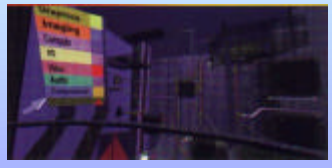
Realtime 3D Computer Graphics & Virtual Reality



OpenGL Performer

What is OpenGL Performer?

- A real-time scene graph based graphics library
- OpenGL Performer is an extensible software toolkit for creating real-time 3D graphics.

Use OpenGL Performer to:

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

Why OpenGL Performer?

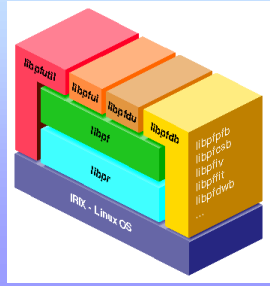
- If your application requires:
  - real-time visuals
  - free-running or fixed-frame-rate display
  - high-performance rendering
- OpenGL Performer drastically reduces the work required to tune your application's performance. General optimizations include:
  - Use of highly tuned routines for all performance-critical operations
  - Reorganization of graphics data and operations for faster rendering

Resources

- *OpenGL Performer Programmer's Guide*
- *OpenGL Performer Getting Started Guide*
- To read these online books, point your browser at:
  - http://techpubs.sgi.com/library/dynaweb_bin/0620/bin/nph-dynaweb.cgi/dynaweb/SGI_Developer/Perf_PG/@Generic__BookView
- For general information about Performer, point your browser at:
  - http://www.sgi.com/software/performer
- Electronic forum for discussions about OpenGL Performer:
- The info-performer mailing list
  - info-performer-request@sgi.com.
  - http://reality.sgi.com/performer (archives)

Performer structure

Set of libraries for different purposes, e.g. for scene graph management, rendering, loading of files...



High-Performance Rendering Library libpr

- Consists of many facilities generally required in most visual simulation and real-time graphics applications, such as:
  - High-speed geometry rendering functions
  - Efficient graphics state management
  - Comprehensive lighting and texturing
  - Simplified window creation and management
  - Immediate mode graphics
  - Display list graphics

High-Performance Rendering Library libpr

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

Visual Simulation Application Library - libpf

- Multiple graphics pipeline capability
- Multiple windows per graphics pipeline
- Multiple display channels and video channels per window
- Hierarchical scene graph construction and real-time editing
- Multiprocessing (parallel simulation, intersection, cull, draw processes, and asynchronous database management)
- System stress and load management

Visual Simulation Application Library - libpf

- Asynchronous database paging
- Morphing
- Level-of-detail model switching, with fading or morphing
- Rapid culling to the viewing frustum
- Intersections and database queries
- Dynamic and static coordinate systems
- Fixed-frame-rate capability
- Shadows and spotlights
- Visual simulation features
  - Environmental model, light points, both raster and calligraphic, animation sequences, sophisticated fog and haze control, landing light capabilities, billboarded geometry

Visual Simulation Application Library - libpf

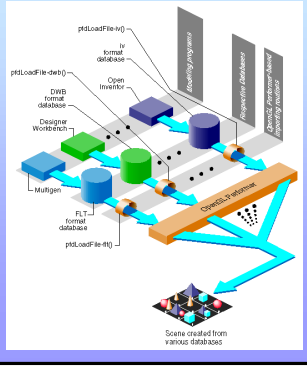
- Visual simulation features
  - Environmental model
  - light points, both raster and calligraphic
  - animation sequences
  - sophisticated fog and haze control
  - landing light capabilities
  - billboarded geometry

Geometry Builder Library (libpfdu)

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

Geometry Builder Library libpfdu

- databases import:
  first, a user creates a database with a modeling program, and then an OpenGL Performer-based application imports that database using one of the many importing routines



Utility Library (libpfutil)

- Processor isolation routines
- GLX mixed mode utilities
- Device input and event handling
- Cursor control
- Simple and efficient GUI and widgets
- Scene graph traversal utilities
- Texture animation or "movies"
- Smoke and fire effect simulation

User Interface Library (libpfui)

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

Database Loader Library (libpfdb)

