

---

# *RM Scene Graph Technical White Paper*

**December 1999**



*Wes Bethel, [wbethel@r3vis.com](mailto:wbethel@r3vis.com)*

**R3vis Corporation**

**P.O. Box 979**

**Novato CA, 94948**

**(415) 898-0814**

**[www.r3vis.com](http://www.r3vis.com)**

---



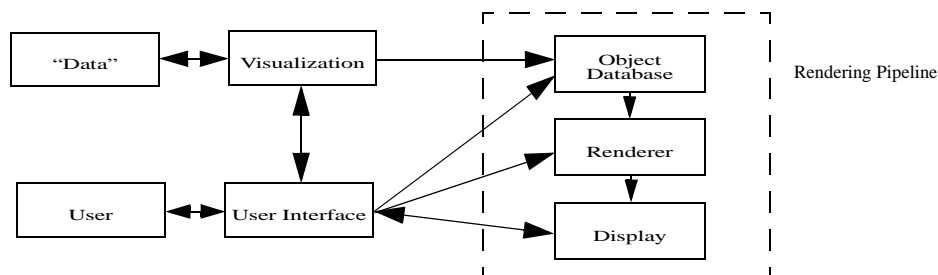
## 1.0 Abstract

This white paper provides a technical overview and outline of RM Scene Graph<sup>1</sup>, a scene graph API developed by R3vis Corporation that is used by developers to create high performance, portable graphics applications that uses OpenGL<sup>2</sup> as the underlying graphics platform. We first introduce fundamental concepts, such as the definition of a scene graph, and discuss the relationship of scene graph software to applications and graphics platforms. With these basic definitions, we describe the technical design goals used to create RM Scene Graph.

## 2.0 Introduction

Whether they are first-person games, flight simulators or stock price chart generators, all graphics applications have a number of common components. A *graphics platform*, or rendering engine, transforms or renders mathematical descriptions of surfaces into arrays of pixels, or images, that are viewable by a human. An *object database* is a data source for the rendering engine. The object database is not a database in the traditional sense, but is a set of data structures that are designed to hold information about the things to be rendered. Together, the graphics platform and the object database, along with some means to display images, form the *rendering pipeline*.

**FIGURE 1. The Rendering and Visualization Pipeline.**



Outside of the rendering pipeline, additional processes and structures combine to reflect the design goals of a particular application. Visualization applications first transform abstract data, such as pressure or temperature, into renderable geometry using visualization techniques such as isocontouring, then pass the results on to a rendering pipeline that in turn creates an image. Stock price chart generators sift through and subset from mountains of historical data, providing a small relevant portion to a visualization tool (a chart generator) that in turn creates renderable geometry (positions and heights of bars) that is in turn transformed into an image. Computer-based games often employ extreme optimizations that result in a compact and quickly-rendered object database.

It is often the case that human user is part of the entire feedback loop. The user manipulates rendering controls to see the data from a different position or orientation. The user can manipulate controls on data selection and subsetting tools, as well as manipulate controls on the visualization techniques themselves. Indirect user manipulation can occur in some applications when the data being accessed changes as a function of a user action. For example, some terrain visualization systems, such as flight simulators, need not keep data for the entire globe in memory at once. Only high-resolution subsets of the globe are needed in the vicinity of the user. Elsewhere, low resolution models will suffice. As the user navigates over the terrain surface, the data changes as a function of view. In these applications, specific render state information, such as viewer's position and orientation, is used to subset and select from a larger model, then visualization or other operations are performed on the data to transform it into renderable geometry in the object database, where it is then rendered by the graphics platform.

