SiliconGraphics
Computer Systems

IRIS Performer 2.0 Technical Report

Silicon Graphics, Inc.

A description of the functionality and operation of IRIS Performer, the Silicon Graphics toolkit for developers of real-time, 3D graphics software.

TABLE OF CONTENTS

1.0 Introduction

1.1 Scope

This report provides a complete description of the core functionality of IRIS Performer™, and many Performer utility functions. It is intended to help developers determine whether IRIS Performer is suitable for their application, and to provide an introductory overview for programmers. For more detailed information on writing applications using IRIS Performer, see the *IRIS Performer Programming Guide*.

1.2 Related Documents

IRIS Performer Programming Guide, Silicon Graphics, Inc., 007-1680-020.

IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics, John Rohlf and James Helman, SIGGRAPH ‘96 Conference Proceeding.

Designing Real-Time Graphics for Entertainment, Course notes for SIGGRAPH ‘96 Course #6.

Graphics Library Programming Guide, Volumes I and II. Silicon Graphics, Inc.

Graphics Library Programming Tools and Techniques, Silicon Graphics, Inc.

1.3 Description of IRIS Performer

IRIS Performer is a toolkit for developers of real-time, 3D graphics applications. It is a set of libraries which run on all Silicon Graphics (graphics-equipped) computer systems. Performer comes with documentation, sample code and example visual databases. IRIS Performer provides a consistent set of features and application programming interface (API) across all Silicon Graphics platforms.

1.4 Relationship to IRIX™ and the Graphics Library

Performer encompasses the functionality of the graphics library (OpenGL or IRIS GL) and IRIX with REACT™, and provides additional high-level functionality that is useful in real-time graphic applications. Performer applications have full, direct access to REACT and the graphics library. Figure 1 shows the relationship of IRIS Performer to the other elements of the software system.

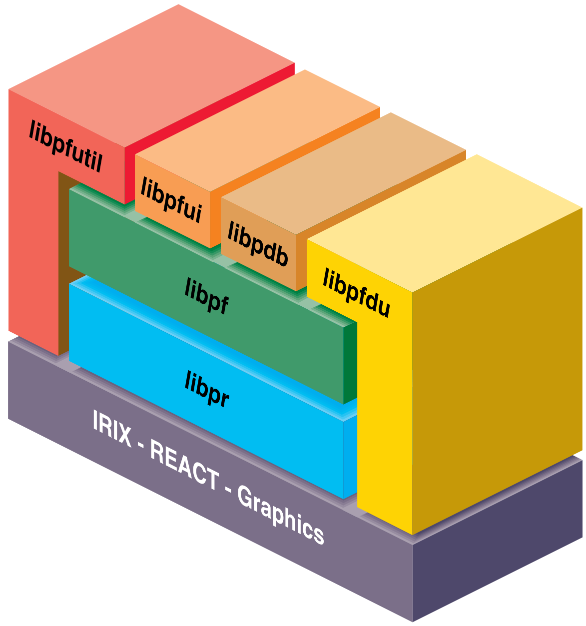

FIGURE 1. IRIS Performer Layer Diagram

1.5 Target Applications

IRIS Performer is designed for use by applications which allow their users to interactively "fly through" a visual database in real-time, viewing a subset during each frame.

Performer is designed for applications with the following characteristics:

- Require maximum graphics performance from a Silicon Graphics system
- Render views of a dynamic hierarchical scene graph description
- Require fast and/or fixed visual frame rate
- May run on a multiprocessor system
- Support interactive user interfaces or immersive displays

Some specific applications that Performer is well-suited for include:

- Classic visual simulation for training and engineering
- Development and deployment of interactive games
- Scripted or live character animation
- Virtual sets and other broadcast video applications
- Walk-through of large architectural and CAD databases

1.6 IRIS Performer Design Objectives

IRIS Performer was created to help software developers use Silicon Graphics systems successfully in real-time graphics applications. In meeting this overall goal, Performer fulfills a set of interrelated objectives that are described in the following sections.

1.6.1 Maximum Graphics Performance

The primary objective of Performer is to enable applications to achieve maximum graphics performance on all Silicon Graphics hardware platforms, both current and future. All engineering trade-offs encountered during the development of Performer are resolved in a way that optimizes performance.

1.6.2 Scalable Application Performance

The Silicon Graphics product family spans a broad range from personal desktop workstations to large multiprocessor racks with multiple graphics subsystems. A corollary to providing maximum performance on every Silicon Graphics platform is providing scalable performance across platforms. That is, the same application should run at maximum speed on every platform without modification. Performer enables an application developed on an Indy™ to run faster when it is moved to an Indigo^2 IMPACT™, and much faster still when it is moved to a multiprocessor Onyx RealityEngine^2™. As Silicon Graphics introduces systems with faster CPUs, increased memory bandwidth, and more powerful graphics subsystems, Performer will enable an existing application to take full advantage of these improvements without recompilation or source code changes.

1.6.3 Image Generator Features

Silicon Graphics platforms and the OpenGL library are suitable for a wide range of 3D graphics applications; including mechanical design, molecular modelling, image processing, visual simulation, film and video, location-based entertainment, and scientific data visualization. The large volume of systems that Silicon Graphics manufactures allows Silicon Graphics hardware to be significantly less expensive than special-purpose Image Generators (IGs) of comparable performance. Enhancing the general purpose functionality of OpenGL, Performer adds specialized functionality that is useful for visual simulation and other real-time graphics applications. A Silicon Graphics platform with Performer provides the best of both worlds; low cost, open hardware with a robust and complete IG feature set.

1.6.4 Simple Access to Complex Features

Implementing certain advanced graphics features requires a complex sequence of graphics library commands.

For each feature supported, Performer provides an identical API for that feature on every Silicon Graphics platform. This ensures that an application can run across the product line without modifications, and take full advantage of the acceleration provided by each platform's hardware.

Similarly, Performer enables Silicon Graphics to provide a smooth transition path for applications using advanced features which will evolve in future hardware generations. The goal of Performer is to make improved hardware result in immediate improvement in the application's performance, without requiring application code changes.

1.6.5 Assist Third-Party Developers

Performer incorporates functionality from which all real-time graphics applications can benefit, such as the ability to select and render only the visible scene from a larger visual database. By incorporating commonly used functionality, Performer frees third party developers to focus on differentiated functionality specific to their target applications.

A developer's license for Performer is priced low to ensure that its cost never interferes with a decision to use Performer. Silicon Graphics allows developers to include a run-time version of Performer with their applications without a license fee.

2.0 Summary of Features

This chapter describes the features of IRIS Performer, and the benefits those features offer to developers of real-time graphics applications.

2.1 Graphics Performance Optimizations

To achieve maximum graphics performance in an application, a developer requires not only mastery of real-time graphics techniques, but also specialized knowledge of the target graphics subsystem. Performer incorporates both types of expertise. It incorporates techniques for reducing the amount of data passed to the graphics subsystem, skill in OpenGL and IRIS GL programming, and detailed knowledge of the operation of the graphics subsystem's hardware and firmware. Performer is the vehicle Silicon Graphics uses to make the benefits of detailed knowledge of its graphics subsystems available to developers.

While prior experience may lead developers to expect that using a toolkit will simplify development at the cost of reduced performance, this is not true with Performer. Because of the specialized knowledge of the hardware that Performer incorporates and the difficulty of fully matching its performance optimizations (including multiprocessing), developers will achieve higher graphics performance using Performer than using either OpenGL or IRIS GL alone.

The following sections provide an overview of the techniques used by Performer to optimize graphics performance.

2.1.1 Static Data Optimizations

Performer includes two techniques which increase drawing speed by optimizing the representation of visual data in memory. These techniques are usually performed at the time the visual database is initially loaded from disk.

2.1.1.1 Geometric Primitive Optimizations

The graphics primitive that all current Silicon Graphics systems draw fastest is a triangle strip. Performer will convert as many other graphics primitives as possible into triangle strips in order to optimize drawing performance. This is a hardware-specific optimization. If a future graphics subsystem is optimized for drawing some other type of primitive, an additional database optimization module will be added to Performer to support this graphics subsystem.

2.1.1.2 Flattening

Another technique for static data optimization is called flattening. Flattening increases drawing speed at the expense of increased memory usage. This technique can be applied to objects which appear multiple times in the database, and which are represented as a master template plus a transformation matrix for each instance. The speed at which these objects are processed in real-time can be increased by pre-applying the transformation and storing each instance of the object separately. For example, trees are often represented as a single object located at the origin, then instanced using a set of transformation matrices which cause each instance of the tree to be drawn at a different location in the scene. When flattening is applied, each transformation matrix is pre-applied to the object, and the set of objects which result are each stored separately.

Flattening can be used in a similar manner to increase the drawing speed of self-advancing geometry sequences. These are described in the section “Self-Advancing Geometry” on page 7.

2.1.2 Visibility Culling

A common technique for improving graphics subsystem performance is to sort the visual database prior to drawing each frame to determine which objects will be visible in that frame. Only visible objects are subsequently passed to the graphics subsystem, thus reducing its load. Because it “culls out” all objects not visible in the current frame, this technique is called visibility culling. Performer uses the CPU subsystem to perform visibility culling, which effectively shifts work to the CPUs from the graphics system. As will be discussed in the section “Transparent Multiprocessing” on page 7, this technique is easily adapted to utilizing multiple processors when they are available.

2.1.3 Drawing Optimizations

An additional performance benefit of visibility culling is that it enables Performer (optionally) to sort the geometry in each visual frame into an order that optimizes drawing performance. Sorting the data in each frame before drawing it offers several opportunities for performance optimization, as described in the following sections.

2.1.3.1 Minimizing Graphics Subsystem State Changes

Each OpenGL or IRIS GL command passed to the graphics subsystem inherits the state established by the preceding sequence of OpenGL or IRIS GL commands. Changing graphics subsystem state always incurs overhead that decreases performance, but some state changes are much more costly than others. Performer can draw a frame in the order which minimizes the most costly state changes.

