IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics

John Rohlf and James Helman
Silicon Graphics Computer Systems*

Abstract

This paper describes the design and implementation of IRIS Performer, a toolkit for visual simulation, virtual reality, and other real-time 3D graphics applications. The principal design goal is to allow application developers to more easily obtain maximal performance from 3D graphics workstations which feature multiple CPUs and support an immediate-mode rendering library. To this end, the toolkit combines a low-level library for high-performance rendering with a high-level library that implements pipelined, parallel traversals of a hierarchical scene graph. While discussing the toolkit architecture, the paper illuminates and addresses performance issues fundamental to immediate-mode graphics and coarse-grained, pipelined multiprocessing. Graphics optimizations focus on efficient data transfer to the graphics subsystem, reduction of mode settings, and restricting state inheritance. The toolkit's multiprocessing features solve the problems of how to partition work among multiple processes, how to synchronize these processes, and how to manage data in a pipelined, multiprocessing environment. The paper also discusses support for intersection detection, fixed-frame rates, run-time profiling and special effects such as geometric morphing.

Keywords: Real-time graphics, multiprocessing, visual simulation, virtual reality, interactive 3D graphics

CR Categories and Subject Descriptors: I.3.2 Graphics Systems; I.3.3 Picture/Image Generation; I.3.4 Graphics Utilities, Application Packages, Graphics Packages; I.3.7 Three-Dimensional Graphics and Realism

1 Introduction

Recently, multipurpose workstations have attained graphics performance levels that have customarily been the province of expensive, special-purpose image generators (IGs). Consequently, many visual simulation applications are migrating from IGs to graphics workstations. Additionally, the decrease in the cost/performance ratio of current-generation workstations has opened the door to non-traditional visual simulation applications such as virtual reality and location-based entertainment. These applications are often very cost-sensitive and so demand every drop of speed from the machine.

*2011 N. Shoreline Blvd., Mountain View, CA 94043 USA
jrohlf@sgi.com, jimh@sgi.com.

Proceedings of SIGGRAPH 94 (Orlando, Florida, July 24-29, 1994), In *Computer Graphics* Proceedings, Annual Conference Series, 1994, ACM SIGGRAPH, pp. 381-394.

1.1 Motivation

In our experience, application developers often have problems extracting graphics performance due to inexperience with the system and ignorance of the "new set of rules", some of them quite arcane, which must be followed for peak performance on each new graphics platform. Also, applications often forgo multiprocessing simply because the development of a multiprocessed application proves too difficult or time-consuming. The resulting single-threaded applications sequentially process all tasks, leaving an expensive graphics subsystem idle while the application carries out non-graphics processing.

Existing general purpose 3D libraries and toolkits tend to address different problems. Immediate-mode rendering libraries such as OpenGL[9], Starbase[6], and XGL provide an efficient interface to hardware, but leave the definition of geometry, scene content and multiple eye points to the application. Object-oriented toolkits such as PHIGS+[13], HOOPS, Doré[7] and IRIS Inventor[12] provide scene structures based on display lists and objects, but for most efficient rendering they retain an internal copy of the geometric data. Since applications often need access to the original data for other purposes, a second inaccessible copy inside the toolkit can substantially increase memory usage. In addition, when the application dynamically changes geometry, the retained data must be edited or rewritten. Depending on the toolkit, this can increase program complexity, degrade performance, or both.

Most importantly, none of the aforementioned toolkits addresses multiprocessing. And from our experience, retrofitting a retained-database toolkit with efficient multiprocessing support and parallel traversals proves difficult at best.

In addition to demanding maximum performance, visual simulation and virtual reality applications have real-time requirements and must run at fixed frame rates to avoid the distractions and artifacts caused by frame rate variations. To achieve reasonable performance, these applications require efficient database culling to the viewing frustum, scene complexity management through level-of-detail switching, intersection testing, and run-time profiling for application and database tuning. Toolkits written specifically for visual simulation such as VisionWorks[10] and GVS[8] partially address many of these issues, but neither offers a fully multiprocessed solution.

1.2 Purpose

The fundamental design goal of the toolkit is to provide a software development layer that delivers the greatest possible performance from the graphics workstation, freeing the application developer to concentrate on other matters. We achieve this primarily through:

- Graphics optimizations
- Multiprocessing

Another goal is to simplify the development of virtual reality and visual simulation applications by providing intrinsic support for common graphics and database operations such as multiple views, level-of-detail switching, morphing, intersection testing, picking, and run-time profiling. However, the toolkit does not provide direct support for I/O devices, audio, or motion systems since these are not directly related to the core functions of a rendering platform or a multiprocessing framework. Some applications, such as the fly-through system shown in Figure 17, have added their own device support to IRIS Performer, as have developers of toolkits for particular application domains, e.g. dVS[5] and WorldToolkit[8].

The graphics optimizations and multiprocessing features of the toolkit are targeted for workstations which support immediate-mode graphics and small-scale, symmetric, shared memory multiprocessing.

1.3 Overview

The toolkit's core consists of two libraries: **libpf** and **libpr**. **libpr** consists primarily of optimized graphics primitives as well as intersection, shared memory, and other basic functions. **libpf** is built on top of **libpr** and adds database hierarchy, multiprocessing, and real-time features. This arrangement is illustrated in Figure 1 below:

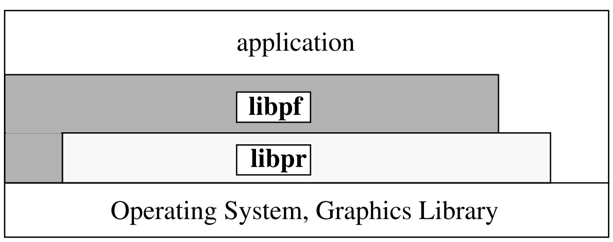

Figure 1. Library Layering

The two-library approach allows developers to choose which layer they wish to program to and also avoids "black box" limitations to flexibility by allowing an application which uses **libpf** to access the underlying **libpr** primitives. An application is also free to access the immediate-mode graphics library and operating system directly for customized rendering or control.

In keeping with our bottom-up design methodology, we discuss **libpr** first, then follow with **libpf** and finish with a description of run-time profiling utilities which facilitate performance tuning.

2 libpr - Efficient Rendering

The **libpr** library provides the high-performance foundation for IRIS Performer. Its specialized graphics primitives are designed to squeeze the highest level of performance from the graphics pipeline by efficiently managing geometry and graphics state for immediate-mode rendering. In addition, **libpr** supports intersection and shared memory utilities that facilitate a multiprocessed visual application.

2.1 pfGeoSet - Efficient Geometry Primitive

In our experience, the data structures used to represent geometry and the code which transfers that data to the graphics hardware very often make or break an immediate-mode graphics application. Scattered memory organizations can result in poor cache behavior and inefficient rendering loops can starve a fast graphics pipeline.

Immediate Mode vs. Display List Mode

The pfGeoSet's purpose is to achieve maximum immediate-mode performance for 3D geometry. In *immediate mode*, the host CPU must feed the graphics subsystem with primitive, vertex, and attribute commands. An alternative to immediate mode is *display list mode* which compiles a list of commands into a data structure that can be very efficiently transferred to the graphics subsystem. However, display list mode has some significant disadvantages that immediate mode does not have:

- A display list is a closed data structure. Geometry data must be duplicated at substantial memory penalty for database queries like intersections which require read access.
- Display lists are costly to compile. This generally requires that geometry be static. Techniques requiring vertex manipulation such as animation do not lend themselves to display list mode.

pfGeoSets utilize application-supplied arrays for attributes such as coordinates and colors, consequently avoiding these disadvantages. Applications are free to modify these arrays for dynamic effects without experiencing degraded rendering performance.

A pfGeoSet is a collection of geometric primitives of a single type defined by its:

- primitive type: points, lines, line strips, triangles, quads, or triangle strips
- attribute lists: coordinates, colors, normals, texture coordinates
- attribute bindings: per-vertex, per-primitive, overall, off.

Figure 2 illustrates a pfGeoSet consisting of two triangles with a per-primitive color binding: the first is red and the second is blue.

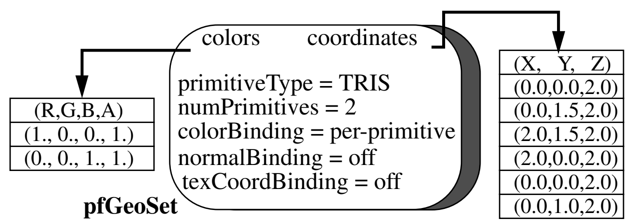

Figure 2. pfGeoSet Structure

On high-end machines in particular, care must be taken to ensure that immediate-mode data transfer is efficient or else the graphics hardware will be starved. pfGeoSets guarantee efficient data transfer by enforcing an *a priori* grouping of geometry by type that facilitates the use of customized, extremely tight rendering loops. Since all primitives within a pfGeoSet are homogeneous, a single, well-tuned rendering routine that is tailored to the specific pfGeoSet type can quickly transfer the primitives with a minimum of overhead. For example, if a pfGeoSet is a collection of triangles which have colors defined per-primitive (i.e., one color per triangle), its corresponding rendering routine doesn't waste precious if-tests determining whether or not a color should be sent down with each vertex. Over 700 of these specialized rendering routines exist (macro-generated) to handle all combinations of primitive types and attribute bindings, and all are indirectly accessed through the single pfDrawGSet() routine.

pfGeoSet Construction

Developers may find pfGeoSet construction messy and may sometimes generate pfGeoSets with sub-optimal performance, e.g., pfGeoSets with a small number of primitives may suffer from excessive setup overhead when transferring them to the graphics subsystem. Or an application may fail to use triangle meshes where possible. Connecting triangles together into a mesh can significantly reduce the amount of data transfer from the CPU to the graphics subsystem as well as the amount of processing required in the graphics hardware. Unfortunately, most databases do not utilize triangle meshing and automatic meshing algorithms are complex. To avoid these pitfalls, the pfuBuilder utility functions provide convenient meshing and performance-oriented construction of pfGeoSets. The application simply feeds independent, potentially concave polygons to a pfuBuilder which returns sorted, meshed, and optimized pfGeoSets on request.