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

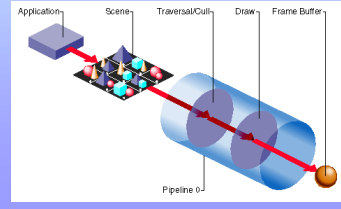
OpenGL Performer Library Structure

- libpf is the visual simulation development library
  - Functions from libpf make calls to libpr functions
  - libpf thus provides a high-performance yet easy-to-use interface to the hardware.
- Multiprocessing Framework
- libpf provides a pipelined multiprocessing model for implementing visual simulation applications. The critical path pipeline stages are:
  - APP
  - CULL
  - DRAW

OpenGL Performer Stages

- APP: update and query scene
- CULL: traverse scene, adds all potentially visible geometry to a special libpr display list, which is then rendered by the draw stage
- DRAW: draw the scene -> OpenGL

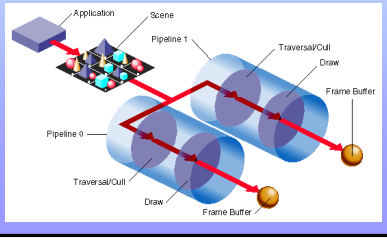
Process flow for a single-pipe system. The application constructs and modifies the scene definition associated with a channel. The traversal process associated with that channel's pfPipe then traverses the scene graph, building a libpr display list.



OpenGL Performer Stages

- Rendering pipelines can be split into separate processes to tailor the application to the number of available CPUs.

Process flow for a dual-pipe system

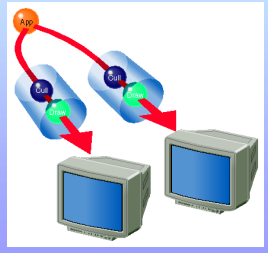


Parallel Pipeline Processes

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

Multiple Pipes

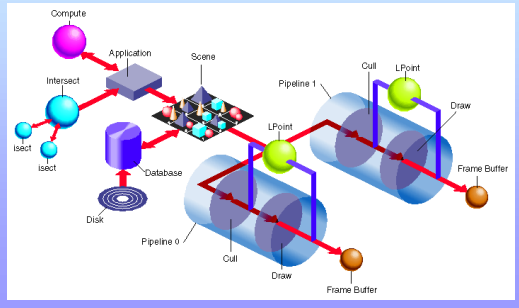
- You may find it appropriate to display your data over more than one display system. For example, you might want to present the left side and right side of a scene on two different monitors. The CULL and DRAW stages are specific to each pfPipe object; the APP stage, however, is shared by both pfPipe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



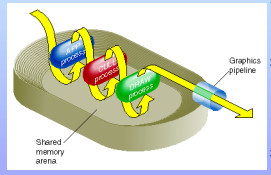
Parallel Pipeline Processes

- An application might have multiple rendering pipelines drawing to multiple graphics pipelines with separate processes. The CULL task of the rendering pipeline can itself be multithreaded.
- Multiprocess operation is largely transparent because OpenGL Performer manages the difficult multiprocessing issues for you, such as process timing, synchronization, and data exclusion and coherence

Parallel Pipeline Processes



Shared Memory



1. After the APP process updates the frame, the process places a copy of unique data for the frame in the shared memory arena.
2. The CULL process takes the frame from the shared memory arena, culls out data invisible to the viewer, and places a revised copy of the frame back in the shared arena memory in the form of a libpr display list for that frame.
3. The DRAW process uses the updated frame and renders the scene to the display system.

More OpenGL Performer features

- Display
  - libpf provides software constructs to facilitate visual database rendering. A pfPipe is a rendering pipeline that renders one or more pfChannels into one or more pfPipeWindows. A pfChannel is a view into a visual database, equivalent to a viewport, within a pfPipeWindow.
- Frame Control
  - Designed to run at a fixed frame rate specified by the application. Measures graphics load and uses that information to compute a stress value. Stress is applied to the model's level of detail to reduce scene complexity when nearing graphics overload conditions.
- Multiple pfChannels on a single pfPipeWindow, multiple pfPipeWindows on a single pfPipe, and multiple pfPipes per machine for *multichannel*, *multiwindow*, and *multipipe* operation. Frame synchronization between channels and between the host application and the graphics subsystem is provided. This also supports simulations that display multiple simultaneous views on different hardware displays.

