

Hierarchical Parallelism in a Scene Graph

Wes Bethel
R3vis Corporation
wbethel@r3vis.com

J. Dean Brederson
Scientific Computing and Imaging Institute
University of Utah
jdb@cs.utah.edu

Abstract

We present a scene graph architecture that features and supports parallelism at multiple levels. Hierarchical parallelism increases overall end-to-end throughput at each stage of the graphics pipeline, from database creation through rendering. A thread-safe scene graph is the fundamental data structure in our design model. Multiple rendering engines may simultaneously read from the scene graph, and multiple application threads may simultaneously write to the scene graph. Each rendering engine in this architecture is itself a multistage, multithreaded software component that draws to separate hardware-accelerated displays. We describe the data structures and access methods that provide thread safety amongst multiple rendering engines and outline how an individual multistage, multithreaded rendering engine uses this infrastructure effectively.

1 Introduction

The term *scene graph* describes a collection of data structures and operators that implement retained-mode storage and rendering of graphics data. The scene graph implements construction and traversal of data structures, an element common to all graphics applications. The majority of code in complex graphics and visualization applications is data management, and scene graph systems provide a set of features that handle many common data management and rendering tasks. Over time, graphics hardware and platforms have evolved considerably, along with the applications themselves. It is in this context of evolving platforms and increasingly demanding application requirements that we have undertaken the work discussed in this paper. Specifically, our goal was the design and implementation of a thread-safe scene graph API that leverages pipelining and parallelism in rendering tasks to increase performance.

The focus of this paper is on parallelism within a scene graph. We describe elements of a thread-safe scene graph API, and describe a framework in which multiple multithreaded, multistage rendering engines can simultaneously create images from a single scene graph. Our implementation, OpenRM Scene Graph [OpenRM], is

an open source scene graph system that uses OpenGL as its basis for hardware accelerated graphics rendering.

Our discussion of previous and related work focuses on existing scene graph technology. Next, we investigate the general requirements for a thread-safe scene graph system. We present the design for our implementation of a thread-safe scene graph system meeting the requirements set forth in this paper. Building upon that design, we present an overview of a multithreaded, multistage rendering engine implementation.

2 Previous and Related Work

[Rohlf94] introduces Performer, a multiprocessing-capable scene graph API for use on SMP platforms. A two-stage rendering pipeline increases overall throughput: a view traversal culls portions of the scene that lie outside the view frustum, then a draw traversal renders this visible model into the framebuffer. Rendering load (system stress) is quantified as a scalar value, which may be used by the scene graph system at runtime to reduce the render load by selecting simplified models, for example. The system supports asynchronous scene graph updates as well as intersection/collision detection. Multiple rendering pipes can share a single scene graph, and can be synchronized to draw in concert.

The scene graph model for Java3D defines the state of a given leaf node as the accumulated state changes along a path from the root of the scene graph to that leaf [Sowrizal98]. Based upon that definition, Java3D can perform simultaneous rendering of leaf nodes according to each unique path in the scene graph. Thus, the order in which primitives are drawn is not guaranteed to reflect placement in the scene graph. In other words, the application has no control over the order in which primitives are drawn. This behavior is undesirable in any situation that requires a deterministic ordering of primitives, such as depth-first rendering of transparent geometry, rendering of shadow polygons, and so forth. Field tests that used Java3D for rendering show that it was an order of magnitude slower than a native scene graph toolkit for a multiple display application, although

development time was faster when using Java3D [Christianson00].

The bandwidth limitations of a serial graphics interface may be overcome through the use of a parallel graphics interface in [Igehy98]. This research outlines what can be thought of as extensions to OpenGL that inherently support parallelism. A many-to-one architecture allows multiple renderers to make use of a single graphics interface. It is important to note that this design is predicated on the assumption that a host CPU cannot keep a graphics pipe full, and that the cost of a graphics context switch is negligible.