2.2 Efficient Graphics State Management

Unlike geometry, graphics state commands do not modify the frame buffer; they do not "draw" anything, but instead configure the graphics hardware with a particular mode (e.g. shading model) or attribute (e.g. texture) that modifies the appearance of geometry. Like geometry, efficient management of graphics state is required for optimal graphics performance.

In **libpr** there are 3 ways to set graphics state, each of which offers significant performance advantages:

- Immediate mode
- Display list mode
- Encapsulated mode

In general, applications use immediate mode to set global state such as enabling fog and use encapsulated mode to specify the appearance of geometry at database creation time. Display list mode is primarily intended for use by the **libpf** library to accommodate multiprocessing.

2.2.1 pfState - Immediate Mode

The state management provided by the pfState object is useful for avoiding redundant mode changes. A pfState object maintains all current and previous graphics state in a state stack. The set of managed graphics state is that which can be modified through **libpr** routines and is a subset of that provided by the graphics library. Graphics state is partitioned into:

- Modes such as backface culling, gouraud shading, wireframe on/off
- Attributes such as texture, material parameters

Modes are generally simple integer values that are set by single commands such as pfShadeModel() while *attributes* are objects like pfTexture that encapsulate many graphics characteristics. Modes are "set" and attributes are "applied" by their immediate-mode routines: pfShadeModel() and pfApplyTex() for example.

By shadowing the state of the graphics hardware, a pfState can eliminate costly mode changes. For example, if the current shading model is FLAT then a subsequent attempt at setting a FLAT shading model should be intercepted before being sent to the graphics hardware. Avoiding mode changes is especially useful for parallelized geometry engines which become essentially single-threaded during a mode change because mode changes must be broadcast to all engines. Redundant mode changes become particularly prevalent if the database is sorted by mode (See Section 3.1.3).

2.2.2 pfDispList - Display List Mode

The primary purpose of the pfDispList is to capture an entire frame's worth of data for use in multiprocessing. It captures and buffers **libpr** rendering commands such as pfShadeModel() and pfApplyTex(). As will be discussed in Section 3.2.2, two processes can communicate via a pfDispList to increase throughput. One *producer* process fills the pfDispList and a *consumer* process draws it by traversing it and sending appropriate commands to the graphics subsystem. Throughput is enhanced because the producer process off-loads expensive database processing from the time-critical consumer process which performs immediate-mode rendering. A pfDispList may be configured as a FIFO or ring buffer for concurrent producer/consumer configurations.

A pfDispList is different from a typical display list in that it captures only references to **libpr** objects and does not contain individual vertex or primitive commands; instead the **libpr** objects themselves contain and transfer these commands. Consequently a pfDispList can be quickly built and traversed. Additionally, a pfDispList is somewhat editable (it may be reused and appended to) and can also contain references to function callbacks for user-defined rendering.

2.2.3 pfGeoState - Encapsulated Mode

The pfGeoState object provides the primary mechanism for specifying graphics state in an IRIS Performer application. It encapsulates all state modes and attributes managed by **libpr**. For example, a pfGeoState may be configured to enable lighting and reference a wood pfTexture and a shiny pfMaterial. Then after it is applied to the graphics subsystem, subsequent geometry will have the appearance of a finished wood surface. A pfGeoState can be attached to a pfGeoSet so that together they define geometry with a specific appearance.

The pfGeoState has some special features that either directly or indirectly enhance rendering performance:

Locally Set vs. Globally Inherited State

It is possible to specify every **libpr** mode and attribute of a pfGeoState, in which case the pfGeoState becomes a true graphics context that fully defines the appearance of geometry. However, a full graphics context is fairly expensive to evaluate and is almost never required. The key observation is that many state settings apply to most geometry in the database. For example: fog, lighting model, light sources and lighting enable flag are often applied to the entire scene since they are global effects by nature. Conversely, attributes such as materials and textures are likely to change often within a database. pfGeoStates support these two kinds of state by distinguishing between *globally inherited* and *locally set* state respectively. By globally inheriting state, a pfGeoState can reduce the amount of state it sets, i.e.- it becomes sparse. A sparse pfGeoState is more efficiently managed because fewer pieces of state need be examined. State is inherited simply by not specifying it. However, an important point discussed below is that state is *never* inherited between pfGeoStates. As an important result, pfGeoState rendering becomes order-independent.

Order Independence

In many immediate-mode graphics libraries, geometry inherits previously set graphics modes. As a result, rendering is order-dependent; graphics state and geometry must be organized in a specific order to produce the desired appearance. Order dependence is undesirable for high-level database manipulations such as view culling and sorting which frequently modify rendering order.

To ensure order independence, the application must either completely specify the graphics state of all geometry or it must be aware of the current graphics state and change state when necessary. The former solution seriously compromises performance if the graphics context is non-trivial and the latter is a bookkeeping nightmare.

pfGeoStates guarantee order independence for rendering as a direct consequence of not inheriting state from each other. When applied, a pfGeoState implicitly saves and restores state so that its state modifications are insulated from other pfGeoStates. Furthermore, if a global state element is modified by a pfGeoState, it will be restored for those pfGeoStates which inherit that element.

Lazy Push/Pop

If a pfGeoState explicitly pushed and popped all graphics state, significant performance would be lost due to unnecessary mode setting. Instead, a pfGeoState pushes only those global state elements that it needs to change and pops only those global state elements that it needs to inherit and that were changed by a previously-applied pfGeoState. Lazy popping eliminates useless mode changes since a mode is not restored if a pfGeoState is going to change it anyway.

2.3 Multiprocessing Support

The **libpr** library is designed to fully support, but not require, a multiprocessing environment. To this end, **libpr** provides mechanisms for creating and maintaining shared data.

2.3.1 Shared Memory

libpr provides mechanisms for sharing memory between related (forked from the same image) and unrelated processes. Allocations are reference counted to support operations such as deletion in a multiprocessed environment (See Section 3.2.3).

2.3.2 pfMultibuffer - Multibuffered Arrays

When a process needs to modify a piece of data for consumption by other processes, data must be passed or multiple copies (buffers) must be maintained. To facilitate this, **libpr** provides multiprocessing constructs such as queues and multibuffered memory. The pfMultibuffer object provides data synchronization and data exclusion for multi-stage software pipelines by managing multiple copies of a single data array. pfMultibuffer is particularly useful for dynamic and morphing geometry. A global index for each process indicates the currently active pfMultibuffer buffer, e.g., process A may be working on buffer0 while process B is simultaneously working on buffer2. By changing the global index, processes can "pass" work to each other, simulating a processing pipeline. Since buffers are recycled rather than copied, the mechanism is efficient regardless of the amount of data which changes and independent of the number of consuming processes. When the contents of a pfMultibuffer stop changing, the most recent version is copied into each buffered instance so the application does not need to write every pfMultibuffer every frame.

2.4 Database Intersection

Most applications require intersection testing for purposes such as picking and collision detection. Since the target of these tests is often the visual data already represented in pfGeoSets, **libpr** provides the ability to intersect line segments against the polygons inside a pfGeoSet, thereby avoiding expensive duplication of the database. We chose line segments as the first primitive to implement because the tests are fast and they provide the most natural expression of common queries such as picking, line-of-sight visibility, and terrain following. Many simple collision detection mechanisms can be implemented by intersecting a set of line segments that describe the swept volume of a moving object with the database. The racing car simulator shown in Figure 15 uses two segments for following the track height and four segments for detecting collisions with walls and other cars. Several line segments can be grouped into a single intersection request to reduce processing overhead. Performance may be further improved by specifying an optional bounding cylinder which encompasses all line segments and by caching plane equations for static pfGeoSets.

pfSegsIsectGSet() returns the nearest or farthest intersection along each line segment. Applications can use a *discriminator callback* to examine each intersection individually during traversal of the geometry. Discriminator callbacks can direct the intersection traversal and/or modify the intersecting line segments for fine-grained intersection control. Intersection information available to the application includes the actual triangle within the hit pfGeoSet, the intersection position and geometric normal.

3 libpf - Adds Database Hierarchy and Automated Multiprocessing to libpr

Representing a visual database involves more than just geometry and its associated graphics state. A higher-level library, **libpf**, built on top of **libpr** provides a hierarchical scene graph of nodes which organizes **libpr** geometry for improved modeling and processing efficiency.

IRIS Performer accomplishes most database processing through *traversals* of the scene graph hierarchy. Much of **libpf**'s programming interface handles traversal configuration and control. Typically, an application updates scene graph and viewing parameters for a frame and then activates one or more processing traversals. For improved performance on multiprocessor systems, **libpf** can automatically execute these traversals in parallel with little extra programming burden on the application.

3.1 Database

A *scene graph* consists of *nodes* connected in a directed, acyclic fashion. Geometry lies at the leaves of the scene graph while internal nodes support notions such as grouping, transformation, selection, and sequencing as well as special operations such as level-of-detail switching, and morphing.

3.1.1 Class Hierarchy

While both **libpf** and **libpr** libraries are object-oriented, the flat class hierarchy of **libpr** allowed us to write it in C. However, the natural expression of scene graph nodes requires a deeper class hierarchy as shown in Figure 3. Consequently **libpf** is written in C++.