General Naming Conventions

- **Prefixes**
  - The prefix of the command tells you in which library a C command or C++ class is found. All exposed OpenGL Performer C commands and C++ classes begin with "pf". The utility libraries use an additional prefix letter, such as "pfu" for the libpfutil general utility library, "pfi" for the libpfui input handling library, and "pfd" for the libpfdu database utility library. Libpr level commands still have the `pf' prefix as they are still in the main libpf library.

General Naming Conventions

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

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

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

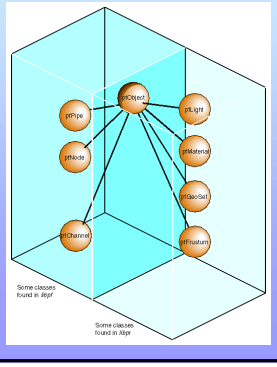
```
C: pfAddChild(node,child);
C++: node->addChild(child);
```

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

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

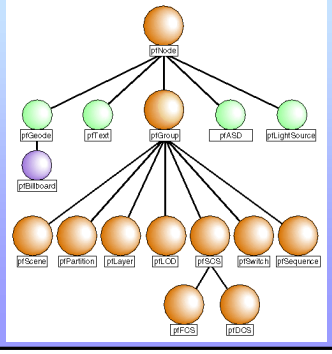
Inheritance Graph

- The relations between classes can be arranged in a directed acyclic inheritance graph in which each child inherits all of its parent's attributes (does not use multiple inheritance)

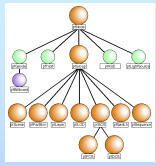


Scene graph nodes

- OpenGL Performer's node hierarchy begins with the pfNode class



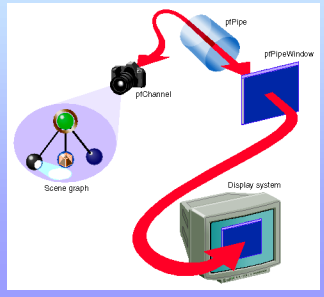
Scene graph nodes



pfNode	Abstract	Basic node type.
pfGroup	Branch	Groups zero or more children.
pfScene	Root	Parent of the visual database.
pfSCS	Branch	Static coordinate system.
pfDCS	Branch	Dynamic coordinate system.
pfFCS	Branch	Flux coordinate system.
pfDoubleSCS	Branch	Double-precision static coordinate system.
pfDoubleDCS	Branch	Double-precision dynamic coordinate system.
pfDoubleFCS	Branch	Double-precision flux coordinate system.
pfSwitch	Branch	Selects among multiple children.
pfSequence	Branch	Sequences through its children.
pfLOD	Branch	Level-of-detail node.
pfLayer	Branch	Renders coplanar geometry.
pfLightSource	Leaf	Contains specifications for a light source.
pfGeode	Leaf	Contains geometric specifications.
pfBillboard	Leaf	Rotates geometry to face the eyepoint.
pfPartition	Branch	Partitions geometry for efficient intersections.
pfText	Leaf	Renders 2D and 3D text.
pfASD	Leaf	Controls transition between LOD levels.

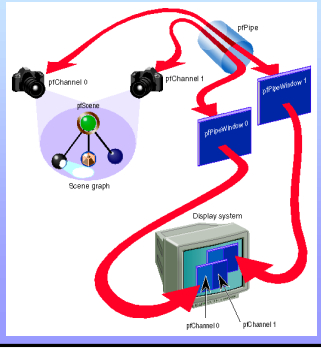
Creating a Display

- Scene-to-Screen Path
  A description of your world is encapsulated in the scene graph, and a view into the world is described by a pfChannel



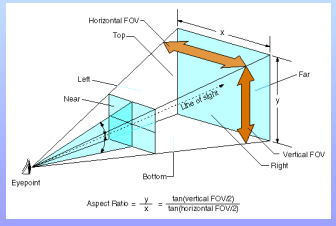
Creating a Display

- A `pfChannel` is a view into a scene graph based on the following:
  - Location and orientation of the camera in the scene
  - Viewing frustum



Creating a Display

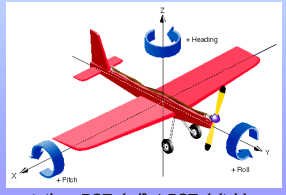
Parameters that define a symmetric viewing frustum (for asymmetric frusta refer to the pfChannel(3pf) or pfFrustum(3pf) man pages for further details.