1. RM Scene Graph is a Registered Trademark of R3vis Corporation.

2. OpenGL is a Registered Trademark of Silicon Graphics, Inc.

## 2.1 What is Scene Graph Software?

In a nutshell, scene graph software is a set of data structures and support software that are used to implement the object database portion of a graphics application. There are a number of hallmarks of scene graph software. First is the concept that the scene graph data structures are designed to be optimal for rendering, as opposed to optimal for searching or editing once created. Second is that the scene graph software provides some additional value above and beyond the graphics platform itself. The graphics platform is a rendering engine or platform, such as OpenGL. The interface to graphics platforms is often simplistic and detail oriented, reflecting design goals that emphasize maximum rendering performance. Programming interfaces to graphics platforms are often verbose in terms of the number of lines of code needed to draw a pixel. Scene graph software is often characterized as a set of tools that extend the base functionality of a graphics platform in a number of ways, most often by providing complex services that are encapsulated behind a simpler and more compact interface. Additionally, there may be fundamental primitive types present in the scene graph model that are not present in the graphics platform, such as procedural surfaces or volumetric objects. A powerful use of scene graph models stems from the ability to create a pseudo-framework for performing view-dependent operations, such as changing the object database as a function of viewpoint.

Often, scene graph tools will provide some amount of window system abstraction. Graphics platforms such as OpenGL traditionally don't include support for window systems or event management. This is for good reason: window systems are complex and have many moving parts. Support for things like fonts and input devices and file browsers are orthogonal to high performance graphics systems. So, while scene graph systems may provide a thin veneer of window system abstraction, most often their focus is on the data structures and processing needed to support high performance as well as platform independent graphics rendering.

An interesting characterization of scene graph software is "just add data, and you have an application." There is a little more to it than that, but that one sentence accurately captures the essence of scene graph software. In this case, the additional details that must be attended to by applications include interfacing to the native window system for window and event management, as well as shepherding data into and possibly out of the scene graph system.

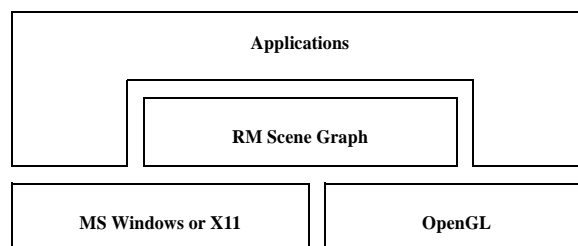
From a design perspective, scene graph models are often thought of as trees. Leaf nodes in the tree contain renderable data, such as triangles or line segments. Interior nodes contain render state information, such as object color, light sources, cameras and so forth. Scoping rules are used to confine the propagation of render state parameters only to descendants in the tree, but not to sibling branches or ancestors. Not all scene graph systems are implemented in this fashion. There is variation from package to package concerning scoping rules, traversal order within a given node, the degree to which data within a node may be modified once created, and so forth.

The fundamental usage model for scene graph systems is that applications first create a "scene graph" using tools provided by the system, populate the tree with data, then ask the system to render the tree. Such a model is intuitive and easy to use. Applications developers benefit from this model because they have a single interface to regardless of the underlying platform or of the specific application domain. Stock chart generators can use the same scene graph model just as well as an immersive virtual reality exploratorium.

## 2.2 RM Scene Graph

RM Scene Graph (RMSG) is a general purpose scene graph tool created by R3vis Corporation. Like other scene graph tools, RMSG is *middleware* that is architecturally positioned between high-level applications and low-level graphics platforms such as OpenGL (Figure 2). Like other scene graph systems, applications use RM by first creating a scene graph, populating it with data, and then requesting rendering of objects in the scene.

FIGURE 2. RM Scene Graph, OpenGL and the Windowing System.



A substantial number of considerations have influence the RMSG design resulting in a powerful and general purpose tool that exceeds the capabilities of other scene graph systems. The balance of this document serves to elaborate upon the RMSG design goals and considerations. These goals can be decomposed into two broad categories. In Section Three, we consider those goals that influence the design of the scene graph itself, including the application interface. In Section Four, we build upon those fundamental scene graph design considerations to define goals that address the needs of high performance and general purpose rendering. Complementary to the core scene graph technology, Section Five describes a window system abstraction layer that may be of interest to some applications developers, but this layer is not a part of the scene graph core technology. Similarly, Section Six describes a collection of tools for scientific visualization that are intimately compatible with the core RMSG technology. In Section Seven, we discuss the future of RM Scene Graph, including OpenRM, an Open Source sibling of RM Scene Graph.