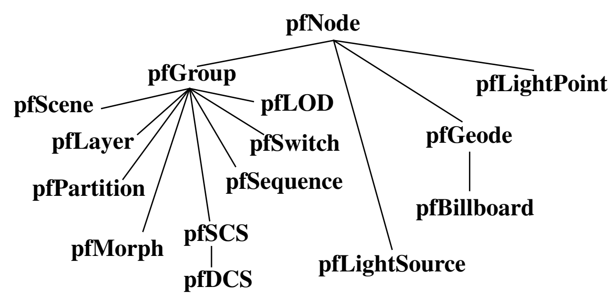

Figure 3. Node Class Hierarchy

Nodes fall into three groups: abstract, internal and leaf. pfNode is the abstract base class for all nodes and is itself derived from an internal class called pfUpdatable which creates and maintains multiple copies of the node for multiprocessing as described in Section 3.2.

The internal node types are:

- pfGroup: references pfNodes as children
- pfScene: group that roots a scene graph.
- pfSwitch: group with none, one, or all children active
- pfSequence: sequences through its children for animation effects
- pfSCS: applies an unchangeable transformation (static coordinate system) to its children
- pfDCS: applies a changeable transformation to its children (dynamic coordinate system)
- pfLayer: renders coplanar geometry, e.g. pictures on a wall.
- pfLOD: selects one or more children based on distance to eye, viewport pixel size, and field-of-view (level-of-detail).
- pfMorph: interpolates geometry, color, etc. between models
- pfPartition: spatially partitions geometry beneath it into an efficient data structure

The leaf node types are:

- pfGeode: references zero or more pfGeoSets
- pfBillboard: rotates pfGeoSets to face the eyepoint

- pfLightPoint: draws visible, but non-illuminating points of light, e.g. stars, runway lights
- pfLightSource: non-visible but illuminating light source

In a scene graph, pfGeodes typically contain most of the visual geometry. Each pfGeode references a set of **libpr** pfGeoSets. Specialized geometry is contained within pfLightPoints and pfBillboards.

3.1.2 Node Hierarchy

State Inheritance

In addition to providing organizational and instancing capability, a hierarchy of nodes (scene graph) also allows state inheritance. Within a scene graph, inheritance is strictly top-down. The absence of any left-right or bottom-up inheritance allows arbitrary pruning of the scene graph during traversal. This also facilitates parallelization of a single traversal because subgraphs of the scene graph can be traversed independently. The primary type of inherited state is 3D transformations, although user callbacks may also affect inherited state during traversals. Graphics state such as that defined by the pfGeoState primitive is *not* inherited through the scene graph. Grouping the primary specification of graphics state with leaf geometry rather than with internal nodes of the scene graph greatly facilitates tasks such as sorting by graphics mode.

Bounding Volume Hierarchy

The node hierarchy also defines a hierarchy of bounding volumes which are used to accelerate intersection and culling. Each node has a bounding sphere which encloses the node as well as any children it may have. The toolkit automatically recalculates these bounding volumes when geometry or scene graph topology changes.

All node types except pfScene retain parent lists. This allows a change to a child in a scene graph, such as a bounding volume change, to be propagated to all its ancestors in the scene graph. To eliminate redundant updates, internal state is marked using dirty bits which are propagated to the root of the scene graph so the cleaning of dirty state can be deferred until required.

3.1.3 Traversals

After the application configures the scene graph and viewing parameters, three basic traversals may process the scene graph:

- Intersection traversal (ISECT) — processes intersection requests for collision detection and terrain following.
- Culling traversal (CULL) — rejects objects outside the viewing frustum, computes level-of-detail switches, sorts geometry by modes
- Drawing traversal (DRAW) — sends geometry and graphics commands to the graphics subsystem.

TABLE 1. Traversal Characteristics

	ISECT	CULL	DRAW
Controller	pfSegSet	pfChannel	pfChannel
Global Activation	pfSegsIsectNode	pfCull	pfDraw
Modes	pfSegSet mode	pfChanTravMode	pfChanTravMode
Masks	pfNodeTravMask pfSegSet mask	pfNodeTravMask pfChanTravMask	pfNodeTravMask pfChanTravMask
Process Callback	pfIsectFunc	pfChanCullFunc	pfChanDrawFunc
Node Callbacks	pfNodeTravFuncs	pfNodeTravFuncs	pfNodeTravFuncs

Table 1 lists the **libpf** routines which define major characteristics of these 3 traversals

The default CULL and DRAW traversals are completely automatic and are triggered by pfFrame() (See Section 3.2.2). However, pfFrame() first triggers a partial traversal of the scene graph which cleans the internal state of the scene graph. Portions of the scene graph may have already been cleaned if the application called a routine which attempted to read a piece of state which was dirty.

Cull Traversal

The CULL traversal precedes the DRAW and uses many techniques to improve rendering performance by reducing load on both the DRAW traversal and on the graphics subsystem:

- Culling to the viewing frustum (pfChannel)
- Computing state specific to a pfChannel, e.g. level-of-detail
- Sorting for performance and visual quality
- Generating a simple display list (pfDispList) for the DRAW traversal

For applications with an eye point in the midst of the database, culling to the viewing frustum can reject the majority of geometry, substantially reducing the amount of data sent to the graphics subsystem. Viewing state and frustum are encapsulated by the pfChannel object. IRIS Performer supports multiple views, e.g. stereo, through multiple pfChannels which may view the same or different pfScenes.

The CULL traversal uses the hierarchical bounding volumes provided by the scene graph (See Section 3.1.2). Bounding spheres are used within the scene graph because they are fast to update, transform and test against. Axially aligned bounding boxes are used for each pfGeoSet to provide tighter bounds around the actual geometry.

During the CULL traversal the bounding sphere of each node is transformed as necessary and compared against the viewing frustum. The action taken depends on the result of the bounding volume test as follows:

- Completely outside the frustum: traversal continues without traversing any of the node's children — the node is *pruned*
- Completely inside the frustum: continue down the scene graph with no further culling tests
- Partially or potentially intersecting: continue testing and traversing down the scene graph

The ultimate output of the CULL traversal is the geometry and graphics state information to be sent to the graphics hardware. When enabled to do so, the CULL traversal first generates sorted lists of the pfGeoSets to be rendered. Each frame, these lists are sorted by graphics mode to increase rendering performance by minimizing expensive graphics mode changes such as transformation and texture changes. It is here that the order-independence offered by pfGeoStates (see Section 2.2.3) is especially useful. Next, the CULL traversal converts these sorted lists into a single pfDispList which eventually contains the entire frame. Transparent geometry is placed into the display list last, after a limited depth sort which improves both pixel-fill performance and the visual quality of the transparency. In our experience, mode sorting can significantly improve rendering throughput, sometimes more than 50%.

Draw Traversal

For each visual channel, the DRAW traverses the display list generated by its associated CULL traversal and sends commands to the graphics subsystem. The DRAW traversal differs from the CULL and ISECT traversals in that it does not involve traversing the actual scene graph. We designed the pfDispList format to be very simple, so the DRAW traversal has very little work other than issuing graphics calls. The scene graph traversal overhead is absorbed by the CULL which increases rendering throughput when multiprocessing. When not multiprocessing, we can combine the CULL and DRAW traversals into a single traversal which both culls and issues graphics commands to avoid the small overhead of pfDispList generation.

Traversal Control

Nodes have separate traversal masks for each traversal type to allow the application to "mask off" subgraphs of the scene for traversal. A node is only traversed if the logical AND of the traversal mask and the node mask is non-zero. This allows multiple databases to coexist in the same scene graph. For example, a scene graph may contain simpler geometry for collisions than for rendering in order to reduce intersection times. In this case, the DRAW traversal mask for the collision geometry and the ISECT traversal mask for the visual geometry would both be zero.

Traversal Callbacks

Traversal callbacks provide even finer control on traversals. Each node can have its own pre- and post-traversal callbacks corresponding to each traversal type. These allow the application to prune or terminate the traversal at any time. The pre-CULL callback also allows the application to specify the result of the cull test for customized culling. The application may use the pre- and post-DRAW callbacks for custom rendering using **libpr** or the underlying graphics library, or to change and restore the graphics state for a portion of the scene graph. Figure 16 shows a real-time video effects program which uses DRAW callbacks to apply video texturing.

Intersection Traversal

ISECT traversals differ from the CULL and DRAW in that they are not automatic but are directly invoked by the application. Currently, intersections are based entirely on sets of line segments. The pfSegSet structure embodies an intersection request as a group of line segments, an intersection mask, discriminator callback, and traversal mode. The traversal consists of testing the pfSegSet against the hierarchical bounding volumes in the scene graph. Intersection "hits" can be returned for pfNode bounding volumes, pfGeoSet bounding boxes and the actual geometry inside pfGeoSets. In addition to the traversal callbacks described above, intersections also provide a discriminator callback so that the application can examine each "hit" during traversal and accept or reject the intersection as well as terminate traversal. Because ISECT traversals usually require a pfSegSet to be tested against many triangles, the traversal transforms the pfSegSet into local object coordinates rather than transforming the bounding volumes and pfGeoSets into world coordinates. Since intersections do not modify the database, applications may invoke many intersection requests in parallel.

Efficiency of Bounding Volume Hierarchy

The efficiency of both CULL and ISECT traversals is largely dependent on the depth and balance of the scene graph hierarchy. For example, a scene graph arranged as a balanced octree will cull more quickly than a flat scene graph. A scene graph with poor spatial hierarchy can be rearranged as a result of database profiling as described in Section 4.2 or be imposed with an improved secondary partitioning with pfPartition as described in Section 3.1.4.