[MacIntyre98] presents a scene graph model that uses distributed shared memory (DSM) as the foundation for a distributed rendering system. Scene graph nodes are replicated across nodes of a distributed memory system, such as a PC cluster. Notification objects can be attached to any scene graph node, and they communicate changes in local state to the collective state of the scene graph on the host. This particular scene graph model design appears to emphasize capability rather than capacity. Such an approach would be best suited for a collaborative visualization or visual simulation application in which participants on separate machines are viewing or interacting with a single scene. A recent variation of this theme is presented in [Zelevnik00], but this design places an implementation-neutral layer atop specific scene graph implementations to achieve heterogeneous collaboration.

[Tigges00] describes a system that uses custom Python code to build and manage a scene graph that is subsequently rendered in parallel. Modeling and scene graph management are performed by Python, and rendering is “outsourced” to an external service, such as a software ray tracer running on a PC cluster. This parallel rendering framework uses a worker pool model for distributing rendering of scanlines across nodes of the cluster, and is implemented by interfacing Python with the native MPI¹ service provided by the cluster. Thus, the entire cluster is used to render a single image. Unfortunately, the article does not provide details about important scene graph management topics such as view frustum culling or dynamic updates to the distributed scene graph.

Our work is aimed at a slightly different target than previous efforts. We chose a threaded programming model to implement parallelism, and thus benefit from a single

memory address space. Our implementation narrowly focuses on the constructs required for a thread-safe scene graph API using OpenGL as the basis for rendering. To improve performance, we capitalize on OpenGL retained mode mechanisms, and adopt a multithreaded, multistage design.

The use of OpenGL in a parallel rendering environment requires special care, since multiple rendering threads cannot simultaneously share a single OpenGL rendering context. Furthermore, performance in OpenGL is maximized when the application can make use of retained mode constructs, such as display lists, which are compact, “compiled” representations of OpenGL commands that include geometry and state changes. Use of display lists in applications is optional, but can result in substantial performance improvements, depending upon the OpenGL implementation. In contrast to display lists, which are optional, other types of retained mode constructs are a required part of an OpenGL application. Any application that uses texture mapping must create a texture object, which is a type of OpenGL retained mode object. The texture object is an identifier that refers to and encapsulates all attributes of an OpenGL texture. Texture objects, display list indices, and rendering state (e.g., current color, material values, lights, matrix and attribute stacks) are context-specific. By considering design options based on the notion of an OpenGL context, we can make choices that both support thread safety and increase rendering performance.

3 Scene Graph Thread Safety Requirements

For our purposes, *thread safety* means that a software component or data structure may be used concurrently from multiple threads without disastrous consequences [Butenhof97]. Considering the specific domain of scene graph models and rendering, we observe that most graphics and visualization applications generally require two types of thread safe behavior from graphics infrastructure software: support for multiple simultaneous readers and writers.

In a multiple-readers architecture, multiple rendering threads simultaneously read and render from a single scene graph. Such an architecture is commonly used to create parallel rendering applications to support tiled display surfaces or collaborative interaction. This is an effective way to achieve parallelism in rendering for graphics and visualization applications.

In a multiple-writers architecture, multiple application threads may create or modify a single scene graph.

1. The MPI Consortium. Message passing interface. <http://www-unix.mcs.anl.gov/mpi/>

Application performance may be effectively increased by parallelizing data input operations or by distributing scene graph updates (e.g., input devices, dynamic model animation). One of the key challenges in supporting thread safety for multiple writers lies in providing a mechanism for orderly access to the scene graph. Our approach for providing thread safety makes use of both implicit and explicit synchronization operations.

Our requirements for a thread-safe scene graph API can be succinctly summarized as follows: the scene graph system must provide support for multiple simultaneous reader and writer threads.

4 Multiple Readers and Writers: The Context Cache and the Component Manager

There are two fundamental problems that we wish to overcome in our thread-safe scene graph system. The first challenge is to remove state data from the scene graph itself. The second challenge is to enforce orderly write access to a single scene graph by multiple, asynchronous threads.