2.1.3.2 Eliminating Redundant State Specifications

A related feature of Performer is that it tracks the current state of the graphics subsystem and avoids issuing state specification commands which would be redundant. This improves performance because even commands which re-specify the existing state will incur graphics subsystem overhead.

2.1.3.3 Optimized Rendering Routines

Performer can group the data to be drawn in a frame according to graphics primitive type. To fully exploit this grouping, Performer includes a set of over 700 unrolled rendering routines; one for each combination of graphics primitive type and graphics subsystem state. Each rendering routine contains no “if” tests. Once activated, a routine passes a group of OpenGL or IRIS GL commands to the graphics subsystem using the tightest code loop possible.

2.1.4 Level of Detail Switching

Another common technique for conserving resources in the graphics subsystem is level-of-detail (LOD) switching. For each object to which the technique is applied, the developer creates a series of models covering a range of polygonal complexity. Performer will automatically draw progressively simpler models as the object moves farther away from the eyepoint. Performer enables the developer to specify the range over which each model is drawn, and offers a choice of techniques for visually smoothing the transition between models. When managed properly, LOD switching can significantly reduce the workload on the graphics subsystem without reducing the perceived resolution of the scene.

In order for Performer to use level of detail switching when drawing an object, a developer must include multiple models of the object in the visual database, each with a different level of detail. (Among applications which use this technique, three or four different levels of detail is common.)

2.1.5 Transparent Multiprocessing

Visibility culling conserves graphics subsystem resources by using the CPU subsystem to pre-process each frame, thus moving work from the graphics subsystem to the CPU subsystem. The CPU subsystem also executes the code which passes vertex data to the graphics subsystem's hardware. In a typical real-time graphics application, the application program runs on the CPU subsystem as well. Performer enables these CPU subsystem tasks (and others described in subsequent sections) to be spread across multiple CPUs without requiring changes to the application software. By increasing the amount of work that can be off-loaded from the graphics subsystem to CPU subsystem without causing it to become the system bottleneck, multiprocessing increases graphics performance. More generally, Performer enables applications which are much larger than a single CPU can support to achieve the maximum performance available from the graphics subsystem.

2.2 Real-Time Graphics Features

Performer is tailored for a particular class of applications; i.e. real-time graphics. The way in which Performer organizes data, and what Performer "knows" about how that data will be used enables it to incorporate high-level functionality that is useful in real-time graphics applications. These features are described in the following sections.

2.2.1 Intersection Testing

Performer will determine the point of intersection between a group of line segments and a scene. For example, an application can use intersection testing to detect collisions between objects in a scene, to maintain contact between the "ground" and a vehicle being driven through the scene, or to pick objects with the mouse. To use intersection testing, the application passes a set of line segments to Performer. For each intersection, Performer returns the intersection point and the identification of the intersected primitive.

2.2.2 Self-Advancing Geometry

Performer supports self-advancing sequences of geometric transformations. These can be used to add animation to geometric models. During modelling, the programmer defines the sequence of transformations to be applied to the geometry. When the self-advancing geometry sequence is triggered, Performer steps through the sequence of transformations at a fixed rate (defined in wall clock time). To the person watching the visual display, the object appears to move or change, just as cartoon characters created using cell animation appear to move.

A simple example of the application of self-advancing geometry is explosions. When an explosion is triggered, a self-advancing sequence can be used to display polygons flying out from the center, followed by rising flames, and then smoke. Self-advancing geometry can be used to handle any non-interactive animated behavior. The behavior is non-interactive because the self-advancing sequence of spatial translations cannot be steered by a user, but instead will proceed the same way each time it is triggered.

2.2.3 Fixed Frame-Rate Operation

For applications in which maintaining a fixed frame rate is a primary objective, Performer provides support for dynamically reducing detail within the frame as needed to maintain the frame rate. In general, maintaining a fixed frame rate requires that the CPU and graphics subsystems can complete the processing and drawing of every possible scene in the visual database within the frame time. Database tuning is the most important factor in meeting this requirement. From a practical standpoint, however, tuning for every possible visual scene in a database is usually impossible. The LOD switching capability provided by Performer can be used to provide additional margin to ensure that an application with a well-tuned database can maintain a fixed frame rate in all cases.

There are two capabilities which underlie the fixed-frame-rate support in Performer. One is the ability to determine when the graphics subsystem is approaching the limit of its capacity. The other is the ability to dynamically reduce the load on the graphics subsystem to ensure that it can draw each visual frame within the frame time.

In order to determine when the graphics subsystem is approaching the limit of its capacity, Performer can monitor the percentage of the frame time that is required to draw each visual frame. This parameter is called *stress*.

LOD switching (see “Level of Detail Switching” on page 6) provides a means for Performer to dynamically reduce the load on the graphics subsystem in high-stress scenes by drawing less detailed models. When properly applied, the overall visual experience can be optimized by preventing frame rate changes at the expense of somewhat lower visual detail.

Performer also makes the stress parameter available to the application. This enables the application to help maintain a fixed frame rate, provided it has a means of reducing graphics subsystem load. For example, the application could move the far clipping plane closer to the eyepoint, or dynamically designate some objects to be omitted from the rendered frame.

2.2.4 Database Paging

Performer 2.0 provides support for an application which uses a visual database that is larger than the physical memory of the system on which it executes. An application can invoke Performer to replace some portion of the database in memory with a new portion of the database from disk. These operations can occur in the background without affecting real-time performance. The application is responsible for determining when to load additional data into memory from disk, and which portion of the existing data in memory should be overwritten.

2.3 Architectural Features

Silicon Graphics hardware platforms include architectural features that make them inherently well suited for real-time graphics applications. The following sections describe Performer functionality that is based on these architectural features.

2.3.1 Immediate Mode Rendering

Onyx incorporates a fully symmetric, shared memory architecture that uses a 1.2GB/S system bus. (The system bus interconnects the CPUs, graphics subsystem(s), and I/O subsystems with the shared memory.) Onyx is especially well suited to real-time graphics applications because its architecture and high speed bus enable it to employ immediate mode rendering. That is, the entire frame of data is drawn from shared memory during each frame time.

By contrast, display list systems cache frame data within the graphics subsystem. This is done to reduce the required data bandwidth into the graphics subsystem. The display list technique is necessary in systems where a physically separate image generator is interconnected with the CPU subsystem using a network or other low-speed link. Display list techniques pose a significant limitation to the performance of interactive real-time graphics applications, because they lengthen the time needed for the application to read the visual data in the current frame. Because the graphics subsystem has the only copy of the currently visible data, the application must send an inquiry over the low-speed link to the graphics subsystem, then wait for the graphics subsystem to process the inquiry and return the results. This approach increases transport delay, and can cause database inquiries to negatively affect drawing performance.

The shared memory architecture of Onyx allows the application to have direct read/write access to the same copy of the data that is used by the IG. This approach minimizes transport delay, and enables the application to make inquiries without adding any additional workload to the IG. The extremely high bus bandwidth of the Onyx architecture ensures that the application's memory accesses do not affect the performance of the IG, and vice versa.

Performer delivers the full benefit of the Onyx immediate-mode architecture to real-time graphics applications. Performer maintains a single copy of the visual data which can be shared among the application, the draw process, and any other process that needs access to the data. Instead of copying data between the CPU subsystem and the graphics subsystem, Performer passes pointers to the single copy of the data, which resides in shared memory. An application can directly make inquiries of the data displayed in the current frame, or modify data which will be drawn during the next frame.

Performer enables an application to modify the position, shape, color, lighting and texture of objects in the visible scene.

2.3.2 Support for Multiple Graphics Subsystems

Onyx systems can be configured with up to three RealityEngine^2 graphics subsystems, and Performer greatly simplifies utilization of multiple graphics subsystems in an application. For example, multiple graphics subsystems can be slaved together to provide multiple out-the-window views from a single eyepoint. Once the relationship among the subsystems is established, the application specifies a single viewpoint for each frame, and Performer renders the correct views.

2.3.3 Support for Multiple Channels

When configured with a Multi-Channel Option (MCO), Onyx can source multiple video outputs from a single graphics subsystem. Performer enables an application to generate multiple views into a visual database every frame, and display each view on a separate video output channel.

The views can either be independent or linked. For example, independent views are useful in a game application where each video output supports a different player. Linked views could be used to provide multiple out-the-window views for a single player. Another common use of linked views is to create sensor simulations in military aircraft. In this case, the sensor viewpoint may be the same as the out-the-window view appearing on a different channel, but a different look-up-table can be used to simulate sensor effects such as night vision or thermal scans.

Performer applications can employ multiple channels without having separate video output channels. For instance, a separate channel can be used to generate a radar display which replaces part of the out-the-window scene in the display of a flight simulation game.

2.3.4 Scalable Graphics Performance

Performer incorporates a layered architecture which enables application performance to scale up with increasing hardware capability. The upper layer (*libpf*) is the same for all Silicon Graphics platforms, while the lower layer (*libpr*) comprises a set of hardware-specific rendering modules. *libpf* provides a consistent API across platforms and supports transparent multiprocessing. *libpr* enables Performer to provide fully optimized rendering on each graphics subsystem. As a result, the performance of an application based on Performer will increase when it is moved to a higher performance graphics subsystem, and will also increase when additional CPUs are made available.

2.3.5 Database Independence

Performer does not define its own database format. Instead, it can use data that has been stored in any of a variety of formats. This enables an application to be independent of the modelling tools used to create the database the application displays. Performer also enables applications to combine models generated by different tools seamlessly into a single visual scene.

2.4 Visual Effects

2.4.1 Environmental Model

Performer includes a set of functions that clear the screen between each visual frame, and implement various atmospheric effects. Collectively, these functions are known as the earth/sky model. The colors of the sky, horizon and ground can be changed in real-time to simulate a specific time of day.

The complexities of atmospheric effects on visibility are approximated within Performer using a multiple-layer sky model, set up as part of the earth/sky. Separate layers represent the effects of ground fog, clear sky, and clouds.