3.1.4 Performance Optimizations

pfFlatten - Eliminating Transformations

Taking a single model and placing it under multiple static transformations (e.g. trees, houses) in the scene graph is convenient for modeling, but not always necessary at run time. During rendering, a transformation typically requires the hardware matrix stack to be pushed, the new transformation applied, the geometry drawn and then the matrix stack to be popped. For small models, these matrix operations can consume as much time or more than the actual rendering. pfFlatten() can improve graphics performance at a cost in memory usage by duplicating static, instanced geometry, applying the current static transform to the geometry, and setting all static coordinate systems (pfSCSes) to the identity matrix.

pfLOD - Level of Detail

Next to view frustum culling, the most important mechanism for reducing and managing the graphics load is level-of-detail (LOD) switching. When an object is only a few pixels large on the screen, it's wasteful to render a model with a high polygon count; rather, a coarser model with a lower level-of-detail should be rendered instead. The pfLOD node uses distance to the eye point, field-of-view, viewport pixel size, and graphics stress (see Section 3.3.2) to select among models of varying geometric complexity.

To make LOD changes as inconspicuous as possible, the pfLOD node can gradually fade between two models when switching. A drawback to fade LOD is that it requires rendering both models during the transition which temporarily increases the graphics load. An alternative LOD mechanism provided by the pfMorph node is described in Section 3.1.5 and can avoid this penalty by smoothly migrating vertices from one LOD to another.

pfSequence - Animation Sequences

Most high-quality animation requires moving vertices every frame. But for the highest performance with minimal CPU loading, most real-time applications make extensive use of precomputed animation sequences such as a sequence of textures to simulate a flickering torch. The pfSequence node supports this by automatically sequencing through its children. Each child is assigned a period of time, rather than a number of frames, during which it should be displayed so that the sequence is immune to frame rate variations. An example of pfSequence use is the dragon seen in the background of Figure 13.

pfBillboard - Billboarded Geometry

Rotating geometry, usually a single textured polygon, so that it always faces the eye is a trick from visual simulation used for axially and radially symmetric objects such as trees, clouds and special effects such as smoke or fire. Using a billboarded polygon instead of a full three-dimensional model reduces both geometry and pixel fill demands on the graphics pipe. A pfBillboard can be constrained to rotate about an axis or a point. The trees and lamp posts in Figure 14 are examples of pfBillboards.

pfPartition - Spatial Data Structure

IRIS Performer relies on the hierarchical bounding volumes of a scene graph to accelerate intersection and culling traversals. However, a user-constructed scene graph may exhibit poor spatial arrangement, obviating the benefits of hierarchical bounding volumes. In this case a specialized spatial data structure imposed on the default scene graph can provide much higher performance, particularly for intersections. The pfPartition group node analyzes geometry underneath it at database load time and partitions pfGeoSets into a 2D grid with multiple membership. During the intersection traversal, line segments in a pfSegSet are scan converted onto the grid to quickly determine which pfGeoSets need to be tested against. Other types of spatial data structures may be added in the future.

3.1.5 Special Features

pfMorph - Morphing

The pfMorph node provides a mechanism for interpolating geometry between many sources. A pfMorph takes a set of input arrays and weights and places the linear combination of the input arrays into an output array. Typically, the morphed arrays are the vertex, color, normal or texture coordinate arrays of a pfGeoSet in the scene graph beneath the pfMorph node. The two main applications are for continuously varying animated geometry such as the head of the creature in the foreground of Figure 13 and for continuous LOD switching [3]. The latter allows nearly invisible LOD transitions and can be more efficient than fade LOD if the cost of morphing is small compared to the cost of drawing two models during a fade transition.

3.1.6 Database Importation

IRIS Performer is strictly a runtime programming interface with an in-memory scene representation and currently has no database file format. An application calls toolkit routines to create and assemble a scene graph from various elements such as pfNodes, pfGeoSets and pfGeoStates. Because the task of creating pfGeoSets can be tedious, a utility library built on top of the toolkit provides routines (pfuBuilder) to simplify the construction and triangle meshing of pfGeoSets. Using these, database loaders have been written for various database formats including Autodesk DXF, Wavefront OBJ, Software Systems FLT, Coryphaeus DWB, and LightScape LSB. Database formats with a hierarchical scene graph and visual simulation extensions (e.g. level-of-detail, billboards) map directly to the toolkit scene graph. For those database formats without any hierarchy, the utility library provides spatial octree-based breakup of geometry (pfuBreakup) so that even large, monolithic models can be organized into a scene graph for efficient culling and intersecting.

3.2 Multiprocessing

A fundamental design criterion of the toolkit was to improve performance through multiprocessing while hiding the programming complexities that multiprocessing creates. This section describes our solutions to the following multiprocessing problems:

- How to partition work among multiple processes
- How to synchronize process execution
- How to manage data in a pipelined, multiprocessing environment

3.2.1 Pipelined Multiprocessing

IRIS Performer employs a *coarse-grained*, *pipelined*, multiprocessing scheme, i.e., a relatively small number of processes work concurrently on different stages of one or more processing pipelines. This configuration favors workstations with a relatively small number of processors (tens) over massively parallel systems (thousands). The partitioning of work into multiple processes is based on *processing stages*. A processing stage is a discrete section of a processing pipeline and encompasses specific types of work. Processing stages are tightly coupled to the scene graph traversals described in Section 3.1.3. The ISECT, CULL, and DRAW processing stages consist of zero or more intersection, culling, and drawing traversals respectively in addition to application-specific processing that is accessed through function callbacks. An additional processing stage, the APP, consists primarily of application code as well as database, viewpoint, and system modifications made through toolkit routines. Together, these four stages define two kinds of processing pipelines:

- rendering pipeline: APP $\rightarrow$ CULL $\rightarrow$ DRAW
- intersection pipeline: APP $\rightarrow$ ISECT

pfPipe - Rendering Pipeline

The APP stage is the head of all pipelines and controls their execution. A rendering pipeline consists of the CULL and DRAW stages and is encapsulated by the pfPipe primitive. An application may use one or more parallel pfPipes that each renders zero or more viewpoints into a single graphics window. The multipipe feature is provided for machines with multiple graphics subsystems and includes support for time-multiplexing the output of multiple hardware renderers to a single display. The intersection pipeline consists of the ISECT stage. Only one intersection pipeline is supported.

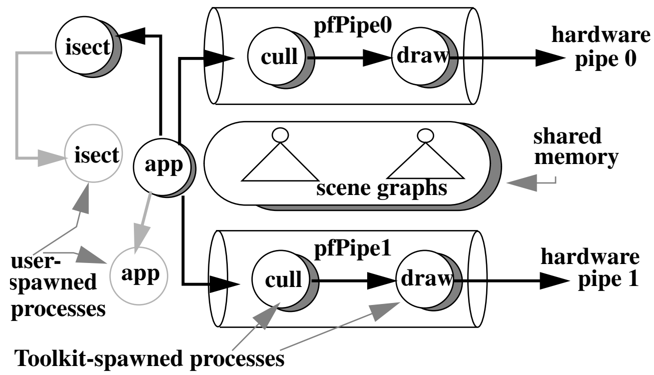

Figure 4. Multiprocessing Multipipe Configuration

Multiprocess Partitioning - pfMultiprocess

Multiprocessing in IRIS Performer is achieved by splitting the rendering and intersection pipelines at stage boundaries into multiple processes. For example, the APP and CULL stages may be combined into a single process while the DRAW stage is split into a separate process, resulting in a 2-process configuration which is suitable for a 2-processor machine. The application specifies this partitioning through pfMultiprocess(), allowing applications to choose a process partitioning based on the number of available CPUs. Figure 4 illustrates a processing configuration consisting of two rendering pipelines and an intersection pipeline where each stage has been split into a separate process. Figure 5 illustrates different multiprocess partitionings of the rendering pipeline that range from 1 to 3 processes.

Multiprocessing With Shared vs. Non-shared Address Space

All pipelined processes are created by pfConfig() using the fork() mechanism. We chose fork() over mechanisms which allow a fully shared virtual address space so we could selectively share memory and support multiple graphics pipes, since not all immediate-mode graphics libraries allow multiple rendering contexts within a single virtual address space. Synchronization for all processes created by pfConfig() is handled internally.

Additional Multiprocessing

Additional multiprocessing is easily acquired if the application itself creates extra processes. The ISECT and APP stages particularly lend themselves to this kind of multiprocessing. For example, multiple ISECT processes may concurrently execute calls to pfSegsIsectNode() which intersects a set of line segments with a scene graph (see Section 3.1.3). However, synchronization for these processes is the responsibility of the application. The stippled circles in Figure 4 depict these user-spawned processes.

3.2.2 Process Synchronization

Process synchronization defines the execution order of multiple processes. It is responsible for enforcing periods of mutual exclusion between processes and for ensuring concurrent execution of processes. Most process synchronization in the toolkit is achieved through well-known mechanisms such as semaphores and locks.

Throughput vs. Latency

IRIS Performer enforces pipelined synchronization of processes created by pfConfig(). Pipelined multiprocessing trades increased throughput for increased latency. *Rendering latency* is defined as the time elapsed from viewpoint specification until the display is completed for that viewpoint. *Rendering throughput* is defined as the amount of geometry processed in unit time. The size of the throughput vs. latency trade-off is dictated by the number of processes in the pipeline (its *depth*) and increases with process count. Pipeline depth is configurable and can range from 1 to 3. For example, a configuration combining the APP and CULL into a single process and separating the DRAW will generate a rendering pipeline whose depth is 2. If all pipeline stages are well-utilized, performance can be increased over the single-processed case by a factor equal to the pipeline depth.

