

---

*White Paper: OpenRM Scene  
Graph Thread Safety and  
Multistage Rendering*

**E. Wes Bethel, R3vis Corporation**

*July, 2001*

**Copyright (C) 2001-2003, R3vis Corporation, All Rights  
Reserved.**

**R3vis Corporation  
P.O. Box 979  
Novato, CA, USA  
94948-0979**



## 1.0 Appendix A: Phase I SBIR Technical Report - Draft

*This technical document is a supplement to the Final Report submitted at the conclusion of Phase I SBIR work activities. This report describes the technical issues we addressed during the Phase I period, research challenges, along with an overview of our software implementation.*

## 2.0 Abstract

We present a scene graph architecture that features what we refer to as *hierarchical parallelism*. The objective of the design is to increase overall end-to-end throughput of graphics, visualization and visual simulation applications. At the center of the design is a single scene graph, or data cache containing data to be rendered. On the back end, each of an arbitrary number of rendering engines is each itself a multistage and multi-threaded, pipelined-parallel subsystem. On the front end, multiple application processes may interact with the scene graph. The system is designed for use in parallel rendering and processing environments. The results of this project are distributed under an Open Source license, and may be freely downloaded over the web from [openrm.sourceforge.net](http://openrm.sourceforge.net).

## 3.0 Overview and Background

OpenRM Scene Graph is an application programming interface (API) that implements a *scene graph model*, and is used by software developers to create high performance and portable graphics, visualization, visual simulation and Virtual Reality applications. A scene graph model is somewhat analogous to a traditional database: the application incrementally builds the database by supplying data to the scene graph subsystem; once built, the application can invoke operations on the data, such as frame rendering, intersection calculation, and so forth.

The bulk of the code in complex graphics and visualization applications is data management, and scene graph systems provide a set of features that implement the most common data management and rendering features. As time goes on, graphics hardware and platforms have evolved considerably, along with the applications themselves. It is in the context of evolving platforms and increasingly demanding applications requirements that we have undertaken the work discussed in this paper: to implement a thread-safe scene graph API.

The nature of the work we performed during the Phase I SBIR was to design and implement a specific set of enhancements to OpenRM Scene Graph<sup>1</sup>. The enhancements we designed and implemented are:

---

1. OpenRM Scene Graph was released to under an Open Source License in March 2000. OpenRM was derived from RM Scene Graph, a commercial scene graph API created and licensed by R3vis Corporation.

- Thread safety;
- Pipelined and parallel, multithreaded and multistage rendering;
- Support for offscreen rendering;
- Support for tiled display systems.

In the balance of this document, we describe the technical issues and implementation details of our work. We begin with a discussion of topics related to thread safety. Building upon thread safety, we discuss a pipelined, parallel architecture rendering engine implementation. We extend the resulting system to support offscreen rendering, as well as simultaneous rendering to multiple display devices configured as a tiled display surface.

## 4.0 Thread Safety

Our primary requirements for a “thread safe” scene graph system focus on support for multiple simultaneous reader threads, and multiple simultaneous writer threads. Scene graph rendering corresponds to a read-only scene graph operation, while multiple application threads loading data into the scene graph corresponds to a write operation. After introducing the concept of thread safety, we define our targeted deployment environment. Given a definition of thread safety and a deployment environment, we define requirements for a thread safe scene graph system. Building upon those requirements, we describe our approach, which makes use of two complementary technologies: the context cache and the component manager.

### 4.1 What is Thread Safety?

The term *thread safety* is often used to describe specific measures of correctness for software components or code blocks that are executed by multiple, concurrent threads. A software component that is thread safe will produce “correct results,” free from unexpected side-effects when executed by multiple, concurrent threads. In contrast, software components that are not thread safe are not guaranteed to produce correct results when executed by multiple, concurrent threads. Most often, code that is “thread unsafe” stores application state into static variables, and reuses the static state information when sequentially invoked. Most strategies for achieving thread safety remove the static state information from the called routine, and push the burden of state management back to the caller.

### 4.2 Targeted Deployment Environment

Our implementation of a thread safe scene graph is targeted at environments where parallelism is achieved using the *threads* programming model: multiple application threads share a common address space. All modern operating systems, including Unix/Linux/Win32, support some notion of threaded programming models. As architectures evolve, it is often the case that graphics applications are more constrained by i/o to the disk or

network than by graphics rendering rates. Those systems that can overlap processing with graphics rendering will achieve the best overall throughput.

### **4.3 OpenRM Scene Graph Thread Safety Requirements**

Most graphics and visualization applications generally require two types of thread safe behavior of graphics infrastructure software: “multiple readers” and “multiple writers.”

#### **4.3.1 Multiple Readers**

*Multiple readers* describes an architecture consisting of multiple rendering threads that simultaneously read from a single scene graph to produce multiple images. Such an architecture is commonly used to create rendering applications for tiled display surfaces. Tiled display surfaces consist of several display devices that can be used to present multiple views of a single scene. The multiple readers architecture is an effective way to achieve parallelism in graphics and visualization applications that are targeted for use in tiled display environments.

One of the biggest challenges in providing for thread safety in a scene graph infrastructure that supports multiple readers is the absence of state variables in the scene graph itself. Our approach for a thread safe scene graph in a multiple reader environment uses two related mechanisms, called the “context cache” and “component manager,” which are described later.

#### **4.3.2 Multiple Writers**

*Multiple writers* describes an architecture in which multiple application threads create or update a single scene graph. Multiple writers is an effective means to increase application performance by parallelizing data input operations. One of the challenges in supporting thread safety for multiple writers lies in providing orderly access to the scene graph. Our approach for providing thread safe support to multiple writers makes use of implicit and explicit synchronization operations, also described later.

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

### **4.4 Multiple Readers: 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 the separation of render state data from the scene graph itself. We call this the “one-to-many” problem - there is only one scene graph, but there may potentially exist many rendering threads. The second challenge is orderly access to the single scene graph by multiple threads. We call this the “many-to-one” problem. In this section, we describe the scope of our solution to the “one-to-many” prob-