The earth/sky model in Performer enables an application to easily move from a featureless background to a natural outdoor background.

2.4.2 Billboards

“Billboards” refers to a technique commonly used in visual simulation applications to reduce the graphics subsystem resources needed to display complex objects that are roughly symmetrical about one or more axes. A classic application of billboards is to represent a tree using a single textured quadrilateral polygon. The billboard automatically rotates to face the viewer at all times. A billboard can produce visually acceptable results using far fewer polygons than would be required for a solid model. Accordingly, billboards reduce both transformation and pixel fill demands on the graphics subsystem at the expense of some additional host processing. Other objects that are well suited to billboards are smoke, fire and clouds.

2.4.3 Light Points

Light points are light sources which are visible, but which do not illuminate geometry in the frame. Light points are used in flight simulation applications to simulate runway lighting, taxiway lights, and street lights. Performer 2.0 can automatically attenuate the brightness of light points as a function of their distance and angle from the viewpoint.

(Intentionally left blank.)

3.0 Basic Real-Time Operation

This chapter introduces the dynamic interaction among the functional elements of Performer, as well as a number of terms. Both of these will be described in more detail in subsequent chapters.

The following diagram shows the functional elements and data flow required for basic Performer operation; i.e. loading a visual database from disk and then drawing frames in real-time. The relationship among these functions is shown in Figure 2. They are described in the following sections.

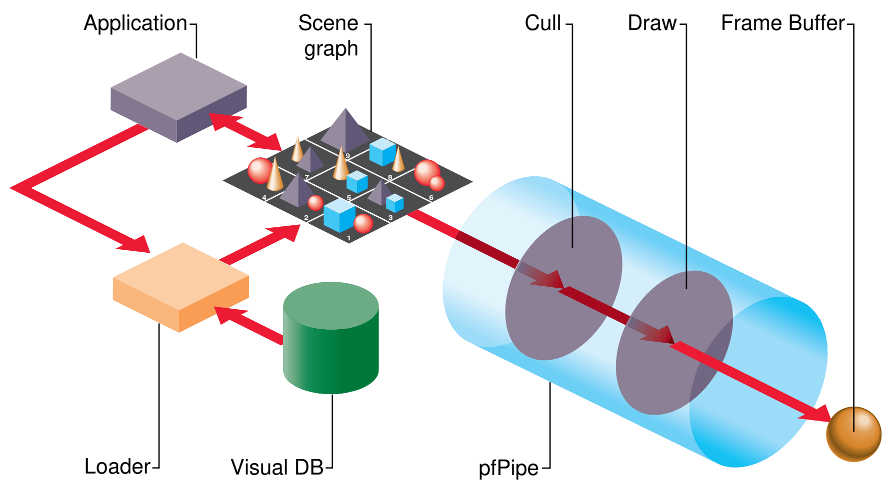

FIGURE 2. Functional Block Diagram for Basic Real-Time Operation

3.1 Loading the Visual Database

Prior to beginning real-time operation, the application invokes a loader to read the visual database from disk into memory. In some cases the target platform is configured with enough physical memory to contain the entire database, and database loading is completed during initialization. In other cases, the application loads a portion of the database during initialization, then begins real-time operation and concurrently loads additional segments as needed.

As Performer loads the visual database from disk, the data is converted to a set of objects called nodes. Performer stores the nodes in a hierarchy called a scene graph. A scene graph is the in-memory representation of a disk-based visual database. Both the nodes and their organization contain information that Performer uses to efficiently render visual frames.

Once the scene graph has been created, the application can invoke Performer functions to perform static optimizations which increase the application's performance during subsequent real-time operation.

3.2 Visibility Culling

Visibility culling is performed by the cull traversal module. (The performance advantage provided by visibility culling is described on page 6.)

To use visibility culling, the application first defines a viewing frustum in terms of its angular field-of-view in two dimensions, and its near and far clipping planes. Geometrically, the viewing frustrate is a truncated pyramid like the one in Figure 3.

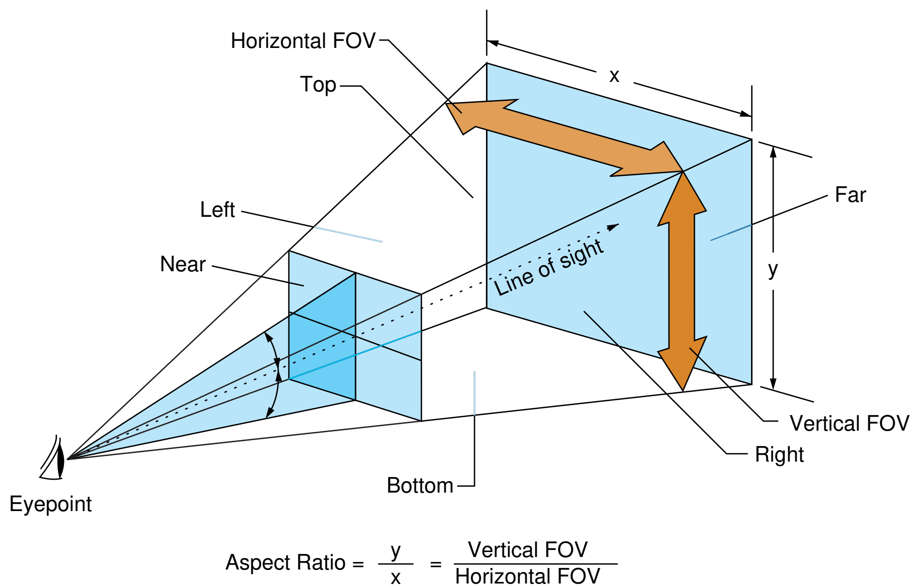

FIGURE 3. Specification of the Viewing Frustum

For each frame drawn, the application specifies the placement of the viewing frustum within the visible database by passing eyepoint and gaze direction information to the cull traversal. The cull traversal then processes ("traverses") the scene graph to determine which objects are at least partially visible in the current frame. It does this by recursively comparing the viewing frustum to the bounding volume which contains all of the geometry within each node encountered during the traversal. For more information on the cull traversal and hierarchical bounding volumes, see the sections "Scene Graph Characteristics" on page 17, and "Cull Traversal" on page 27.

Performer supports two operational modes of the cull traversal. In one mode, the cull traversal passes a pointer to each visible object to the draw function as soon as the object's visibility is determined. In the other mode, the cull traversal builds a list of pointers to nodes containing geometry that is visible in the current frame. This list of pointers is called a pfDispList (Performer display list)[1].

As a pfDispList is built, the items in it are sorted by their graphics mode (i.e. primitive type, attributes, and texture). The sorting order is hardware-specific and corresponds to the order that will achieve maximum drawing performance on the target graphics subsystem. Creating a sorted pfDispList enables Performer to organize the nodes in the scene graph for optimum performance of the cull traversal, yet draw the frame in the order that is optimal for the hardware. Because the cull traversal sorts pointers to the data, the actual data is never copied in memory.

The performance benefits of creating a pfDispList are described in the section "Drawing Optimizations" on page 6. For more information of how pfDispList sorting is implemented, see the section "Sorting Objects To Optimize Drawing Performance" on page 28.

3.3 Drawing

Pointers to nodes which contain visible geometry are passed from the cull traversal to the draw traversal.

The draw traversal makes OpenGL or IRIS GL library calls, which in turn pass commands to the graphics subsystem hardware. The draw function consists of over 700 routines, each of which is optimized to draw one combination of specifications (primitive, attribute list, attribute binding). Because the Cull Traversal has already sorted the display list by specifications, all "if" tests can be eliminated from the draw routines. For example, for a collection of triangles which have colors defined per-primitive (i.e. on color per triangle), the corresponding rendering routine does not need to test whether a color should be sent down with each vertex.

1. Performer employs immediate mode rendering, not display list rendering. Whereas a conventional display list contains the data to be drawn, a pfDispList contains only pointers to the data. The draw function in Performer reads data from the scene graph and passes it directly to the graphics subsystem hardware. The data is never copied from one memory location to another.

(Intentionally left blank.)

4.0 Scene Graph Characteristics

A scene graph is a hierarchical structure composed of interconnected database units called nodes. Scene graphs and nodes have an implicit set of characteristics which are integral to the functioning of Performer. These are described in the following sections.

4.1 Traversal Order

Performer processes its database using functions which visit the nodes and operate on them in a predetermined order: top to bottom; left to right. This database processing is generically referred to as traversing or traversals. Specific types of traversals are described in more detail in Section 6.0, "Traversals," on page 27.

4.2 Traversal Control

Nodes include a traversal mask to allow the application to mask off subgraphs of the scene from the traversal functions. Separate masks are used for each type of traversal supported. A node (and the subgraph beneath it) is traversed only if the node's mask bit for that traversal is set. This allows multiple databases to coexist in the same scene graph. For example, a scene graph may contain simpler geometry for intersection testing than for rendering in order to reduce the time spent in the intersection traversal.

4.3 State Inheritance

In addition to providing a logical ordering of its nodes, the scene graph hierarchy defines how state is inherited between nodes during traversals. Specifically, state is inherited from top down only. The absence of any left-right or bottom-up inheritance allows arbitrary pruning of the scene graph during a traversal. It also allows parallelization of a single traversal because subgraphs can be traversed independently.

The only type of state for which Performer directly supports inheritance is 3D transformations, but user callbacks may cause other types of state to be inherited. 3D transformations (or more simply, "transformations") are 4x4 homogeneous matrices which specify scaling, rotation, and translation. Both static and dynamic transformations are inherited, and transformations that inherit other transformations are combined, allowing chained articulations and complex modelling hierarchies.

Graphics subsystem state variables which the Cull Traversal uses to sort OpenGL or IRIS GL commands are specifically NOT inherited, since they are always contained in leaf nodes at the bottom of the hierarchy. Grouping the primary specification of graphics state in leaf nodes rather than internal nodes greatly simplifies the task of sorting by graphics mode.