### 3.0 RM Scene Graph Design Goals

The intended use of a scene graph model has a substantial impact upon the design of the model. A scene graph model designed for maximum performance will likely emphasize the use of simple graphics primitives such as triangles, and will discourage, if not prohibit, user code that extends the system in the primary rendering execution path. Among our primary design considerations is to balance flexibility with performance. To that end, RM is extremely flexible yet efficient. Applications may extend RM using application defined primitive types, including raw OpenGL code as well as user code that can affect or change the render time traversal order of the scene graph. A flexible shared data management framework promotes efficient use of system resources and avoids the creation of unnecessary copies of data. The scene graph node design encourages aggregation of primitives at the node level. Such a design promotes overall efficiency. Access to OpenGL's numerous rendering parameters is managed by RM in an intuitive and easy-to-use framework. A rich set of primitives take RM beyond a simple "triangle engine," and provides direct support for contemporary rendering techniques including volume rendering and image based data. RM is written in the C language, and is intended to be used from C and C++ applications.

#### 3.1 Extensibility

Broadly speaking, an *extensible* system is one that gracefully accommodates new and unanticipated functionality that comes in the form of application supplied extensions or modifications to the behavior of the base system. One useful way to extend a rendering system is to include application rendering code that co-exists with the scene graph system. Another useful extension is the ability for applications to modify the scene graph traversal path at render time.

An application can use one of the many RM-defined primitives, or can extend RM to include a "user defined primitive" within a scene graph node. The user defined primitive consists of an callback that will be invoked during scene graph traversal. The callback will be provided parameters that include a handle to the scene graph node and a handle to an object containing the current render state. The render state is a snapshot of all rendering parameters at that point during scene graph traversal, including model and projection matrices, their inverses, window and viewport size, current material properties, polygon rendering modes, the current stereo channel being rendered, and many other parameters.

The callback associated with the user defined primitive is the primary framework RM provides for placing raw application OpenGL rendering code<sup>3</sup> into the scene graph. Because of this design feature, RM is completely application-extensible at the primitive level. In addition, for those developers who want a "quick and dirty" route to write OpenGL code, this approach is nearly ideal, since all internal render state management is performed by RM. In other words, the scene graph subsystem performs the "dirty work" of setting up lights, camera position and so forth and the developer can focus just on the OpenGL draw code.

In addition, there are other not-so-obvious benefits of this design choice. With an application rendering callback, it is possible to render essentially an infinite amount of data, subject to a caveat. When you hear claims that such-and-such graphics library can render 2 giga-triangles, you should think "immediate mode rendering." The appli-

---

3. This type of architecture is similar to that of the CAVE library. The CAVE library manages the model and projection matrix stack, and invokes user code to actually perform the rendering. However, the CAVE library itself does not implement any rendering services - it is designed as a framework for interfacing to VR devices.

cation callback will rely upon a z-buffer for occlusion resolution, thus only opaque data will be rendered correctly. Since current graphics hardware is not capable of sort-last<sup>4</sup> rendering of semi-transparent data, it is crucial that semi-transparent data be rendered in either front-to-back or back-to-front order. This necessitates sorting the data before sending it to the graphics pipe (sort first). Such a sort, in the application callback, is the responsibility of the application code. The clear trade-off is that performance may be an issue when such processing is placed directly in the rendering execution path.

During rendering, objects in the scene graph are rendered according to an order defined by a top-down, left-to-right traversal. Applications can extend the base scene graph model through the use of callbacks that can modify the scene graph traversal order at render time. Level-of-detail or model switching can be performed by the application using this mechanism by first examining the render state parameters, such as the current transformation matrix or its inverse, then making a decision as to which child of a given scene graph node to render based upon distance to the child, projected screen area, or some other view-dependent criteria. This type of “select one child for rendering” activity is commonly called a “switch callback.” Similarly, a callback that is positioned at the start of when a given node is processed can be used to abort further processing of the subtree containing that node.

### 3.2 Shared Data Management

One detrimental hallmark of other scene graph systems is they make a copy of application data. When a triangle list is added to the scene graph, the system creates a copy of the triangle list. Multiple unnecessary copies of data is a preventable source of system inefficiency. One solution that avoids creating multiple copies of data is the notion of *sharing data* between the application and the infrastructure. In such a model, the scene graph system would not make a copy of data unless directed to do so by the application. The proposition of sharing data raises several implementation issues, such as when is it safe to free the data, how the application signals to the scene graph system when the data has changed, and so forth.