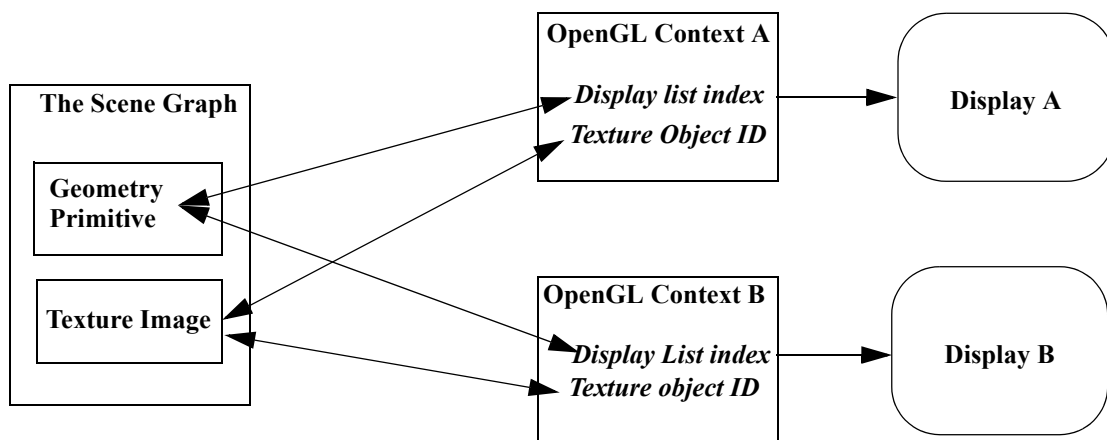
lem, and present implementation details in a later section. We discuss the “many-to-one” problem and implementation details in a later section.

#### 4.4.1 The Context Cache

The *context cache* is an object that manages data specific to a rendering context, and which is associated with each RMPipe object. The OpenGL graphics library provides support for retained mode objects, and our focus in this discussion is on OpenGL display lists and texture objects.

The context cache provides thread safe storage of values that are specific to a rendering context, and also associated with a single scene graph objects (Figure 1). One common pitfall encountered in some scene graph systems is that context-specific data is stored as part of the scene graph itself. The fundamental issue is that there exists data that is dependent upon both the scene graph primitives (the things being drawn), and the rendering context (the environment in which they’re drawn). The context cache is our remedy to this fundamental problem.

**FIGURE 1. One-to-Many Relationship Between Scene Graph Objects and Rendering Contexts**



OpenRM builds retained mode objects from scene graph objects whenever possible in order to increase rendering performance. Each retained mode object is specific to a rendering context.

We define the context cache as a data structure that holds data that is dependent upon both the scene graph and the rendering context. Thus, the OpenRM context cache contains display list indices and texture object identifiers. The context cache is a part of the RMPipe object, which is the rendering context specific object in an application that uses the scene graph system, but is not a part of the scene graph itself. The context cache is created when the RMPipe object is “bound” to an OpenGL rendering context. The size of the context cache is a function of the size of the scene graph itself: as the number of objects in the scene graph grows, the number of potentially cacheable objects likewise grows.

#### 4.4.2 The Component Manager

The component manager is a “central clearing house” for all scene graph objects and performs three crucial functions in the multithreaded scene graph. First, all creation and destruction requests for scene graph objects are processed by the component manager; the component manager performs memory management of scene graph objects. Second, the size of the context cache is inherently a function of the size of the maximum number of scene graph objects, a number that is determined by the component manager. This relationship closely binds the component manager and the context cache. Third, the component manager is a control nexus that provides orderly access to the scene graph for multiple reader and writer threads.

When the application initializes the scene graph, the component manager preallocates a large block of objects, and then manages its own internal free and alloc object lists. This approach allows for synchronization of the size of the component manager object pools, and the size of the rendering context cache.

#### 4.5 Multiple Writers: Coordinating Access to the Scene Graph

In the many-to-one problem, multiple application threads may attempt to simultaneously modify the scene graph. A thread safe system will provide orderly access to multiple writer threads that attempt to modify the single scene graph. The scope of the many-to-one problem is quite large and includes a number of interrelated topics: scene graph object creation and destruction; scene graph object modification; and scene graph topology modifications.

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 synchronization to provide thread safe object creation and destruction requests. Implementation details follow in a subsequent section.

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 the number of scene graph objects and the number of 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 scene graph geometry. 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 writer threads. Neither of these solutions addresses hierarchical locking, where a lock at a single scene graph node locks all nodes in the subgraph 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<sup>2</sup>.”

All are made thread safe by mutex locks in the component manager; no special application code is required to ensure thread safety for topology changing operations.

At the time of this writing (March 2001), the OpenRM renderer ignores any mutex locks that might be set by the application. In the future, the renderer may be modified to respond to the presence of a locked node by either pruning traversal of the scene graph at the locked node, or to block and wait for the node to become unlocked.

## 4.6 Typical Application Architecture

It is assumed that the application that uses the thread safe features of OpenRM (multiple readers or writers) will itself be a parallel application composed of multiple execution threads. A multiple renderer (reader) application will create and manage multiple threads, each of which will create an RMPipe object, and invoke the OpenRM frame rendering method to produce an image. Most applications will want some form of per-thread view computation, such as a sheared-frustum for tiled surface displays (each tile uses a unique view specification for its view of the scene). The application must compute and assign such a view specification - the scene graph system is not intended to be a framework for computing views for multiple displays - it is a framework for running multiple, simultaneous rendering threads that draw to each display from one or more scene graph object databases. Similarly, multiple writer threads must be created and managed by the application.

## 5.0 Thread Safety Implementation Details

Applications developers need not be concerned with the details of context cache and component manager implementation in OpenRM scene graph - the benefits provided by these services are realized at the application level without any explicit code.

### 5.1 Context Cache

The context cache exists on a per-RMPipe basis. The RMPipe contains all the information and structures needed to perform rendering: an OpenGL context, and open drawable, etc. The context cache is a mechanism used to hold data that is specific to an OpenGL rendering context. At the present time, this data consists of display list indices and texture object identifiers.

Associated with each context specific datum in the context cache are what we call "context cache keys." The context cache key is an identifier, or fingerprint, that is associated with each of the display list indices or texture object identifiers. The cache key is merely an identifier that is assigned to each display list index or texture object identifier. When an application writer thread assigns data to a "cacheable object," e.g., an RMprimitive, a

---

2. Each RMnode may contain an arbitrary number of RMprimitive objects. This design encourages applications to aggregate drawable objects.

cache key is automatically generated and stored inside the RMprimitive. Later, at render time, the rendering thread compares the value of the cache key in the RMprimitive with the value of the cache key stored in the context cache of the RMpipe. If they differ, the OpenGL display list is regenerated for the RMprimitive. The purpose of the cache key stored in the scene graph object is to act as a unique identifier associated with the contents of the scene graph object. We don't need to know what the contents are, precisely, but we do need to be able to detect when they have changed, in order to take appropriate action (download a texture to OpenGL or regenerate a display list).

The cache key is simply a 32-bit integer value. Our implementation is a mutex-protected counter, which is guaranteed to be unique each time a new cache key is requested. In the unlikely event that an application exhausts all cache keys, the cache key can be changed to a 64-bit integer by modifying only a few lines of OpenRM code. No application code need be modified, as the cache key is used only by OpenRM internals.