4.4 Hierarchical Bounding Volumes

The scene graph also defines a hierarchy of bounding volumes which are used to accelerate the cull and intersection traversals. Each node has a bounding volume that encloses all geometry defined in the sub-tree lying underneath that node. Performer automatically recalculates these bounding volumes when geometry or scene graph topology changes.

While hierarchical bounding volumes support optimization of the cull and intersection traversal, the application's scene graph must be structured properly for this optimization to be realized. In general, objects that are spatially close together should be organized into one node. For the same reason, nodes that are spatially clustered should be organized into one group. Also, traversal performance can be better optimized for a deep hierarchy that for a flat one.

For more information on how to organize data in a scene graph, refer to the *IRIS Performer Programming Guide*.

5.0 Nodes

Performer defines approximately 15 node types. (For a complete list, refer to the *IRIS Performer Programming Guide*.) All node types fall into three basic classes: root; internal; and leaf. Node type names follow the convention pf<name>; e.g. pfSequence.

5.1 Root Nodes

A root node is the top node of a scene graph and is used to refer to the entire scene graph when invoking a traversal. pfScene is the one type of root node supported by Performer.

5.2 Leaf Nodes

All visual geometry data contained in a scene graph is contained in leaf nodes. These data consist of geometry, attributes of the geometry, and elements of graphics subsystem state at the time the geometry is drawn (e.g. texture). The combination of the data in a leaf node and the global graphics subsystem state data (e.g. lighting model) maintained by pfState fully specify the rendered appearance of a visual object. See “Managing the State of the Graphics Subsystem” on page 39 for more details.

pfGeodes (GEometry nODES) are the most commonly occurring leaf nodes because pfGeodes are used to represent most types of visual geometry that can occur in a scene graph. In addition to pfGeodes, there are three other types of leaf nodes: pfBillboard nodes, pfLightSource nodes, and pfLightPoint nodes. These three node types implement the visual simulation features for which they are named, as described in the section “Visual Effects” on page 11.

5.2.1 pfGeodes

pfGeodes contain a list of data structures called pfGeoSets, which in turn contain visual geometry data. Each pfGeoSet references a pfGeoState, which contains the specification of graphics subsystem state required to correctly render the geometry in the pfGeoSet. (Note that pfGeoSets and pfGeoStates are not nodes themselves, but are elements of a PfGeode node.)

5.2.1.1 pfGeoSets

A pfGeoSet is a collection of geometry of a single type, where a type is defined by its:

- Geometric primitive: points, lines, line strips, triangles, quadrilaterals (quads), or triangle strips
- Attribute lists: vertex coordinates, colors, normal, and texture coordinates
- Attribute bindings: per-vertex, per-primitive, overall, or off

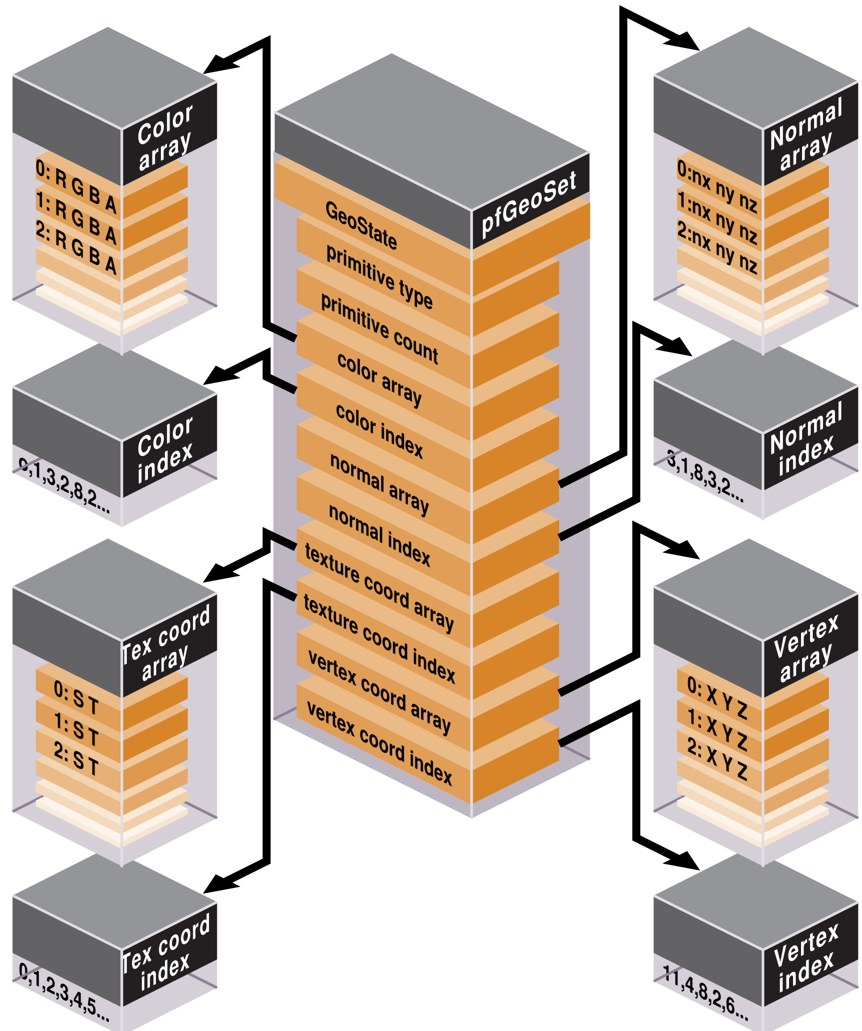

FIGURE 4. pfGeoSet Structure

By creating *a priori* groupings of geometric primitive types, pfGeoSets enable Performer to use a set of specialized, extremely tight rendering loops to pass the data to the graphics subsystem. The loops are specialized and extremely tight in that they contain no "if" statements for determining which geometric primitive type is to be drawn. Instead, Performer includes a separate rendering routine for each geometric primitive type. Once Performer has selected the appropriate rendering loop for drawing a pfGeoSet, no further "if" tests are required to draw that pfGeoSet.

Enumerating all combinations of characteristics of graphics primitives produces over 700 unique graphics primitive types. Performer includes a specialized, macro-generated rendering routine for each type.

5.2.1.2 pfGeoStates

A full specification of how a visible object should appear must include not only the description of the geometry included in a pfGeoSet. It also must include a description of the state of the graphics pipeline at the time the geometry is drawn. Graphics state is partitioned into two categories which are called modes and attributes. Modes are graphics subsystem state variables that have simple ON or OFF values, such as transparency enable or texture enable. Attributes define the more complicated elements of graphics subsystem state, such as texture or lighting model.

While it is possible to define all state variables directly using pfGeoStates, Performer also allows an application to establish global default values which prevail unless a pfGeoState explicitly overrides them. See “Managing the State of the Graphics Subsystem” on page 39 for more a complete explanation.

5.2.2 pfBillboard

pfBillboard nodes implement billboards, which are described on page 11. A pfBillboard is a pfGeode in which each pfGeoSet rotates to follow the eyepoint.

A pfBillboard rotates its children’s geometry to follow the view direction or the eyepoint. pfBillboards can either be constrained to rotate about an axis, as is done for a tree or a lamp post, or constrained only by a point, as when simulating a cloud or puff of smoke. Since rotating the geometry to the eyepoint doesn’t fully constrain the orientation of a point-rotating billboard, modes are available to use the additional degree of freedom to align the billboard in eye space or world space.

5.2.3 pfLightSource

A pfLightSource node defines a light source which illuminates a scene’s geometry, but is itself invisible.

In order to illuminate the geometry in a scene graph, a light source must be specified to the graphics subsystem hardware. The pfLightSource node class allows the developer to add a graphics subsystem lighting specification to a scene graph. While the Performer primitive pfLight can also be used to define a light source, pfLight is not a node. Accordingly, it does not benefit from the features provided by a scene graph; e.g. transformation hierarchy, switches, and animation sequences.

pfLightSource nodes can specify either local or infinite light sources. A local light source has both a location and spotlight direction. Accordingly, transformations affect both the location and direction of a local light source. An infinite light source has only a direction. It emits parallel rays because it is considered infinitely far away. Transformations change only the direction of an infinite light source. The scope, or area of illumination of a pfLightSource is global and isn’t affected by the node’s location in the scene graph, unless it is culled during the cull traversal. If it is not culled, the pfLightSource illuminates everything in the pfScene of which it is a member.

A pfLightSource has no default bounding volume, and so by default is not subject to being removed from the scene by view frustum culling. The developer can assign a bounding volume to a pfLightSource, thereby subjecting it to view frustum culling.

Like all nodes, a pfLightSource node inherits transformations and switch values from its parents in the scene graph. This allows the developer to attach a light source to a moving object and easily turn it on and off. For example, a pfLightSource node could be used to attach a headlight to a moving automobile.

5.2.4 pfLightPoint

A pfLightPoint is a leaf node that represents a point light or a set of point lights. Point lights are visible, but do not provide illumination of the scene. Airport runway lights are a typical use of pfLightPoints.

A pfLightPoint node can contain one or many light points that share common attributes such as color, intensity, direction and shape.

5.3 Internal Nodes

Internal nodes contain control and state information that is used by the traversal functions. Internal nodes can be used to direct the path of a traversal (and to provide this capability to the application), to apply transformations to geometry, or to embed macros of OpenGL commands in a scene graph.

A number of the features offered by Performer are implemented as internal nodes. (See "Summary of Features" on page 5.) These internal nodes are described in the following sections.

5.3.1 pfSCS – Static Transformation

A pfSCS node applies an unchangeable transformation to its children. pfSCS nodes are useful for positioning models within a database. For example, a house that is modelled at the origin could be placed in the world using a pfSCS.

A pfSCS contains a transformation matrix that cannot be changed once it is created. For best graphics performance, matrices should be orthonormal (translations, rotations, and uniform scales). Non-uniform scaling requires renormalization to be performed by the graphics subsystem. Projections and other non-affine transformations are not supported.