Our approach for a thread-safe scene graph in a multiple reader environment uses two related mechanisms, the *context cache* and *component manager*. These two fundamental objects are the basis for hierarchical parallelism in a scene graph. Since the context cache and component manager are cooperative but architecturally disjoint, they must be synchronized during the course of normal application execution. This required synchronization is accomplished with a *context cache key*.

Maintaining orderly write access to the scene graph amongst multiple threads presents technical challenges. Our approach is to use a combination of explicit locking mechanisms with a policy that serves to delimit when application code may write into the scene graph.

4.1 The Context Cache

In a typical multiple readers architecture, there may be several rendering engines asynchronously traversing the scene graph in order to create images. It should be clear that there can be no rendering state information in a thread-safe scene graph. Nonetheless, each of the many renderers requires thread-safe storage of render state information and retained mode data. We define the *context cache* to be an object that is associated with each rendering thread that is used to manage information that is specific to each rendering context.

The context cache contains retained mode data that is private to each rendering thread. In OpenRM, the context cache is integrated into the RMPipe object, which is comprised of a render context, a drawable (window), and other rendering-specific information. OpenRM uses OpenGL for rendering, and so we are referring to OpenGL display list indices and texture object identifiers when we speak of retained mode data. The OpenGL context itself maintains render state information, such as current color, material properties, and so forth.

The context cache provides thread-safe storage of values that are specific to a rendering context, and also associated with single scene graph objects. One common pitfall encountered in thread-unsafe systems is that context-specific data is stored as part of the scene graph itself. The fundamental issue is that data exists that is dependent upon both the scene graph objects (what is things being drawn), and the rendering context (the environment in which drawing takes place).

The size of the context cache is a function of the total number of scene graph objects that may be reduced to a retained mode representation. In our implementation, the context cache holds OpenGL display list indices and texture object identifiers. The OpenRM rendering engine will automatically reduce all drawable objects into OpenGL display lists for improved rendering performance¹. Note that the size of the context cache is a function of the number of scene graph objects, not the size of each of the objects. In large-data visualization applications, there are typically a small number of scene graph objects, but each object consists of a large number of triangles (or other primitives). Scene graph nodes consist of “scene data” (material properties, lights, cameras, transformations, and so forth) along with aggregations of “primitive data” (a draw method and raw geometry data: vertices, normals, colors). Therefore, the size of the context cache is a function only of the number of objects in the scene graph, and not a function of the size of each of the individual objects.

4.2 The Component Manager

The component manager is the “central clearing house” for all scene graph objects. Creation and destruction requests for all objects must be processed by the component manager. At the time of scene graph initialization, the component manager preallocates a large block of scene graph objects, and then manages its own internal free and allocated object lists. This approach allows for

1. This automatic behavior may be overridden by the application.

synchronization of the size of the component manager object pools with the size of the rendering context cache, which is a requirement for thread-safe multiple reader access to the scene graph. The primary function of the component manager is to serialize accesses to scene graph resources. This function is especially relevant for multiple application threads, requiring write access to the scene graph.

The component manager mediates a number of operations on scene graph objects, and enforces orderly access through the use of internal mutual exclusion locks. The operations include construction, destruction, attribute modification and changes to scene graph topology. In general, none of these atomic scene graph operations is computationally expensive, and none are part of the rendering process, so they cannot adversely affect frame rate.

4.3 Interaction Between the Context Cache and the Component Manager: The Context Cache Key

During normal operation, an application will create a scene graph then request rendering. The rendering operation may cause retained mode objects to be constructed. If the scene graph contains textures, the rendering process will build OpenGL texture objects for each texture and store them in the context cache. Other primitives can be compiled into OpenGL display lists, and the display list indices are stored in the context cache.

We have not yet stated how we address the problem of indicating when a retained mode object is “stale” with respect to the data in the scene graph. We will explain how the need to rebuild retained mode objects is communicated to each separate rendering thread from the scene graph system.