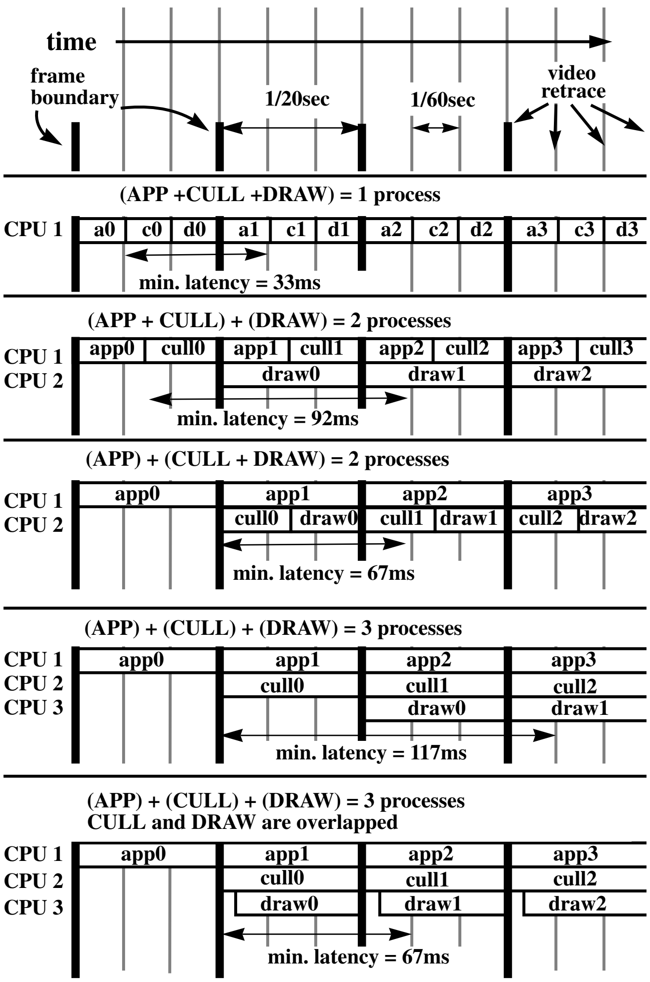

Figure 5. Multiprocess Partitioning and Timing Diagram

Figure 5 illustrates timing diagrams for different multiprocess configurations ranging from 1 to 3 processes that are running at 20Hz. Boxes represent the execution time of individual stages and each row of boxes corresponds to a single process. Thus, multiple rows of timing boxes illustrate parallel execution of pipeline stages. The text inside the boxes specify the stage or stages that the process handles while the numbers indicate the frame that the process is currently working on. Notice how the amount of time available to each stage (throughput) increases as the number of processes (pipeline depth) increases.

Frame Control

The toolkit typically synchronizes the application to a user-specified frame rate, e.g. 30Hz. This frame rate defines a series of frame boundaries that demarcate the beginning and ending of a frame. The APP stage is responsible for synchronizing to the specified frame rate and for triggering all processing pipelines once per frame by calling pfSync() and pfFrame() respectively.

pfSync() suspends the calling process until the next frame boundary and is discussed in more detail in Section 3.3.1. pfFrame() indicates that all rendering and intersection pipelines should begin processing a new frame. If a pipeline stage is not ready to begin processing a new frame because the processing time for the previous frame exceeded the allotted frame time, the stage has *frame-extended.* In this event, pfFrame() does not block but returns control to the application. If the APP process frame-extends, then pfFrame() is not called often enough and the update rate drops even if the rendering pipeline can keep up. For this reason, application processing *must* be kept to within a frame time.

Improving Latency

Certain applications like "man-in-the-loop" flight simulation and virtual reality applications utilizing a head-tracked display require very low latencies [14]. The latencies listed in Figure 5 are timed from the end of the APP processing until video scanout of the last pixel. To ensure this minimal latency even in cases when the APP takes less than its full allotment of time, the toolkit allows latency-critical updates such as the viewpoint to be made just before kicking off the CULL traversal with pfFrame(). Figure 6 depicts a close-up view of how pfSync() and pfFrame() work together to synchronize process execution. Latency-critical updates are made in the shaded portions of the APP processing time and may reduce throughput by delaying the triggering of the processing pipelines.

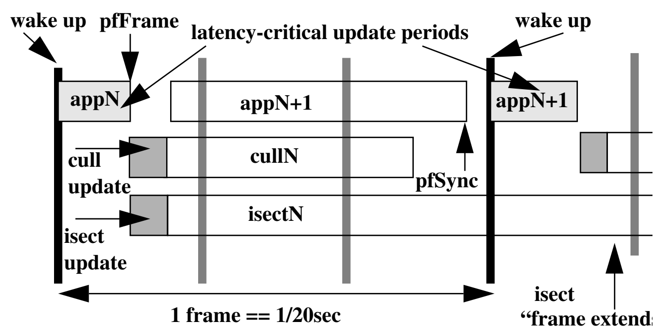

Figure 6. pfSync and pfFrame

The following pseudo-code fragment illustrates the use of pfSync() and pfFrame() in a typical simulation loop:

```
while(!Done)
{
    updateSim();     /* Make non-latency-critical updates */
    pfSync();        /* Sleep until next frame boundary */
    updateView();    /* Read input devices and update eyepoint */
    pfFrame();       /* Trigger new frame */
}
```

A special multiprocessing mode illustrated by the last timing diagram of Figure 5 eliminates an entire frame of latency by overlapping the CULL and DRAW processes that are working on the same frame. The two processes communicate via a FIFO which stalls a process on empty and full conditions. Although the DRAW has to wait for the CULL to begin filling the FIFO and will stall if it is faster than the CULL, in practice neither of these drawbacks are significant. In this overlapped case, latency is reduced to a single frame, generally the lowest possible. When CULL and DRAW are not overlapped, latency can still be reduced to a single frame by culling to a slightly larger viewing volume and sampling a new viewing position just before drawing.

A lower latency alternative to pipelined multiprocessing would be a single, multithreaded scene graph traversal. We chose against this method due to the *much* higher complexity and overhead arising from the necessary fine-grained synchronization. Also, the threads would have to be single-threaded when the application makes random access modifications to the database and when rendering, if the graphics pipeline does not allow multiple writers.

Pipeline Bottlenecks

Ideally, each process in the pipeline takes exactly one frame time to complete its work. This situation indicates a balanced pipeline that is getting maximum utilization of its processors and is the one depicted in Figure 5. An out-of-balance situation arises when a particular process takes longer than all other processes in the pipeline and becomes a bottleneck. In most graphics intensive applications, the process handling the DRAW stage is the bottleneck. In this case, draw times can be reduced through the stress management techniques described in Section 3.3.2. If the bottleneck is due to the CULL stage, times can be reduced by disabling one or more culling modes. Bottlenecks due to the APP stage are largely the responsibility of the application.

Process Callbacks

By default, IRIS Performer performs all rendering processing when triggered by pfFrame(); culling and drawing functions are carried out in "black box" fashion. *Process callbacks* provide the user with the ability to execute custom code both before and after default processing, and to execute the code in the appropriate process when multiprocessing.

Process callbacks are provided for the ISECT, CULL, and DRAW stages. Default processing for these stages is triggered by pfSegsIsectNode, pfCull, and pfDraw respectively. If a callback is specified, default processing is disabled and must be explicitly triggered by the callback. This arrangement allows the user to "wrap" default processing with custom code, allowing save/restore, before/after, and multipass rendering methods which use techniques such as projective textures [11]. Figure 12 is from an application which uses multipass renderings with projective textures to simulate a spotlight with real-time shadows. In practice, the DRAW callback is often used for 2D graphics, textual annotations and specialized rendering that requires the full flexibility of the underlying graphics library. A typical DRAW callback is illustrated below:

```
void
drawCallback(pfChannel *chan, void *data)
{
    clearFrameBuffer();
    pfDraw();
    drawSpecialStuff();
}
```

3.2.3 Data Management

Three problems plague data management in a pipelined multiprocessing environment:

1) Data visibility. Processes need to share data.
2) Data exclusion. A process must not modify data while other processes are simultaneously reading and/or writing it.
3) Data synchronization. Data modifications must be propagated down processing pipelines in a "frame-accurate" fashion.

1) is handled by the shared memory mechanisms described in Section 2.3.1. **2)** can be handled with hardware spin locks but fine-grain locking becomes expensive and as we shall see, the data exclusion problem is solved by the solution to **3)**. First, let us examine the data synchronization problem more closely.

Data Synchronization

In the toolkit's multiprocessing pipelines, multiple processes work on different frames at the same time. For example, the APP process works on frame 33 while the DRAW is on frame 31. Suppose a single matrix in shared memory represents the position of a database model. If the APP process updates this matrix while the DRAW process is sending it to the graphics hardware, the matrix might be partially updated when sent to the graphics, resulting in an unintended combination of two matrices. Alternatively, the model might be drawn at the position it should have at frame 33, rather than frame 31. In this case we say that the matrix update is not *frame-accurate* since it does not affect the displayed model at the appropriate time.

Note that hardware pipelines exemplified by graphics subsystems such as RealityEngine[1] solve the data synchronization problem by copying the entire database down through the pipeline. While wide, fast data paths make this practical for hardware pipelines, software pipelines do not have this luxury and require another approach.

Multibuffering

We solve the problem of data exclusion and data synchronization with a technique called *multibuffering*. Multibuffering employs multiple copies of data structures known as pfUpdatables (or updatables) that are logically partitioned into buffers known as pfBuffers. All **libpf** objects including pfNodes are pfUpdatables so that each pfBuffer contains a full copy of the scene graph. A pfBuffer is associated with a single process and that process may access only those pfUpdatables in its pfBuffer, thereby solving the data exclusion problem.