While pfSCS nodes are useful in modelling, too many of them in a scene graph can reduce culling, rendering, and intersection traversal performance. For this reason, one of the optimizations provided by pfFlatten is to apply static transformations directly to the geometry. For more information, refer to the section "pfFlatten" on page 29.

5.3.2 pfDCS – Dynamic Transformation

A pfDCS node applies a changeable transformation matrix to its children. It can be used to articulate moving parts and show object motion.

The initial transformation applied by a pfDCS node is the identity matrix. The application can modify the transformation (as frequently as each frame) either by specifying a new transformation matrix, or by replacing the rotation, scale, or translation value in the current transformation matrix.

5.3.3 pfLOD – Level-of-Detail Control

pfLOD nodes implement level-of-detail switching, described on page 6. A set of pfGeodes is placed beneath the pfLOD node. Each pfGeode contains a model of the same object at a different level of detail, as illustrated in Figure 5.

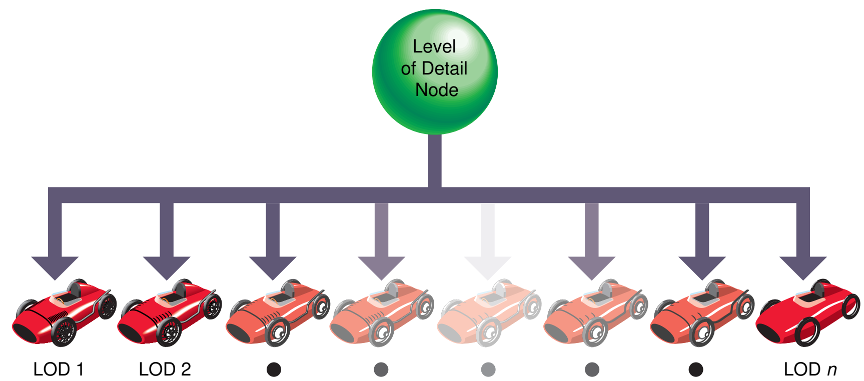

FIGURE 5. pfLOD Node

The application defines the distance for switching between models (called a switch point) in terms of distance from the eyepoint in world coordinates. For each LOD model, the pfLOD node contains a “center-of-LOD” value in x,y,z coordinates and a “switch range” value. The pfLOD node determines which model to draw by comparing the difference between the eyepoint and center-of-LOD values with the switch range value, as illustrated in Figure 6.

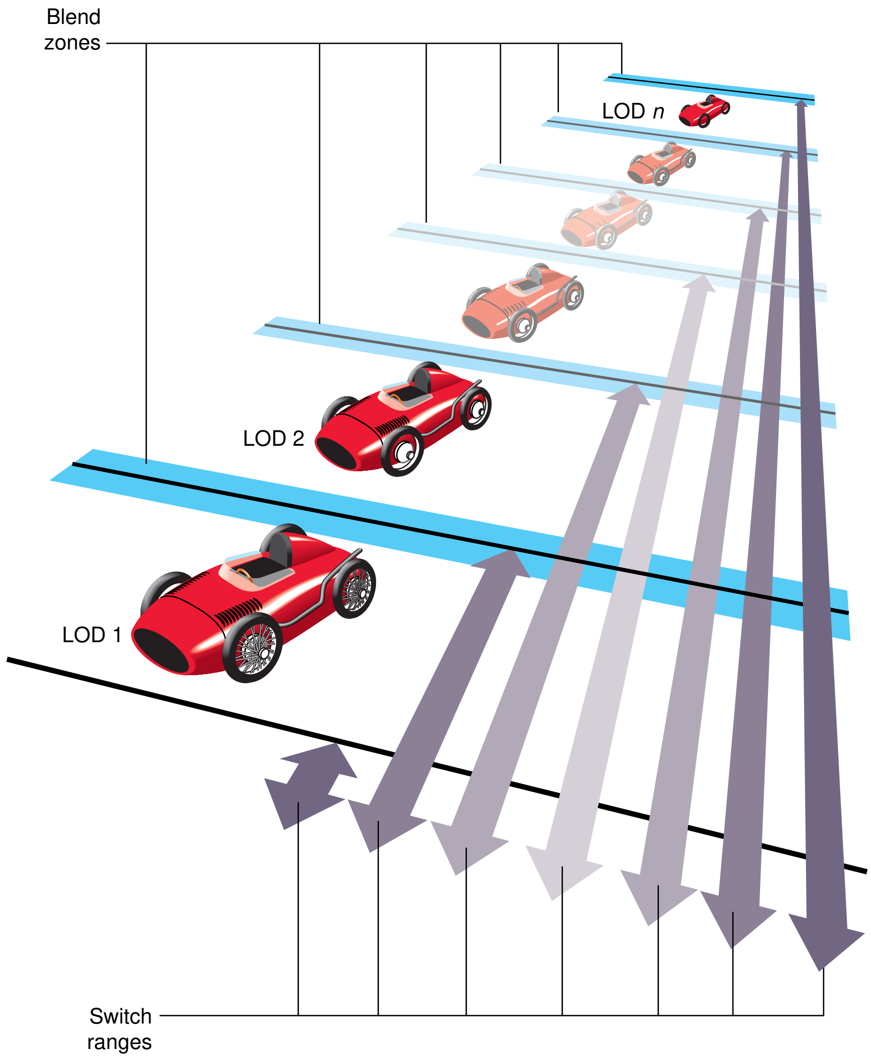

FIGURE 6. LOD Switching

Optionally, the application also defines a blend zone. As the object passes through the blend zone, Performer gradually switches from one model to the other. If the blend zone value is not defined (or is zero), no blending will occur, and one model will replace the other in successive frames when the switch range is crossed. This can produce annoying visible flashes, especially if the eyepoint is moving frequently back and forth across the switch point. Blending reduces this effect.

Performer 2.0 supports two techniques for LOD blending: fading; and morphing.

5.3.3.1 Level-of-Detail Blending Using Fading

When fading is used, Performer draws both models while the object is in the blend zone. The transparency of the two models is varied linearly as the blend zone is transitioned, so that one model becomes increasingly transparent and the other model becomes increasingly opaque. Because fading requires drawing the object twice, it increases the load on the graphics subsystem during the transition.

Currently, LOD blending using fading is supported only on RealityEngine and RealityEngine^2 graphics subsystems.

5.3.3.2 Level-of-Detail Blending Using Morphing

LOD switching using morphing requires that a mapping be established between the vertices in one model and the vertices in the other model. When the object enters the blend zone, Performer begins morphing the geometry of one model into the geometry of the other and drawing the hybrid model. The morphing proceeds linearly across the blend zone. While morphing requires additional work when modelling a database, it imposes no additional load on the graphics subsystem during real-time operation.

Support for blending using morphing is first included in Performer 2.0, and is not available in earlier versions.

5.3.4 pfSequence – Self-advancing Geometry

pfSequence nodes implement self-advancing geometry, described on page 7. The pfSequence node automatically sequences through its children. Each child is assigned a period of wall clock time during which it is displayed. A sequence can consist of any number of children, and each child has its own duration. A sequence can be set to proceed from start to end then either repeat, terminate, or proceed from end to start.

5.3.5 pfLayer – Coplanar Geometry

A pfLayer node resolves the visual priority of coplanar geometry. pfLayer nodes can be used to overlay markings on a polygon. For example, a pfLayer node can be used to apply markings to a runway.

pfLayer allows the application to define a set of base geometry and a set of layered or decaled geometry. The base geometry and the decal geometry should be coplanar, and the decal geometry must lie within the extent of the base polygons.

5.3.6 pfPartition

A pfPartition node can improve intersection and cull traversal performance on a scene graph which exhibits poor spatial arrangement. It does this by imposing a specialized spatial data structure on the scene graph. At the time a database is loaded, the pfPartition node analyzes the geometry underneath itself and partitions pfGeoSets into a 2D grid with multiple membership. During the intersection traversal, pfPartition scan-converts the line segments in the pfSegSet onto the grid to quickly determine which pfGeoSets must be tested and which can be eliminated.

pfPartition is useful in applications which simulate objects which primarily move in 2D, such as land vehicles moving across terrain.

5.4 Instancing Nodes

Instancing is a technique for replicating a node one or more times without duplicating all of the node's data in memory. Instancing offers the benefit of reduced memory usage and simpler modelling. On the other hand, excessive use of instancing can adversely affect performance. Consequently, one function of pfFlatten is to improve performance by fully duplicating the data that is shared by instanced nodes. (For more information see "pfFlatten" on page 29.)

Performer supports two types of instancing: shared and cloned. The difference between these two lies in how transformations affect their geometry. In both types of instancing the geometry is shared, rather than duplicated, in memory.

5.4.1 Shared Instancing

Using shared instancing, a node is duplicated by simply adding a pointer to the instanced node into the new parent node. No data is duplicated. Once instanced, the entire subgraph rooted by the node that was duplicated appears beneath the new parent node.

Shared instancing provides the most efficient use of resources, and is appropriate for duplicating static geometry.

Shared instancing is not appropriate in cases where the duplicate copies of the geometry must move independently. Because all nodes are shared, all transformations applied to one instance of the node will also apply to all other instances of the node. For example, consider an application containing a fleet of identical airplanes which are to be flown independently. Using shared instancing would result in multiple planes that share the same articulations. Consequently, it would be impossible for one plane to fly with its landing gear retracted while another is on a runway with its landing gear down.

Cloned instancing provides the solution for situations where duplicate copies of the geometry must move independently.

5.4.2 Cloned Instancing

Nodes which are created using cloned instancing share the same leaf nodes (i.e. visible geometry and graphics subsystem state), but each has its own copy of any internal nodes (e.g. pfSCS nodes). This enables the visible geometry instances to move independently of each other. Since leaf nodes consume most of the memory in a scene graph, cloned instancing provides most of memory-saving benefit provided by shared instancing.

6.0 Traversals

6.1 Real-Time Traversals

Real-time traversals are ones which must be completed within a single frame time during real-time operation.

