwise OR-ed into the collision mode of *xf*. Possible modes are **PFICOLLIDE_GROUND** and **PFICOLLIDE_OBJECT**. *val* specifies the minimum distance from *xf*'s origin to an object or to the ground at which *xf* is considered to have collided with that object. *node* specifies the root of the subtree of the scene graph against which *xf* will check for collisions. **pfiGetXformerCollisionStatus** returns whether or not *xf* is currently colliding with something.

pfiXformerLimits 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 **pfBox** *dbLimits*. **pfiGetXformerLimits** returns everything set by **pfiXformerLimits**.

pfiXformerModel sets *model* to put at location *index* in the motion model list of *xf*.

pfiSelectXformerModel causes the motion model at location *which* in the motion model list of *xf* to be the current active motion model of *xf*. **pfiGetXformerCurModelIndex** will return the index of the current active motion model for *xf*.

pfiRemoveXformerModel will remove *model* from the motion model list of **xf**.
pfiRemoveXformerModelIndex will replace the motion model at *index* from the motion model list of *xf* with NULL.

pfiXformerMat sets the current pfiXformer matrix to *mat* thus resetting *xf*'s transformation matrix to *mat*. Note that this also causes the *xf*'s position as specified in the coord field of the **pfiXformer** to be updated. **pfiGetXformerMat** returns *xf*'s current transform matrix by copying it into *mat*.

pfiXformerModelMat sets the current tranform matrix of the current motion model of *xf* to *mat* thus resetting *xf*'s transformation matrix to *mat*. Note that this also causes the *xf*'s position as specified in the coord field of the **pfiXformer** to be updated. **pfiGetXformerModelMat** returns the transform matrix of the current motion model of *xf* by copying it into *mat*.

pfiXformerCoord sets the current position of the **pfiXformer** *xf* to *coord* and then recalculates the overall **pfiXformer** matrix again based on this new position. Note that positions are absolute and not relative to previous positions. **pfiGetXformerCoord** returns the position currently specified by the **pfiXformer** *xf* by copying it into *coord*.

pfiXformerNode causes the pfiXformer to use *node* to automatically compute database limits and size based on the scene graph rooted at *node*. If the database limits have been previously explicitly specified with **pfiXformerLimits** or **pfiIXformDBLimits** they will not be overwritten.

pfiResetXformer causes the motion models, the collision model, and the current position and transformation matrices to be reset to their initial state.

pfiStopXformer causes the transformation matrix in *xf* to stop changing by zeroing velocity and acceleration.

pfiResetXformerPosition causes the position of the pfiXformer and its current motion model to be reset to the initial position. This stored position can be set with **pfiXformerResetCoord** and queried with **pfiGetXformerResetCoord**.

pfiCenterXformer will cause the position of the pfiXformer and its current motion model to be reset to the center of the database, computed using the current database limits as defined by **pfiXformerNode** or **pfiIXformDBLimits**.

pfiEnableXformerCollision enables the current requested collisions on **xf** which will be computed upon the call to **pfiCollideXformer**. **pfiDisableXformerCollision** disables the requested collision and a call to **pfiCollideXformer** will have no effect.

pfiCollideXformer performs collision checks of *xf* against all relevant objects (as specified in **pfiXformerCollision**) and return TRUE if *xf* has collided with something. This routine is commonly called from a forked intersection process.

NOTES

The libpfui source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfiTDFXformer, pfChannel, pfChanViewMat, pfCoord, pfDCS, pfGetSharedArena, pfMatrix

libpfutil

libpfutil is a library for facilitating application development.

This library provides functions including GUI widgets, and motion models, and window system support.

NAME

pfuBoxLOD, pfuMakeBoxGSet – Calculate node bounding boxes and build LODs from boxes.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

pfLOD * **pfuBoxLOD**(pfGroup *grp, int flat, pfVec4 *color);

pfGeoSet * **pfuMakeBoxGSet**(pfBox *box, pfVec4 color, int flat);

DESCRIPTION

These functions can be used to automatically generate very simple level-of-detail representations of a subgraph from the bounding boxes of the geometric objects contained in that subgraph. Sending a node to **pfuBoxLOD** creates multiple representations for that node by finding bounding boxes at different levels of the scene graph and then using those boxes as level-of-detail models under **pfLOD** nodes. The highest level-of-detail under the **pfLOD** is simply the original subgraph. The next highest level-of-detail consists of the subgraph with each **pfGeode's pfGeoSets** replaced by a single **pfGeoSet** depicting a box. Next every **pfGeode** is replaced by a single "box" **pfGeode**. This continues to progressively lower levels-of-detail as the "box" **pfGeodes** contain larger and larger subgraphs, until the box rendered for the lowest level-of-detail contains the entire subgraph. The new partial subgraphs are created using **pfClone**. The LOD transition ranges (see **pfLODRange**) are set as integer multiples of the spatial extent of the subgraph.

pfuBoxLOD takes a subgraph *grp*, a color vector *color*, and a shading type *flat*. *grp* specifies the existing scene graph hierarchy to be processed. The *color* argument provides the color definition used for the boxes that will be built. The shading type *flat* determines each **pfGeoSet**'s primitive type and normals. If *flat* is non-zero, the generated **pfGeoSet**'s are made up of **PFGS_FLAT_TRISTRIP** primitives with face directed normals. If *flat* is zero, they consist of **PFGS_TRISTRIP** primitives with normals directed radially outwards. The former produces sharply defined cubes, the latter marshmallows.

To determine the appropriate size of the box geometry, **pfuBoxLOD** uses **pfuTravCalcBBox** to get a tighter bound on the subgraphs than **pfGetNodeBSphere** provides.

pfuMakeBoxGSet takes a **pfBox** *box* and returns a **pfGeoSet** containing drawable geometry that represents the box with color *color* and a shading type determined by *flat*, as explained in the description of **pfuBoxLOD** above. The **pfGeoSet** and associated arrays are allocated from the current shared memory arena (see **pfGetSharedArena**).

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

BUGS

The LOD transition range setting is very primitive.

SEE ALSO

pfGeoSet, pfLOD, pfLODRange, pfuTravCalcBBox, pfNodeBSphere, pfScene, pfClone, pfGetSharedArena

NAME

pfuCollisionChan, pfuGetCollisionChan, pfuCollideSetup, pfuCollideGrnd, pfuCollideObj, pfuCollideGrndObj – Terrain following and collision routines.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

void **pfuCollisionChan**(pfChannel *chan);

pfChannel * **pfuGetCollisionChan**(void);

void **pfuCollideSetup**(pfNode *node, int mode, int mask);

int **pfuCollideGrnd**(pfCoord *coord, pfNode *node, pfVec3 zpr);

int **pfuCollideObj**(pfSeg *seg, pfNode *objNode, pfVec3 hitPos, pfVec3 hitNorm);

int **pfuCollideGrndObj**(pfCoord *coord, pfNode *grndNode, pfVec3 zpr, pfSeg *seg, pfNode *objNode, pfVec3 hitPos, pfVec3 hitNorm);

DESCRIPTION

This collection of simple routines provides intersection traversals for basic collision detection and ground following. Examples of **pfuCollide** usage can be found in the IRIS Performer transformation utility, **pfuXformer**.

pfuCollisionChan sets the channel from which to derive LOD scale and viewing information. This is used to determine which LOD to collide against.

pfuGetCollisionChan returns the current channel set by **pfuCollisionChan.**

pfuCollideSetup sets the intersection mask of the subgraph rooted by *node* to *mask*. The values *mask* can take are the same as in **pfNodeTravMask**. *mode* is either

PFUCOLLIDE_STATIC
Geometry below *node* is considered static and intersection caching will be enabled (see **pfNodeTravMask**) or

PFUCOLLIDE_DYNAMIC
Geometry below *node* is considered dynamic and intersection caching will not be enabled (see **pfNodeTravMask**)

Best intersection performance is achieved for static geometry.

pfuCollideGrnd fires a ray downward (in negative Z direction) from *coord* and returns **PFUCOLLIDE_GROUND** if an intersection was found with the subgraph rooted by *node* and **FALSE** otherwise. The height and orientation of the intersected geometry is returned in *zpr* which contains the height, and the pitch and roll angles of the surface at the intersection point.

pfuCollideObj intersects *seg* with the subgraph rooted by *objNode* and returns **PFUCOLLIDE_OBJECT** if

an intersection was found and **FALSE** otherwise. The intersection point and surface normal is returned in *hitPos* and *hitNorm* respectively. *seg* is typically a velocity vector used to detect collisions in the direction of travel.

pfuCollideGrndObj is the combination of **pfuCollideObj** and **pfuCollideGrnd**. It may provide better performance over calling these routines separately when *grndNode* and *objNode* are identical. The return value is a bitmask of **PFUCOLLIDE_GROUND** and **PFUCOLLIDE_OBJECT** indicating whether the downward-directed ray and/or the line segment hit anything.

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfHit, pfNode, pfNodeIsectSegs, pfNodeTravMask

NAME

pfuCreateDftCursor, pfuLoadWinCursor, pfuLoadPWinCursor, pfuGetInvisibleCursor, pfuCursor, pfuGetCursor – Create and load cursors for pfWindows and pfPipeWindows.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

Cursor **pfuCreateDftCursor**(int index);

void **pfuLoadWinCursor**(pfWindow *win, int index);

void **pfuLoadPWinCursor**(pfPipeWindow *pwin, int index);

Cursor **pfuGetInvisibleCursor**(void);

void **pfuCursor**(Cursor c, int index);

Cursor **pfuGetCursor**(int index);

DESCRIPTION

These functions allow different types of cursors to be created and associated with **pfWindows** and **pfPipeWindows**.

pfuCreateDftCursor creates and returns a default cursor corresponding to type *index*.

pfuLoadWinCursor loads the cursor of type *index* into the **pfWindow** *win*. **pfuLoadPWinCursor** does the same for the **pfPipeWindow** *pwin*. Definitions are provided in pfutil.h for the standard cursor types. Such cursors have definitions of the form PFU_CURSOR_*name*, where *name* is one of:

circle, hand1, arrow, based_arrow_down, based_arrow_up, boat, bogosity, bottom_left_corner, bottom_right_corner, bottom_side, bottom_tee, box_spiral, center_ptr, clock, coffee_mug, cross, cross_reverse, crosshair, diamond_cross, dot, dotbox, double_arrow, draft_large, draft_small, draped_box, exchange, fleur, gobbler, gumby, hand2, heart, icon, iron_cross, left_ptr, left_side, left_tee, leftbutton, ll_angle, lr_angle, man, middlebutton, mouse, pencil, pirate, plus, question_arrow, right_ptr, right_side, right_tee, rightbutton, rtl_logo, sailboat, sb_down_arrow, sb_h_double_arrow, sb_left_arrow, sb_right_arrow, sb_up_arrow, sb_v_double_arrow, shuttle, sizing, spider, spraycan, star, target, tcross, top_left_arrow, top_left_corner, top_right_corner, top_side, top_tee, trek, ul_angle, umbrella, ur_angle, watch, or xterm.

pfuGetInvisibleCursor creates and returns an invisible cursor.

pfuCursor allows the user to change the default cursor associated with a given type. **pfuCursor** sets the cursor of type *index* to be *c*. **pfuGetCursor** returns the cursor of type *index*.

Use the **xfd** command with the "-fn cursor" option to see a list of the different available cursor types. The IRIS Performer header file "/usr/include/Performer/pfutil.h" contains a list of all the cursor names.

NOTES

Only even numbered cursor types are supported, even though **xfd** will display cursors of both odd and even types.