We use a simple data object called a *context cache key* that is associated with each scene graph object that may have a corresponding entry in the context cache¹. The context cache key is created by the component manager when the scene graph object is created or updated by the application. Later, each rendering process compares its copy of the cache key to that in the scene graph. When they differ, the retained mode object must be rebuilt, and the rendering thread must update its copy of the context

cache key. When the context cache keys in the scene graph and the context cache are equivalent, the retained mode structure in the context cache is current with respect to the corresponding object in the scene graph.

The roles of the component manager, the context cache, and the context cache key are listed below in Table 1. The interrelationship of these three subsystems are shown in Figure 1. Note that there is a unique context cache associated with each rendering pipe, and that the context cache maintains retained-mode data specific to each rendering context. This is an example of the system architecture for a simple scene graph example with two separate rendering pipes..

Table 1. Thread-Safe Scene Graph Constructs

Component	Role in Scene Graph
Component Manager	The component manager brokers application requests to the scene graph. There is one component manager in the scene graph system.
Context Cache	The context cache contains a table of data that is specific to a rendering context. The table contains context cache keys and retained mode object identifiers.
Context Cache Key	The context cache key is an identifier that is set by the component manager when the application sets values in the scene graph. Each renderer compares the value of its context cache key with the value in the component manager. When they differ, the associated retained mode object is out-of-date with respect to the data in the scene graph.

There are two assumptions that must be valid in order for this strategy to function reliably. First, each time the component manager requests a new context cache key, the new key must be unique with respect to any other key that has been or will be generated. Second, the rendering engine must be prevented from comparing its copy in the context cache with the context cache key that is stored in the scene graph.

To satisfy the first assumption, uniqueness, we have implemented a counter that is protected by a mutex. The mutex prevents multiple threads from obtaining identical values for a context cache key. Each time a new key

1. Not all scene graph objects are reducible to a retained mode object. Text properties, for example, have no direct equivalent in OpenGL.

is requested, the calling thread first obtains ownership of a mutex lock, is assigned a new context cache key, then releases the mutex. To satisfy the second assumption, we must define an access policy that will prevent simultaneous read and write operations in the scene graph. This policy is defined in the next section.

4.4 Multiple Writers: Coordinating Access to the Scene Graph

In the multiple writers scenario, multiple application threads may attempt to simultaneously modify the scene graph. A thread-safe system will provide orderly access to multiple writer threads that wish to modify the single scene graph. Providing thread-safe access to multiple writers requires careful attention to several topics: creation and destruction of scene graph objects, modification of scene graph objects, modifications to scene graph topology, and ensuring coordinated access between readers and writers. Scene graph thread safety amongst multiple writer threads is even more difficult in the case of a scene graph system that provides a multistage rendering pipeline. This latter issue, which we call “frame accurate scene graph data,” is the subject of ongoing work and will not be addressed in any substantial detail in this paper.

The component manager acts as a broker for scene graph object creation and destruction operations. When the application requests a new scene graph object, the request is processed by the component manager. The component manager uses internal thread-safe synchronization mechanisms (mutexes) that effectively serialize object construction and destruction operations.

Scene graph objects are modified by applications during the normal course of execution. Geometry is added to primitives, primitives are added to nodes, cameras define views into the scene, and so forth. Because of the substantial number of scene graph objects and their attributes, the product of scene graph objects and ways they can be modified is quite large. Rather than place serialization code in each of these methods, our implementation focuses on a pair of related solutions that simplify this aspect of thread safety. The first solution is to automate serialization for what may be the most common type of multiple write access: updates to “heavy payload” data, such as vertices, normals and texture data. The second solution is to provide a general purpose scene graph node lock that applications may use to explicitly protect a scene graph node by preventing access by other reader or writer threads. Neither of these solutions addresses hierarchical locking, where a lock at

a single scene graph node locks all nodes in the sub-graph rooted at the locked node.

Operations that change the parent/child relationship between two nodes modify scene graph topology. Such operations include “add child,” “remove child” and “add primitive.” All such operations are made thread safe by mutex locks in the component manager. No special application code is required to ensure thread safety for operations that alter scene graph topology.