6.1.1 Cull Traversal

The cull traversal determines whether or not a node (and thus the sub-tree underneath that node) is within the current viewing frustum. Nodes that are not visible are culled (not drawn) so that the graphics subsystem doesn't spend time processing primitives that couldn't possibly appear in the final image. The output of the cull traversal is a set of pointers to pfGeoSets and pfGeoStates which contain the geometry and local graphics state information necessary for drawing the current frame.

When enabled by the application, the cull traversal produces a pfDispList in which it has arranged the list of pfGeoSets and pfGeoStates in an order which optimizes drawing performance. A pfDispList is always enabled when executing on a multiprocessor, and can be enabled on a single processor. This is described in more detail in the section "Rendering Pipeline Models" on page 32. More information on the sorting of a pfDispList is given later in this section.

6.1.1.1 Using Hierarchical Bounding Volumes

Each node in a scene graph includes a hierarchical bounding volume that spatially encompasses all geometry in the sub-tree underneath that node. To optimize performance, bounding volumes are chosen to be simple geometric shapes whose centers and edges are easy to locate. A sphere is used as the bounding volume for all internal nodes because it can quickly be updated, transformed, and tested. Axially aligned boxes are used as the bounding volume for pfGeoSets because they provide tighter bounds around the actual geometry.

6.1.1.2 Node Testing

The cull traversal uses the hierarchical bounding volumes to efficiently traverse the scene graph. Proceeding top-to-bottom, left-to-right, the cull traversal tests the viewing frustum against the bounding volume of each node. Each test has three possible outcomes:

1. Node is completely outside the viewing frustum: the node is culled. This means that the geometry underneath the node will not be drawn, and the cull traversal will proceed no further down the sub-tree underneath the node. The cull traversal proceeds to the next node to the right of the culled node.
2. Node is completely inside the viewing frustum: all geometry will be drawn. Again, no further traversal down the sub-tree is required, and the traversal proceeds to the next node to the right.
3. Node is partially inside the viewing frustum: proceed down the sub-tree. In this case, more testing is needed to determine what portion of the geometry underneath the node should be drawn. The cull traversal proceeds recursively to the left-most node underneath the node that tested partially visible. If the node is a leaf node, the pfGeoSets contained in the node are tested against the frustum individually.

6.1.1.3 Sorting Objects To Optimize Drawing Performance

Changing the state of the graphics subsystem can be quite costly in terms of the time required. For example, loading a new texture description into the RealityEngine^2 is a costly state change. While the state change is underway, drawing is suspended.

When a pfDispList is enabled, the output of the cull traversal is a pfDispList which contains a list of pointers to all pfGeoSets that will be drawn during a frame (see "Rendering Pipeline" on page 31). The cull traversal places objects in the pfDispList in an order which minimizes the most costly graphics subsystem state changes.

Generically, the optimization is a 2-level sort. The first sort level minimizes the most time-consuming state change by grouping pfGeoSets which share the same value of this state. For example, pfGeoSets which share the same texture description are grouped when the target platform is RealityEngine^2. The second-level sort groups pfGeoSets which share the same pfGeoState. A further optimization places transparent geometry last in the pfDispList.

Sorting the pfDispList minimizes the number of state changes. A related optimization eliminates redundant state specifications. (A redundant state specification command is one which re-specifies the current state.) This optimization is performed by the draw traversal, and is described in Section 6.1.2.2, "Avoiding Redundant Changes of Graphics Subsystem State," on this page.

6.1.2 Draw Traversal

The draw traversal has been kept logically simple in order to minimize drawing time. All logically complex operations are performed during the cull traversal to help ensure that the graphics subsystem is never starved for data while waiting for the draw traversal.

6.1.2.1 Basic Operation

The draw traversal does not traverse the scene graph directly, but instead traverses the pfDispList that was generated by the cull traversal. Based on the contents of the pfDispList, the draw traversal grabs geometry and state data from the scene graph. The draw traversal includes a set of specialized drawing routines, one for each geometric primitive type. These are described in more detail in the section "pfGeoSets" on page 19.

The output of the draw traversal is a sequence of OpenGL or IRIS GL commands which render the visible frame.

6.1.2.2 Avoiding Redundant Changes of Graphics Subsystem State

The pfState object in Performer mirrors the state of the graphics subsystem. The draw traversal compares each state specification in the pfDispList with the current setting of that state variable in pfState. If the two are the same, the draw traversal does not issue a (redundant) OpenGL or IRIS GL command to the graphics subsystem. (pfState is described more fully in the section "Minimizing OpenGL and IRIS GL State Commands" on page 39.)

6.1.3 Intersection Traversal

The intersection traversal {pfSegsIsectNode()} tests for intersections between application-defined line segments and geometry in the scene graph. The intersection traversal differs from the cull and draw traversals in that it must be explicitly invoked by the application.

Intersection testing is based entirely on sets of line segments. The application specifies a spatially grouped set of line segments to be tested using pfSegSet. Optionally, the application can specify a bounding volume which pfSegsIsectNode() will use for intersection testing. Specifying a bounding volume speeds the intersection traversal. Each pfSegSet tested requires a separate traversal of the scene graph.

During each traversal, pfSegsIsectNode() tests the pfSegSet against the hierarchical bounding volumes in the scene graph. The application can direct pfSegsIsectNode() to return a list of intersection "hits". The application can choose to receive a list of "hits" for node bounding spheres, pfGeoSet bounding boxes, or the actual geometry inside pfGeosets (in order of increasing precision). As the precision of the hit report increases, so does the intersection traversal time.

pfSegsIsectNode() provides a discriminator callback that enables the application to examine each "hit" as it is encountered. The callback directs Performer to either accept or reject the intersection, and then to either continue or terminate the traversal.

For efficiency, pfSegsIsectNode() converts the pfSegSet into local object coordinates rather than transforming the node bounding volumes and pfGeoSets into world coordinates.

Because intersection traversals do not modify the database, an application may invoke multiple intersection traversal requests in parallel. Performer will assign each traversal to a separate processor in the CPU subsystem, if one is available.

6.2 Static Traversals

These traversals modify the structure of a scene graph in order to reduce traversal time, improve drawing performance, or modify the configuration of the visual database.

6.2.1 pfFlatten

Performer uses the graphics subsystem hardware to apply non-identity pfSCS transformation matrices to geometry as it is drawn. However, this usually requires sending a new transformation matrix to the hardware. In turn, this requires the hardware to push its matrix stack, apply the new matrix to the geometry under the pfSCS node as it is drawn, then pop its matrix stack. For small models, these stack and matrix operations can consume more time that the actual rendering.

Applying pfFlatten() improves graphics subsystem performance at a cost of increased memory usage. pfFlatten() traverses the scene graph and duplicates static, instanced geometry in memory. It applies the current pfSCS matrix to the geometry, then sets the pfSCS to the identity matrix.

(Intentionally left blank.)

7.0 Pipelining, Synchronization and Multiprocessing

Performer implements coarse-grained multiprocessing using a data-pipelined architecture. That is, a relatively small number of processors work concurrently on different stages of a data pipeline. The processing requirements of real-time graphics mesh well with the characteristics of a pipelined multiprocessing architecture, in that each frame is data-independent of all others, and all frames progress through the same set of computational stages. In a shared memory environment (such as Onyx), data can progress through the stages of the pipeline without ever being copied.

Figure 2 on page 13 shows a functional block diagram of the real-time elements of a Performer application. Each functional block repeats its operation for every visual frame that is drawn. Architecturally, Performer organizes the functional blocks into two types of data processing pipelines: the rendering pipeline shown in Figure 7; and the intersection pipeline shown in Figure 9 on page 37. The rendering pipeline displays a view into the scene graph, and the intersection pipeline identifies intersections between line segments and geometry in the scene graph.

Performer allows each pipeline stage to be assigned to a separate processor. In general, adding processors to the pipeline increases throughput, but also increases latency. This chapter discusses the issues which must be considered when structuring an application to optimize the trade-off between throughput and latency.

7.1 Rendering Pipeline

The application (APP) is the first stage of all pipelines, and it controls each pipeline's execution. A rendering pipeline consists of the APP plus CULL and DRAW stages. The CULL and DRAW are encapsulated by the pfPipe primitive, as shown in Figure 7.

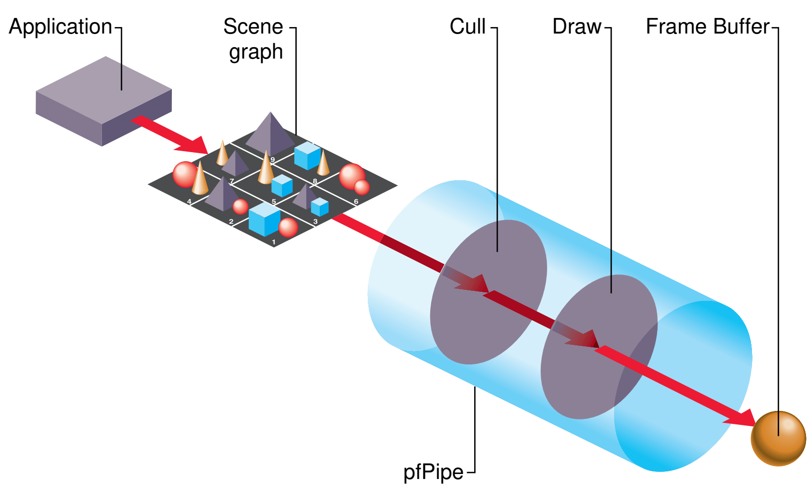

FIGURE 7. Rendering Pipeline

7.1.1 Assigning Pipeline Stages to Unix Processes

Each processor in the pipeline must run a separate Unix process. Applications which are targeted to run on a multiprocessor platform typically assign each pipeline stage to a separate Unix process. These multiple-process applications will run correctly on any number of processors, and will automatically take advantage of multiple processors when they are available. (To achieve steady frame rate, each Unix process should be locked onto a processor that is configured for real-time operation. See the *REACT Technical Report* for more information.)