SEE ALSO

xfd, pfuGUI, pfuXFont

NAME

pfuNewEventQ, pfuResetEventStream, pfuResetEventQ, pfuAppendEventQStream, pfuAppendEventQ, pfuEventQStream, pfuGetEventQStream, pfuGetEventQEvents, pfuIncEventQFrame, pfuEventQFrame, pfuGetEventQFrame, pfuIncEventStreamFrame, pfuEventStreamFrame, pfuGetEventStreamFrame – Event queue and event stream management utilities

FUNCTION SPECIFICATION

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

pfuEventQueue *     pfuNewEventQ(pfDataPool *dp, int id);
void                pfuResetEventStream(pfuEventStream *es);
void                pfuResetEventQ(pfuEventQueue *eq);
void                pfuAppendEventQStream(pfuEventQueue *dst, pfuEventStream *src);
void                pfuAppendEventQ(pfuEventQueue *dst, pfuEventQueue *src);
void                pfuEventQStream(pfuEventQueue *eq, pfuEventStream *es);
pfuEventStream *    pfuGetEventQStream(pfuEventQueue *eq);
void                pfuGetEventQEvents(pfuEventStream *events, pfuEventQueue *eq);
void                pfuIncEventQFrame(pfuEventQueue *eq);
void                pfuEventQFrame(pfuEventQueue *eq, int val);
int                 pfuGetEventQFrame(pfuEventQueue *eq);
void                pfuIncEventStreamFrame(pfuEventStream *es);
void                pfuEventStreamFrame(pfuEventStream *es, int val);
int                 pfuGetEventStreamFrame(pfuEventStream *es);
```

These functions provide a means to manage streams and queues of X and GL input events. **pfuEventQueue**s are compressed frames of events. **pfuEventStream**s are sequences of events. They offer a way to manipulate **pfuEventQueue**s.

pfuNewEventQ creates, initializes and returns a new event queue. *dp* is the **pfDataPool** from which memory for the new event queue is allocated. *id* is the handle by which the event queue is identified. New memory is allocated for the event queue only if an event queue with the same *id* is not already present in *dp*.

pfuResetEventStream resets the event stream *es* to a null event stream. **pfuResetEventQ** resets the event stream associated with the event queue *eq*.

pfuAppendEventQStream appends the event stream *src* to the event queue *dst*. This is done by copying the events in *src* to the end of the event stream associated with *dst*. **pfuAppendEventQ** appends the event stream associated with *src* to *dst*.

pfuEventQStream sets the event stream associated with *eq* to be *es*. **pfuGetEventQStream** returns a pointer to the event stream associated with *eq*.

pfuGetEventQEvents copies the event stream associated with *src* into *dst*.

pfuIncEventQFrame increments the frame stamp of the event stream associated with *eq* by 1 or sets the frame stamp to 0 if it is negative.

pfuEventQFrame sets the frame stamp of the event stream associated with *eq* to *val*. **pfuGetEventQFrame** returns the frame stamp of the event stream associated with *eq*.

pfuIncEventStreamFrame increments the frame stamp of the event stream *es* by 1 or sets the frame stamp to 0 if it is negative.

pfuEventStreamFrame sets the frame stamp of the event stream *es* to *val*. **pfuGetEventStreamFrame** returns the frame stamp of the event stream *es*.

SEE ALSO

pfDataPool, pfuGUI, pfuInitInput

NAME

pfuOpenFlybox, pfuReadFlybox, pfuGetFlybox, pfuGetFlyboxActive, pfuInitFlybox – Routines to read the BG Systems flybox.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

int **pfuOpenFlybox**(char *port);

int **pfuReadFlybox**(int *dioval, float *inbuf);

int **pfuGetFlybox**(float *analog, int *but);

int **pfuGetFlyboxActive**(void);

int **pfuInitFlybox**(void);

DESCRIPTION

These routines provide a simple interface to the BG Systems flybox but do not provide a flight model based on the flybox.

pfuOpenFlybox opens the flybox port specified by *port*.

pfuReadFlybox reads the values of the flybox digital inputs into *dioval* and stores the eight analog input values into *inbuf*.

pfuGetFlybox reads the values of the flybox analog inputs into *analog* after setting them to zero if their absolute values are very small. It also stores the flybox digital inputs into *but*.

pfuGetFlyboxActive returns whether the flybox is currently active.

pfuInitFlybox initializes the flybox. It uses **pfuOpenFlybox** and **pfuGetFlybox**.

NOTES

BG Systems Inc. is located in Palo Alto, California. Their telephone number is (415) 858-2628.

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

NAME

pfuOpenXDisplay, pfuGLXWinopen, pfuGetGLXWin, pfuGetGLXDisplayString, pfuGLXAllocColormap, pfuGLXMapcolors, pfuGLMapcolors, pfuMapWinColors, pfuMapPWinColors, pfuPrintWinFBConfig, pfuPrintPWinFBConfig, pfuChooseFBConfig – Open X display, open GLX window, get GLX window handle.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

pfuXDisplay * **pfuOpenXDisplay**(int screen);

pfuGLXWindow * **pfuGLXWinopen**(pfPipe *pipe, pfPipeWindow* pipewin, const char *name);

void **pfuGetGLXWin**(pfPipe *pipe, pfuGLXWindow *win);

const char * **pfuGetGLXDisplayString**(pfPipe *pipe);

int **pfuGLXAllocColormap**(pfuXDisplay *dsp, pfuXWindow win);

void **pfuGLXMapcolors**(pfuXDisplay *dsp, pfuXWindow win, pfVec3 *colors, int loc, int num);

void **pfuGLMapcolors**(pfVec3 *colors, int loc, int num);

void **pfuMapWinColors**(pfWindow *win, pfVec3 *colors, int loc, int num);

void **pfuMapPWinColors**(pfPipeWindow *pwin, pfVec3 *colors, int loc, int num);

void **pfuPrintWinFBConfig**(pfWindow *win, FILE *_file);

void **pfuPrintPWinFBConfig**(pfPipeWindow *pwin, FILE *_file);

pfFBConfig **pfuChooseFBConfig**(Display *dsp, int screen, int *constraints, void *arena);

```
typedef Window pfuXWindow;
typedef uint   pfuXDisplay;

typedef struct _pfuGLXWindow
{
    pfPipeWindow *pw;
    pfuXDisplay *dsp;
    pfuXWindow xWin;
    pfuXWindow glWin;
    pfuXWindow overWin;

} pfuGLXWindow;
```

DESCRIPTION

pfuInitUtil should be called immediately after **pfConfig** when using these routines to initialize shared memory used by the utility library.

pfuOpenXDisplay opens and returns a handle to a connection to the X server on *screen*. If *screen* is 1, then the default display, or that specified by the shell environment variable **DISPLAY** is used. **pfuOpenXDisplay** returns NULL if the X display cannot be opened. These routines are provided only for compatibility with previous IRIS Performer releases and new development should use the window system utilities in libpr, such as **pfOpenWSConnection** and **pfOpenScreen**.

pfuGLXWinopen opens and returns a handle to a GLX window for **pfPipe** *pipe*, using the existing **pfPipeWindow** *pipewin* or making a new **pfPipeWindow** for *pipe* if *pipewin* is NULL. The window's title bar is set to *name*.

pfuGetGLXWin returns a handle in *win* to the GLX window for pfPipe *pipe*, and returns NULL if no such window can be found.

pfuGetGLXDisplayString returns the name of the X Display stored for *pipe*.

pfuGLXAllocColormap allocates color map entries for the window specified in *win* and returns the number of colors available. *dsp* is *win*'s connection to the X server.

pfuGLMapcolors replaces *num* IRIS GL color map entries. *win*, starting at color map index *loc*. The *colors* array contains the color values to map, normalized in the range [0, 1]. The red component is stored in *colors[i][0]*, the green component in *colors[i][1]*, and the blue component in *colors[i][2]*.

pfuGLXMapcolors replaces *num* color map entries in the X window *win*, starting at color map index *loc*. *dsp* is a valid X connection. The *colors* array contains the color values to map, normalized in the range [0, 1]. The red component is stored in *colors[i][0]*, the green component in *colors[i][1]*, and the blue component in *colors[i][2]*.

pfuMapWinColors replaces *num* color map entries in the pfWindow *win*, starting at color map index *loc*. *win* can be pure a IRIS GL, GLX, or OpenGL/X window. If *win* is a pure IRIS GL window, this call must be made in the rendering process. The *colors* array contains the color values to map, normalized in the range [0, 1]. The red component is stored in *colors[i][0]*, the green component in *colors[i][1]*, and the blue component in *colors[i][2]*. **pfuMapPWinColors** is the analogous routine for pfPipeWindows. If *pwin* is a pure IRIS GL window, this call must be made in the rendering process.

pfuPrintWinFBConfig prints the framebuffer configuration of the pfWindow, *win*, to the specified file *file*. If *file* is NULL, stderr is used. **pfuPrintPWinFBConfig** prints the framebuffer configuration of the pfPipeWindow, *pwin*, to the specified file *file*.

pfuChooseFBConfig is an OpenGL X framebuffer configuration chooser for selecting X Visuals for OpenGL/X windows. This chooser limits performance critical attributes: multisamples, size of depth buffer, RGB color, and stencil. If the provided pfWSConnection is NULL, the current pfWSConnection libpr will be used. See the **pfGetCurWSConnection** reference page for more information. If the provided screen is (-1), the default screen of the pfWSConnection will be used.

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfuInitUtil, pfMultipipe

NAME

pfuInitGUI, pfuExitGUI, pfuEnableGUI, pfuUpdateGUI, pfuRedrawGUI, pfuGUIViewport, pfuGetGUIViewport, pfuInGUI, pfuInitGUICursors, pfuGUICursor, pfuGetGUICursor, pfuGUIHlight, pfuGetGUIHlight, pfuGUICursorSel, pfuGetGUICursorSel, pfuUpdateGUICursor, pfuFitWidgets, pfuGetGUIScale, pfuGetGUITranslation, pfuNewPanel, pfuEnablePanel, pfuDisablePanel, pfuGetPanelOriginSize, pfuNewWidget, pfuDisableWidget, pfuEnableWidget, pfuGetWidgetType, pfuGetWidgetId, pfuWidgetDim, pfuGetWidgetDim, pfuWidgetLabel, pfuGetWidgetLabelWidth, pfuGetWidgetLabel, pfuWidgetRange, pfuWidgetValue, pfuGetWidgetValue, pfuWidgetDefaultValue, pfuWidgetActionFunc, pfuGetWidgetActionFunc, pfuWidgetSelectFunc, pfuGetWidgetSelectFunc, pfuWidgetDrawFunc, pfuGetWidgetDrawFunc, pfuWidgetSelections, pfuWidgetSelection, pfuGetWidgetSelection, pfuWidgetDefaultSelection, pfuWidgetDefaultOnOff, pfuWidgetOnOff, pfuIsWidgetOn, pfuResetGUI, pfuResetPanel, pfuResetWidget, pfuDrawMessage, pfuDrawMessageCI, pfuDrawMessageRGB, pfuDrawTree – High-performance graphical user interface routines.

FUNCTION SPECIFICATION

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

void                    pfuInitGUI(pfPipe *pipe);
void                    pfuExitGUI(void);
void                    pfuEnableGUI(int en);
void                    pfuUpdateGUI(pfuMouse *mouse);
void                    pfuRedrawGUI(void);
void                    pfuGUIViewport(float l, float r, float b, float t);
void                    pfuGetGUIViewport(float *l, float *r, float *b, float *t);
int                     pfuInGUI(int x, int y);
void                    pfuInitGUICursors(void);
void                    pfuGUICursor(int target, int c);
int                     pfuGetGUICursor(int target);
void                    pfuGUIHlight(pfHighlight *hlight);
pfHighlight *           pfuGetGUIHlight(void);
void                    pfuGUICursorSel(int c);
int                     pfuGetGUICursorSel(void);
void                    pfuUpdateGUICursor(void);
void                    pfuFitWidgets(int val);
```