When building the scene graph, the application may choose to allow RM to perform data management (of scene graph or geometric primitives), or it may choose to manage the data. *When the application manages the data, there is no need for multiple copies of the data in RM.* Borrowing a concept from current data modeling efforts, RM models geometric primitive data (vertices, colors, normals, texture coordinates) with *data blobs*. Data blobs are essentially large blocks of homogeneous data; large flat arrays. Separate objects hold the metadata, or information about the blobs. The blob model facilitates data sharing between cooperative processes, efficient data transfer, and modular coding interfaces.

In the visualization pipeline model, the visualization tool generates the raw blob data as part of the visualization process. For example, an isosurface algorithm generates a surface that is a three dimensional contour within a scalar field. It is common to represent such a surface with a collection of disjoint triangles or triangle strips. The geometric vertex data, normals and optional surface colors can each be aggregated into a data blob. Since this data may be potentially quite large, duplication should be avoided. A close coupling between visualization tools, scene graph software and rendering engine is required to avoid data duplication.

### 3.3 Nodes and Primitives and Scene Graph Topology

As previously stated, the scene graph can be thought of as a tree-like structure; a directed, acyclic graph. Nodes contain renderable data and/or scene parameters. Renderable data may consist of surface geometry, vector data, or image based data. Scene parameters include viewer positions, lighting models and lighting environment parameters, textures and texture environment parameters, polygon fill mode, and many other parameters.

In RM, only leaf nodes of the graph may contain renderable data. More precisely, only leaf nodes will have their underlying primitive data rendered. However, all nodes, interior nodes and leaf nodes alike, may contain render state parameters. There is only one type of node in RM Scene Graph. There are no group nodes, switch nodes, material nodes and so forth. All of these features are available to all RM scene graph nodes. Inside the RM scene graph

---

4. By “sort-last”, we mean the process of ordering drawable objects from front-to-back or back-to-front for the purpose of alpha blending. In other literature, the terms sort-first, sort-middle and sort-last refer to the process of partitioning primitives into groups that are rendered in parallel. In this latter usage, a better term might be partition-first, partition-middle and partition-last since the activity is designed not to *sort* but to *partition*.

node, render state parameters and primitive data are implemented in such a way that when no features are used, no memory is consumed. The RM node is a compact object that expands only when the application requests more features.

RM nodes may contain renderable data in the form of a list of *primitives*. A single primitive is a collection of homogenous objects, such as an array of spheres or disjoint triangles of an arbitrary length, or a single triangle strip of arbitrary length. The collection of primitives at an RM node may contain a heterogeneous mixture of fundamental primitives. In other words, it is possible to have at a single node primitives consisting of triangle strips, quad meshes, sprites, two dimensional text and so forth. Whereas it is possible to add and delete children of a given node, the programming interface used to manipulate the primitive list at a node is much more limited. Applications may only add primitives to a node, or obtain the handle of one of the primitives at an RMnode. The intent is to encourage applications to create a compact scene graph wherever possible. Some types of optimizations, such as primitive grouping and aggregation, are better performed by the application than scene graph system.

### 3.4 Scene Parameters

Rendering parameters, or *scene parameters*, are those that have a “global” effect on rendering over an entire subtree. Scene parameters include two- or three-dimensional cameras, “global” object color, current texture and texturing environment parameters, current font and other text rendering properties, polygon fill mode, lights and lighting models, viewport, background color or background image, clipping planes, and so forth.

Scene (rendering) parameters should have scope only over the subtree rooted at the node containing the scene parameter. Scene parameters should not have any effect on sibling nodes. The primary significance of parent-child relationship in the scene graph is the notion of the *scope* of scene rendering parameters. Scene parameters have effect *only* within the subtree in which they reside. Therefore, a camera must be placed at a “more shallow” level in the tree than a node containing renderable geometry. Since a given node may have more than one parent, it is possible to have multiple cameras refer to the same subtree of geometry. Texturing (2D or 3D) is active when there is a combination of texture coordinates within a given primitive, and when there is an active texture. Since only one texture can be active at any given instant, the placement of the texture within the scene graph defines which texture is applied to any given primitive.