For applications which will run only on uniprocessors, configuring all pipeline stages within a single Unix process eliminates the overhead of performing a Unix context switch between pipeline stages. (A Unix context switch requires on the order of 100 microseconds.)

Developers can utilize multiple processors within the APP stage, but Performer provides no support for automatically spreading the APP across multiple processors.

7.1.2 Assigning Pipeline Stages to Processors

On a multiprocessor platform, Performer supports a range of possibilities for assigning pipeline stages to processors. The optimum solution depends upon the number of processors available and the relative amount of computational time required for each stage. In the case where the number of processors is not a constraint, the trade-off is purely one of rendering throughput versus rendering latency.

Rendering latency is defined as the elapsed time from viewpoint specification until the display update is completed. Rendering throughput is defined as the amount of geometry processed per unit of time.

7.1.3 Rendering Pipeline Models

Performer enables an application to increase the throughput of its rendering pipeline as more processors are made available for use by the pipeline, up to a maximum of three. The options for assigning pipeline stages onto 1–3 processors are enumerated below, and illustrated in Figure 8.

- PFMPAPPCULLDRAW – Single processor. Each stage is allowed 1/3 frame of execution time. Pipeline latency is one frame.
- PFMPAPPCULL_DRAW – Two processors; APP and CULL assigned to share a processor. APP and CULL are each allowed 1/2 frame execution time; DRAW is allowed a full frame of execution time. Rendering latency is two frames.
- PFMP_APP_CULLDRAW – Two processors; CULL and DRAW assigned to share a processor. APP is allowed one frame of execution time; CULL and DRAW are each allowed 1/2 frame of execution time. Rendering latency is two frames.
- PFMP_APP_CULL_DRAW – Three processors; Each stage has its own processor and is allowed a full frame of execution time. Rendering latency is three frames.
- PFMP_APP_CULLoDRAW (CULL overlap DRAW) – Three processors; APP is allowed a full frame of execution time; CULL and DRAW are allowed more than 1/2 but less than 1 frame time. Rendering latency is two frames.

Notice how the amount of time available to each stage (throughput) increases as the number of processors increases, without any decrease in the frame rate. Notice also that the time required for data to make its way through the pipeline (latency) increases with the number of processors. Specifically, the rendering latency of a pipeline (in video frames) will be equal to the number of processors in the pipeline, except in the PFMP_APP_CULLoDRAW mode. This mode allows the DRAW stage to begin processing the pfDispList before the CULL stage has completed building it, and there by reduces the rendering latency to two frames. Provided that all processors are well-utilized, the rendering throughput will increase proportionally as the number of processors is increased.

To specify a pipeline model, the application calls pfMultiprocess().

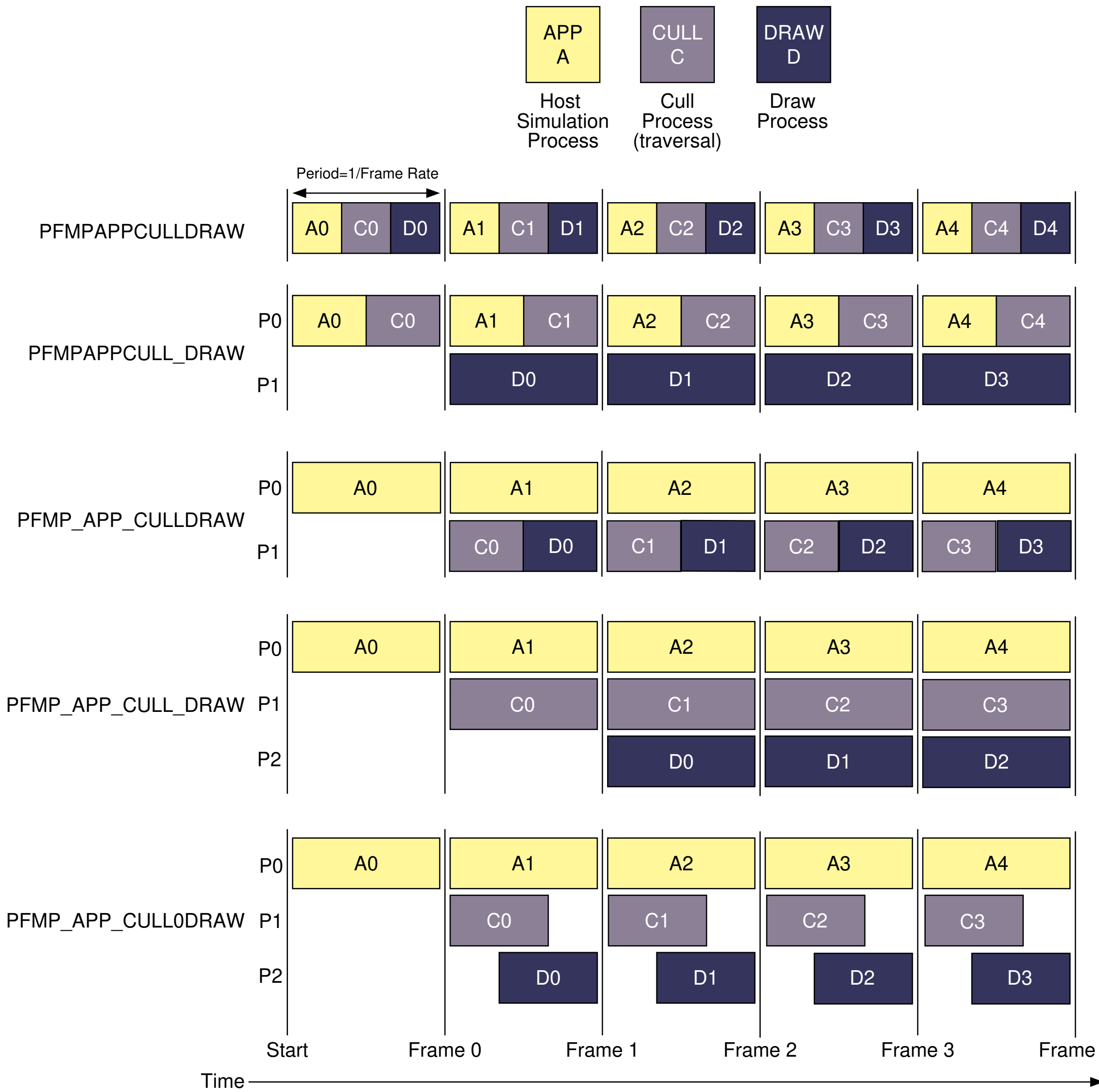

FIGURE 8. Multiprocessor Partitioning and Timing Diagram

7.1.4 Synchronizing Pipeline Stages

A pipelined architecture requires a global synchronization mechanism to coordinate the progression of data through the pipeline stages. (Or, more literally in this case, to coordinate when each pipeline stage begins processing a new buffer of data.) In a Performer rendering pipeline each stage is allocated the same amount of time.

Performer uses the V_Sync[2] interrupt as a timing signal for synchronizing the rendering pipeline. All synchronization among processors is done at intervals which are an integer multiple of the V_Sync interval. This has a number of implications: data can be passed from one stage to another only at video frame boundaries, all pipeline stages execute for a time that is an integer multiple of the V_Sync interval, the minimum latency imposed by each processor in the pipeline is one V_Sync interval, any frame overrun by a pipeline stage will delay the pipeline by a minimum of one V_Sync interval.

The Performer library call which processes use for synchronizing with V_Sync is pfSync(). For example, when a process completes its work for a frame, it calls pfSync(), which suspends the process awaiting a Unix signal. When the next V_Sync interrupt occurs, the Performer interrupt service routine sends the Unix signal which wakes up each process suspended using pfSync().

7.1.5 Data Movement

Two data movement issues must be addressed in the design of a rendering pipeline. One issue is passing the output of one stage on to the next stage. The other issue is ensuring that each pipeline stage refers to a version of the scene graph that is identical to the one referred to by the previous stage during the previous frame. This is known as ensuring "frame accurate" behavior. These issues are discussed in the following two sections.

7.1.5.1 Connecting Pipeline Stages

The APP stage passes viewpoint and gaze direction information to the CULL stage as arguments to the pfPipe() library call.

Performer supports two options for passing data from the CULL stage to the DRAW stage. One option is to create a pfDispList which references all of the data needed to draw a visual frame. In this case, the CULL stage completes the pfDispList for a frame, then passes it to the DRAW stage. As described in "Sorting Objects To Optimize Drawing Performance" on page 28, creating a pfDispList enables the CULL stage to sort the list into optimum drawing order before passing it to the DRAW stage. On the other hand, it requires that CULL stage processing be completed before DRAW processing begins.

The other option is for the CULL stage to pass visible data to the DRAW stage piece-by-piece as soon as it is identified, before continuing with the cull traversal. This enables the operation of the CULL and DRAW stages to be overlapped, but does not allow the data to be sorted prior to being drawn.

---

2. The V_Sync signal is a pulse generated by the graphics subsystem hardware at the start of each vertical blanking interval. V_Sync triggers an interrupt to the CPU subsystem which is serviced by Performer. The V_Sync interval is determined by the video frame rate, and is typically either 16.7mS (60 Hz) or 20mS (50 Hz). For more information, see the *setmon()* man page.

It is usually more efficient not to create a pfDispList when running on a uniprocessor. This allows the CPU subsystem to proceed with the CULL while the graphics subsystem hardware processes the OpenGL or IRIS GL commands needed to draw the previously identified visible geometry. The benefit of the concurrency achieved by this arrangement usually outweighs the disadvantage of not sorting the objects in the frame prior to drawing.

7.1.5.2 Ensuring Frame Accurate Behavior

In a pipeline containing three processors, the DRAW stage is working on frame N while the APP stage is working on frame N-2. This is shown in Figure 8. Because Performer utilizes immediate mode rendering, pointers to data in the scene graph are passed from one stage to the next, rather than the actual data. For example, pointers to visible geometry are passed from the CULL stage to the DRAW stage. However, the pointers received by the DRAW stage refer to the scene graph as it existed one frame earlier. If the DRAW references the same copy of the scene graph currently being modified by the APP, it can draw outdated or partially updated objects, resulting in gibberish on the screen.