void	**pfuGetGUIScale**(float *x, float *y);
void	**pfuGetGUITranslation**(float *x, float *y);
pfuPanel *	**pfuNewPanel**(void);
void	**pfuEnablePanel**(pfuPanel *p);
void	**pfuDisablePanel**(pfuPanel *p);
void	**pfuGetPanelOriginSize**(pfuPanel *p, float *xo, float *yo, float *xs, float *ys);
pfuWidget *	**pfuNewWidget**(pfuPanel *p, int type, int id);
void	**pfuDisableWidget**(pfuWidget *w);
void	**pfuEnableWidget**(pfuWidget *w);
int	**pfuGetWidgetType**(pfuWidget *w);
int	**pfuGetWidgetId**(pfuWidget *w);
void	**pfuWidgetDim**(pfuWidget *w, int xo, int yo, int xs, int ys);
void	**pfuGetWidgetDim**(pfuWidget *w, int *xo, int *yo, int *xs, int *ys);
void	**pfuWidgetLabel**(pfuWidget *w, const char *label);
int	**pfuGetWidgetLabelWidth**(pfuWidget *w);
const char *	**pfuGetWidgetLabel**(pfuWidget *w);
void	**pfuWidgetRange**(pfuWidget *w, int mode, float min, float max, float val);
void	**pfuWidgetValue**(pfuWidget *w, float newval);
float	**pfuGetWidgetValue**(pfuWidget *w);
void	**pfuWidgetDefaultValue**(pfuWidget *w, float val);
void	**pfuWidgetActionFunc**(pfuWidget *w, pfuWidgetActionFuncType func);
pfuWidgetActionFuncType	**pfuGetWidgetActionFunc**(pfuWidget *w);
void	**pfuWidgetSelectFunc**(pfuWidget *w, pfuWidgetSelectFuncType func);
pfuWidgetSelectFuncType	**pfuGetWidgetSelectFunc**(pfuWidget *w);
void	**pfuWidgetDrawFunc**(pfuWidget *w, pfuWidgetDrawFuncType func);
pfuWidgetDrawFuncType	**pfuGetWidgetDrawFunc**(pfuWidget *w);
void	**pfuWidgetSelections**(pfuWidget *w, pfuGUIString *selectionList, int *valList, void (**funcList)(pfuWidget *w), int numSelections);
void	**pfuWidgetSelection**(pfuWidget *w, int index);

int	**pfuGetWidgetSelection**(pfuWidget *w);
void	**pfuWidgetDefaultSelection**(pfuWidget *w, int index);
void	**pfuWidgetDefaultOnOff**(pfuWidget * w, int on);
void	**pfuWidgetOnOff**(pfuWidget *w, int on);
int	**pfuIsWidgetOn**(pfuWidget *w);
void	**pfuResetGUI**(void);
void	**pfuResetPanel**(pfuPanel *p);
void	**pfuResetWidget**(pfuWidget *w);
void	**pfuDrawMessage**(pfChannel *chan, const char *msg, int rel, int just, float x, float y, int size, int cimode);
void	**pfuDrawMessageCI**(pfChannel *chan, const char *msg, int rel, int just, float x, float y, int size, int textClr, int shadowClr);
void	**pfuDrawMessageRGB**(pfChannel *chan, const char *msg, int rel, int just, float x, float y, int size, pfVec4 textClr, pfVec4 shadowClr);
void	**pfuDrawTree**(pfChannel *chan, pfNode *tree, pfVec3 panScale);

```
typedef void (*pfuWidgetDrawFuncType)(pfuWidget *widget, pfuPanel *panel);
typedef pfuWidget* (*pfuWidgetSelectFuncType)(pfuWidget *widget, pfuPanel *panel);
typedef void (*pfuWidgetActionFuncType)(pfuWidget *widget);
```

DESCRIPTION

These functions define a simple graphical user interface library for the IRIS Performer sample application **perfly**.

Call **pfuInitGUI** to initialize the GUI module. Call **pfuExitGUI** on exit to deallocate GUI data structures and print brief frame statistics through **pfNotify** at notification level **PFNFY_INFO**.

pfuEnableGUI disables or enables GUI processing and GUI panel updates depending on the value of *en*. **pfuUpdateGUI** uses the current mouse data in *mouse* to draw the GUI according to its current configuration. The mouse data can be collected by your own input handling routines and stored in a pfuMouse structure to give to the GUI or retrieved from the libpfutil input collector using **pfuGetMouse**. **pfuRedrawGUI** redraws the GUI.

pfuGUIViewport sets the GUI coordinates relative to the GUI window. **pfuGetGUIViewport** retrieves the current coordinates.

pfuInGUI returns true if the point *(x, y)* lies within the GUI. *x* and *y* should be given relative to the

window, not the screen.

Your application should call **pfuFitWidgets** if you change your panel layout after initialization so that the widgets in the window are accurately scaled and translated to fit inside the window. This function is called automatically from **pfuUpdateGUI** if a change in window size is detected. *val* is currently ignored.

pfuGetGUIScale can be called to retrieve the relative scale of the GUI to the display window. Similarly, **pfuGetGUITranslation** will retrieve the current translation.

pfuNewPanel allocates and initializes a new GUI panel, **pfuEnablePanel** is used to enable a panel, and **pfuDisablePanel** disables a panel. GUI panels are essentially a region containing widgets, rectangular regions which can be customized as buttons, menus, and other interface items. **pfuResetPanel** restores the widgets in *panel* to their default values. **pfuGetPanelOriginSize** retrieves the origin and size for *panel*.

pfuNewWidget allocates and initializes a new widget of type *type*. *id* is the handle by which the new widget is identified. **pfuGetWidgetType** returns *w*'s type while **pfuGetWidgetId** returns *w*'s id. Use **pfuEnableWidget** and **pfuDisableWidget** to enable or disable the action of a widget *w*.

pfuWidgetDim sets *w*'s lower left corner at *(xo, yo)* and its top right corner at *(xo+xs, yo+ys)*. **pfuGetWidgetDim** returns *w* in *xo* and *yo* and its size in *xs* and *ys*.

pfuWidgetLabel sets *label* to be the label by which *w* will be identified. **pfuGetWidgetLabelWidth** returns the length of *w*'s label while **pfuGetWidgetLabel** returns the label itself.

pfuWidgetRange sets the range and initial value of *w* when *mode* takes the values **PFUGUI_SLIDER** or **PFUGUI_SLIDER_LOG**. When *mode* is **PFUGUI_SLIDER**, *w*'s range is set to be (*min*, *max*) and its initial value is set to *val*. When *mode* is **PFUGUI_SLIDER_LOG**, the base-10 logarithms of *min*, *max* and *val* are used.

pfuWidgetValue essentially sets *w*'s value to *newval*. The actions are different depending on *w*'s type.

PFUGUI_SLIDER
w's value is set to *newval*.

PFUGUI_SLIDER_LOG
w's value is set to the base-10 logarithm of *newval*.

PFUGUI_SWITCH,
w's value is set to its minimum possible provided *newval* is close to the minimum. Otherwise, *w*'s value is set to its maximum possible.

PFGUI_BUTTON
w is highlighted if *newval* is non-zero, otherwise it is unhighlighted.

PFUGUI_MENU_BUTTON
w's value is set to the index of the selection whose value is close to *newval*.

PFUGUI_RADIO_BUTTON or **PFUGUI_RADIO_BUTTON_TOGGLE**
w's value is set to the index of the selection whose value is close to *newval*.

pfuGetWidgetValue returns *w*'s value. **pfuWidgetDefaultValue** works just like **pfuWidgetValue** but sets *w*'s default value to *newval* instead.

pfuWidgetActionFunc is used to set the callback function when the mouse button is clicked on the widget. **pfuGetWidgetActionFunc** will return the widget's action callback function. **pfuWidgetSelectFunc** and **pfuWidgetDrawFunc** are used to set up custom widgets. **pfuWidgetSelectFunc** is called with a custom selection function in *func*. *func* is called when the mouse button is clicked on the widget; *func* returns a pointer to another widget, whose action function is called instead of the custom widget's action function. If **pfuWidgetDrawFunc** is called to set a custom draw function, that function will be called instead of the built-in draw function when that widget needs to be redrawn. **pfuGetWidgetDrawFunc** and **pfuGetWidgetSelectFunc** will return the custom draw and selection callback functions, respectively.

pfuWidgetSelections sets the selection choices for the widget *w*, which must be of type **PFUGUI_RADIO_BUTTON** or **PFUGUI_MENU_BUTTON**. *val* contains a list of the values assigned to each of the selections; if *val* is NULL, then the selection value will be set to the ordinal index of the selection. *funcList* contains a list of action functions corresponding to each selection; if *funcList* is NULL, then the widget's default action function will be used for all of the selections. Use **pfuWidgetSelection** to set the current selection. Use **pfuGetWidgetSelection** to retrieve the selection last chosen or set. Use **pfuWidgetDefaultSelection** to set the selection which will be set when the widget (or the panel that contains it) is reset.

pfuWidgetOnOff changes various fields of *w* depending on whether *on* is 0 or 1. These changes essentially correspond to setting the widget off or on. Exactly how *w* is changed depends on its type as follows.

PFUGUI_SWITCH
If *on* is 0, *w*'s value is set to its minimum. Otherwise, it is set to the maximum possible value.

PFUGUI_BUTTON
If *on* is 1, *w*'s value is set to 1 and *w* is highlighted. If *on* is 0, *w*'s value is set to 0 and *w* is restored to its normal, unhighlighted state.

PFUGUI_RADIO_BUTTON
If *on* is 1, *w* is set on; otherwise it is set off.

pfuWidgetDefaultOnOff works essentially like **pfuWidgetOnOff**. If *w* is of type **PFUGUI_SWITCH**, *w*'s

default value is modified. If *w* is of type **PFUGUI_RADIO_BUTTON**, *w*'s default on value is modified.

pfuIsWidgetOn returns the status of *w* - whether it is on or off.

pfuResetWidget resets *w* to its default values and calls *w*'s action function if that function has been set.

pfuDrawMessage draws the message *msg* justified with respect to (*x*, *y*) in a font of size *size*. *x* and *y* must range between 0 and 1 and *size* should take the value **PFU_FONT_SMALL**, **PFU_FONT_MED**, or **PFU_FONT_BIG**. *just* specifies the justification and can take the value **PFU_CENTER_JUSTIFIED**, **PFU_LEFT_JUSTIFIED** or **PFU_RIGHT_JUSTIFIED**. Set *cimode* to **PFU_CI** to use color-indexing mode or **PFU_RGB** to use direct color mode. *rel* controls whether *x* and *y* are specified as absolute values or relative to *chan*'s origin.

pfuDrawMessageCI works just like **pfuDrawMessage** except that *cimode* is set to **PFU_CI**. The color indices are specified in *textClr* and *shadowClr*.

pfuDrawMessageRGB also works just like **pfuDrawMessage** except that *cimode* is set to **PFU_RGB**. The RGB color values are specified in *textClr* and *shadowClr*.

pfuDrawTree draws each node of the subtree of the scene graph rooted at *tree* inside the view frustum defined by *chan*.

The following functions are not normally called by user applications but are used by the GUI module to manipulate its internal state and are provided here for completeness.

pfuInitGUICursors initializes the cursors used by the GUI module.

pfuGetGUICursor and **pfuGUICursor** are used to get and set cursor definitions, respectively. **pfuGetGUICursorSel** and **pfuGUICursorSel** are used to get and set the current cursor, respectively. **pfuUpdateGUICursor** is called to load the current cursor bitmap into the display window.

pfuGUIHlight and **pfuGetGUIHlight** are used to get and set the *pfHighlight* definition used to highlight selected geometry.

NOTES

pfuInitUtil should be called immediately after **pfConfig** when using these routines to initialize shared memory used by the utility library. Additionally, the GUI relies on receiving mouse information via **pfuGetMouse**. This input can be collected automatically via input handling utilities in libpfutil. See the **pfuInitInput** man pages.

These functions use **pfDataPools** to store multiply accessed data so as to work well in multiprocessing applications. See "/usr/share/Performer/src/libpfutil/gui.c" for further details.

Sample source code programs using the GUI utilities include "/usr/share/Performer/src/pguide/libpf/C/detail.c" and the perfly sample application.