The following table shows the OpenRM primitives that require context-specific data to be stored in the context cache. Of these objects, text and quadrics do not require a cache key because the contents of these OpenGL display lists are invariant - these are in effect pre-tessellated or pre-rendered representations of models (or character glyphs) that are simply instantiated as needed by the renderer.

**TABLE 1. OpenRM Scene Graph Objects Requiring a Cache Key**

Object or Process	Role in Rendering
RMprimitive	All draw code inside RMprimitives can be reduced to a single OpenGL display list index.
Text and Fonts	A single display list identifier is used to render a bitmap glyph for each character. An internal font manager keeps track of display list indices for bitmap glyphs, and causes new lists to be built at render time when the application draws text. The font manager uses an internal cache to hold bitmaps for all characters of all font, style and size combinations. Display lists are built only when the application requests text rendering. The font manager will build display lists only when it encounters a new font, size or family combination. Applications do not have direct access to the font manager.
Quadrics	When OpenRM is initialized, internal display lists are built to represent spheres, cones and cylinders. A number of lists are built for each quadrics object, representing varying degrees of tessellation. Applications do not have direct access to these display lists, but can control which of the tessellation models is rendered via a flag set at the RMprimitive level.

**TABLE 1. OpenRM Scene Graph Objects Requiring a Cache Key**

<b>Object or Process</b>	<b>Role in Rendering</b>
RMtexture	Texture object identifiers are stored inside the context cache. The texture object identifiers are obtained from OpenGL at the time when the texture is downloaded to OpenGL (at render time).
RMimage	The <code>glDrawPixels()</code> used to draw the image is stored as an OpenGL display list index inside the RMimage object. The RMimage display list index is used by draw code for <code>RM_SPRITES</code> primitives and when drawing background image tile scene parameters. Even though RMimages specify pixel data for RMtexture objects, the display list identifier inside the RMimage object is not used within the context of downloading pixel data for textures.

The context cache is created by OpenRM at the time that an RMPipe is “bound” to an OpenGL rendering context. The current design requires each RMPipe to use a unique OpenGL rendering context, and that the contexts used by multiple RMPipes do not share display lists or texture object identifiers. This approach is not the most efficient on platforms that support multiple graphics pipes that are capable of sharing display lists. There are few such platforms, however. Our design will facilitate deployment in distributed memory environments, such as PC clusters equipped with commodity graphics hardware.

When created by OpenRM, the context cache contains enough space to hold display lists for all possible permutations of text, as well as all possible combinations of pre-tessellated quadrics models. These tables are fixed in size, and can never grow or shrink. The initial cache also contains entries for display lists to be associated with each of the RMprimitive objects, RMimage objects and RMtextures. To address the problems inherent with a growing and shrinking population of objects of these types, we preallocate a large amount of space to hold display list identifiers (and cache keys) for many objects. The amount of space allocated is a function of the number of objects present in the Component Manager object pool.

## 5.2 Component Manager

The primary function of the component manager is to serialize access to constructor and destructor functions for a “critical set” of scene graph objects. The critical set is defined as those objects whose population size may have an effect upon the size required by the context cache. When initialized, the component manager will preallocate a large number of objects (RMnodes, RMprimitives, RMimages and RMtextures). Later, when a context cache is created for an RMPipe, the size of the component manager object pool dictates the size of the context cache for certain classes of objects. The component manager is initialized when the scene graph system is initialized; there may be multiple context caches, one per RMPipe, but there is only one component manager.

The component manager maintains its own internal lists of free and allocated objects. When the application invokes a constructor function (e.g., `rmNodeNew()`), a handle to the first item on the free list is returned to the caller. Internal to the component manager,

the object is removed from the free list and added to the alloc list. Conversely, when an object is destroyed (e.g., `rmNodeDelete()`), the object is moved from the alloc list back to the free list. This approach replaces numerous `mallocs` of small objects with a single `malloc` for a large pool of objects.

In the current implementation, the initial size of the object pool is controlled by a compile-time constant. When all available objects have been exhausted, an error message is issued. In the future, we will implement a reallocation strategy that will grow the size of the object pool, and also produce growth in the context cache of the corresponding object type. At this point in time, if the size of the component manager's object pool is too small for a given application, the only remediation is to increase the size of the pool that is allocated when OpenRM is first initialized. The scope of modifications required to the Component Manager to grow the object pool is substantial, but straightforward.

## 6.0 Multistage, Multithreaded Rendering

In the context of rendering, the term multistage describes a divide-and-conquer decomposition of the rendering process into some number of smaller components. The components are executed in assembly-line fashion, where the output of one component along the pipeline becomes input to the next component. The term multithreaded means that all components execute concurrently. In theory, if there are  $N$  such components in the pipeline, and all execute concurrently, then the expected throughput of the pipeline approaches an  $N$ -fold increase, but is accompanied by an  $N$ -frame latency.

The extent to which the speedup of the pipeline approaches the theoretical limit is largely a function of how well the overall problem is decomposed, and to a large degree, the contents of the scene being rendered. A pipeline in which the subproblems are approximately equal will exhibit better overall performance than one in which the time requirements of one subproblem dominates the others. Similarly, a scene that is well decomposed spatially will lend itself to better frustum culling, which will in turn result in faster rendering rates due to reduced load on the graphics hardware.

The  $N$ -frame latency describes the amount of time required for a datum to move from the start of the pipeline to the end. We are assuming that the process of rendering a frame means that each stage of the pipeline executes once. Therefore,  $N$  such invocations must occur before a datum presented at stage one emerges from stage  $N$ .

The preferred platform for deployment of a multistage, multithreaded rendering engine is an SMP computer endowed with a hardware graphics accelerator. An SMP platform, by definition, provides a shared address space. This means that all processes that run on the SMP use the same physical memory. Specifically, the software components of the multistage pipeline can communicate their results via shared memory - one component writes into memory, the next component reads from the same memory. In contrast, in a distributed memory architecture, the data contents would have to be migrated from one physical memory to another. Memory-to-processor bus bandwidth is typically much faster than memory-to-memory migration speeds in distributed memory architectures.

For example, many PC clusters use Fast Ethernet for the node interconnect fabric. That fabric provides transfer rates on the order of 100Mbps, whereas processor/memory bandwidth is typically on the order of 100-500MBps. In some implementations, the I/O channel to the graphics device is on a bus separate from the processor-to-memory bus, so the underlying architecture lends itself well to a multistage architecture.

The motivation behind our design and specification of a multistage, multithreaded rendering engine is to realize an overall improvement in rendering rates on SMP platforms. Our implementation is being made available at no cost to developers as OpenRM Scene Graph, licensed under the terms of the GNU Lesser General Public License (LGPL), and Open Source license.