Any scene graph node may contain scene parameters as well as geometric primitives. Such a methodology contrasts with other scene graph designs in which each node contains either a single render state parameter or a single type of primitive. Rendering and processing is accelerated when the state changes induced by scene parameters are aggregated. Such aggregation is under application control: multiple state changes can be placed into a single scene graph node. The overall result is a simplified and more streamlined scene graph design that places emphasis on the *batching* of data for rendering, including both raw geometry data as well as rendering parameters.

### 3.5 Rich Set of Primitives

RM provides a set of primitives that directly support scientific visualization as well as many common operations in computer games. In addition to the primitives commonly supported by low-level graphics APIs (points, lines and line strips, triangles and triangle strips, quads and quad strips), support is provided for a number of procedural 3d surfaces: spheres, cylinders, cones, 3D boxes; image-based data including bitmaps, pixmaps/sprites, 3D image (volumetric) data and images consisting of color and depth data; 2D bitmap text and glyphs; semi-procedural 2D geometric markers; an “octmesh” (analogous to a structured hexahedral mesh) for volume rendering.

### 3.6 C Interface

RM Scene Graph is written in the C programming language, and is intended to be used by C and C++ applications. Why the C-based interface, since many other scene graph APIs have been written in C++? The primary reasons for a C-based interface include portability, simplicity and performance. There is a wide variation in vendor implementations of the C++ specification, resulting in code that runs only on some platforms depending upon the usage of certain C++ constructs, such as arrays and templates. C is arguably a simpler programming language model, and RM has been written to be “object like” in the sense that applications do not directly access fields within structures, but use a programming interface to access this data. For this reason, the same object oriented concepts that apply within C++ apply in the C-based RM interface. In the C language, memory management is explicit rather than

implicit in the language specification. For this reason, C-based programs often exhibit better performance than their C++ counterparts, but at the cost of burdening the implementation with careful memory management.

## 4.0 RM Scene Graph: Rendering Design Goals

The RM renderer is designed to achieve a balance between flexibility and speed. Configurable multipass rendering is used to display scene containing both geometry and volumetric data, but also to support stereoscopy and integration of both 3D and 2D data and view models into unified scene graph model. The renderer provides direct support for common visualization primitives, including image and volume based data, including “deep images” to support image-based rendering. To maximize rendering speed, the RM renderer aggressively uses display lists whenever possible to realize the benefits of retained mode rendering. The design philosophy for RM’s rendering engine reflects a balanced approach that blends the needs of visualization with a high-performance, yet portable and stable renderer.

### 4.1 Configurable Multipass Rendering

Multipass rendering refers to the process of making multiple passes through the object database using some criteria to select objects for rendering during each rendering pass. One pass may render only 3D opaque objects, another pass may render only semi-transparent 3D objects, while another pass may render only 2D data, or objects that can be seen only in the left or right channel of a stereo rendering pipe. Multipass rendering is necessary to render certain types of scenes. Those scenes that contain a mixture of opaque and semi-transparent objects require multipass rendering in order to achieve a correct final appearance; opaque objects are rendered first, followed by semi-transparent objects. A user-configurable multipass rendering framework allows an application to specify the order of rendering passes.

### 4.2 Stereoscopy

Studies have shown a significant and measurable increase in comprehension of depth relationships when the viewer is presented with a combination of motion parallax and stereo image pairs as compared to viewing the same scene rendered using a monoscopic camera. RM provides direct support for stereoscopic rendering. *Anaglyphs*, or images rendered in different colors to present an impression of three dimensions, are directly supported by RM, as are *multibuffered* stereo image pairs. RM can directly generate anaglyph stereo pairs on any framebuffer supported by OpenGL. Multibuffered stereoscopy is supported whenever supported by the underlying graphics hardware. RM also provides support for per-frame output of pixel buffers, both in anaglyph and multibuffered stereo rendering modes, and has been used to provide input to MPEG encoders to make stereoscopic movies for publishing on the web. With the RM source distribution, the frame rendering code can be modified and extended to support arbitrary pairs of anaglyph colors. The default anaglyph color pairs supported by RM are Red/Blue<sup>5</sup> and Blue/Red.