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfDataPool, pfuXFont, pfuCursor, pfuInitInput, pfuInitUtil, pfuGetMouse

NAME

pfuNewHTable, pfuDelHTable, pfuResetHTable, pfuEnterHash, pfuRemoveHash, pfuFindHash, pfuHashGSetVerts, pfuCalcHashSize – Hash table utility library.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

pfuHashTable * **pfuNewHTable**(int numb, int eltsize, void* arena);

void **pfuDelHTable**(pfuHashTable* ht);

void **pfuResetHTable**(pfuHashTable* ht);

pfuHashElt * **pfuEnterHash**(pfuHashTable* ht, pfuHashElt* elt);

int **pfuRemoveHash**(pfuHashTable* ht, pfuHashElt* elt);

int **pfuFindHash**(pfuHashTable* ht, pfuHashElt* elt);

int **pfuHashGSetVerts**(pfGeoSet *gset);

int **pfuCalcHashSize**(int size);

```
typedef struct _pfuHashElt
{
    int        id;
    int        listIndex;
    uint       key;
    void       *data;
} pfuHashElt;

typedef struct _pfuHashBucket
{
    int        nelts;
    pfuHashElt *elts;
    struct _pfuHashBucket     *next;

} pfuHashBucket;

typedef struct _pfuHashTable
{
    void       *arena;
    int        eltSize;
    int        realeltSize;
    int        numBuckets;
    pfuHashBucket     **buckets;

    /* Flat list of hash elements provides linear ordering */
```

```
    int        listCount;
    int        listAvail;
    pfuHashElt **list;
} pfuHashTable;
```

DESCRIPTION

pfuNewHTable returns a new hash table allocated from the shared memory arena *arena* with *numb* elements each of size *eltsize*. *eltsize* is in bytes and must be a multiple of four.

pfuDelHTable deletes the hash table *ht*.

pfuResetHTable resets the hash table *ht*.

pfuEnterHash puts element *elt* into hash table *ht*. If the element is already in the table, it returns the address of that element, otherwise it returns NULL and adds the element to the *list* member of the **pfuHashTable** structure.

pfuRemoveHash removes element *elt* from the hash table *ht*, returning **TRUE** if *elt* was found and **FALSE** otherwise.

pfuFindHash looks for element *elt* in hash table *ht*, returning **TRUE** if *elt* was found and **FALSE** otherwise.

pfuHashGSetVerts takes a **pfGeoSet** of type **PFGS_TRIS** and attempts to share all **PFGS_PER_VERTEX** attributes. **pfuHashGSetVerts** will convert a non-indexed **pfGeoSet** into an indexed one and may delete the old attribute and index arrays and create new ones. Consequently you may wish to **pfRef** your arrays to avoid their deletion.

An example of **pfuHashGSetVerts** usage is found in **pfdMeshGSet**.

pfuCalcHashSize returns the smallest prime number larger than *size*. This is useful since hash tables are more memory efficient when their table size is prime.

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfGeoSet, pfRef, pfuMeshGSet, pfuEventQueue, pfuGUI

NAME

pfuInitInput, pfuExitInput, pfuGetMouse, pfuGetEvents, pfuInputHandler, pfuCollectInput, pfuCollectGLEventStream, pfuCollectXEventStream, pfuMapMouseToChan, pfuMouseInChan – Initialize, process and reset input devices.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

void **pfuInitInput**(pfPipeWindow *pipeWin, int mode);

void **pfuExitInput**(void);

void **pfuGetMouse**(pfuMouse *mouse);

void **pfuGetEvents**(pfuEventStream *events);

void **pfuInputHandler**(pfuEventHandlerFuncType userFunc, uint mask);

void **pfuCollectInput**(void);

void **pfuCollectGLEventStream**(pfuEventStream *events, pfuMouse *mouse, int handlerMask, pfuEventHandlerFuncType handlerFunc);

void **pfuCollectXEventStream**(pfWSConnection dsp, pfuEventStream *events, pfuMouse *mouse, int handlerMask, pfuEventHandlerFuncType handlerFunc);

int **pfuMapMouseToChan**(pfuMouse *mouse, pfChannel *chan);

int **pfuMouseInChan**(pfuMouse *mouse, pfChannel *chan);

```
typedef struct _pfuMouse
{
    int        flags;         /* for PDEV_MOUSE_*_DOWN and PFUDEV_MOD_* bitmasks */
    int        modifiers;     /* modifier keys only  */

    int        xpos, ypos;    /* Screen coordinates of mouse */
    float      xchan, ychan;  /* Normalized coordinates of mouse */
    double     posTime;       /* msec timestamp on current mouse position */

    /* These are used by the GUI and pfiXformer
     * GUI needs Last click positional info
     * Xformers need last and middle click and release info
     */
                   /* Mask of clicks seen last frame */
    int        click;
                   /* Last click position for each mouse button */
    int        clickPos[PFUDEV_MOUSE_DOWN_MASK][2];
                   /* Screen coordinates where a mouse button was last clicked */
    int        clickPosLast[2];
```

```
                /* mask of mouse releases seen last frame */
    int         release;
                /* last release position for each mouse button */
    int         releasePos[PFUDEV_MOUSE_DOWN_MASK][2];
                /* Screen coordinates where a mouse button was last released*/
    int         releasePosLast[2];
                /* Last click time for each mouse button */
    double      clickTime[PFUDEV_MOUSE_DOWN_MASK];
                /* Time of last button click */
    double      clickTimeLast;
                /* Last release time for each mouse button */
    double      releaseTime[PFUDEV_MOUSE_DOWN_MASK];
                /* Time of last button reelase */
    double      releaseTimeLast;


    int         winSizeX;       /* Window Size */
    int         winSizeY;

    int inWin;          /* Window focus flag */

} pfuMouse;


typedef void (*pfuEventHandlerFuncType)(int dev, void* val,
    pfuCustomEvent *pfuevent);
```

DESCRIPTION

There are a variety of automatic and explicit event collection utilities in libpfutil. Automatic X or GL input event collection is started with **pfuInitInput** and the resulting events can be queried with **pfuInputHandler** and **pfuGetEvents**.

pfuInitInput initializes mouse and keyboard input to be read from the specified **pfPipeWindow** *pipeWin*. *mode* is one of:

PFUINPUT_X
Read mouse and keyboard from a forked process using X device commands. *pipe* must have a GLX window. See the **pfuGLXWinopen** reference page for more information.

PFUINPUT_GL
Read mouse and keyboard from the draw process using GL device commands. *pipe* must have a GL window.

pfuCollectInput should be called from the draw process if the mode is **PFUINPUT_GL** and will poll the mouse and collect all queued devices. The first time it is called, **pfuCollectInput** will queue the following GL devices:

```
WINQUIT
REDRAW
KEYBD
LEFTMOUSE
MIDDLEMOUSE
RIGHTMOUSE
INPUTCHANGE
```

Any other required GL devices, such as function keys, should be queued explicitly by the application.

If the mode is **PFUINPUT_X** then **pfuCollectInput** does not need to be called since the device input is automatically collected by the forked process.

pfuInputHandler installs the custom handler *userFunc*, which will then be called to process each input event included in *mask*. If the mode is **PFUINPUT_X**, then *mask* can be set to the bitwise-or of an X input mask with **PFUINPUT_CATCH_UNKNOWN**, **PFUINPUT_CATCH_SIM**, or **PFU_CATCH_ALL**. If mode is **PFUINPUT_GL**, then *mask* may be set to either **PFU_CATCH_ALL** or **PFU_CATCH_UNKNOWN**.

pfuGetMouse copies the current mouse values from the libpfutil event collector (initially triggered with **pfuInitInput**) into *mouse* and **pfuGetEvents** copies the events of the current frame into *events*. **pfuGetEvents** also resets the internal event queue.

pfuMapMouseToChan maps the mouse screen coordinates (mouse->xpos, mouse->ypos) into coordinates in the range [-1, 1] (mouse->xchan, mouse->ychan) based on *chan*'s viewport. Either **TRUE** or **FALSE** is returned to indicate that the mouse is in or out of the *chan*'s viewport.

pfuMouseInChan does the **pfuMapMouseToChan** mapping. In addition, its return value considers mouse focus if a mouse button is recorded as being down in *mouse*. In this case, the recorded position of where the mouse button was clicked will determine if the current channel has focus. The mouse will be considered to be "in" the channel of focus.

pfuCollectGLEventStream will do immediate IRIS GL input collection into the provided pfuEventStream and pfuMouse structures. If the provided event stream or mouse pointer is NULL, it will be ignored. This routine must be called in the draw process.

pfuCollectXEventStream will do immediate X input collection from the provided pfWSConnection into the provided pfuEventStream and pfuMouse structures. If the provided event stream or mouse pointer is

NULL, it will be ignored.

pfuExitInput must be called to terminate the forked X input process.

NOTES

pfuInitUtil should be called immediately after **pfConfig** when using these routines to initialize shared memory used by the utility library. IRIS Performer recommends that you use X device input. The IRIS Performer sample application, **perfly**, is shipped with **PFUINPUT_X** as the default. X device input is recommended for the following reasons:

OpenGL does not contain device input routines and all input must be managed through X.

Collecting GL device input in the draw process can reduce rendering throughput.

Collecting X device input in an asynchronous process can improve real-time characteristics.

See the **pfuEventQueue** man page for a description of the **pfuEventStream** structure.

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfuGLXWinopen, pfuInitUtil, pfuEventQueue, pfuGUI

NAME

pfuInitUtil, pfuGetUtilDPool, pfuExitUtil, pfuDPoolSize, pfuGetDPoolSize, pfuFindUtilDPData – Initialize and reset IRIS Performer utility library.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

void	**pfuInitUtil**(void);
pfDataPool *	**pfuGetUtilDPool**(void);
void	**pfuExitUtil**(void);
void	**pfuDPoolSize**(long size);
long	**pfuGetDPoolSize**();
volatile void*	**pfuFindUtilDPData**(int id);

DESCRIPTION

pfuInitUtil must be called before making any calls to the utility library. **pfuInitUtil** creates a **pfDataPool** which **libpfutil** uses for multiprocess operation. The **pfDataPool** is created in "/usr/tmp" or the directory specified by the environment variable, **PFTMPDIR**, if it is set.

In order to change the amount of memory that **pfuInitUtil** will allocate for the **libpfutil** data pool, call **pfuDPoolSize**. **pfuGetDPoolSize** returns the size of the data pool. Note that the default data pool size is optimal for **libpfutil's** memory allocation. You should only use **pfuDPoolSize** to increase the size of the data pool. Changes to the data pool size only take effect when your application calls **pfuInitUtil**.

pfuGetUtilDPool returns a pointer to the utility library **pfDataPool**.

pfuFindUtilDPData returns a pointer to the block of memory identified by *id* in the utility library **pfDataPool** or NULL if *id* is not found.

pfuExitUtil removes the utility library **pfDataPool** from the file system.

NOTES

If a calling program exits abnormally, the **pfDataPool** will not be deleted.

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfDataPool

NAME

pfuMakeLPStateShapeTex, pfuMakeLPStateRangeTex – Sample functions to derive a texture image from light point specifications.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

void **pfuMakeLPStateShapeTex**(pfLPointState *lps, pfTexture *tex, int size);

void **pfuMakeLPStateRangeTex**(pfLPointState *lps, pfTexture *tex, int size, pfFog *fog);

DESCRIPTION

pfMakeLPStateRangeTex and **pfMakeLPStateShapeTex** are provided to compute a texture image which accurately mimics certain characteristics of **pfLightPoints**. These functions are provided as sample code in **libpfutil**.