Creating a Display

Setting Up a Viewpoint: use **`pfChanView(chan, point, dir)`** or **`pfChanViewMat(chan, mat)`** to define the viewpoint for the pfChannel identified by chan.

Heading is a rotation about the *Z* axis
Pitch is a rotation about the *X* axis
Roll is a rotation about the *Y* axis



The value of *dir* is the product of the rotations ROTy(*roll*) * ROTx(*pitch*) * ROTz(*heading*), where ROTa(*angle*) is a rotation matrix about axis *A* of *angle* degrees

Creating a Display

```
main() {
   pfInit(); ...
   pfConfig(); ...
   InitScene();
   InitPipe();
   InitChannel();
/* Application main loop */
   while(!SimDone()) { ... }
}
```

Creating a Display

```
void InitChannel(void) {
     pfChannel *chan;
     chan = pfNewChan(pfGetPipe(0));
/* Set the callback routines for the pfChannel */
     pfChanTravFunc(chan, PFTRAV_CULL, CullFunc);
     pfChanTravFunc(chan, PFTRAV_DRAW, DrawFunc);

/* Attach the visual database to the channel */
     pfChanScene(chan, ViewState->scene);
/* Attach the EarthSky model to the channel */
     pfChanESky(chan, ViewState->eSky);
/* Initialize the near and far clipping planes */
     pfChanNearFar(chan, ViewState->near, ViewState->far);
/* Vertical FOV is matched to window aspect ratio. */
     pfChanFOV(chan, 45.0f/NumChans, -1.0f);
/* Initialize the viewing position and direction */
     pfChanView(chan, ViewState->initView.xyz,
              ViewState->initView.hpr);
}
```

Creating a Display

```
/* CULL PROCESS CALLBACK FOR CHANNEL*/
/* The cull function callback. Any work that needs to be * done in
   the cull process should happen in this function. */
void CullFunc(pfChannel * chan, void *data) {
   static long first = 1;
   if (first) {
        if ((pfGetMultiprocess() & PFMP_FORK_CULL) &&
        (ViewState->procLock & PFMP_FORK_CULL))
        pfuLockDownCull(pfGetChanPipe(chan));
   first = 0;
   }
   PreCull(chan, data);
   pfCull(); /* Cull to the viewing frustum */
   PostCull(chan, data);
}
```

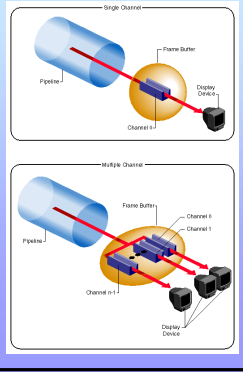
Creating a Display

```
/* DRAW PROCESS CALLBACK FOR CHANNEL*/
/* The draw function callback. Any graphics functionality * outside
   OpenGL Performer must be done here. */
void DrawFunc(pfChannel *chan, void *data) {
   PreDraw(chan, data); /* Clear the viewport, etc. */
   pfDraw(); /* Render the frame */
   /* draw HUD, or whatever else needs * to be done post-draw. */
   PostDraw(chan, data);
}
```

Multiple channel support

- Single-Channel and Multiple-Channel Display:
  - e.g., if multiple channels are needed when inset views must appear within an image
  - for stereo support
  - multiple video outputs per pipeline

  channels can share attributes (grouping of channels)

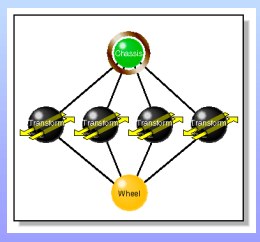


Channel groups

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

Creating a scene graph

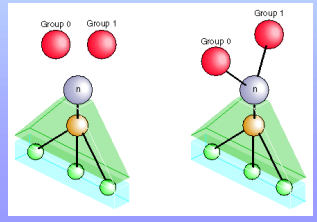
- On your own



Creating a scene graph

Instancing: A scene graph is typically constructed at application initialization time by creating and adding new nodes to the graph. If a node is added to two or more parents it is termed *instanced* and is shared by all its parents

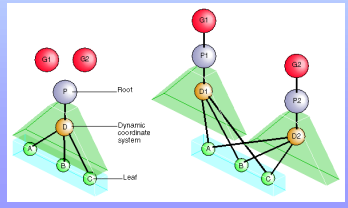
Shared instancing



Creating a scene graph

The cloned instancing operation constructs new copies of each internal node of the shared hierarchy, but uses the same shared instance of all the leaf node.