Modifications made to pfUpdatables by the APP process are recorded in an update list. Each frame these updates are applied to all downstream pfUpdatables so the updates propagate down all pipelines in frame-accurate fashion, thereby solving the data synchronization problem. Propagating only database modifications significantly reduces the amount of data that flows through the processing pipelines.

This update-based multibuffering mechanism is most useful when making sparse modifications to largely static data structures. This is in contrast to the pointer-switching type of multibuffering provided by pfMultibuffer (see Section 2.3.2) which is most suitable for data structures with large changes, such as vertex arrays used in morphing. In this case, swapping pointers is much more efficient than copying large amounts of data.

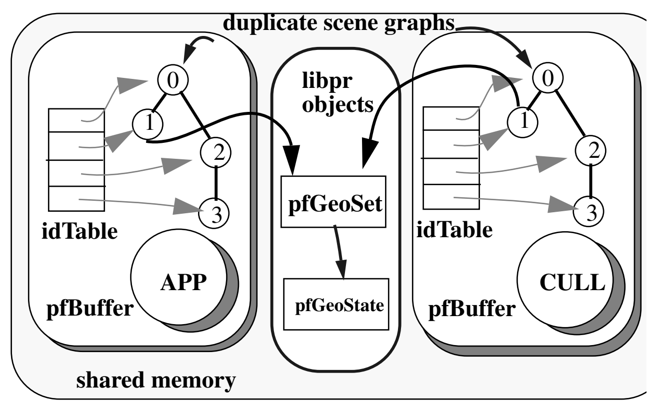

Figure 7. Multibuffering of a Scene Graph for APP and CULL

pfBuffer and pfUpdatable

In addition to forking multiple processes, pfConfig() creates and associates a pfBuffer with each process (except the DRAW as is discussed below). Each pfBuffer has an id table which associates the address of a pfUpdatable with its id. When created, a pfUpdatable is assigned a unique integer id and is added to the id table of the creating process' pfBuffer. Then during the period when updates are exchanged, corresponding pfUpdatables are created in all downstream pfBuffers. Figure 7 depicts the referencing of two copies of the scene graph (one each for the APP and CULL processes) through the pfBuffer's idTable.

Selective Multibuffering

The net result is that N "images" of each pfUpdatable are created: one for each pfBuffer in use. At first glance this may seem to be an extravagant use of memory. However, only pfUpdatables are multibuffered and only **libpf** objects are pfUpdatables, e.g. pfNodes, pfChannels. Thus all **libpr** primitives such as pfGeoSets and pfGeoStates are *not* multibuffered and do not suffer the memory penalty that multibuffering introduces. This design decision relies on the following assumptions:

- Geometric primitives like pfGeoSets and pfGeoStates represent the vast majority of database memory. Thus, duplicating only the scene graph skeleton does not drastically increase memory usage.
- Most geometry is static and does not require the frame-accurate behavior provided by multibuffering. (In Figure 7 the pfUpdatable numbered "1" is a pfGeode that references a non-multibuffered pfGeoSet.)

Although the first assumption has proven reasonable in most circumstances, we are currently exploring a "copy-on-write" extension to the multibuffering mechanism which would create extra copies only when an updatable is modified. The second assumption however, is restrictive in applications which use sophisticated morphing techniques like continuous terrain level-of-detail that require vertex-level manipulations of geometry [3]. Without multibuffering, the APP process may modify geometry at the same time the DRAW is sending the geometry to the graphics subsystem, resulting in cracks between adjacent polygons. To solve this problem we have offered a solution with the pfMultibuffer primitive described in Section 2.3.2.

Data Exclusion Revisited

In addition to frame-accurate behavior, multibuffering provides data exclusion which is essential to robust multiprocessing. Since each process is guaranteed exclusive access to updatables in its pfBuffer, it need not worry for example, that the APP process has removed a node from the scene graph. Otherwise, the process might collide with the modification and dereference a bad pointer with disastrous results.

Update List

An *update* consists of an updatable id and another integer id which defines what has changed. For example an update of [31, 12] might mean "update the transform of the pfDCS whose id is 31." Recording updates by reference has significant advantages over recording updates by value, which in the above example would mean copying the transformation matrix into the update list:

- Updates are homogeneous, thereby simplifying code and data structures
- Updates are small, resulting in quick recording and memory conservation
- Updates have a unique key which allow them to be efficiently managed by a hash table. Specifically, duplicate updates are discarded, keeping the update list from growing without bound.

The primary disadvantage of this update form is that it requires blocking the upstream process during the update period described below.

In order to provide frame-accurate behavior, updates must propagate in an orderly fashion down all processing pipelines. This propagation period occurs during pfFrame(). At this point all processes downstream of the APP (all CULL and ISECT processes) traverse the update list generated by the APP process and update their pfUpdatables. Each update consists of copying a portion of a pfUpdatable in the upstream pfBuffer into the corresponding pfUpdatable in the downstream pfBuffer. For the pfDCS example mentioned above, we would copy only the transformation matrix between pfDCS copies. At the end of the update period, all pfUpdatables in the downstream pfBuffer are identical to those in the upstream pfBuffer.

During the update period, the upstream process (the APP) must be blocked so that it cannot modify updatables in its buffer and possibly corrupt the update data exchange; we must ensure data exclusion. This update period is illustrated in Figure 6 as the shaded portions of the CULL and ISECT processes.

Figure 8 illustrates an APP feeding two pipelines: one intersection and one rendering pipeline. In this case there are three pfBuffers - one each for the APP, ISECT, and CULL processes.

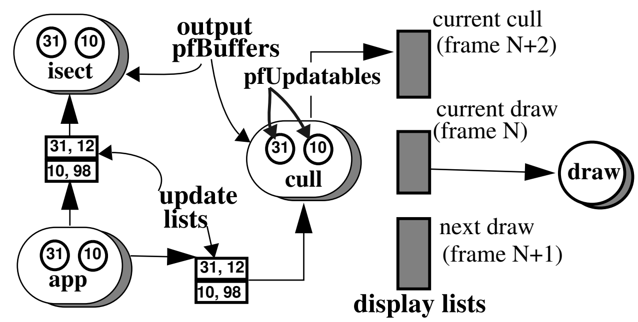

Figure 8. Interprocess Communication for Processing Pipelines Using Update Lists and Display Lists

Pipeline Frame Extension

The APP pfBuffer maintains an update list for each processing pipeline and appends all updates to all update lists. If a downstream pipeline is not ready to accept the update list when pfFrame() is called because it has frame-extended, the APP does not block but continues with the next frame. In this case, the update list corresponding to the frame-extended pipeline is not reset so that further updates are appended to the list and previous updates are not lost; they will be consumed later when the pipeline is ready. If the APP is feeding multiple pipelines, all ready pipelines update themselves in parallel.

Cull/Draw Communication

Note however that the DRAW process in Figure 8 does not have a pfBuffer and uses a different communication mechanism with the upstream CULL process. This is not precluded by the pfBuffer/pfUpdatable mechanism but was chosen to reduce memory requirements and performance degradation. When the CULL and DRAW stages are in separate processes, the CULL process traverses the scene graph and renders visible geometry into a **libpr** display list (See pfDispList in Section 2.2.2). This is very important because it off-loads scene graph traversal overhead from the time-critical DRAW process. However, this means that there is no need for a scene graph in the DRAW process. Also, maintaining a pfBuffer in the DRAW process would require an update period that would steal precious drawing time.

As illustrated in Figure 8, the CULL and DRAW communicate via three display lists. In a perfectly balanced pipeline, only two display lists would be required — the classic double-buffered configuration. However, both CULL and DRAW processes may frame-extend. As a result, a third display list is required to keep the non-extending process from waiting until the extending process is finished with its display list.

pfDelete - Object Deletion

Deletion of a hierarchical scene or subgraph that supports instancing can be tricky. Care must be taken to ensure that an object's memory is not freed until all references to it are removed. To do otherwise would open the possibility of corrupted memory and ungraceful program cessation. IRIS Performer employs a reference counting scheme to avoid such results.

Whenever an "attachment" is made between two objects, the reference count of the "attachee" is incremented by one. Reference count modifications are locked to ensure data exclusion between multiple processes. pfDelete() deletes objects whose reference counts are non-positive and follows all reference chains, deleting objects until it reaches one whose reference count is greater than zero. The reference count of a pfNode is simply the number of its parents. User-allocated memory such as the attribute arrays of pfGeoSets (See Section 2.1) are reference-counted if the memory is allocated by **libpr** routines since they maintain internal reference counts.

Multiprocessed Delete

Unfortunately, multiprocessing adds another dimension to reference counting. Non-multibuffered objects such as pfGeoSets are "referenced" by the processes which are accessing them. For example, the ISECT and DRAW processes may be concurrently intersecting with, and rendering a given pfGeoSet. Consequently, a simple reference counting scheme is inadequate.

One possibility would be for processes to reference/dereference objects as they need them. This is unacceptable from a performance standpoint since locks are not free and the number of objects needing locking is large. IRIS Performer's solution takes advantage of its pipelined configuration. An object is not immediately deleted; rather, a frame-stamped deletion request is added to a special list. Meanwhile, the back ends of all pipelines (ISECT and DRAW processes) record the frame count of their most-recently-completed frame. Then when pfFrame() is called, each deletion request on the list is examined. If its frame stamp is less than the frame counts of all pipelines, the deletion request is safely carried out since all pipelines have flushed themselves of the object.

3.3 Achieving Real-Time

3.3.1 Achieving Real-time Synchronization

Real-time behavior is often required of graphics applications, both for human and hardware (sensor) perception. Real-time in this context implies more than a reasonable frame rate. Equally important is a *fixed frame rate* which ensures a solid, consistent update rate without glitches or hiccups. In fact, many visual simulation applications sacrifice peak frame rates for a fixed frame rate.