## **6.1 OpenRM Scene Graph Multistage, Multithreaded Rendering Requirements**

Our multistage pipeline is decomposed into two primary components - a view stage and a render stage. The goal of the view stage is to reduce the load on the render stage by culling objects that lie outside the field of view and by minimizing the amount of processing performed during the render stage. The render stage will ideally only be tasked with transmission of data to the graphics pipeline.

We consider three primary trade-offs in our design of the data structures used between components of a multistage rendering pipeline: frame accuracy of the underlying data, and render state “compaction” and “sorting”, and minimizing the amount of data copying between stages of the pipeline.

A multistage renderer that is “frame accurate” is one that preserves the integrity of data as it moves along the multistage pipeline. In other words, data that enters the pipeline will remain unchanged as it traverses the stages of pipeline. Consider an example: a terrain flyover application. The viewer's position changes in each frame, and the new viewpoint is stored in the scene graph during each frame. The first stage of the pipeline reads a viewpoint, performs view dependent object culling, then passes the results along to the rendering stage of the pipeline. To be frame accurate, the view stage must create a copy of the viewpoint, and transmit the copy to the render stage. Otherwise, the viewpoint used in the render stage will be clobbered by the application when the viewpoint is updated for the next frame. For “small” data, such as viewpoints, the overhead caused by the copying process is negligible. Creating multiple copies of “large” data, such as large geometry arrays, may be impractical or impossible.

Render state sorting seeks to minimize the cost associated with changing rendering state. Rendering state is the collection of current material properties, textures, draw modes and so forth. In OpenGL, the size of the render state is quite large, consisting of more than 200 different parameters. Some parameters may require significant setup time in the graphics hardware, and OpenGL implementations vary in the amount of time required to change state. The amount of variation of cost associated with state changes are significant: one of the OpenRM demonstration programs, tfly draws frames at better than 30fps on an Intel i810 motherboard (a platform costing a few hundred dollars), but

can achieve only about 20fps on an SGI Onyx2 with InfiniteReality2 graphics (a platform costing on the order of \$500,000). A generally accepted guideline in graphics programming to improve performance is to minimize the number of state changes. Often this is achieved by reorganizing the scene graph to aggregate primitives that can be drawn using the same state. Tfly, in the pre-OpenRM 1.4.0 implementation, requires extensive state changes in the OpenGL pipeline (there is no state aggregation or incremental state change - the entire OpenGL state is saved then restored for each of several hundred objects during each frame).

Some scene graph systems (Performer, Java3D) implement state sorting by reordering the scene graph contents. We feel this is, in general, an undesirable quality. In many instances, the application requires a specific draw order (shadow polygons, for example). We compromise by allowing for the possibility of “state accumulation”, where multiple state changes along a depth traversal of the scene graph can be accumulated into a single state change. This optimization is planned for the Phase II effort.

For the purposes of our design, we use rather narrow and conservative requirements: the view stage will cull objects that are completely outside the field of view, and generate information that describes incremental state changes that reflect the inherited render state resulting from parent-child relationships of scene graph nodes. These two requirements serve to reduce the load on the render stage by removing unseen objects from the scene, and by decreasing the amount of render state changes in the pre-1.4.0 OpenRM implementation.

With regard to frame accurate scene graph data, our implementation in 1.4.0 will preserve the frame accuracy of transformations and viewpoints. These attributes and data will be fully multi-buffered so that data “in transit” in the rendering pipeline is insulated from changes made by the application. During Phase II, we will implement a general solution that will support frame accurate scene graph data.

## **6.2 Detailed Technical Design**

### **6.2.1 Communication Between View and Render**

The primary purpose of the multistage design is to lessen the load on the graphics pipe by reducing the amount of information to be drawn during each frame, and by reducing the amount of non-rendering activities performed by the render stage (such as matrix multiplies and matrix inversions). Render stage performance improvement follows from a number of potential operations: model culling, where objects that lie outside the view frustum are not drawn at all; state consolidation, where graphics state changes are minimized; and processing minimization, where all matrices and matrix inversions are computed by the view traversal.

## 6.2.2 View-Render Draw List

We use the term Draw List for the data structure created by view and consumed by render. The Draw List consists of an array of opcode/index pairs. The opcode defines an action that is to be taken by render, such as “change render state” or “draw some primitives.” The complete list of opcodes is described in the following section.

The meaning or use of the “index” component of the Draw List is context-specific. In some instances, the index refers to an RMnode, and the index is an entry in the global RMnode table maintained by the Component Manager. In other instances, the index is an entry in an array of RMmatrix's, which is included as an adjunct structure of the Draw List. All transformation matrices are computed by view, and consumed by render: render is relieved of the burden of computing transformations.

**TABLE 2. Draw List Components**

---

<b>Component</b>	<b>Description</b>
Opcode	An action to be executed by render.
Index	An index into the global RMnode table (maintained by the Component Manager), or an entry into a local RMmatrix table. Some opcodes do not require an associated index value.
RMmatrix Table	Contains transformation matrices computed by view, and consumed by render.

A single draw list represents the results of a view traversal of the scene graph. Since OpenRM intrinsically uses a multipass rendering model consisting of three rendering phases (opaque 3D, transparent 3D, opaque 2D), there exists one Draw List for each of these rendering passes. Furthermore, each Draw List is double buffered, thereby allowing for concurrent, pipelined operation of view and render. Thus, there are a total of 12 Draw Lists associated with each RMpipe object.

The maximum theoretical size of each Draw List is a function of the total number of RMnodes maintained by the Component Manager. In typical large scene graph applications, the number of scene graph nodes might reach the thousands, therefore the size of each Draw List is (arguably) negligible, even for large scene graphs.

In the Phase I SBIR implementation, the initial size of the Draw List is defined by a compile-time constant, and remains fixed for the lifetime of the application. Future work will include dynamic growing and shrinking of the Draw List in response to dynamic resource requirements.

The Draw List opcodes represent actions to be taken by render. They are grouped into two broad categories: render state change and drawing operations. Render state changes encompass changes to current rendering parameters, such as transformation,

draw color, draw style, clip planes, and so forth. Draw operations consist of framebuffer clear operations, and actual rendering of RMprimitives.