### 4.3 Multiple Windows and Graphics Pipes and Threads

A variety of workstation vendors provide hardware that use multiple processors and graphics pipes. This class of machines has been used for multi-display environments, such as the CAVE and the Powerwall. RM provides direct support for multiple graphics pipes on SMP architectures. Multiple OpenGL rendering contexts can share display lists and texture object indices, but can be executed by separate rendering threads. RM allows multithreaded applications to simultaneously traverse one scene graph, each rendering with a different OpenGL context which share display lists. RM is thread-safe, and has been tested with pthreads. All concurrent application rendering processes that invoke the RM rendering engine must share a common address space, as is the case with pthreads and other threads models. RM does not use System V shared memory segments to share data between processes, and is not inherently a parallel rendering system. Parallelization is properly the domain of the application designer. In the future, RM may pipeline certain operations with a threads programming model.

Each pipe in a multi-pipe system can be configured for monoscopic or stereoscopic rendering independent of the other rendering pipes. While Win32 applications are incapable of supporting multiple graphics pipes, application code based upon RM will transparently scale from the laptop to the multi-piped SMP.

---

5. To maximize the use of color bandwidth, RM uses Red/Cyan, or Cyan/Red. Without the use of the green phosphor, the “blue” channel is “too dark” and is overpowered by the red channel.

## 4.4 Frame-Based Output

At times it will be necessary for the application to obtain the pixels from the framebuffer, perhaps for the purpose of saving an image file or for reuse later as part of an image-based rendering application. Applications may specify a callback that will be invoked after a frame has been rendered. When invoked, the callback will be provided a handle to an *RMImage* object that contains the framebuffer pixels. Applications may request that the *RMImage* object contain data pixel data from the framebuffer color planes, the depth buffer, or both.

The second form output from RM is true vector PostScript<sup>6</sup>. Using a proprietary algorithm, the entire scene is “rendered” to a PostScript file. The contents of the PostScript file is not simply a raster dump of the framebuffer. It contains a true vector-based, display-device independent representation of the scene. Due to significant functionality differences between OpenGL and PostScript, it is not possible to capture all aspects of the scene graph. For example, PostScript does not support any notion of texturing or Gouraud shading. Other features, such as transparency, are very difficult to implement in PostScript. The vector PostScript output option was created at the request of our customers as a way to generate publication quality “business graphics.”

## 4.5 Imaging

Image data and operations on image data are often shortchanged in scene graph architectures. RM provides support for image data as first class objects in the scene graph. An area of growing emphasis for RM in the future will be direct support for hardware-accelerated image operations using the graphics pipeline itself. The new OpenGL 1.2 specifications reveals an API to support imaging operations, such as convolution. At this time, RM implements rudimentary imaging operations, and gives the application flexibility and control over how the raw pixel data in image is managed as it flows through the OpenGL imaging pipeline.

The OpenGL architecture uses a multistage rendering and imaging pipeline. Early stages in the pipeline perform geometric operations, such as vertex lighting calculations, vertex transformations, and face culling. Later stages in the pipeline operate on pixel fragments, performing operations such as alpha blending and texturing. Operations on raw pixel data include conversion to floating point, manipulation with scalar operators and color mapping. Many contemporary OpenGL implementations support both classes of operations (upon geometry and pixel fragments) in hardware. RM supports a number of imaging operations using OpenGL, thus these imaging operations may be accelerated by hardware (depends upon the OpenGL implementation).

OpenGL requires the size of image texture data, including three dimensional texture volumes, to be an even power of two. Rarely is source image data of such a size, hence one task which is common across many applications is the need to resize images to meet OpenGL’s texture image size criteria. RM implements an image resize operator through OpenGL, and can achieve dramatic throughput and processing rates when supported by hardware. One early test of this feature in RM called for resizing a source volume from 1.5Gbytes to 256x256x256. Using the imaging pipeline, RM performed this resize operation in just a few seconds; the same operation in software took many minutes.

RM supports control over pixel data expansion and format conversion within the OpenGL pipeline. Pixel data is first converted to floating point format by OpenGL as it enters the pipeline. Then, OpenGL applies a scale factor then adds a bias to the raw floating point pixel data. RM includes these scale and bias parameters as part of the RM image object. Next, pixel data is optionally transformed by an RGBA to RGBA color table lookup. This color table is also a parameter of the RM image object. These imaging transformations apply to all RM image objects, whether they are used as textures or as sprite primitives or background images.