Maintaining order between multiple writer threads, as well as reader/writer combinations, is a complex matter and difficult to clearly define. One difficulty stems from the hierarchical nature of the scene graph itself. If a lock is placed on a scene graph node, how is the state of that lock propagated downward to all children? A similar difficulty exists with respect to objects that are attributes of larger structures.

In our implementation, we focus our attention on a commonly occurring subset of the larger problem. We rely on the component manager to serialize access to all scene graph objects for all writer threads. The component manager uses a mutex lock that will effectively serialize all modifications (writes) to the scene graph. In most instances, application performance will not be adversely affected by this approach as scene graph write operations are not computationally or resource-intensive. This approach is a simple but effective means to coordinate access to multiple writer threads, but does not address the issue of coordinating access between reader and writer threads.

Coordination between reader and writer threads is partially addressed through a policy that must be implemented in the application code. The application must ensure that scene graph updates and rendering do not occur simultaneously. This is a straightforward matter, as OpenRM’s frame rendering call is a blocking operation. Applications that require simultaneous scene graph updates and rendering must themselves be parallel, and can use an explicit “merge” type operation. A merge is implemented by having an asynchronous model loading process construct a detached scene graph which is later merged into the main scene graph when the renderer is not active.

5 OpenRM Scene Graph Multistage and Multithreaded Rendering

Our multistage rendering pipeline is decomposed into two primary components: a view stage and a render stage. Like the architecture used in Performer, the goal

of our view stage is to reduce the load on the render stage. Load is reduced by culling objects that lie outside the field of view, and by moving view dependent computations out of the rendering pipeline. The render stage will transmit data to the graphics pipeline as quickly as possible. For this article, we focus on the problem of minimizing the amount of communication traffic between the view and render stages of the pipeline.

The view stage traverses the scene graph, and generates a “draw list” that is consumed by the render stage. Draw list entries are composed of opcode/value/data triplets, and the maximum size of the draw list is bounded by the size of the scene graph. The draw list is a compact representation of render state transitions and objects to be drawn, and may be significantly smaller in size than the scene graph itself due to view frustum culling or other view dependent operations. The draw list itself is multi-buffered, as the view and render stages are by design processing different frames. There exists one multi-buffered draw list for each pass of multipass rendering: opaque objects are processed separately from transparent objects, and stereo modes support left- or right-channel operations. The compact draw list can be processed much more quickly than a scene graph traversal, and the render stage spends most of its time keeping the graphics pipe full, as opposed to scene graph processing. .

Table 2 Multistage Processing Modes

Mode	Feature
Multistage	View and render traversals occur sequentially in one frame. Zero frame latency.
Multistage Parallel	View and render execute concurrently in detached threads. One frame latency.
Multistage, View-Parallel	View and render execute concurrently. View is in a detached thread, render is in the same thread as the caller. One frame latency.

The draw list contains only “metadata” representations of the objects to be drawn. The actual heavy payload data (vertices, normals, textures, and so forth) are not duplicated in the draw list. This approach facilitates economical and efficient use of space, but does not address the issue of “frame accurate” scene graph data. Frame accuracy in a general sense is the subject of ongoing work. The current implementation does allow for frame accuracy of geometric transformations by including matrix data as an adjunct to the draw list.

We have implemented several processing models in the multistage and multithreaded renderer. A “multistage, serial” mode will cause the view and render stages to execute sequentially within one thread. A “multistage, parallel” mode places view and render into separate threads, and both execute concurrently. In this mode, when view is processing frame N , render is processing the results from frame $N-1$. A third mode, called “multistage, view-parallel,” places view into a separate thread, but render remains in the same execution thread as the caller. In this mode, both view and render execute concurrently. This mode offers advantages in tiled display environments, where a control environment precomputes view transformations for each display screen, then invokes application rendering code (see [Bethel01])

6 Conclusions and Future Work

We have presented a design for a thread-safe scene graph that supports multiple, simultaneous rendering threads reading from a single scene graph, and also supports multiple threads writing into the scene graph. To achieve thread safety, we have introduced several related concepts that provide a framework for thread-safe operations. The context cache manages render specific data, and is resident with each rendering thread. The component manager acts as a broker for scene graph object creation and destruction, and mediates operations that modify scene graph topology. When the application updates the scene graph, a unique context cache key is created and stored inside the scene graph. Each rendering thread maintains its own collection of cache keys, and can determine by key comparison when any retained mode objects are out-of-date with respect to the contents of the scene graph.