**TABLE 3. Draw List Opcodes**

<b>Opcode</b>	<b>Description</b>
Push Attribute	Indicates a rendering stage change: new material properties, new lights, new clip planes, new draw modes (line style, line width), new polygon face mode, etc. In render, this opcode will produce a glPushAttrib() call. The push bitmask is a function of which modes change, and is computed as modes changes are dictated by the application. The minimal set of OpenGL attributes is pushed to restore the previous rendering state. The bitmask is contained in the RMnode referenced by the index field of the Draw List.
Pop Attribute	Indicates a return to a previous rendering state. In render, this opcode will produce a glPopAttrib() call.
Push/Load ModelView matrix	Indicates a change to the OpenGL projection matrix. In the current OpenRM implementation, cameras (2d & 3d) are the only scene parameters that can produce a change to the projection matrix (the transformation produced by a camera produces a modelview and a projection matrix). The index field of the Draw List refers to a matrix in the matrix table.  In render, produces a glPushMatrix followed by a glLoadMatrixf to the OpenGL projection matrix stack. Note that most OpenGL implementations have a projection matrix stack that is only two deep. Therefore, developers are urged to NOT have multiple cameras nested in depth in the scene graph. Multiple cameras at a sibling level in the scene graph (for multiple views) are OK.
Pop Projection Matrix	Indicates a return to a previous OpenGL projection transformation. Produces a glPopMatrix to the projection matrix stack in render.
Push Texture Matrix	Indicates a new transformation to be applied to the OpenGL texture matrix stack. In render, produces a glPushMatrix() followed by a glLoadMatrixf() to the texture matrix stack. The index field of the Draw List refers to a matrix in the matrix table that will be loaded
Pop Texture Matrix	The draw opcode will cause RMprimitives contained in an RMnode to be rendered. The index field of the Draw List refers to an RMnode owned by the component manager. All RMprimitives of the referenced RMnode will be rendered.
Framebuffer Clear	Causes a buffer clear operation to occur. A buffer clear could be one of: solid color fill to the color planes of the framebuffer; constant value clear of the depth buffer; painting/tiling of an RMimage to the color buffer; or painting/tiling of an RMimage to the depth buffer.  Framebuffer clears are logically considered as “scene parameters” of an RMnode. Unlike other scene parameters, the framebuffer clear does not result in a change to rendering state, and so this class of operations gets a separate opcode in the Draw List.

**TABLE 3. Draw List Opcodes**

<b>Opcode</b>	<b>Description</b>
Push Text Properties	<p>Indicates a change to the current text properties. The text property state is a stack that is internally maintained by render, and is completely external to OpenGL. A text property state change (new font, new size, etc.) a scene parameter, but does not affect any OpenGL state.</p> <p>In this implementation (Jan. 2001), this opcode is causes the text properties within the current RMstate to be updated (no text properties stack is implemented). When new text properties are encountered, at least two opcodes are generated by view: a “Push State” opcode (see below) causes the entire RMstate to be pushed, then the “Push Text Properties” opcode causes the text attributes within the RMstate to be refreshed.</p>
Pop Text Properties	<p>Causes render to pop the text properties stack. In the current implementation, this opcode is effectively a no-op. As the notes for “Push Text Properties” indicate, a change in text properties is encapsulated within a “Push State” / “Pop State” opcode pair. Thus, the text properties stack is effectively subsumed by the RMstate stack. This design was intentional, and produces an overall cleaner design with more predictable behavior, and fewer “moving parts.”</p>
Push State	<p>This opcode causes render to push the current RMstate onto stack that is maintained internal to render. “State” in this case means render “RMstate”, not OpenGL render state. RMstate is a superset of OpenGL rendering attributes, and is queryable by application callbacks. A change in any of the render state attributes (OpenGL attributes, cameras, transformations, text properties) will cause an RMstate state push.</p>
Pop State	<p>Inside render, causes an RMstate stack pop.</p>

### 6.3 View Stage Processing

The view stage of rendering performs a traversal of the scene graph, and creates a Draw List consisting of actions to be performed by the render stage of rendering. Each scene graph node is processed using an algorithm represented in the following pseudocode:

## FIGURE 2. OpenRM View Traversal

```
void rmViewTraversal(RMnode *n)
{
    If the node is not scheduled for processing return;

    Process view stage "pre-traversal" callback if present

    Compute bit flags for:
        Change in text properties, change in transformations, change
        in viewpoint, change in any OpenGL render state.

    If any bits set, load opcode pushing render state.

    If text properties change, load opcode pushing text properties state.

    If there's a change in transformations or viewpoint
    {
        Compute new matrices.
        Load opcode(s) to push change in matrix state for ModelView,
        Projection or texture matrix, depending upon which of
        these matrices change.
    }
    If there's a change in OpenGL state
        Load opcode to push the OpenGL attribute state.
    If there are any Framebuffer Clear operations for this node
        Load opcode to initiate framebuffer clear operations.
    If this node has any children
    {
        If this node has a switch callback
        {
            index = Evaluate switch callback.
            rmViewTraversal(child[index]);
        }
        else
            For each child, perform view traversal on child[i].
    }
    else
        If this node has RMprimitives, Load a draw opcode.

    Load pop opcodes for all push attributes generated by this node
}
```

### 6.3.1 View Frustum Culling

Frustum culling is enabled when the application attaches a view stage pre-traversal callback to an RMnode. The routine `rmNodeFrustumCullCallback` is provided by OpenRM (as part of the Phase I SBIR work) to perform frustum cull tests, and is intended to be used by applications for the purpose of enabling frustum cull testing at a specific RMnode. Note that view frustum culling is not enabled by default, but that applications must enable frustum culling by assigning a view-stage pre-traversal callback. Applications can provide their own custom view-stage culls, if desired.

### 6.4 Render Stage Processing

Render stage processing is simply a straightforward processing of the opcodes produced by the view stage. At this time (January 2001), there is no attempt to reorganize or optimize the contents of the draw list: there is no render state sorting, no migration of

framebuffer clears to the head of the list, no attempt to move texture loads to the start of the list, etc. Optimizations of this type will be performed during the Phase II effort.

## 6.5 RMPipe Processing Modes

Blocking vs. non-blocking frame rendering calls. Non-blocking requires full solution to frame-accurate data issue.

**TABLE 4. Multistage Render Processing Modes for RMPipes**

Processing Mode	Details
RM_PIPE_SERIAL	Uses pre-version 1.4 serial rendering code. Used internal to OpenRM for picking.
RM_PIPE_MULTISTAGE	View and render stages run sequentially. This mode is not multithreaded.
RM_PIPE_MULTISTAGE_PARALLEL	View and render stages run concurrently in separate <i>pthreads</i> . This mode can be used only when the application can relinquish control of the OpenGL rendering context.
RM_PIPE_MULTISTAGE_VIEW_PARALLEL	View and render stages run concurrently. Only view is in a separate <i>pthread</i> . Render is in the same thread as the caller. Use this mode when the application must retain control of the OpenGL rendering context. This mode is designed for use with Tiled display environments.

## 7.0 Offscreen Rendering

As a general rule, images created by applications that use hardware-accelerated graphics are resolution-limited. During onscreen rendering, the graphics hardware draws into a mapped window, and applications callbacks can be used to read pixels from the window and to save the pixel data to a raster file. The size of the window is constrained to the size of the display device (the screen).