EXAMPLES

The following example illustrates how to build a comprehensive light point structure that uses texture mapping to accelerate directionality computations. The texture maps are generated using the **pfuMakeLPStateShapeTex** described here.

```
/*
 * Create pfLPointState and pfGeoState.
 */
pfGeoState     *gst = pfNewGState(arena);
pfLPointState  *lps = pfNewLPState(arena);
pfGStateMode(gst, PFSTATE_ENLPOINTSTATE, 1);
pfGStateAttr(gst, PFSTATE_LPOINTSTATE, lps);

/*
 * Light point projected diameter is computed on CPU. Real world
 * size is 0.07 database units and projected size is clamped be
 * between 0.25 and 4 pixels.
 */
pfLPStateMode(lps, PFLPS_SIZE_MODE, PFLPS_SIZE_MODE_ON);
pfLPStateVal(lps, PFLPS_SIZE_MIN_PIXEL, 0.25f);
pfLPStateVal(lps, PFLPS_SIZE_ACTUAL, 0.07f);
pfLPStateVal(lps, PFLPS_SIZE_MAX_PIXEL, 4.0f);

/*
 * Light points become transparent when their projected diameter is
 * < 2 pixels. The transparency falloff rate is linear with
 * projected size with a scale factor of 0.6. The transparency
```

```
 * multiplier, NOT the light point transparency, is clamped to 0.1.
 */
pfLPStateVal(lps, PFLPS_TRANSP_PIXEL_SIZE, 2.0f);
pfLPStateVal(lps, PFLPS_TRANSP_EXPONENT, 1.0f);
pfLPStateVal(lps, PFLPS_TRANSP_SCALE, 0.6f);
pfLPStateVal(lps, PFLPS_TRANSP_CLAMP, 0.1f);

/*
 * Light points will be fogged as if they were 4 times
 * nearer to the eye than actual to achieve punch-through.
 */
pfLPStateVal(lps, PFLPS_FOG_SCALE, 0.25f);

/* Range to light points computed on CPU is true range */
pfLPStateMode(lps, PFLPS_RANGE_MODE, PFLPS_RANGE_MODE_TRUE);

/*
 * Light points are bidirectional but have different (magenta)
 * back color. Front color is provided by pfGeoSet colors.
 */
pfLPStateMode(lps, PFLPS_SHAPE_MODE, PFLPS_SHAPE_MODE_BI_COLOR);
pfLPStateBackColor(lps, 1.0f, 0.0f, 1.0f, 1.0f);

/*
 * 60 degrees horizontal and 30 degrees vertical envelope.
 * Envelope is rotated -25 degrees about the light point
 * direction. Falloff rate is linear and ambient intensity is 0.1.
 */
pfLPStateShape(lps, 60.0f, 30.0f, -25.0f, 1.0f, 0.1f);

/*
 * Specify that light points should use texturing hardware to simulate
 * directionality and use CPU to compute light point transparency and
 * fog punch-through. Note that if light points are omnidirectional,
 * you should use PFLPS_TRANSP_MODE_TEX and PFLPS_FOG_MODE_TEX instead.
 */
pfLPStateMode(lps, PFLPS_DIR_MODE, PFLPS_DIR_MODE_TEX);
pfLPStateMode(lps, PFLPS_TRANSP_MODE, PFLPS_TRANSP_MODE_ALPHA);
pfLPStateMode(lps, PFLPS_FOG_MODE, PFLPS_FOG_MODE_ALPHA);

/*
 * Make directionality environment map of size 64 x 64 and attach
 * it to the light point pfGeoState. We assume that a pfTexEnv of
```

```
 * type PFTE_MODULATE has been globally applied with pfApplyTEnv.
 */
tex = pfNewTex(arena);
pfuMakeLPStateShapeTex(lps, tex, 64);
pfGStateAttr(gst, PFSTATE_TEXTURE, tex);
pfGStateMode(gst, PFSTATE_ENTEXTURE, 1);

/*
 * Make SPHERE_MAP pfTexGen and attach to light point pfGeoState.
 * pfGeoSet normals define the per-light light point direction.
 */
tgen = pfNewTGen(arena);
pfTGenMode(tgen, PF_S, PFTG_SPHERE_MAP);
pfTGenMode(tgen, PF_T, PFTG_SPHERE_MAP);
pfGStateAttr(gst, PFSTATE_TEXGEN, tgen);
pfGStateMode(gst, PFSTATE_ENTEXGEN, 1);

/*
 * Configure light point transparency. Use PFTR_BLEND_ALPHA for high
 * quality transparency. Set pfAlphaFunc so that light points are not
 * drawn unless their alphas exceed 1 when using 8-bit color resolution.
 */
pfGStateMode(gst, PFSTATE_TRANSPARENCY, PFTR_BLEND_ALPHA);
pfGStateVal(gst, PFSTATE_ALPHAREF, 1.0/255.0);
pfGStateMode(gst, PFSTATE_ALPHAFUNC, PFAF_GREATER);

/*
 * Disable pfFog effects since light points are fogged by
 * the pfLPointState.
 */
pfGStateMode(gst, PFSTATE_ENFOG, 0);
/*
 * Disable lighting effects since light points are completely
 * emissive.
 */
pfGStateMode(gst, PFSTATE_ENLIGHTING, 0);

/*
 * Attach the pfGeoState to a pfGeoSet of type PFGS_POINTS and
 * you've got light points!
 */
pfGSetPrimType(gset, PFGS_POINTS);
pfGSetGState(gset, gst);
```

For further details, see the **libpr** routines **pfMakeLPStateShapeTex** and **pfMakeLPStateRangeTex**.

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfLPointState, pfMakeLPStateShapeTex, pfMakeLPStateRangeTex

NAME

pfuFreeCPUs, pfuRunProcOn, pfuLockDownProc, pfuLockDownApp, pfuLockDownCull, pfuLockDownDraw, pfuPrioritizeProcs – Priority, processes and processor assignment functions.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

int **pfuFreeCPUs**(void);

int **pfuRunProcOn**(int cpu);

int **pfuLockDownProc**(int cpu);

int **pfuLockDownApp**(void);

int **pfuLockDownCull**(pfPipe *);

int **pfuLockDownDraw**(pfPipe *);

int **pfuPrioritizeProcs**(int pri);

DESCRIPTION

These routines assign processes to CPUs and implement a policy specifically designed for locking down the IRIS Performer application, cull, and draw processes. The routines implementing these features utilize the IRIX REACT facilities. Refer to the IRIX REACT technical report, and the **sysmp(2)** reference page for detailed information on these concepts.

The routine **pfuRunProcOn** can be used to force a process to run on a specified CPU and does not require super-user permission. This is often used to force extra processes that can run asynchronously from the draw, such as those receiving and generating input, onto CPU 0 without isolating that CPU from standard UNIX scheduling.

All of the **pfuLock<*>** routines force a process to run on the specified CPU. They also attempt to isolate the processor to run only those processes that have specified that they must run on that CPU. Isolating a CPU also protects it from seeing unnecessary cache and TLB flushes generated by processes that have not specified that they must run on this CPU.

The **pfuLockDown<App,Cull,Draw>** routines implement a policy for selecting CPUs for different processors given the program and machine configuration.

These routines are used in the IRIS Performer **nextfly** and **perfly** sample applications.

The locking and assignment policy implemented by these routines is implemented in the various stages of an IRIS Performer application as follows.

1. CPU 0 is never isolated.

2. In the **APPCULLDRAW** mode, the processor assignment is handled by the **APP** process which takes

CPU 1.

3. In the **APP_CULLDRAW** mode, the processor assignment is handled separately by the **APP** and **DRAW** processes.

4. In the **APP_CULL_DRAW** mode, each process handles itself.

When there is only one pipe, processors are mapped to processes as follows. If there are three CPUs, each of **APP, CULL** and **DRAW** gets its own process. If there are only two CPUs, **APP** is put on CPU 0, which is not isolated, and **DRAW** and **CULL** share CPU 1.

Multipipe mappings are as follows.

```
If NumCPUs >= 2 + 2*NumPipes
then each cull and draw process can have its own CPU, with the application
getting CPU 1, and UNIX getting CPU 0.

If NumCPUs == 1 + 2*NumPipes
then the application shares CPU 0 with UNIX.

Otherwise, if NumCPUs >= 2 + NumPipes
then cull and draw processes for each pipe are paired together.
```

If there are fewer CPUs then indicated above, then the application is assigned to CPU 0 with UNIX, APP and CULL process are paired, and when only one free CPU remains, all remaining processes are assigned to the last CPU.

Each of these routines return 1 if successful and 0 if an error is encountered.

pfuFreeCPUs frees any CPUs which may have been previously restricted.

pfuRunProcOn forces the calling process to run on the specified CPU and does not require super-user permission.

pfuLockDownProc locks the calling process onto CPU *cpu*. The CPU is isolated to running only processes that have specified that they must run on this CPU. This CPU isolation requires super-user permission.

pfuLockDownApp locks the APP process to a CPU determined by the policy above. The CPU is isolated to running only processes that have specified that they must run on this CPU. This CPU isolation requires super-user permission.

pfuLockDownCull locks the CULL process of a pfPipe (there is one CULL process per pipe) to a CPU determined by the policy above. This routine should be called the first time through the channel cull callback for a given process. The CPU is isolated to running only processes that have specified that they must run on this CPU. This CPU isolation requires super-user permission.

pfuLockDownDraw locks the DRAW process of a pfPipe (there is one draw process per pipe) to a CPU determined by the policy above. This routine should be called from application's **pfInitPipe** callback. See the **pfInitPipe** reference page for more information. The CPU is isolated to running only processes that have specified that they must run on this CPU. This CPU isolation requires super-user permission.

pfuPrioritizeProcs should be called after **pfConfig** and will set or remove non-degrading priorities from all Performer processes. *pri* is a boolean: if TRUE all IRIS Performer processes will be assigned a non-degrading priority of NDPHIMAX+2 (see **schedctl**); if FALSE any non-degrading priorities will be removed. You must have super-user permission to enable non-degrading priorities but not to remove them.

If you wish to assign different priorities to different processes, simply modify **pfuPrioritizeProcs** to suit your needs.

NOTES

Isolating a CPU to specific processes requires super-user permission.

CPU 0 should *never* be isolated.

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfConfig, pfInitPipe, sysmp, schedctl

NAME

pfuConfigMCO, pfuGetMCOChannels, pfuTileChans, pfuTileChan – Multi-Channel Option configuration utilities.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

void **pfuConfigMCO**(pfChannel **chn, int nChans);

int **pfuGetMCOChannels**(pfPipe *p);

void **pfuTileChans**(pfChannel **chn, int nChans, int ntilesx, int ntilesy);

void **pfuTileChan**(pfChannel **chn, int thisChan, int nChans, float l, float r, float b, float t);

DESCRIPTION

These functions serve as a generic way of initializing channels when using the Multi-Channel Option (MCO) available for RealityEngine graphics systems. The MCO divides the frame buffer into several tiles and then outputs these different tiles to different displays through different video outputs. The MCO can be configured to drive up to six display channels at resolutions varying from 1280x1024 60Hz non-interlaced to 640x480 30Hz interlaced. These functions use a basic knowledge of how the MCO tiles the frame buffer to configure **pfChannels** appropriately for most configurations.

pfuConfigMCO takes the array of **pfChannel** pointers *chn* of size *nChans* and formats the viewports such that each channel's viewport divides the framebuffer in the same way the MCO does. Each channel is thus mapped to a different display.

pfuGetMCOChannels returns the number of channel subdivisions the MCO is currently configured to use. This is the number of displays that the MCO has tiled the frame buffer into based on the current video output format specification (see **setmon**).

pfuTileChans is a tiling routine that takes *nChans* channels and divides them into a grid of *ntilesx* by *ntilesy* viewports that all fit into the range 0 to 1 in both the x and y directions.

pfuTileChan is a utility function that sets the *chn[thisChan]* viewport to the specified left (*l*), right(*r*), bottom(*b*), and top(*t*) values where each value is between 0 and 1. *nChans* is usually the number of channel subdivisions the MCO is configured to use. *thisChan* should be at most *nChans*.

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfChannel, pfPipe, setmon

NAME

pfuManageMPipeStats – Multipipe/multichannel stats utility.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

int **pfuManageMPipeStats**(int nframes, nSampledPipes);

DESCRIPTION

This utility obtains time stamp stats for each Performer channel/pipe in an automatic fashion. The complete stats log is stored in a file for later review. Timing information for pre- and post-callbacks is also provided.

The routine **pfuManageMPipeStats** can be used to obtain timing information during a number of frames per each channel for pre-draw, **pfDraw**, post-draw, pre-cull, **pfCull**, post-cull as well as **ISECT** and **APP**.

nframes specifies the number of frames to be measured. *nSampledPipes* is the number of pipes for which statistics are to be gathered.

The first time that **pfuManageMPipeStats** is called it automatically determines the pipe and channel configuration. It also sets up a minimally configured **pfStats** in order to obtain the data.

During the requested number of frames, the utility grabs the latest **pfStats** buffer into main memory. After the requested number of frames, the utility creates a new file called mpstats**0**.data. This the file number is incremented each time that the utility is used from the same execution.

pfuManageMPipeStats returns 1 while it is getting time stamps and 0 when it has finished his collection and created the historical file.

pfuManageMPipeStats must be called in the APP and it could be used multiple times.