The context cache and component manager address thread safety in the context of supporting multiple asynchronous rendering engines that read from a single scene graph. Thread safety for multiple writer threads is implemented using a mechanism that serializes all write operations. Scene graph updates of this sort are typically not computationally or resource intensive, so application performance is not adversely affected in most cases. The problem of hierarchical locking remains, and we have not addressed that issue in this paper.

Our system implements a multistage and multithreaded rendering engine that is based upon the thread-safe scene graph design, and which has been implemented on Unix and Windows platforms, using OpenGL for hardware acceleration. Together, the thread-safe scene graph and multithreaded rendering engine combine to implement hierarchical parallelism in a scene graph system.

Our design partially addresses the issue of frame accurate scene graph data in a multistage rendering environment. While transformations are currently fully frame accurate, a solution to general frame accuracy is the subject of ongoing work.

A thread-safe scene graph is a central component of scalable and high performance rendering applications, such as scientific visualization and visual simulation (Figure 2). The solution we have presented was created for shared-memory architectures. In the future, we will explore deployment on distributed memory architectures, such as PC clusters equipped with commodity graphics hardware. This future work will rely on parallel graphics interfaces, such as the design presented in [Humphreys00].

7 Acknowledgements

The authors wish to thank Randall Frank and the Scientific Computing and Communications Department at Lawrence Livermore National Laboratory for many helpful technical suggestions and for providing access to facilities during the course of this project. This work was funded by the U.S. Department of Energy, Office of Advanced Scientific Computing Research in the Office of Science, through the Small Business Innovation Research (SBIR) program under contract number DE-FG03-00ER83083.

8 References

- [Bethel99] W. Bethel and J. Bastacky, "Measurement of Perceived Objects," in Late Breaking Hot Topics, Proceedings of IEEE Visualization 99, San Francisco, CA, October, 1999.
- [Bethel00] W. Bethel, B. Tierney, J. Lee, D. Gunter, S. Lau, "Using High-Speed WANs and Network Data Caches to Enable Remote and Distributed Visualization", in Proceedings of SC00, Dallas, Texas, November 2000.
- [Bethel01] W. Bethel, R. Frank and J. D. Brederson, Combining a Multithreaded Scene Graph System with a Tiled Display Environment, submitted to *IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2001*.
- [Butenhof97] D. Butenhof, Programming with Posix Threads. Addison-Wesley, Reading MA, 1997.
- [Christianson00] B. Christianson and A. Kimsey, Comparison of Java3D in a Virtual Environment Enclosure (2000). Master's Thesis, Naval Postgraduate School, Monterey, CA, March 2000.
- [Humphreys00] G. Humphreys, I. Buck, M. Eldridge and P. Hanrahan, Distributed Rendering for Scalable Displays. In *Proceedings of Supercomputing '00*. Dallas, Texas, November 2000.
- [Igehy98] H. Igehy, G. Stoll and P. Hanrahan, "The Design of a Parallel Graphics Interface." In *Computer Graphics (Proc. of ACM SIGGRAPH 98)*, pp. 141-150, August 1998.
- [MacIntyre98] B. MacIntyre and S. Feiner. "A Distributed 3D Graphics Library". In *Computer Graphics (Proc. ACM SIGGRAPH 98)*, pp. 361-370, August 1998.
- [OpenGL] OpenGL 1.2 Specification, <ftp://ftp.sgi.com/opengl/doc/opengl1.2/opengl1.2.1.pdf>
- [OpenRM] <http://openrm.sourceforge.net/>
- [Rohlf94] J. Rohlf and J. Helman, IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Computer Graphics (Proc. ACM Siggraph 94)*, pp 381-394, August 1994. (Performer)
- [Sowrizal98] H. Sowrizal, K. Rushforth and M. Deering. The Java3D API Specification. Addison-Wesley, Reading MA, 1998.
- [Strauss92] P. Strauss and R. Carey, "An Object-Oriented 3D Graphics Toolkit, In *Computer Graphics (Proc. ACM Siggraph 92)*, pp. 341-349, August 1992. (Inventor)
- [Tigges00] M. Tigges and B. Wyvill, "Python for Scene and Model Description for Computer Graphics", In *Proceedings of the Eighth International Python Conference*, Arlington, Virginia, January 2000. <http://www.python.org/workshops/2000-01/proceedings.html>
- [Zelevnik00] B. Zelevnik, L. Holden, M. Capps, H. Abrams and T. Miller, "Scene-Graph-As-Bus: Collaboration between Heterogeneous Stand-alone 3-D Graphical Applications," *Computer Graphics Forum*, 19, 3, (*Proc. Eurographics 2000*), 2000.