The term *offscreen rendering* refers to the process of rendering to an offscreen buffer. In most instances, the maximum resolution of offscreen buffers is significantly greater than that for mapped windows. Combining increased resolution with hardware accelerated rendering, offscreen rendering can be useful for quickly creating extremely high resolution images. An offscreen rendering engine forms the basis for a remote rendering or visualization server.

During our Phase I work, we implemented support for offscreen rendering. Our work provides support for hardware accelerated rendering to an offscreen buffer. Our implementation provides offscreen rendering support using a single API that is consistent across Unix, Linux and Win32 environments.

## 7.1 Unix/Linux Implementation

GLX, the “glue layer” between OpenGL and the X11 windowing system, provides a set of APIs for creating offscreen rendering areas. All OpenGL programs that implement offscreen rendering will use these special offscreen drawables provided by GLX. The current version of GLX, version 1.3, unifies the process of creating OpenGL rendering contexts and OpenGL-capable drawables. However, GLX version 1.3 is implemented only on SGI systems<sup>3</sup>. The older GLX 1.2 API is ubiquitous and stable, but the APIs for creating drawables is somewhat more convoluted.

Our implementation assumes version 1.2 of the GLX API, which is supported on all platforms we encountered during development and testing (Linux, IRIX, Solaris). We create a GLXPixmap and use that drawable for all subsequent drawing operations.

For testing, we added a new demonstration program that exercises offscreen drawing features. This demonstration program creates and renders into a high resolution offscreen buffer (2048 by 2048 pixels). The resulting image is saved into a raster file on disk. During testing, we experimented with several image sizes and scene complexities, and conclude that offscreen rendering rates are comparable to those for mapped windows.

During testing, we discovered that offscreen rendering does not work at all with the current nVidia Linux OpenGL drivers<sup>4</sup>. We sent email to nVidia to report the problem - they responded by saying the bug would be fixed in the next release of the drivers. Note: this problem was resolved in March 2000 when nVidia released their 0.9-769 drivers for Linux.

## 7.2 Win32 Implementation

The general approach described above for Unix environments also applies to Win32. However, there is little published work describing how to implement offscreen buffers using OpenGL. The OpenGL “Red Book” [Woo99] provides hints, as do other sources, but none provide a complete code example.

Our implementation uses Win32 internals to create a “device independent bitmap” (or DIB), which is used as the offscreen drawable. Unlike the X11 implementation, the Win32 DIB code initialization sequence requires a regular Win32 window to be created, and the DIB inherits device characteristics from the window. The “mother window” is never mapped, as the normal Win32 event loop is never executed.

The demonstration program used for testing works as well in Win32 as it does in Unix/Linux: it renders using OpenGL hardware acceleration to produce a pixel map that is much larger than screen resolution, and writes the results into a raster file on disk.

---

3. Mesa has function prototypes for the GLX 1.3 API, but the routines are implemented only as stubs.

4. We were using the 0.9-6 nVidia OpenGL drivers, released in early January 2001.

### 7.3 Application Initialization: RMpipe Channel Formats

Applications that want to perform offscreen rendering can use a new `RMaux` routine to create the offscreen drawable, and must also use an `RMpipe` channel format that is compatible with an offscreen drawable. When `RMaux` is used to create drawables, the `RMpipe` must be initialized first, as the OpenGL rendering context is created first, then the drawable is created later.

To support offscreen rendering, during the Phase I work, we added three new channel formats, which are listed in the following table.

**TABLE 5. RMpipe Offscreen Rendering Channel Formats**

Format	Description
<code>RM_OFFSCREEN_MONO_CHANNEL</code>	For single pass, offscreen rendering.
<code>RM_OFFSCREEN_REDBLUE_STEREO_CHANNEL</code>	For red-blue anaglyph, multipass, offscreen rendering.
<code>RM_OFFSCREEN_BLUERED_STEREO_CHANNEL</code>	For blue-red anaglyph, multipass, offscreen rendering.

In addition to selecting the appropriate OpenGL visual, these channel formats also initialize a set of frame-related callbacks that are appropriate for offscreen drawables (e.g., `swapbuffers()`, etc.). These channel formats support monoscopic and anaglyph stereo rendering. There is no OpenGL support on any platform for offscreen multibuffered stereo rendering.

## 8.0 Tiled Displays

We use the term *tiled display* in a general way to refer to a collection of physically separate display devices that are used logically as a single display device. Tiled display devices are useful for creating a very high resolution display device using (relatively) inexpensive low-cost, and lower-resolution components.

We have deployed OpenRM in two specific tiled display environments: `CAVELib` and `VDL`. `CAVELib`<sup>5</sup> is a commercial software package used to build applications that run in the Cave Automatic Virtual Environment, or CAVE [Cruz-Niera93]. The Virtual Display Library, or `VDL`, is a software package developed by researchers in the ASCI/Views program at Lawrence Livermore National Laboratory. `VDL` is used in-house at LLNL as framework for developing applications to be run on any of LLNL's several tiled display devices.

Since there already exist several environments that perform tiled display abstraction and a framework for applications development (e.g., `VDL` and `CAVELib`), our approach is to

---

5. VRCO of Chicago, Illinois licensed `CAVELib` from the University of Illinois, and sells commercial `CAVELib` licenses. <http://www.vrco.com/>

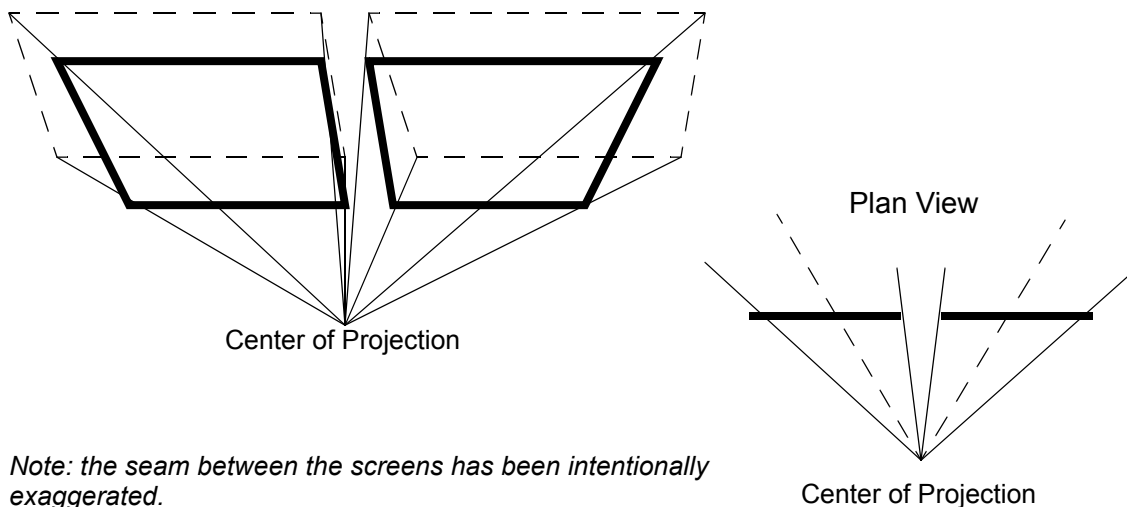
enhance OpenRM so that it is compatible with any such environment in a general way. This is a more cost-effective and a more sound technical approach: there is no need to create yet another such device abstraction layer; OpenRM (and scene graphs in general) should be divorced from event management, and in some cases, display device management<sup>6</sup>. Our approach leverages others' work, and introduces fewer "moving parts" into OpenRM. In the discussion that follows, we use the term **host environment** to refer to the software that performs tiled display device management and provides the application framework. We use this term to refer to both VDL and CAVELib.