The first step in achieving real-time behavior is accessing a timer that runs at wall-clock time, i.e., it runs at the same rate as the clock on your office wall. Since the graphics update rate is restricted to integral fractions of the video refresh rate, the video clock provides a natural real-time clock for synchronizing a graphics application.

pfVClockSync - Synchronizing to Video Retrace

The kernel maintains a video retrace counter and also provides a synchronizing feature that is accessed through the pfVClockSync() call. This routine takes two arguments, [*interval, offset*] that together specify the frame synchronization boundary. Put arithmetically, pfVClockSync() puts the calling process to sleep until the video retrace count modulo the *interval* equals the *offset*. For example, if pfVClockSync() is called with arguments of [3, 0] when the current video clock is 658, the process will sleep until the video clock is 660.

An application specifies its desired fixed frame rate and synchronizes the APP process to that rate by invoking pfSync() which calls pfVClockSync() to sleep until the next frame boundary. Note that pfSync() alone does not guarantee a fixed frame rate. First, the APP cannot take longer than a frame time because it would then synchronize to an integral multiple of the desired field rate such as 30Hz dropping to 15 Hz or even 10Hz. Second, the processing pipelines must be able to complete their work within a frame time as is discussed in more detail below.

3.3.2 Achieving A Fixed-Frame Rate

Once synchronization to wall-clock time is achieved, the next step in attaining real-time behavior is to ensure a fixed frame rate. Many things can compromise a fixed frame rate on a multiuser workstation:

1) Graphics context switching
2) Process context switching
3) Process frame extension (e.g. APP, CULL extensions)
4) Graphics pipeline frame extension (DRAW extension)

1) can be remedied by ensuring that only the application of interest is running: no clocks or performance meters allowed. **2)** may be solved by running the application with super-user privileges and using OS commands to isolate and restrict processors. **3)** is more difficult to solve and requires rearranging database hierarchies, disabling of modes, and further multiprocessing to unload the burdened process(es). **4)** is often the most prevalent enemy to a fixed frame rate and it is that which we address in this section.

Graphics pipelines have hard limits on the amount of geometry they can process in a given time. Ideally, the throughput of a graphics pipeline is always enough to render the desired amount of scene geometry in the desired amount of time. In this case a fixed frame rate is easily achieved. However, most scenes have varying geometric complexities due to varying scene density and/or moving models which may come into view. If a frame rate is chosen such that the view of highest complexity may be rendered within a frame time, then the expensive graphics hardware will be underutilized for less complex scenes. On the other hand, if a higher frame rate is chosen, complex scenes will take longer to render than the allowed frame time and distracting visual anomalies, technically referred to as "hiccups", will occur. Consequently, many applications choose a frame rate that can handle the average scene and rely on other mechanisms to artificially reduce more complex scenes so that they can be rendered within a frame time.

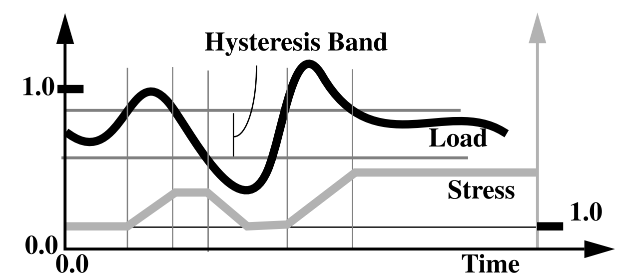

Figure 9. Stress Feedback Filtering

Stress management is the technique used to reduce scene complexity that relies on the level-of-detail mechanism described in Section 3.1.4. When the system is in stress, LODs are artificially reduced; coarser than normal models are chosen, so that overall graphics load is reduced. Stress is based on *load*, the fraction of a frame time taken to render a frame, and increases as load exceeds a user-specified threshold. The load for frame N is used in conjunction with user-specified parameters to define the stress value for frame N+1, thus defining a feedback network. As discussed in [4], this method works reasonably well for relatively constant scene densities but suffers because the stress is always a frame late and can exhibit oscillatory behavior. As illustrated in Figure 9, a hysteresis band can reduce stress oscillations but a more sophisticated stress management technique such as that described in [4] has better characteristics.

3.3.3 Overload Management

While stress management seeks to fit DRAW processing into a frame time, *overload management* dictates what happens when stress management has failed and the DRAW exceeds a frame time — it has frame extended. The application may choose differing overload behavior by selecting the *phase* of the DRAW process. Phase dictates the type of synchronization used by the DRAW process: if the phase is *locked*, the DRAW process is guaranteed to begin only on a frame boundary. Thus if the DRAW takes just slightly longer than a frame time, the aggregate frame rate will drop in half. If the phase is *float*ing, a frame-extended DRAW will start to draw again as soon as it can (at the next vertical retrace) and try to "catch up", relying on stress management to reduce scene complexity. In practice, floating phase is used more often than locked phase since it does not sacrifice an entire frame time if the DRAW takes just slightly longer than a frame. However, locked phase offers deterministic latencies and can produce a steadier frame rate.

4 Run-Time Profiling

Without proper profiling and diagnostic utilities, it is difficult to ascertain the performance of a given application. "Is it running as fast as it can go?" is the most pertinent question. To answer, the developer must be able to answer other questions concerning potential bottleneck areas:

- CPU processing, e.g., is the APP taking longer than the DRAW?
- CPU to graphics transfer, e.g., is the bus saturated or is the DRAW suffering from overhead due to small pfGeoSets?
- Geometry transform, e.g., are excessive mode changes thrashing the Geometry Engines? Are my triangle strips too short?
- Geometry fill, e.g., is the pixel depth complexity too high?

To further complicate matters, bottlenecks change and shift as the visual scene changes, making them moving targets for the tuner.

To aid application and database tuning, IRIS Performer provides extensive profiling information that is collected at run-time and may be graphically displayed for easy comprehension. Run-time collection provides a display of up-to-date information as you fly through the database, facilitating an interactive and time-saving approach to tuning. Figure 10 is the statistics display for the scene in Figure 14 and shows both process and database statistics measurements that are examined in the following sections.

4.1 Process Statistics

Due to the concurrent, time-dependent nature of multiprocessing, it is often difficult to understand the behavior of a multiprocessed application. IRIS Performer records the times spent by each processing stage and displays the results in a timing diagram which quickly exposes any bottlenecks. In Figure 10, the upper portion of the display defines a timing diagram analogous to those in Figure 5. Vertical lines indicate vertical retrace and frame boundaries. Horizontal lines indicate the processing times for different stages and their color indicates the stage's frame count.

Example Analysis

From Figure 10 we see that the application is configured as 4 processes, one each for ISECT, APP, CULL and DRAW, which all run in parallel. Additionally, the processing times for CULL and DRAW are roughly equivalent and occupy most of a frame time indicating that 30Hz is a reasonable frame rate and load balancing is good. (Note that the time required to draw the statistics display itself pushes the draw time over 1/30 sec.) However, the APP and ISECT stages take little time so we could free a CPU by combining these two stages into a single process.

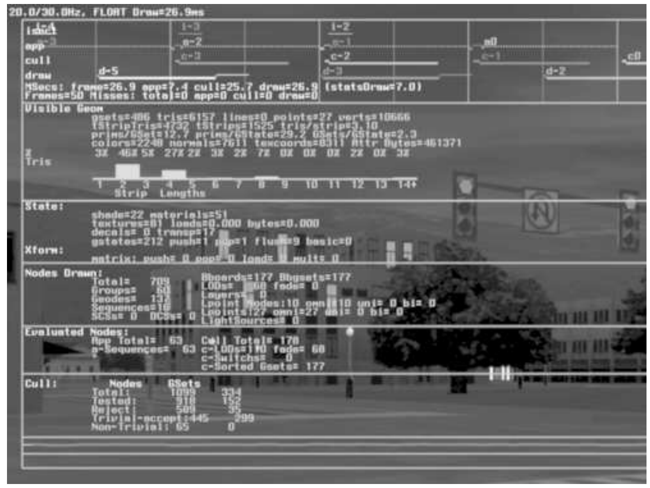

Figure 10. Display of Process and Database Statistics

4.2 Database Statistics

Although the toolkit strives to achieve maximum performance with a given database, a significant amount of performance gain may lurk within the database itself. For example, a scene graph without hierarchy will suffer from poor intersection and culling performance, both of which rely on hierarchical bounding volumes to accelerate processing. Also, a pfGeoSet which contains few triangles will suffer from overhead in pfDrawGSet(). These problems and more can be easily inferred from the statistics display of Figure 10.

Example Analysis

The ratios of primitives to pfGeoState (12.7) and pfGeoSets to pfGeoState (2.3) are reasonably high, indicating that pfGeoSet and pfGeoState overhead is not likely a problem. However, the average number of triangles per strip is low at 3.1 which indicates that the hardware geometry processing stage may be a bottleneck. This fragmentation of the database is likely due to the large number of textures (81) since a strip cannot span multiple textures.

Figure 11. Profiling Display Depicting Pixel Depth Complexity

Although Figure 10 reveals much about the database, it says nothing about the pixel fill bottleneck which is the most important one for the majority of full-screen applications. The toolkit provides a special mode for visualizing pixel *depth complexity*, the number of times each pixel is touched. Figure 11 is a false-color visualization of the depth complexity for the scene of Figure 14. Depth complexities of up to 7 are represented by colors of increasing brightness (some areas have complexities > 7 and wrap). Additionally, the total number of pixels rendered and the average depth complexity is displayed. All of these statistics are computed and displayed at a run-time, albeit at a reduced frame rate.