Performer offers the developer a choice of two techniques to ensure that this problem never occurs. In both cases, each stage in the pipeline processes a copy of the scene graph that is identical to the one processed by the previous stage during the previous frame time. The two techniques are called pfMultibuffer and pfBuffer.

Using the pfMultibuffer technique, a pointer to a buffer containing a copy of the scene graph is passed down the pipeline. After being processed by the DRAW stage, each buffer is recycled to match the state of the APP stage’s current buffer, then started down the pipeline again. Using the pfBuffer technique, each pipeline stage has its own copy of the scene graph, and a list of changes to the scene graph is propagated down the pipeline between each frame. Which method is optimal depends on the amount of change to the scene graph which typically occurs each frame. Both methods ensure frame accurate behavior.

7.2 Intersection Pipeline

The intersection pipeline, shown in Figure 9, implements the intersection traversal described in the section “Intersection Traversal” on page 29.

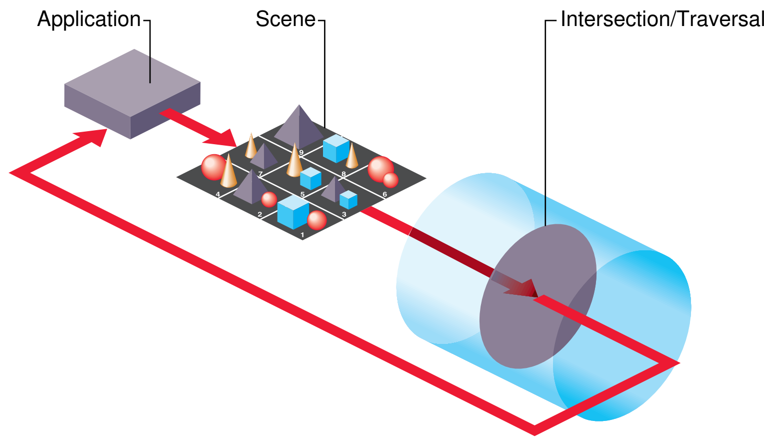

FIGURE 9. Intersection Pipeline

Because intersection traversals do not modify the database, multiple intersection pipelines can be created in parallel. Performer will assign each intersection pipeline to a separate processor, if one is available.

The intersection traversal is invoked using pfSegsIsectNode().

(Intentionally left blank.)

8.0 Managing the State of the Graphics Subsystem

Graphics subsystem state is comprised of the additional information needed to fully specify how the geometry in a pfGeoSet should appear visually. The elements of graphics subsystem state (referred to in this section as simply “state”) are listed in the section “pfGeoStates” on page 21.

8.1 Minimizing OpenGL and IRIS GL State Commands

OpenGL is an immediate-mode state machine, as is IRIS GL. This means that once a state is set, all subsequent geometry is drawn using that state until a command is received to modify the state. Consequently, drawing is order-dependent. On the other hand, the high-level data optimizations provided by Performer (e.g. view frustum culling) require pfGeodes to be order-independent. Given these conflicting requirements, there are several choices for ensuring that the graphics subsystem will always be in the appropriate state to render geometry correctly.

One choice would be to always issue the commands necessary to fully specify the graphics subsystem state prior to drawing a pfGeoSet. Since many pfGeoSets share much or all of their state specification, this would result in issuing many redundant state commands which would greatly reduce drawing performance. Performer does not use this approach.

Another choice is to group pfGeoSets which share elements of graphics subsystem state, then insert the state specification commands among the pfGeoSet groups to modify state only as needed for correct drawing. As described in Section 6.1.1.3, “Sorting Objects To Optimize Drawing Performance,” on page 28. Performer uses this technique when operating in pfDispList mode.

When the application has specified an operational mode in which no pfDispList is created, Performer tracks the current state of the graphics subsystem hardware using the pfState object. Every OpenGL or IRIS GL state command passes through pfState, which compares it with an internal table that contains the current state of the hardware. If the command will result in a state change, pfState issues the command and updates its internal table. If the command is redundant, pfState does not issue it to the graphics subsystem.

pfState enables Performer to draw pfGeoSets in any order without ever incurring the overhead of a redundant graphics subsystem state change.

8.2 Local and Global State

While it is possible to completely specify graphics subsystem state using a pfGeoSet, Performer allows the application to split state definition into two pieces: global state and local state. This split is based on the observation that many elements of graphics state apply to the entire scene graph. Examples of global state are lighting model, lights, fog, and transparency.

Because it is known to apply to the entire scene, global state can be dropped from the pfGeoSet. This improves performance by reducing the amount of state information which must be handled.

The pfState object maintains the current definition of global state, which is initialized by the database loader. Local state elements are stored within the scene graph in pfGeoStates.

Each time the cull traversal adds a pfGeoSet to a pfDispList, it evaluates the complete state associated with that pfGeoSet. To create the complete state definition, the cull traversal combines the global state and the pfGeoState as follows: any element of state that is specified in the pfGeoState takes precedence over the global state, any element of state that is not specified in the pfGeoState reverts to the global state. Once it has the complete state for each pfGeoSet, the cull traversal can reduce the set-state commands in the pfDispList to a minimum.

During the draw traversal, the pfState object combines global and local state into a complete state definition, as follows: any element of state that is specified in the pfGeoState takes precedence over the global state; any element of state that is not specified in the pfGeoState assumes the global state.

The pfState object ensures that an element of global state which is overridden by a pfGeoState does not incorrectly affect drawing of subsequent pfGeoSets. pfState tracks the global state and the current state of the graphics subsystem and issues OpenGL or IRIS GL commands to restore elements of global state, as needed.

Global state can be updated at any time by Performer commands from the application.

9.0 Database Loaders

A number of commercially available programs exist for creating visual models, including ones tailored for visual simulation, mechanical design, industrial design and entertainment applications. Typically, each modeler defines its own native file format, and some also support standard interchange formats. Instead of defining a file format, Performer defines only an in-memory representation of the visual data (i.e. the scene graph) which can be created from a wide range of existing file formats.

9.1 Existing Loaders

Performer includes an extensible set of loader programs, each of which can create a scene graph from a visual database stored in a particular file format. Using multiple loaders, files in different formats can be combined into a single scene graph. Each type of file format is identified by an extension to the file name (e.g. <filename>.iv). To load a database, an application invokes a generic Performer loader which selects the appropriate loader program for the indicated file.

Loaders for 19 different file formats are shipped with Performer 2.0. Both binary and source code are included. This simplifies development of new loaders by enabling developers to use an existing loader as their starting point for new development.

Additional Performer loaders are commercially available. Refer to the IRIS Performer 2.0 Data Sheet for a list of included loaders and information on the commercially available loaders.

9.2 Developing New Loaders

The loaders included with Performer use the pfuBuilder utility library (called the “builder”) to handle construction and triangle meshing of pfGeoSets. The loader application feeds independent, potentially concave polygons to pfuBuilder, which passes back sorted, meshed, and optimized pfGeoSets.

Database file formats which include a hierarchical scene graph map directly to the Performer scene graph. For those database formats without any hierarchy, the pfuBuilder provides spatial octree-based breakup of geometry so that even large, monolithic models can be organized into a hierarchical scene graph to improve culling and intersection traversal efficiency.

10.0 Glossary

Cull traversal – The Performer function which processes (traverses) a scene graph to determine which objects are potentially visible in the current frame. The cull traversal is an integral part of the real-time functioning of Performer. Also referred to as “CULL”.

Database – See visual database.

Database loader – A program which loads a visual database from disk, converts it to a scene graph, and stores it in memory.

Draw traversal – The Performer function which issues the OpenGL or IRIS GL commands required to render an object or frame to the display device. Also referred to as “DRAW”.

Frame – A real-time graphics frame. The portion of the scene graph that is at least partially contained within the viewing frustum for the current video frame. The geometry that is drawn during a single video frame.

GeoSet (pfGeoSet) – A group of geometric primitives of the same type. A group of geometric primitives which refer to a single pfGeoState. An element of a pfGeode.

Graphics subsystem – The hardware elements of a Silicon Graphics platform which receive OpenGL or IRIS GL commands, perform all functions needed to draw the geometry, and produce video output signals. Examples are RealityEngine^2 and Extreme. Often referred to as a graphics “pipe”, though not in this document.

Image Generator (IG) – A functional subsystem of a training simulator or other visual simulation system. A specialized computer that is used to render frames in real-time, but which does not include a general purpose CPU.

Intersection traversal – The IRIS Performer function which tests for intersections between line segments and geometry in the scene graph. Also referred to as “ISECT”.

Node – An element of a scene graph which can contain geometry, graphics subsystem state data, or traversal control data.

Pipe or Pipeline – See graphics subsystem.

Scene graph – The memory resident version of a visual database, including all geometry and graphics subsystem state information. The data portion of a real-time graphics application.The memory image created by a Performer loader.

Transformations (3D) – Actions specified by a 4x4 matrix which modify the position or orientation of geometry. Dynamic transformations are performed every frame; static transformations are performed infrequently.

Traverse – To process a scene graph. The action taken by a traversal function such as the Cull or Draw traversal. To visit the nodes in a scene graph and select the subset that is relevant to the type of traversal being conducted.

Video frame interval – The duration of each video frame; the complement of the video frame rate. During each video frame interval, the video circuitry updates the screen from the frame buffer. The video frame interval is independent of the rate at which frames are rendered into the frame buffer. Typical video frame intervals are 16.7mS (60 Hz) and 20mS (50 Hz).

Viewing frustum – A truncated pyramid that contains the geometry that is visible given an eyepoint, gaze direction, and horizontal and vertical field of view.

Visual database – The disk-based representation of the visual database, contained in one or more files. As the database is loaded into memory, it is transformed into a scene graph.

V_Sync – A pulse which occurs at the start of each vertical retrace interval. V_Sync is generated by the video circuitry, and is used to generate a CPU interrupt.