In this section, we describe our implementation and the technical issues we faced in bringing support for tiled displays to OpenRM.

## 8.1 Problem Description

The following figure illustrates a simplified tiled display configuration: a powerwall consisting of two display devices.

**FIGURE 3. Tiled Display**



*Note: the seam between the screens has been intentionally exaggerated.*

In this example, the tiled display device is composed of two physical displays; a left and right display. The viewer is positioned at the point labelled "Center of Projection." A different view transformation is applied when rendering each scene. A useful metaphor is to think of looking through a pair of windows into a virtual world. There is one virtual world, and we see a slightly different view of it through each window (display device). The CAVE fits this metaphor, but the screens are arranged so as to surround the viewer, rather than being placed in a plane in front of the viewer as in the above example.

CAVELib and VDL both have a similar software architecture: after initialization, the host environment invokes an application-provided callback that performs rendering. In other

---

6. In contrast, Java3D implements a complete view and event model as part of their scene graph.

words, the host environment knows about all the individual displays that make up the tiled display system. When a frame needs to be rendered, the host environment computes the view transformation that corresponds to a given display, and invokes the application “draw function” to perform rendering. The host environment typically invokes all application draw functions in parallel in order to achieve the maximum possible frame rate.

### 8.1.1 Multiprocessing and Memory Management

The host environments differ in how they are implemented: CAVELib uses *fork()* to create (at least) one process per display, and VDL uses a threaded model to create a separate execution thread, one per display. In CAVELib, the application callbacks for each display device are in separate processes. This has a far-reaching implication for the application designer: the scene must be explicitly placed into a shared memory segment that is globally visible to all participating processes. A VDL implementation is much simpler, as all threads share a common memory address space: there is no need to place the scene, or other common information, into a shared memory segment: the VDL application need not perform any specific memory management, and can use normal *malloc()* and *free()* calls, as heap memory is globally visible to all draw threads.

### 8.1.2 OpenGL Rendering Contexts

When the application initializes the host environment, the host environment typically initializes each of the individual display devices. For each display device, the host environment locates a suitable OpenGL context<sup>7</sup> then opens a drawable on the display device. According to the rules of OpenGL and rendering contexts, a given OpenGL rendering context can be active only in one thread at a time, but can be used by multiple threads if the rendering context is used in round-robin fashion. In order to achieve maximum frame rates, most host environments open one OpenGL rendering context per display.

When the host environment computes the view transformation for each display prior to invoking the application callback, the host environment writes this transformation into the OpenGL transformation stack. In the simplest case, the application draw callback simply traverses issues OpenGL calls to render into the drawable. The view transformation computed (and loaded) by the host environment ensures that the contents of each individual draw callback are rendered from the correct perspective.

## 8.2 OpenRM and Host Environments

To begin our development, we first identified the “interfaces” between the host environment and OpenRM. We use the term interface in the sense of resources that are shared between the host environment and OpenRM, as opposed to API-style interfaces. These

---

7. In the Unix world, the OpenGL/GLX rendering context is usually created prior to creating a window. The application selects a GLX context by specifying attributes, such as single- vs. double-buffered, number of bits of color planes, depth buffer, whether or not the context will be stereo-capable, and so forth.

include the OpenGL rendering context, the drawable, the OpenGL transformation stack, and any special post-rendering activities, such as swapping buffers. These interfaces are summarized in the following table.

**TABLE 6. Interfaces between Host Environment and OpenRM**

<b>Resource</b>	<b>How Used</b>
OpenGL context	Created and made current by the host environment.
Drawable	The OpenGL-capable drawable is created by the host environment, and mapped onto the display device.
Transformation Stack	The host environment computes the view transformation that reflects the correct view transformation for a single viewer looking “through” a “window” on each display device.
Post-render actions	Buffer swapping for double-buffered rendering contexts. The host environment may try to coalesce all buffer swaps into a single point in order to have all display devices update simultaneously.

### 8.3 Implementation in VDL

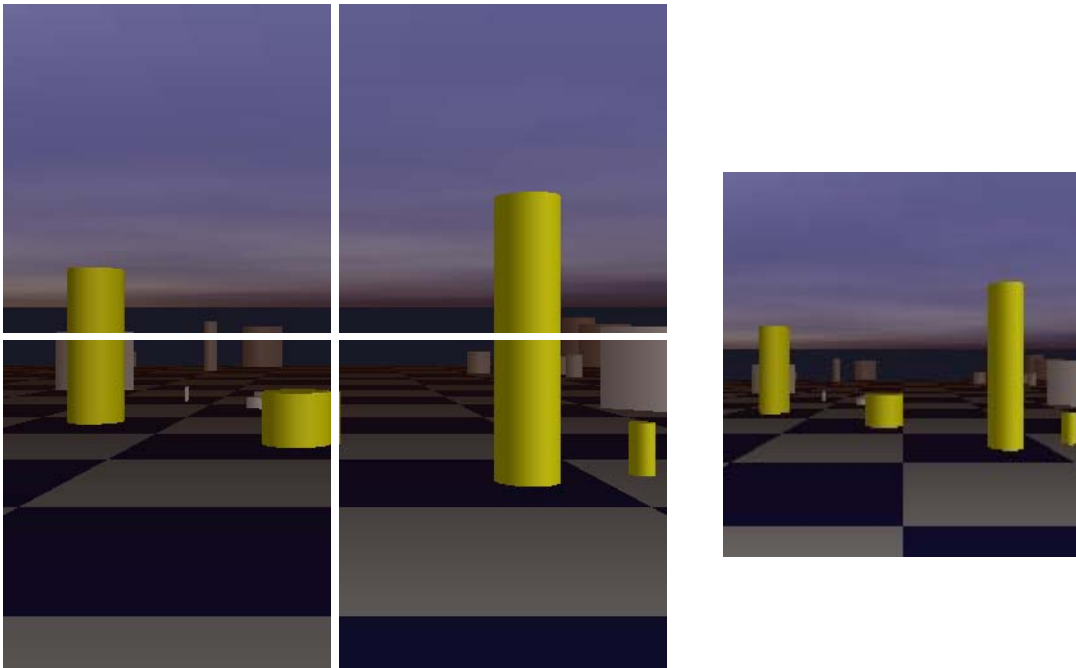
We created three sample applications that use VDL for tiled display management, and OpenRM for scene graph services. All VDL sample applications present a similar interface to the user: when the application launches, VDL opens windows on the tiled projection system, and OpenRM opens a single window on the main graphics console. All interactive transformations are performed within the OpenRM “navigation” window, and the VDL windows can be thought of as “slaves” that are refreshed when the navigation window is redrawn.