Figure 1. System Architecture: Component Manager, Context Cache and Context Cache Keys

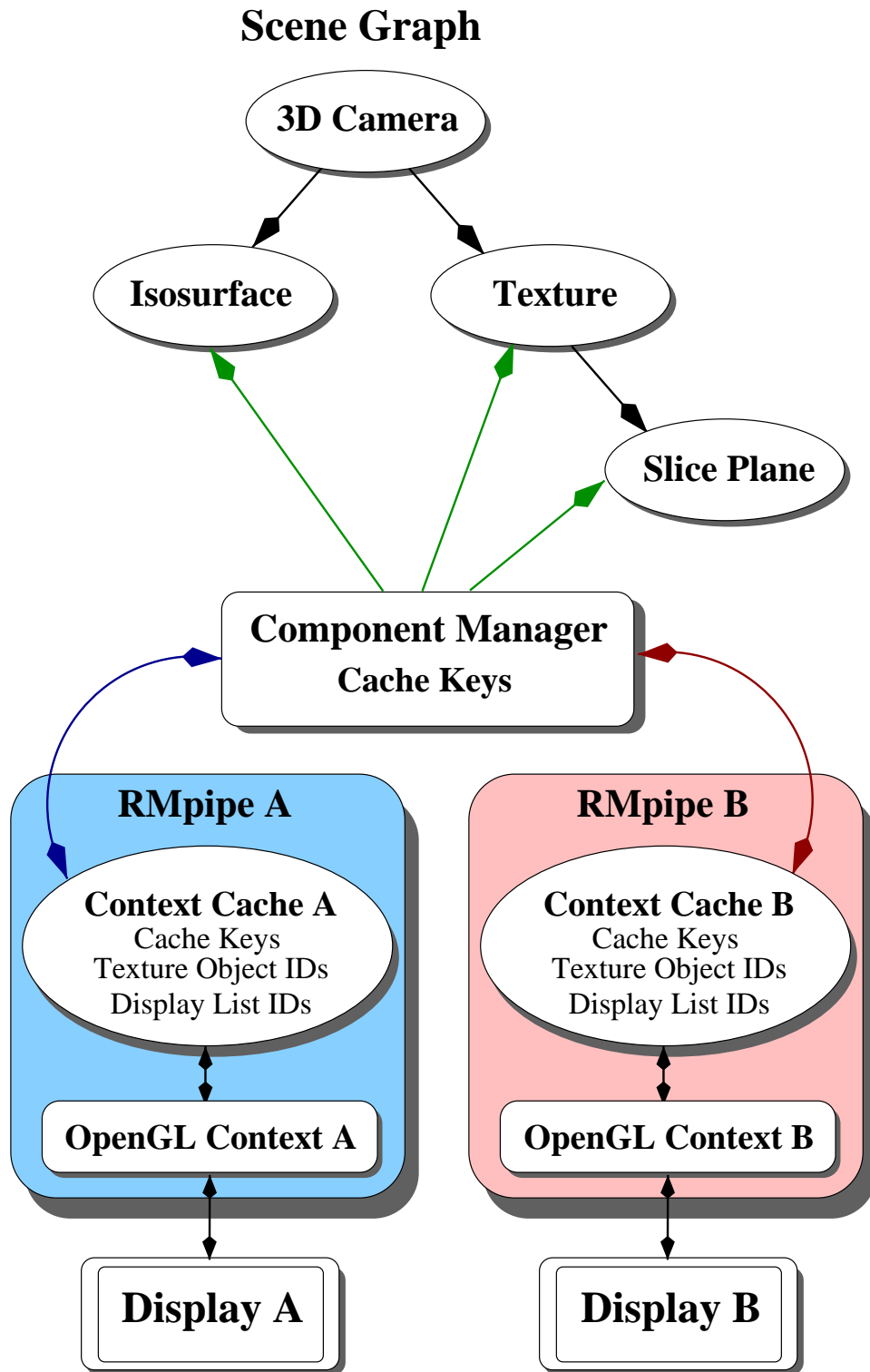
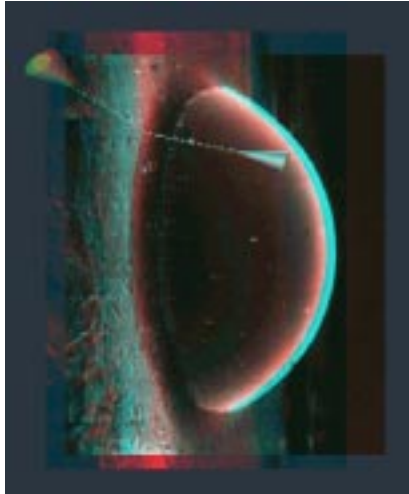
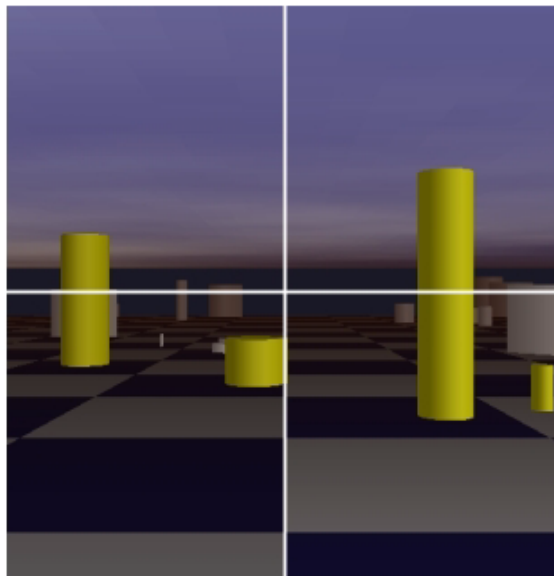
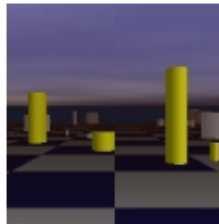


Figure 2. Examples of Scientific Visualization and Visual Simulation



Example 1. A red-blue stereo anaglyph of a scanning electron microscope dataset permits a scientist to measure surface angles using a simple stereoscopic display. The rendering shown is accomplished using multipass rendering and stereo-aware traversal masks. The printed image may be viewed by the reader with a pair of red-blue stereo glasses [Bethel99].



Example 2. Tiled display environments are increasingly popular for scientific visualization and visual simulation applications. This example demonstrates an integration of OpenRM and VDL running on a quad-pipe tiled display. The rendering shown leverages view frustum culling and level-of-detail model switching to increase rendering performance. In addition, multiple-readers render to each of four pipes to produce the composite image. The user navigates with a mouse in the small navigation window shown to the left [Bethel01].

Example 3. This example shows an IBR-assisted volume rendering of a combustion simulation dataset. The data was computed on an AMR grid, and the computational grid is shown in wireframe. A pipeline-parallel, multipass algorithm is exploited to achieve interactive framerates, and multiple writers keep the scene graph objects full of the streaming data for compositing and rendering [Bethel00].