4.3 Future Work

Our design approach has been to focus on the performance and structure of the toolkit's rendering and multiprocessing core. Because of this, we believe the toolkit provides a good foundation for additional functionality.

Database Paging

Many applications use databases which are too large to fit in main RAM memory or even a 32-bit virtual address space so portions of the database must reside on disk. The avoidance of distracting pauses when loading from disk requires a quick-loading database format as well as run-time logic which anticipates the viewpoint so the toolkit can begin paging database regions before they come into view.

Traversals

While the current 3-process rendering pipeline (APP, CULL, DRAW) is adequate for most applications, some require extensive application and cull processing. The addition of an APP traversal would allow user callbacks to be invoked each frame to control object behavior or trigger activity outside the toolkit. And currently, each pipeline's CULL traversal is restricted to a single process. Implementing parallelized traversals for both APP and CULL, where multiple processes concurrently carry out the same traversal, would improve throughput for both. The strict top-down inheritance of state in the scene graph eases this task since multiple processes can traverse individual subgraphs without requiring state information from other subgraphs. However, load balancing issues and allowing APP processing to be conditional on the results of visibility and level-of-detail computations are problematic since these computations are currently made *after* APP processing.

Collision Detection

While intersecting with line segments is useful for terrain following and simple collisions, collisions between objects of substantially different sizes and more detailed interference checking can require very large numbers of segments for adequate spatial coverage. Graph-to-graph intersections of volumes, geometry, and line segments represented by nodes within the scene graph would greatly benefit applications such as MCAD.

5 Conclusions

In this paper, we have presented a toolkit with a novel architecture for building high performance, multiprocessed graphics applications. We have described how the toolkit extracts maximal performance from multiprocessor, immediate-mode graphics workstations primarily through:

- geometric data structures designed for efficient immediate-mode data transfer
- reduction of graphics mode changes
- pipelined multiprocessing for parallel scene graph traversal
- efficient host-based view frustum culling
- stress modified level-of-detail switching
- run-time database and process statistics for tuning

By emphasizing immediate-mode performance without caching, the toolkit lends itself to techniques such as character animation and morphing which require intensive vertex-level modifications.

In the course of writing the toolkit, we developed a number of useful techniques for efficient task and data synchronization in a pipelined, multiprocessing system including a configurable software pipeline with update-driven multibuffering.

Without these performance optimizations, expensive hardware can be substantially underutilized. Since the optimizations described in this paper are non-trivial to implement, providing this functionality in a layered toolkit makes it substantially easier for application and other toolkit developers to reap significant performance benefits.

6 Acknowledgments

We would like to thank both present and past IRIS Performers: Michael Jones, Sharon Fischler, Chris Tanner, Allan Schaffer, Rob Mace, Ben Garlick and especially Craig "Crusty" Phillips and George Kong for their contributions. We would also like to thank Wade Olsen for the video application in Figure 16, Computer Arts and Entertainment of Madrid for the race simulator in Figure 15, Angel Studios and GreyStone Technology for Figure 13, and Paul Mlyniec of Software Systems for Figure 17.

7 References

1. Akeley, Kurt. Reality Engine Graphics. Proceedings of SIGGRAPH 93 (Anaheim, California, August 1-6, 1993). In *Computer Graphics*, Annual Conference Series, 1993, 109-116.
2. Craig, Edgar Phillip. Micropoly 2 vs. Stargazer: the Search for an Exoteric Image Generator. *Proceedings of the 1996 Image VIII Conference,* Dallas, Texas, 26-29 June, 1996. 370-371.
3. Ferguson, Robert, et al. Continuous Terrain Level of Detail for Visual Simulation. In *Proceedings of the 1990 Image V Conference, Phoenix, Arizona, 19-22 June, 1990*, 144-151.
4. Funkhouser, Thomas and Carlo Sequin. Adaptive Display Algorithms for Interactive Frame Rates During Visualization of Complex Virtual Environments. Proceedings of SIGGRAPH 93 (Anaheim, California, August 1-6, 1993). In *Computer Graphics*, Annual Conference Series, 1993, 247-254.
5. Grimsdale, Charles, dVS - Distributed Virtual Environment System. In *Proceedings of Computer Graphics '91 Conference*, London, 1991.
6. Hewlett-Packard Company, *Starbase Graphics Techniques and Display List Programmer's Guide,* Hewlett-Packard, Fort Collins, Colorado, 1991.
7. Kaplan, Michael. The design of the Doré graphics system, *Advances in Object-Oriented Graphics I, Konigswinter, Germany, 6-8 June 1990*. Springer-Verlag, 1991. 177-198.
8. Kawalsky, Roy, *The Science of Virtual Reality and Virtual Environments,* Addison-Wesley, Wokingham, England, 1993.
9. Neider, Jackie, Tom Davis and Mason Woo, *OpenGL Programming Guide*, Addison-Wesley, Reading, Mass, 1993.
10. Paradigm Simulation Inc., *VisionWorks Programming Guide*, Paradigm Simulation, Dallas, Texas, 1992.
11. Segal, Mark, et al. Fast Shadows and Lighting Effects Using Texture Mapping, Proceedings of SIGGRAPH '92 (Chicago, Illinois, July 26-31, 1992). In *Computer Graphics* 26,2 (July 1992, 249-252.
12. Strauss, Paul and Rikk Carey, *An Object-Oriented 3D Graphics Toolkit*, Proceedings of SIGGRAPH 93 (Anaheim, California, August 1-6, 1993). In *Computer Graphics*, Annual Conference Series, 1993, 341-349.
13. van Dam, Andries, et al., PHIGS+ Functional Description Revision 3.0, *Computer Graphics* 22, 3 (July 1988), 124-218.
14. Ward, Mark, et al. A Demonstrated Optical Tracker with Scalable Work Area for Head-Mounted Display Systems, Proceedings of 1992 Symposium on Interactive 3D Graphics (Cambridge, Massachusetts, March 29 - April 1, 1992), 43-52.

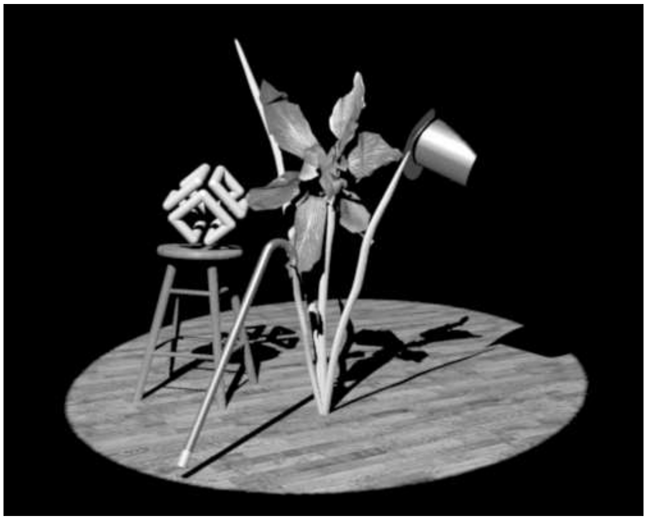

Figure 12. Real-Time Shadows Using Multipass Rendering

Figure 13. Precomputed and Dynamic Geometry Animations

Figure 14. Visual Simulation Scene

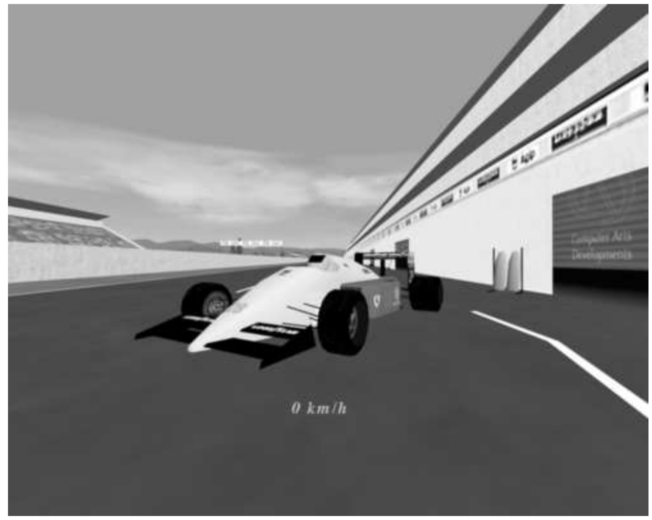

Figure 15. Racing Simulator with Collision Detection

Figure 16. Video Special Effects Using Draw Callbacks

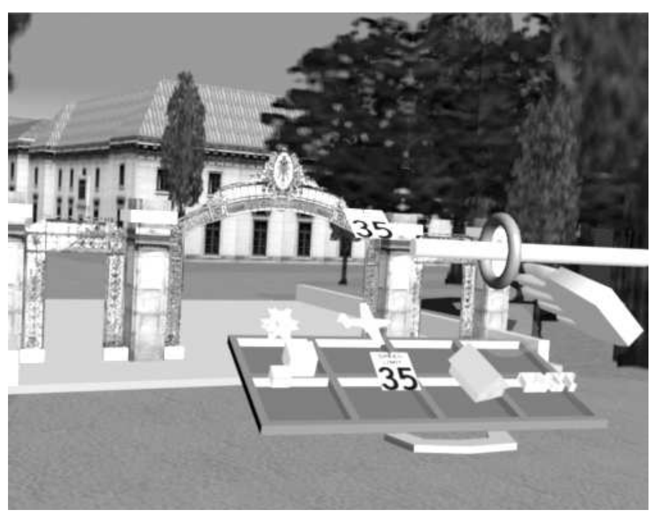

Figure 17. Fly Through with Virtual Reality Interface