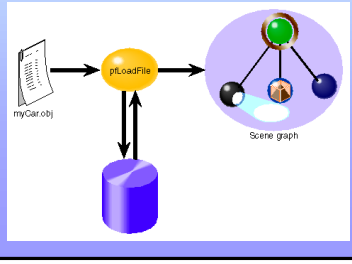
Cloned instancing



Creating a scene graph

- Or by loading it by using

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



Creating a scene graph

- Some common supported loaders:

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

Scene graph traversals

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

Scene graph traversals

- Some nodes, called group nodes, simply pass the traversal to other nodes. In some cases (pfSwitch, pfLOD), the group node passes the traversal only to selected children nodes.
- Other nodes, called leaf nodes, such as a pfGeode node, either encapsulate geometry to be rendered or represent significant computation, such as pfASD.

Scene graph traversals

- **Pipelined Traversals**
  - Several standard traversal operations are usually necessary for basic application operation and for the efficient rendering of a scene
  - Provides automatic and transparent mechanisms for utilizing pipelined and parallel multiprocessing for handling these different traversals.

The following processes can be created for the purpose of handling a specific traversal with its own effective copy of the scene graph nodes:

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

Application Traversal

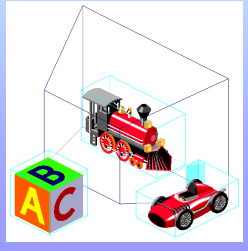
- First traversal that occurs during the processing of the scene graph
- Initiated by calling **pfAppFrame ()**. If **pfAppFrame()** is not explicitly called, the traversal is automatically invoked by **pfSync()** or **pfFrame()**.
- Can be invoked for each channel, but usually channels share the same application traversal (see **pfChanShare()**).
- Updates dynamic elements in the scene graph, such as geometric morphing.
- Used for implementing animations or other custom processing when it is desirable to have those behaviors **embedded in the scene graph** and invoked by OpenGL Performer rather than requiring application code to invoke them every frame.
- The selection of which children to traverse is also affected by the application traversal mode of the channel, in particular the choice of all, none, or one of the children of pfLOD, pfSequence and pfSwitch nodes is possible.

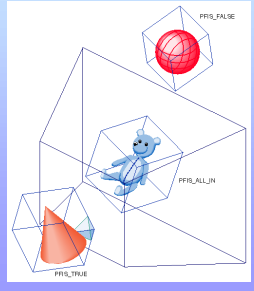
Application Traversal

```
int AttachPendulum(pfDCS *dcs, PendulumData *pd) { pfNodeTravFuncs(dcs,
   PFTRAV_APP, PendulumFunc, NULL);
   pfNodeTravData(dcs, PFTRAV_APP, pd);
}
int PendulumFunc(pfTraverser *trav, void *userData) { PendulumData *pd =
   (PendulumData*)userData;
   pfDCS *dcs = (pfDCS*)pfGetTravNode(trav);
   if (pd->on) {
        pfMatrix mat;
        double now = pfGetFrameTimeStamp();
        float frac, dummy;
        pd->lastAngle += (now - pd->lastTime)*360.0f*pd->frequency;
        if (pd->lastAngle > 360.0f)
                pd->lastAngle -= 360.0f;
   // using sinusoidally generated angle
   pfSinCos(pd->lastAngle, &frac, &dummy);
   frac = 0.5f + 0.5f * frac;
   frac = (1.0f - frac)*pd->angle0 + frac*pd->angle1;
   pfMakeRotMat(mat, frac, pd->axis[0], pd->axis[1], pd->axis[2]);
   pfDCSMat(dcs, mat); pd->lastTime = now; }
   return PFTRAV_CONT;
}
```