Following is an example of use in an application loop:

```
/* DumpMPipeStats is controlled by the user interface */

if(SharedArena->DumpMPipeStats && !mpstats_running)
    mpstats_running = pfuManageMPipeStats(10, 1));

if(mpstats_running)
    mpstats_running = pfuManageMPipeStats(10, 1));
```

The file format used by **pfuManageMPipeStats** is offered only as an example. The meaning and lay out

is as follows:

```
================================================================
Frame:<frame>
_________________________APP_____________________________
Absolute Timeline information (sec)
AppFrame:<frame>
AppStart:<start>
enterSync:<enterSync> afterClean:<afterClean> afterSync:<afterSync>
pfFrameStart:<frameStart> pfFrameEnd:<frameEnd>
_________________________ISECT_____________________________
Absolute Timeline information (sec)
IsectFrame: <frame>
Start:<start> End:<end>

Relative Timeline information (msec)
Total Isect Time:<end-start>
_________________________CULL & DRAW_____________________
Chan: <Channel_number>
Frame: <frame_number>

Absolute Timeline information (sec)
CullFrame: <cullFrame>
    Start: <cullStart> End: <cullEnd>
    BeginUpDate: <begUpdate> EndUpdate: <endUpdate>
DrawFrame: <drawFrame>
    Start: <drawStart> End: <drawEnd>
    pfDrawStart: <pfDrawStart> pfDrawEnd: <pfDrawEnd> AfterSwap: <AfterSwap>

Relative Timeline information (msec)
Total Cull Time:<cullEnd-cullStart>
Total Draw Time:<drawEnd-drawStart>
    PreDraw: <preDraw>
    pfDraw:  <pfDraw>
    PostDraw:<postDraw>
----------------------------------------------------------------
```

You are encouraged to change the default file format as well as the data being collected.

NOTES

Depending on the multiprocess mode being used a number of initial frames could be filled by nonsensical data.

The use of **pfuManageMPipeStats** modifies the **pfStats** mode and after his use it leaves the **pfStats** modes in the default.

If you wish to reuse the previous **pfStats** mode you should save it before the first call to **pfuManageMPipeStats**.

CPU 0 should *never* be isolated.

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfFrameStats

NAME

pfuNewPath, pfuClosePath, pfuCopyPath, pfuSharePath, pfuPrintPath, pfuFollowPath, pfuAddArc, pfuAddDelay, pfuAddFile, pfuAddFillet, pfuAddPath, pfuAddSpeed – Simple path-following utility

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

pfuPath * **pfuNewPath**(void);

pfuPath * **pfuClosePath**(pfuPath *path);

pfuPath * **pfuCopyPath**(pfuPath *copy);

pfuPath * **pfuSharePath**(pfuPath *share);

int **pfuPrintPath**(pfuPath *path);

int **pfuFollowPath**(pfuPath *path, float seconds, pfVec3 where, pfVec3 orient);

int **pfuAddArc**(pfuPath *path, pfVec3 center, float radius, pfVec2 angles);

int **pfuAddDelay**(pfuPath *path, float delay);

int **pfuAddFile**(pfuPath *path, char *name);

int **pfuAddFillet**(pfuPath *path, float radius);

int **pfuAddPath**(pfuPath *path, pfVec3 start, pfVec3 final);

int **pfuAddSpeed**(pfuPath *path, float desired, float rate);

DESCRIPTION

The **pfuPath** functions provide a simple way to move one or more simulated vehicles or eyepoints along a mathematically defined path. The path is a general series of arcs and line segments. Once a path is created, it can be followed each simulation frame to provide position and orientation information.

pfuNewPath allocates and initializes a new **pfuPath** structure. Once the path is opened, segments (both straight and curved) can be added using the **pfuAddPath** and **pfuAddArc** commands as described below. The initial path speed is set at 1.0 database units per second.

pfuClosePath connects the end point of the final segment of path *path* with the start point of the path's first segment. This creates a closed (also known as looping) path that can be followed endlessly. Paths can be either open or closed, but they should only be closed once. The closed path is returned.

pfuCopyPath creates and returns a new **pfuPath** structure that has path segments which are an exact copy of those in the path indicated by the *path* argument. This is a deep copy, and will create a new instance of each element in the original path definition.

pfuSharePath creates a new **pfuPath** structure, but causes the linked list of path segments in the path structure *path* to be shared by both old and new path structures. This allows multiple simulated objects to follow the same path without the redundant allocation of path storage, and also allows changes to the

path segment definition to effect all **pfuPath** structures that share it.

pfuPrintPath prints the path following control information for path object *path*, and then prints the definition of each segment in the path itself. All printing is done using the IRIS Performer **pfNotify** mechanism with a severity level of **PFNFY_DEBUG**.

pfuFollowPath performs the actual simulation of moving an object along the path indicated by *path*. The *seconds* argument specifies the simulated duration, and internal data in the **pfuPath** structure supplies the speed. From this time and speed information, the vehicle's simulated distance of travel is computed. Then, the vehicle is moved this distance along the path segments defined within the **pfuPath** object. The resulting simulated position is returned in the **pfVec3** argument *where* and the orientation of the vehicle is returned in *orient*.

pfuAddArc adds a circular arc path segment to the **pfuPath** structure *path*. The arc is defined by the *center* and *radius* values, and a pair of *angles*. The first angle, *angle[0]*, is the start angle for the path, and is a point on the circle defined by *center* and *radius* with the indicated counterclockwise angle from the positive X axis. The second angle, *angle[1]*, represents the turn angle, the angle between the arc's start and end points as measured from the designated center point. Positive turn angles indicate counterclockwise turns, and negative angles represent clockwise turns. An arc from the +X axis to the +Y axis would be defined with a start angle of 0 degrees and a turn angle of 90 degrees. The same arc in the opposite direction of travel is defined by a start angle of 90 degrees and a turn angle of -90 degrees.

pfuAddDelay adds a zero-length segment to *path* that causes the simulated vehicle to stop for the *delay* seconds at that point in the path. Once this much simulated time elapses, the simulation will continue with the next segment in the path. Delay segments can be used to simulate motor vehicles paused waiting for traffic signals and similar latent delay sources along a path.

pfuAddFile adds a series of path segments defined by a simple ASCII file format to the **pfuPath** structure *path*. This function provides an easy way to load user specified paths for vehicles into simulation applications, and is used in the popular Silicon Graphics "Performer Town" demonstration program to load the paths for the truck into the program.

pfuAddFillet is used to create circular arcs that join two adjacent straight path segments. Fillet creation is a three step process.

1. A straight segment is added to a path using **pfuAddPath**. This is the segment that will lead into the fillet.
2. A fillet request is added using **pfuAddFillet**. The fillet's radius is defined in this call.
3. A second straight segment is added, again with **pfuAddPath**. This segment must start where the segment of step one ended.

When the second segment is added in step three, a test is performed to see if the first point in the second

segment has the same X, Y, and Z values as the last point in the previous segment. If so, and if there is a fillet request between the two segments, then a matching fillet of the specified radius is computed. In addition, the endpoints (and thus lengths) of the two straight segments are adjusted so as to match with the fillet endpoints.

For example, the request

line from (0,0) to (1,0)
fillet of radius 0.25
line from (1,0) to (1,1)

will cause the fillet arc and line segment lengths to be automatically computed as

line from (0,0) to (0.75,0)
fillet of radius 0.25 with center (0.75,0.25) and angles 270 and 90.
line from (1,0.25) to (1,1)

This automatic fillet construction is seen to be very convenient once the alternative manual fillet construction is attempted. When the fillet radius is too large to allow the arc to be created (as would be the case in the example above with a radius greater than one), a fillet is not constructed and a sharp turn will exist in the path.

pfuAddPath adds a straight line segment to *path*. The line segment is defined as extending from *start* to *final*. As mentioned above, if the segment preceding the line segment is an unevaluated fillet, and the value of *start* is equal to that of the *final* point of the segment before the fillet, then an automatic fillet will be constructed if the fillet's radius specification is sufficiently small to allow it.

pfuAddSpeed adds a speed changing segment to *path*. The new speed is indicated by *desired* and the rate of adjustment from the current path speed to the new speed is given by *rate*.

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfNotify

NAME

pfuRandomize, pfuRandomLong, pfuRandomFloat, pfuRandomColor – Set up and generate random numbers and colors.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

void **pfuRandomize**(int seed);

long **pfuRandomLong**(void);

float **pfuRandomFloat**(void);

void **pfuRandomColor**(pfVec4 color, float minColor, float maxColor);

DESCRIPTION

These functions set up a random number generator and use it to return uniformly distributed random numbers and colors.

pfuRandomize initializes the random number generator with *seed*.

pfuRandomLong returns a random number of type **long** while **pfuRandomFloat** returns a random number of type **float**.

pfuRandomColor returns a random color in *color*. The *r*, *g* and *b* values of *color* are random numbers in the range [minColor, maxColor]. The alpha value of *color* is set to the fully opaque value 1.

See the **srandom** and **random** man pages for further information on the random number generator used.

SEE ALSO

random, srandom

NAME

pfuInitRendezvous, pfuMasterRendezvous, pfuSlaveRendezvous – Multiprocessing master and slave rendezvous routines

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

void **pfuInitRendezvous**(pfuRendezvous *rvous, int numSlaves);

void **pfuMasterRendezvous**(pfuRendezvous *rvous);

void **pfuSlaveRendezvous**(pfuRendezvous *rvous, int id);

```
#define PFURV_MAXSLAVES         2

#define PFURV_GARBAGE           -1
#define PFURV_READY             10
#define PFURV_SYNC              11
#define PFURV_SYNCACK           12
#define PFURV_RESUME            13

typedef struct _pfuRendezvous
{
    int master;
    int numSlaves;
    int slaves[PFURV_MAXSLAVES];
} pfuRendezvous;
```

DESCRIPTION

These rendezvous functions are useful for synchronizing master and slave processes in a multiprocessing environment.

In the case of multiple processes, a rendezvous is a method for synchronizing each independent process. A process chosen as the "master" waits for the remaining processes, designated "slaves", to indicate through the rendezvous that they are ready to synchronize. As each slave indicates its readiness, it then waits on the master process. The master process releases the slaves after all slaves have made the rendezvous. The rendezvous token should be allocated in a shared arena; all processes require access to it.

pfuInitRendezvous initializes the rendezvous token *rvous* for one master process and *numSlaves* slave processes.

In order to synchronize multiple processes, the master should call **pfuMasterRendezvous** and each of the slaves should call **pfuSlaveRendezvous** with its slave ID in *id*. Slave IDs can range from 0 to (-**PFURV_MAXSLAVES** - 1).

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfGetSharedArena, pfMalloc

NAME

pfuCalcNormalizedChanXY, pfuSaveImage – Capture screen images.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

void **pfuCalcNormalizedChanXY**(float* px, float* py, pfChannel* chan, int xpos, int ypos);

int **pfuSaveImage**(char* name, int xorg, int yorg, int xsize, int ysize, int alpha);

DESCRIPTION

pfuCalcNormalizedChanXY normalizes the window coordinates (*xpos*, *ypos*) to *chan*'s viewport and returns the resulting normalized position in (*px*, *py*). The mapping is defined so that (*px*, *py*) range from (0, 0) to (1, 1) when (*xpos*, *ypos*) is within *chan*'s viewport.

pfuSaveImage saves the image rectangle that spans the window coordinates (*xorg*, *yorg*) to (*xorg* + *xsize*, *yorg* + *ysize*) into an image file with the name *name*. The *alpha* boolean argument specifies if only an RGB image (*alpha* = 0) or a complete RGBA image (*alpha* = 1) is desired.

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfChannel, lrectread

NAME

pfuInitSmokes, pfuNewSmoke, pfuSmokeType, pfuSmokeOrigin, pfuSmokeVelocity, pfuGetSmokeVelocity, pfuSmokeDir, pfuSmokeMode, pfuDrawSmokes, pfuSmokeTex, pfuSmokeDuration, pfuSmokeDensity, pfuGetSmokeDensity, pfuSmokeColor – Routines for simulating smoke and fire.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

void **pfuInitSmokes**(void);

pfuSmoke * **pfuNewSmoke**(void);

void **pfuSmokeType**(pfuSmoke *smoke, int type);

void **pfuSmokeOrigin**(pfuSmoke *smoke, pfVec3 origin, float radius);

void **pfuSmokeVelocity**(pfuSmoke *smoke, float turbulence, float speed);

void **pfuGetSmokeVelocity**(pfuSmoke *smoke, float *turbulence, float *speed);

void **pfuSmokeDir**(pfuSmoke *smoke, pfVec3 dir);

void **pfuSmokeMode**(pfuSmoke *smoke, int mode);

void **pfuDrawSmokes**(pfVec3 eye);

void **pfuSmokeTex**(pfuSmoke *smoke, pfTexture *tex);

void **pfuSmokeDuration**(pfuSmoke *smoke, float dur);

void **pfuSmokeDensity**(pfuSmoke *smoke, float dens, float diss, float expansion);

void **pfuGetSmokeDensity**(pfuSmoke *smoke, float *dens, float *diss, float *expansion);

void **pfuSmokeColor**(pfuSmoke *smoke, pfVec3 bgn, pfVec3 end);

DESCRIPTION

This higher level utility is designed to show how to easily generate smoke effects in IRIS Performer applications.

pfuInitSmokes allocates for, loads and sets the geometry for the smoke and fire textures. Call **pfuInitSmokes** after **pfInit** but before **pfConfig** so that all shared data structures are correctly shared among processes.