The first application made use of VDLs orthographic view model, and displayed a 3D cube. In this example, VDL computes and preloads a view and projection transformation onto the OpenGL transformation stack, then invokes the application callback. In this case, the application callback consists of a single call to the OpenRM frame rendering routine.

The second application is modification of the first, and makes use of perspective projection. This tests the accuracy of the sheared view frustum calculations provided by VDL. In this application, the 3D view transformation is computed by OpenRM, and multiplied with the shear transformation computed by VDL: VDL computes a shear-and-translation transformation for each tile. The shear transformation is loaded onto the OpenGL matrix stack, then the application callback is invoked. When the scene is redrawn, the view and projection transformation specified by the OpenRM 3D perspective camera is premultiplied with the shear transformation to produce the correct overall view transformation.

The third application builds upon the second, and consists of a “terrain flyover” where the viewer position is updated each frame. The flyover example uses extensive view-dependent operations to perform frustum culling and distance-based level-of-detail model switching [Clark76].

**FIGURE 4. VDL and OpenRM Flyover Example**



This flyover application uses VDL for tiled display management and OpenRM for data management and rendering. The four windows created by VDL are purposefully separated to emphasize that they are, in fact, four separate windows. The image on the right is from the “navigation” window. The shear transformation for each of the four VDL windows is computed by VDL, but then the view transformation is computed and loaded by OpenRM. Performing view transformations in the scene graph is crucial for operations that occur in eye coordinates, such as clipping planes and fogging (this example uses OpenGL fog).

During testing at LLNL, we ran this application using multistage and multithreaded processing (`RM_PIPE_MULTISTAGE_VIEW_PARALLEL`): view processing was in a separate thread, while render remained in the same thread as VDL (due to vagaries of OpenGL rendering context ownership). During this test, only those objects that passed the view frustum cull test were passed to render for display.

In these images, those cylinders that are within a certain distance to the viewer are displayed in yellow and are more finely tessellated, while those farther away are displayed in white and have only four sides. The frustum test was independently applied in each of the four display channels.

We have provided the source code for these examples to staff at LLNL. We have not made source for these VDL examples publicly available, as VDL is not publicly distributed outside of LLNL (to the best of our knowledge). LLNL does not promote VDL primarily for issues pertaining to software support. LLNL made VDL available to R3vis for the

purpose of testing OpenRM on LLNL equipment. In turn, we created sample code that shows how to use VDL and OpenRM together in a single application.

## 8.4 Implementation in CAVELib

Our CAVELib examples were deployed on a single display machine using the “simulator mode” in CAVELib. That means that CAVELib opens only a single window and causes drawing for all screens defined in the CAVE configuration file to be drawn into a single window. We obtained a time-limited development license from VRCO, Inc. at no cost for use in porting and testing purposes (VRCO wants to use OpenRM as its cross-platform scene graph in future commercial products, so has been helpful in providing resources).

We created two demonstration programs that use CAVELib for tiled display management. One shows a 3D object that spins, the other is a “virtual room” that uses multiple textures to provide some visual detail. In both cases, a single scene graph is created prior to CAVELib going parallel. When the application callback is invoked, the frame rendering call causes each rendering process to draw from the single scene graph.

FIGURE 5. CAVELib/OpenRM Demonstration



## 9.0 Conclusions and Future Work

In this document, we have provided detailed technical information that documents our activities performed during a Phase I SBIR grant. We extended an Open Source scene graph toolkit along four developmental axes. We implemented thread safety in a scene

graph tool by migrating state information out of the scene graph in order to support multiple simultaneous reader and writer threads. We implemented a multistage and multi-threaded rendering engine using a two-stage rendering decomposition. The multistage design improves rendering performance by reducing the load on the graphics pipe in two ways: the rendering stage has less processing work, as transformations and other operations are performed during the view stage, and the actual number of polygons to be drawn is potentially reduced by view frustum culling. Third, we have implemented support for OpenRM use in tiled display environments, and have made demonstration code available to stakeholders. Fourth, we have implemented offscreen rendering support.

There are two technical areas that require further development before the implementation is fully complete. The first is to fully implement “frame accurate” scene graph data. The nature of this problem stems from the fact that scene graph data travels down the multistage rendering pipeline: while the view stage is processing data for frame  $N$ , the renderer is processing data from frame  $N-1$ . Unless data is somehow protected while in transition from view to render, it is possible for the application to corrupt data while in transit. Our Phase I implementation provides for frame accurate transformations, but all other data is subject to corruption. The second topic is to fully implement a dynamic component manager that will grow the size of the scene graph object pool, and will propagate those changes to all active context caches.

## 10.0 Acknowledgement

This work was funded by the U.S. Department of Energy, Office of Science, through the Small Business Innovation Research (SBIR) program under contract number DE-FG03-00ER83083.

## 11.0 References

[Clark76] J. Clark, Hierarchical Geometric Models for Visible Surface Algorithms. Communications of the ACM, 19, 10, pp 547-554. October 1976.

[Cruz-Niera93] C. Cruz-Niera, D. Sandin, T. DeFanti, Surround-screen Projection-based Virtual Reality: The Design and Implementation of the CAVE. In Computer Graphics, *Proceedings of SIGGRAPH 93*, August 1993.

[Schikore00] D. Schikore, R. Fischer, R. Frank, R. Gaunt, J. Hobson, B. Whitlock, “High-resolution Multi-projector Display Walls and Applications”, IEEE Computer Graphics and Applications, Vol 20, 4:38-44, Jul./Aug., 2000.

[Woo99] M. Woo, J. Neider, T. Davis, D. Shreiner, “OpenGL Programming Guide, Third Edition”, Addison-Wesley, 1999. (*The Red Book*)