Image *sprites* and bitmaps are directly supported as first class renderable objects. Bitmaps and sprites are both pixel arrays, the difference being that bitmaps are binary pixel values packed into bytes (one bit per pixel), whereas sprites may be true-color or index-color pixels, with or without depth information. RM image objects may also be used as background “wallpaper.” These background image tiles may or may not contain depth values.

One of the many demonstration programs included with the RM distribution renders a complex molecule using a ball and stick model. When the left mouse buttons is clicked, the program first renders the molecule using only a ball model, then reads the resulting color and depth pixels using a post-render callback. The resulting image,

---

6. PostScript is a registered trademark of Adobe Systems, Incorporated.

including the depth pixels, are reused as a background image and the scene is rerendered with just the stick model. The following interactive transformations of the stick model occur much more quickly than when rendering the ball and stick model, yet the stick model appears to rotate in and out of the static ball model. This is a dramatic demonstration of the power of combining image and geometry based rendering.

## 4.6 Volume Rendering

RM provides support for hardware accelerated direct volume rendering. Hardware acceleration is achieved through the use of a combination of three dimensional texture memory, textured geometry and alpha blending<sup>7</sup>. Modern scientific visualization applications often require rendering a mixture of surface geometry and volumetric data. In many instances, the underlying volumetric grid can be represented as a structured hexahedral mesh. RM uses the notion of a structured hexahedral mesh, called an *octmesh*, as its volumetric primitive (a structured 3D grid of voxels). The application can specify either the minimum/maximum extents of the grid, along with the number of evenly-spaced grid points along each axis, or can use a rectilinear description by providing explicit spacing along each axis.

RM provides direct support for both true-color and index-color volume data. Index-coloring occurs “for free” through the parameters of the RM image object - the RM image object may contain three dimensional image data. RM implements an object called a *visualization colormap*. This object has both a colormap, along with a transfer function which describes how raw scalar data is converted into RGBA-tuples. Or, the application can choose to have render the scalar volumetric data directly.

Data management of volume data can be problematic, especially with time-varying volume data. With RM’s data model, the application can manage the raw volume data, thereby avoiding the creation of multiple copies of volume data. The raw volume data can be modified by the application on-the-fly, thereby providing the means to render time-varying volume data.

## 4.7 Text

One notable shortcoming in OpenGL is the lack of direct support for the rendering of 2D text. One reason for this shortcoming is due to the fact that fonts management is the purview of the window system. RM fills this gap by providing a uniform interface for creating and rendering two dimensional, bitmapped text that may be positioned in either 2D or 3D. A set of enumerated font types, sizes and styles are provided by RM, and produce consistent results in both X and Windows.

## 4.8 Retained Mode Rendering

The foundation of high-performance rendering is built upon retained mode rendering operations. However, management of retained mode rendering is the domain of the rendering engine, not the application. In other words, the application should not be burdened with the task of directing the renderer to build and maintain retained mode rendering constructs, such as OpenGL display lists, or to signal the renderer to regenerate the display lists whenever the underlying data changes.

RM relieves the application of this burden by automatically building and maintaining display lists wherever possible. The application may override this behavior on a per-primitive basis, thereby preventing RM from building display lists if it is known that the underlying vertex or color (or other) information will be changing from frame to frame. No elaborate “primitive compilation schemes” or “capability bits” are required. The OpenGL model rewards this effort with dramatically accelerated rendering rates, particularly over a network connection. RM places as much information as possible into display lists, including textures and other image-based data to get the most performance from the rendering hardware. When applications update the vertex, color or normal information in an RM primitive, the underlying rendering machinery detects the change and forces a regeneration of display lists for that primitive.

---

7. Three dimensional textures are part of the OpenGL 1.2 specification, but are an extension in the 1.1 specification. Not all OpenGL implementations support the 1.2 specification nor the three dimensional texturing extension (such as Win32), hence do not support hardware accelerated volume rendering.