Call **pfuNewSmoke** to get a pointer to a new smoke structure which you then can configure with **pfuSmokeVelocity**, **pfuSmokeDir** and the other smoke functions. Smoke is initialized with several defaults but is not started by default.

Call **pfuDrawSmokes** from the draw process (and **only** from the draw process) so that the active smoke objects are actually drawn.

pfuSmokeType can be set to any of
PFUSMOKE_MISSILE
PFUSMOKE_EXPLOSION
PFUSMOKE_FIRE
PFUSMOKE_SMOKE
PFUSMOKE_DUST

pfuSmokeOrigin sets the origin and radius of where *smoke* is to occur.

pfuSmokeVelocity sets the speed at which *smoke* is to move.

pfuGetSmokeVelocity gets the speed at which *smoke* is moving.

pfuSmokeDir sets the direction that *smoke* will move in (via the vector *dir*).

pfuSmokeMode sets the current mode of operation for *smoke*. Supported values are
PFUSMOKE_STOP
PFUSMOKE_START

pfuSmokeTex sets the texture to be used for the smoke effect when *smoke* is drawn.

pfuSmokeDuration sets the length of time *smoke* should last once it is started.

pfuSmokeDensity sets the density, dissipation rate, and expansion rate of the smoke puff.

pfuSmokeColor sets the beginning and ending colors for the smoke puff.

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

NAME

pfuPreDrawStyle, pfuPostDrawStyle – Functions to produce fancy drawing styles.

FUNCTION SPECIFICATION

#include <Performer/pf.h>

#include <Performer/pfutil.h>

void **pfuPreDrawStyle**(int style, pfVec4 scribeColor);

void **pfuPostDrawStyle**(int style);

PARAMETERS

style identifies a drawing style.

DESCRIPTION

Both **pfuPreDrawStyle** and **pfuPostDrawStyle** use multi-pass rendering to implement special draw styles that are not directly provided by the hardware or graphics library. These effects, such as hidden line rendering and haloed polygons, can however be achieved with high-performance Z-buffered graphics rendering hardware as these functions demonstrate.

pfuPreDrawStyle should be called in the Performer draw callback just before **pfDraw**. **pfuPostDrawStyle** must then be called after **pfDraw**. These two functions are a pair and both must be called as described.

The *style* argument can take the following values.

PFUSTYLE_POINTS
This causes polygon vertices to be drawn as small points. In many cases, internal structural aspects can be observed from the resulting "cloud of points" image, especially when that image is in motion on the screen.

PFUSTYLE_LINES
This causes polygons to be drawn in wireframe mode using the Performer **PFGS_WIREFRAME** draw style. The lines are drawn in the same color as the base geometry.

PFUSTYLE_DASHED
This is a wireframe drawing style that does something slightly tricky: it draws the front-facing polygons of an object in the normal wireframe mode and the back-facing polygons in a dashed line mode, creating a typical draftsman's dashed-occlusion hint style of hidden line drawing.

PFUSTYLE_HALOED
Polygons are drawn in wireframe mode with haloed edges. This is the drawing style of electronic schematics, where lines that cross but are not connected have a slight gap or break in the lower line. This same idea can be extended to 3D geometry, and was proposed as a rendering mode by graphics pioneer Dr. Arthur Appel.

PFUSTYLE_HIDDEN
Hidden polygons are not drawn while visible polygons are drawn in wireframe mode. This is the traditional hidden-line removal display mode.

PFUSTYLE_FILLED
Draws polygons as filled solids. This hidden-surface removal display mode is the standard display style.

PFUSTYLE_SCRIBED
Polygons are drawn filled with hidden surfaces removed. The boundary of each visible polygon is highlighted in a wire frame highlight. This mode can be very useful in understanding the geometric complexity of textured scenes.

When *style* takes the values **PFUSTYLE_POINTS** or **PFUSTYLE_DASHED**, **pfuPostDrawStyle** restores polygon drawing to the normal filled mode. Finally, for every value of *style*, it restores the state before **pfuPreDrawStyle** was called.

The *scribeColor* argument is the desired color for scribed lines. The first three elements of the array represent the red, green, and blue values in the range 0 to 1, and the final element is the alpha value for the scribed lines. This argument is not used in the other style modes.

The following code fragment displays how **pfuPreDrawStyle** and **pfuPostDrawStyle** would typically be used in an application. See the files:

```
/usr/share/Performer/src/sample/C/perfly.c
/usr/share/Performer/src/sample/C++/perfly.C
```

for details.

Example 1:

```
/* convey draw style from localPreDraw to localPostDraw */
static int selectedDrawStyle = 0;

void
localPreDraw(pfChannel *chan, void *data)
{
  :
  :
    /*
     * remember draw style in case it changes between now
     * and the time localPostDraw() gets called.
     */
    selectedDrawStyle = ViewState->drawStyle;
```

```
    /* handle draw style */
    pfuPreDrawStyle(selectedDrawStyle, ViewState->drawColor);
}

void
localPostDraw(pfChannel *chan, void *data)
{
    /* handle draw style */
    pfuPostDrawStyle(selectedDrawStyle);
  :
  :
}
```

SEE ALSO

pfDraw

NAME

pfuNewSharedTex, pfuGetSharedTexList, pfuMakeTexList, pfuMakeSceneTexList, pfuDownloadTexList, pfuGetTexSize, pfuNewTexList, pfuLoadTexListFiles, pfuLoadTexListFmt, pfuNewProjector, pfuProjectorPreDrawCB, pfuProjectorMovie, pfuGetProjectorHandle, pfuProjectorHandle, pfuGetProjectorScreenList, pfuAddProjectorScreen, pfuRemoveProjectorScreen, pfuReplaceProjectorScreen – Create and initialize textures, create and display movies.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

pfTexture *	**pfuNewSharedTex**(const char *filename, void *arena);
pfList *	**pfuGetSharedTexList**(void);
pfList *	**pfuMakeTexList**(pfNode *node);
pfList *	**pfuMakeSceneTexList**(pfScene *scene);
void	**pfuDownloadTexList**(pfList *list, int style);
int	**pfuGetTexSize**(pfTexture *tex);
void	**pfuNewTexList**(pfTexture *tex);
pfList *	**pfuLoadTexListFiles**(pfList *movieTexList, char nameList[][PF_MAXSTRING], int len);
pfList *	**pfuLoadTexListFmt**(pfList *movieTexList, const char *fmtStr, int start, int end);
pfSequence *	**pfuNewProjector**(pfTexture *handle);
int	**pfuProjectorPreDrawCB**(pfTraverser *trav, void *travData);
void	**pfuProjectorMovie**(pfSequence *proj, pfList *movie);
pfTexture *	**pfuGetProjectorHandle**(pfSequence *proj);
void	**pfuProjectorHandle**(pfSequence *proj, pfTexture *new);
pfList *	**pfuGetProjectorScreenList**(pfSequence *proj);
void	**pfuAddProjectorScreen**(pfSequence *proj, pfTexture *screen);
void	**pfuRemoveProjectorScreen**(pfSequence *proj, pfTexture *screen);
void	**pfuReplaceProjectorScreen**(pfSequence *proj, pfTexture *old, pfTexture *new);

DESCRIPTION

These utilities assist in the sharing and downloading of textures.

For consistent frame rates, it is very important to download textures into the graphics pipeline's physical texture memory before beginning simulation. This is so that there is no momentary pause while the textures are processed (**texdef**) and downloaded (**texbind**).

An example of the use of these functions can be found in /usr/share/Performer/src/sample/apps/common/generic.c which is used by a number of sample

applications including the IRIS Performer **perfly** sample application.

pfuNewSharedTex examines the application's global list of previously allocated textures for the file *filename*. If the file has already been loaded, the address of the existing pfTexture structure is returned; if not, a new pfTexture is allocated in *arena*, the named file is read, and the address is returned for reference in future requests.

pfuGetSharedTexList returns the list of all textures allocated using the **pfuNewSharedTex** texture-sharing mechanism described above during the current execution of the process. The list returned is useful for many things including texture downloading.

pfuMakeTexList constructs a list of textures by recursively traversing the IRIS Performer scene graph rooted by *node*. Since this traversal is exhaustive no texture will be missed.

Performer supports the notion of a *scene pfGeoState* to represent common rendering state for a pfScene. When this mechanism is used, the texture list built by **pfuMakeTexList** will not include the texture defined by the scene pfGeoState. The function **pfuMakeSceneTexList** duplicates the function of **pfuMakeTexList** and adds the scene pfGeoState's texture to the list if the scene pfGeoState defines a texture.

pfuDownloadTexList visits each texture in the list provided in *list* and performs one of the following functions:

PFUTEX_APPLY
Download each texture without any on-screen fanfare.

PFUTEX_SHOW
Show each texture in the screen while downloading. This is the source of the "slide-show" seen as the IRIS Performer **perfly** program starts up.

PFUTEX_DEFINE
Perform a partial download, performing only the **texdef** operation and not the subsequent **texbind**. This can be used as the basis of simple texture paging mechanisms.

In most cases, **pfuDownloadTexList** will be called in the first traversal through the channel draw callback on each configured **pfPipe**. This function **must** be called from the draw process since it makes direct graphics function calls.

pfuGetTexSize queries the number of bytes of texture used in *tex*.

pfuNewTexList preallocates a sequence of 16 frames on a **pfTexture** for animation.

pfuLoadTexListFiles fills a list within a **pfTexture** from the array of file names, *nameList*, which contains *len* names. If *movieTexList* is passed as NULL, a new list is automatically allocated. The filled list is returned.

pfuLoadTexListFmt fills a list within a **pfTexture** from a sequence of files indicated by the **printf**-style format string *fmtStr*. **pfuLoadTexListFmt** uses **sprintf** and *fmtStr* to construct filenames ranging sequentially from *start* to *end* and adds the textures in these files to the list.

The following routines are used to create and display a movie. A movie has a projector (a **pfSequence** node with a special **pfUserData** and pre-draw callback), a default base frame (a **pfTexture***), one or more screens (**pfTexture***'s) and a reel of movie frames (a **pfList** of **pfTexture***).

The **pfSequence** API is used to run the projector and control the movie display. Screens can be added, removed, or replaced at will. Each projector can have any number of screens but each screen should be in only one projector. The projector node should be the first child of the **pfScene** node. It draws no geometry, but only configures its **pfTexture*** screen to display the correct image when accessed later via traversal of the normal scene graph (or a direct **pfApplyTex**).

pfuNewProjector creates and returns a **pfSequence** containing the list of textures in *handle*, with the textures stored as the leaves under the **pfSequence** and installs **pfuProjectorPreDrawCB** as the default pre-draw stage callback.

pfuProjectorMovie sets the **pfSequence** passed in *proj* to the list of textures passed in *movie*. This enables *movie* to be played in all the screens of *proj*.

pfuProjectorHandle sets the **pfSequence** passed in *proj* to the list of textures contained in the **pfTexture** handle *new*.

pfuGetProjectorHandle returns the **pfTexture** handle from the **pfSequence** passed in *proj*.

pfuGetProjectorScreenList returns the list of screens in the **pfSequence** passed in *proj*.

pfuAddProjectorScreen adds *tex* to the screen list of *proj*. This causes *tex* to reference *proj*'s movie as its list of frames.

pfuRemoveProjectorScreen removes the screen *screen* from the movie *proj*. This sets *screen*'s texture list to NULL and its frame to -1.

pfuReplaceProjectorScreen replaces the screen *old* in the movie *proj* with the screen *mew*. This sets *old*'s texture list to NULL and its frame to -1.