Cull Traversal

Culling to the frustum



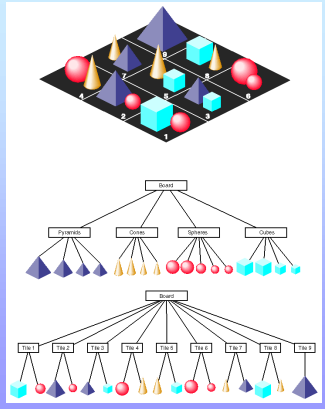


Cull Traversal

1. Prune the node, based on the channel's draw traversal mask and the node's draw mask.
2. Invoke the node's pre-cull callback and either prune, continue, or terminate the traversal, depending on callback's return value.
3. Prune the node if its bounding volume is completely outside the viewing frustum.
4. Traverse, beginning again at step 1, the node's children or geometry (pfGeoSets) if the node is completely or partially in the viewing frustum. If the node is a pfSwitch, a pfSequence, or a pfLOD, the state of the node affects which children are traversed.
5. Invoke the node's post-cull callback.

Culling

Organizing a database for efficient culling: Organizing this database spatially, rather than by object type or other attributes, promotes efficient culling.



Cull traversal

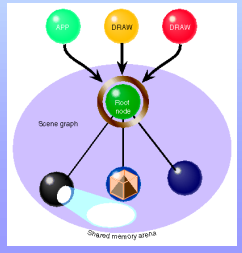
- pfChannel can rearrange the order in which pfGeoSets are rendered for improved performance and image quality by *binning* and *sorting*.
- Binning is the act of placing pfGeoSets into specific *bins*, which are rendered in a specific order.
  - Two default bins: one for opaque geometry and one for blended, transparent geometry. The opaque bin is drawn before the transparent bin so transparent surfaces are properly blended with the background scene

Draw traversal

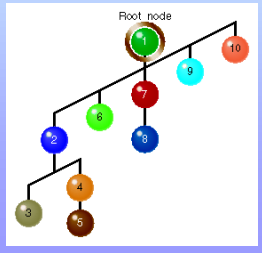
- For each bin the cull traversal generates a libpr display list of geometry and state commands which describes the bin's geometry that is visible from a pfChannel.
- The draw traversal parses all root bins (bins without a parent bin) in the order given by their rendering order value.
- For each root bin, it simply traverses the display list and sends commands to the Geometry Pipeline to generate the image. If a bin has sub-bins, objects that are not in any sub-bin of the bin are rendered first and are followed by objects of each sub-bin. The order in which sub-bins of the bin are drawn is determined by their rendering order value.

Scene graph traversals

- **Stages (sometimes processes) traverse from the root node**

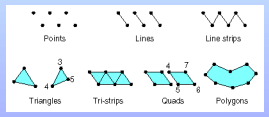


- **Traversal Order: Scene graphs are traversed in a depth-first, left-to-right order.**



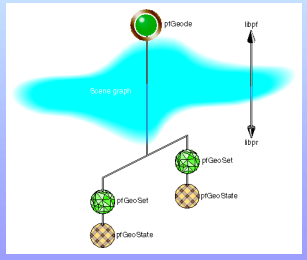
Geometry

- **pfGeoSets** for holding low-level geometric descriptions of objects
- Types:
  - **PFGS_POINTS**
  - **PFGS_LINES**
  - **PFGS_LINESTRIPS**
  - **PFGS_FLAT_LINESTRIPS**
  - **PFGS_TRIS**
  - **PFGS_QUADS**
  - **PFGS_TRISTRIPS**
  - **PFGS_FLAT_TRISTRIPS**
  - **PFGS_TRIFANS**
  - **PFGS_FLAT_TRIFANS**
  - **PFGS_POLYS**



Placing Geometry in a Scene Graph

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



Appearance

- A **pfState** holds the global graphic's state description.
- A **pfGeoState** encapsulates the graphics state elements, such as lighting, transparency, and texture that define the appearance of a pfGeoSet.
- Every **pfGeoSet** must reference a **pfGeoState**. State definitions for the **pfGeoSet** come either from its **pfGeoState**, or from the global, default settings in the global **pfState**.

Appearance

- **pfGeoStates** can specify the following, among other things:

1. Material properties with the **pfMaterial** state attribute object
2. Textures with the **pfTexture** state attribute object
3. Transparency with the transparency mode

Appearance

Defining a pfGeoState:

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

Appearance

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

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

Some reminders

- Open GL Performer supports real time applications
- Strongly related to OpenGL
- Sophisticated scene graph structure
- Database management
- Multiprocessing support
- Multi display support (channel, pipe...)