The following code fragment shows how a movie can be created.

Example 1:

```
/* create a projector with a pfTexture handle for the movie tape.
 * The reel base texture can also be used as a base screen.
 * The movie can be the texture list on the handle here, or can
 * be added/replaced later with pfuProjectorMovie(proj, tape) or with
 * pfuProjectorHandle(proj, newHandle);
 */
  pfTexture *handle = pfNewTex(pfGetSharedArena());
  pfSequence *proj = pfuNewProjector(handle);

/* set AUTO_IDLE mode on the handle - new screens on the projector inherit
 * this mode from the handle
 */
  pfTexLoadMode(handle, PFTEX_LIST_AUTO_IDLE, 1);

/* Create a pfList to hold the frames of the movie */
  pfList *tape = pfuLoadTexListFmt(NULL, fmtStr)
         /* or pfuLoadTexListFiles(NULL, fileNameList); */

/* put the movie in the projector */
  pfuProjectorMovie(proj, tape);
```

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfApplyTex, pfList, pfScene, pfSequence, pfTexture, pfUserData, printf, sprintf, texdef, texbind

NAME

pfuNewTimer, pfuInitTimer, pfuStartTimer, pfuEvalTimers, pfuEvalTimer, pfuStopTimer, pfuActiveTimer – Benchmarking and interval timing facilities.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

pfuTimer * **pfuNewTimer**(void *arena, int size);

void **pfuInitTimer**(pfuTimer *timer, double start, double delta, void (*func)(pfuTimer*), void *data);

void **pfuStartTimer**(pfuTimer *timer);

void **pfuEvalTimers**(void);

int **pfuEvalTimer**(pfuTimer *timer);

void **pfuStopTimer**(pfuTimer *timer);

int **pfuActiveTimer**(pfuTimer *timer);

```
struct _pfuTimer
{
    double tstart, tstop, tdelta;

    int  frames;
    double tnow;
    double fraction;

    void        (*func)(struct _pfuTimer *timer);
    void        *data;
    int  dataSize;
};
typedef struct _pfuTimer pfuTimer;
```

DESCRIPTION

pfuTimers provide a real-time, frame rate independent mechanism for defining time-based behavior. A **pfuTimer** is typically initialized with start and stop times, a callback function, and is triggered with **pfuStartTimer**. Then each time **pfuEvalTimer** is called, the function callback will be invoked and can then carry out actions based on the current time.

pfuNewTimer creates a new **pfuTimer** structure in *arena* as well as a user-data memory block of *size* bytes that is referenced by the *data* member of the **pfuTimer** structure.

pfuInitTimer initializes the starting time (*tstart*), duration (*tdelta*), function callback (*func*), and user data

(*data*) of *timer* to *start*, *delta*, *func* and *data*. User data is copied by value, not by reference.

pfuStartTimer starts *timer* and adds it to a static list of active timers. All timers in this list are evaluated by **pfuEvalTimers** and a single, activated timer may be evaluated by **pfuEvalTimer**. A timer must be evaluated for its function callback to be invoked.

When a timer is evaluated, the current time is checked against its active interval defined by [*tstart*, *tstop*]. If the current time is within this interval, *func* will be invoked with a pointer to the **pfuTimer**. The following elements of the timer will be set:

frames The number of times the pfuTimer has been evaluated,

tnow The current time in seconds,

fraction The fraction of the interval that has passed, (*tnow-tstart*)/(*tstop-tstart*). This ranges from 0 to 1.

data The user-data memory block referenced by the timer.

If the current time is not within the active interval, the timer will be removed from the internal timer list.

pfuStopTimer stops *timer* and removes it from the internal timer list.

pfuActiveTimer returns **TRUE** if *timer* is active and **FALSE** otherwise.

NOTES

pfuTimer utilizes **pfGetTime** and its accuracy is dependent on the resolution of available system timers. See **pfGetTime** for more details.

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfGetTime, pfInitClock

NAME

pfuInitTraverser, pfuTraverse, pfuTravPrintNodes, pfuTravGLProf, pfuTravCountDB, pfuTravNodeHlight, pfuTravNodeAttrBind, pfuTravCalcBBox, pfuTravCountNumVerts, pfuTravCachedCull, pfuCalcDepth – Useful scene graph traversals.

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

void **pfuInitTraverser**(pfuTraverser* trav);

int **pfuTraverse**(pfNode* node, pfuTraverser* trav);

void **pfuTravPrintNodes**(pfNode *node, const char *fname);

void **pfuTravGLProf**(pfNode *node, int mode);

void **pfuTravCountDB**(pfNode *node, pfFrameStats *fstats);

void **pfuTravNodeHlight**(pfNode *node, pfHighlight *hl);

void **pfuTravNodeAttrBind**(pfNode *node, unsinged int attr, unsinged int bind);

void **pfuTravCalcBBox**(pfNode *node, pfBox *box);

int **pfuTravCountNumVerts**(pfNode *node);

void **pfuTravCachedCull**(pfNode* node, int numChans);

int **pfuCalcDepth**(pfNode *node);

```
struct _pfuTraverser
{
    pfuTravFuncType       preFunc;
    pfuTravFuncType       postFunc;
    int                   mode;
    int                   depth
    pfNode               *node;
    pfMatStack           *mstack;
    void                 *data;
};

typedef struct _pfuTraverser pfuTraverser;
typedef int (*pfuTravFuncType)(pfuTraverser *trav);
```

DESCRIPTION

pfuTraverser provides a customizable, recursive traversal of an IRIS Performer scene graph. Traversals are useful for many things including database queries like "find and activate all the **pfSequence** nodes in this scene graph".

The **pfuTraverser** facility is used by initializing a **pfuTraverser** structure and invoking **pfuTraverse** with the **pfuTraverser** and a target **pfNode**. Custom database processing is carried out in the pre- and post-traversal callbacks that are provided by the user.

pfuInitTraverser initializes *trav* to the following:

```
preFunc = postFunc = NULL
mode = PFUTRAV_SW_ALL | PFUTRAV_LOD_ALL | PFUTRAV_SEQ_ALL
mstack = NULL
data = NULL
```

mode is a bitmask indicating how to traverse nodes which have an explicit (**pfSwitch**) or implicit (**pfLOD**) switch. The **ALL**, **NONE**, and **CUR** forms of the **PFUTRAV** tokens indicate that the traversal should traverse all, none, or just the current child of the specified node type, e.g. **PFUTRAV_SEQ_CUR** will traverse the currently selected child of all **pfSequence** nodes.

preFunc and *postFunc* are callbacks which are invoked before and after each node in the hierarchy is visited. Callbacks are passed a pointer to the current **pfuTraverser** structure and may access elements of the **pfuTraverser**.

The *node* member references the current node in the traversal. *data* is a pointer to user-data and if non-NULL, *mstack* will contain the matrix stack of the traversal, i.e. all **pfSCS** and **pfDCS** nodes will push, multiply and pop the **pfMatStack** represented by *mstack*. Memory management of the **pfMatStack**' is the responsibility of the application.

The following functions provide a few specific examples of the **pfuTraverser** utility. The specific traversals implemented are

pfuTravPrintNodes
Recursively prints all nodes below *node* into file *fname*.

pfuTravGLProf
Outputs GLprof output tags. If *mode* is true then it places glprof tag callbacks.

pfuTravCountDB
Accumulates stats for scene graph *node* into stats structure *fstats* by traversing *node* with stats open.

pfuTravNodeHlight
Highlights all nodes below *node* by recursive traversal and calling a highlight routine on each.

pfuTravNodeAttrBind
Recursively traverses *node* setting *attr* of each node under *node* to the value *bind*.

pfuTravCalcBBox
Computes the bounding box of *node* and returns it in the parameter *box*. If the *node* parameter is **NULL**, the returned box will be an empty box.

pfuTravCountNumVerts
Returns the number of vertices in the scene graph rooted by *node*.

pfuTravCachedCull
Installs callback functions for each node which cache CULL results between frame updates.

pfuCalcDepth
Returns the depth of the scene graph pointed to by *node*. A single root node with no children is counted as having depth 1.

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfFrameStats, pfGeoSet, pfNode, pfTraverser

NAME

pfuLoadXFont, pfuMakeXFontBitmaps, pfuMakeRasterXFont, pfuGetXFontWidth, pfuGetXFontHeight, pfuGetCurXFont, pfuSetXFont, pfuCharPos, pfuDrawString, pfuDrawStringPos – X Window string drawing routines

FUNCTION SPECIFICATION

#include <Performer/pfutil.h>

void **pfuLoadXFont**(char *fontName, pfuXFont *fnt);

void **pfuMakeXFontBitmaps**(pfuXFont *fnt);

void **pfuMakeRasterXFont**(char *fontName, pfuXFont *fnt);

int **pfuGetXFontWidth**(pfuXFont *f, const char *str);

int **pfuGetXFontHeight**(pfuXFont *f);

void **pfuGetCurXFont**(pfuXFont *f);

void **pfuSetXFont**(pfuXFont *f);

void **pfuCharPos**(float x, float y, float z);

void **pfuDrawString**(const char *s);

void **pfuDrawStringPos**(const char *s, float x, float y, float z);

```
typedef struct pfuXFont
{
    int   size;
    int    handle;
    XFontStruct *info;
} pfuXFont;
```

DESCRIPTION

These functions do simple text drawing using X Windows fonts and string drawing facilities.

Call **pfuLoadXFont** to load the X Window font named by *fontName* into *fnt*. Application programs will usually call **pfuMakeRasterXFont** rather than calling this function directly.

pfuMakeXFontBitmaps allocates display lists under OpenGL for writing the font *fnt* on the screen. This is a necessary step after calling **pfuLoadXFont** and before calling **pfuDrawString** or **pfuDrawStringPos**. Application programs will usually call **pfuMakeRasterXFont** rather than calling this function directly.

pfuMakeRasterXFont loads the font specified in *fontName* into *fnt* and, under OpenGL, allocates display lists for displaying the font. This function is normally used to load and initialize the font; this can be followed by **pfuDrawString** or **pfuDrawStringPos**.

pfuGetXFontWidth returns the width (X dimension) in pixels of the string *str* drawn on the screen in font *f*.

pfuGetXFontHeight returns the height (Y dimension) in pixels of the font *f*.

pfuGetCurXFont sets *f* to the current X Window font for drawing.

pfuSetXFont sets the current X Window font to be *f*.

pfuCharPos sets the current drawing position on the screen. See **cmov** in IRIS GL and **glRasterPos3f** in OpenGL for more details.

pfuDrawString draws the string *s* at the current drawing position.

pfuDrawStringPos draws the string *s* at the provided coordinates.

NOTES

The libpfutil source code, object code and documentation are provided as unsupported software. Routines are subject to change in future releases.

SEE ALSO

pfDataPool, cmov, glRasterPos3f

NAME

pfuNewXformer, pfuXformerMode, pfuGetXformerMode, pfuStopXformer, pfuXformerAutoInput, pfuXformerMat, pfuGetXformerMat, pfuXformerCoord, pfuGetXformerCoord, pfuXformerLimits, pfuGetXformerLimits, pfuXformerCollision, pfuGetXformerCollisionStatus, pfuUpdateXformer, pfuCollideXformer – Backward compatibility module for basic flight models and transformers.

DESCRIPTION

As of IRIS Performer 2.0, the pfuXformer has been moved to libpfui -- the IRIS Performer User Interface library, and is now the pfiXformer. The original pfuXformer API has been preserved, but now has a pfi- prefix. For compatibility with previous releases of IRIS Performer, there are compatibility defines from the old *pfu* routine names to the new *pfi* routine names in <Performer/pfui.h> that are enabled if the PF_COMPAT_1_2 variable is defined.

See the **pfiNewXformer** man page for information on the pfiXformer.

NOTES

These functions are provided to ease the transition of programs from the older IRIS Performer 1.2 release to the current release. These functions should not be used in new application development since they will be removed in a future release of IRIS Performer.

Index

H

L

M

O

P

X

Z

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2783-001.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389