## 4.8.1 Transformations

A by-product of the tree-like structure and the scoping rules of rendering parameters, transformations accumulate along tree depth. All other rendering parameters accumulate in a top-down fashion. A top-down accumulation means that settings at lower levels in the tree override settings at higher levels. For example, if the root node of the tree specifies the object color is green, but a lower level node specifies the object color is yellow, the object will be drawn as yellow, not green. Transformations accumulate from the bottom of the tree upwards, so those transformations applied lower in the tree are applied before transformations higher in the tree.

RM provides a number of ways to specify a transformation. It can be as simple as a single 4x4 matrix, or much more complex. The complete RM transformation can be represented using the following matrix product:

$$Pre \times -C \times S1 \times R \times S2 \times C \times Post$$

The elements comprising this composite transformation are:

1. *Pre* - a 4x4 transformation matrix applied prior to any other transformation.
2. *-C*, *C* - a 4x4 transformation matrix that first translates the center of the object to the origin (*-C*), then back again (*C*). The matrix is derived from the center point parameter of the RM scene graph node. RM provides tools that compute the bounding box of a node, or a subtree. The node's center parameter need not be the actual center, but is used as the center point about which scaling and rotation occur. These translation matrices are not set by the application directly, they are derived from the node's center point. The application may set the center point directly.
3. *S1*, *S2* - These 4x4 matrices represent scale matrices. One is applied prior to rotation, the other after rotation. Applications specify the entire matrix, so these need not be scale matrices in a strict sense.
4. *R* - a single 4x4 rotation matrix.
5. *Post* - a single 4x4 transformation matrix applied after all other transformations.

Internal to RM, the composite transformation is implemented as a product of matrices. The application may specify any or all of these matrix parameters at an RM node. Like other node parameters, when no matrix is specified, no extra memory is consumed. If only one of the matrices is specified, the others are effectively set to the Identity matrix for the purpose of computing a transformation. The only exception are the *C* and *-C* matrices, which are derived from the node's center point.

## 5.0 Window System Issues and RMaux

OpenGL is a window system independent graphics API, but software that uses OpenGL for rendering must be window system aware, at least for the purposes of initializing OpenGL. In addition, interactive applications are often under user control. The most common input devices are mice and keyboard, and most often these devices are accessed through the windowing system. Therefore, most graphics applications face issues with regard to window creation and placement as well as event management.

One possible solution would be a complete window system abstraction layer that would provide a mediating layer between applications, the window system and OpenGL. Another is to provide absolutely no window system tools whatsoever. The former option forms the basis of several software companies, and is beyond the scope of a scene graph tool. The latter option has the potential for introducing complexity into the application development process. A reasonable compromise takes the form of an auxiliary library that provides an API to common window system features such as window creation and event loop management. RM provides such a library that represents a thin veneer of window system abstraction. This auxiliary library, called *RMaux*, can be used to create windows and to manage input events. A significant design goal of *RMaux* is to facilitate a wide range of application needs, from "quick and dirty" applications with minimal user interfaces, through those applications that have a significant investment in window system user interface code.

The *RMaux* library provides a uniform interface to a minimal set of window system services that are essential for interactive graphics applications. Using *RMaux*, applications can create windows suitable for rendering. Nearly all interactive applications that execute within a windowing system use an *event loop*. The event loop detects window system events, such as mouse motion or button presses, and responds to those events. *RMaux* provides an

event loop handler, as well as a framework for either using a set of predefined callbacks, or application-supplied callbacks. Alternately, applications can use their own event loop and simply invoke RM's frame-based rendering tools at the appropriate time. The RMaux default user interface model implements a virtual trackball interface for rotations along with isometric scaling. These transformations are applied at a scene graph node specified by the application. The RMaux event loop provides hooks to invoke an application callback during idle periods when there are no window system events. Source code for the RMaux library is included with the RM distribution.

RMaux provides for two different user interface models. The first implements a virtual trackball interface. In this model, the objects in the scene may be interactively rotated, while the cameras and lights remain fixed. The application specifies which node in the scene graph is to have its transformation matrix modified as part of the setup process. The second interface is used for navigation through a scene. This interface, called *rmauxFlyUI*, implements a "flight" model, where the camera moves only forward a constant amount during each frame, but the user can modify the pitch and roll of the camera. This flight model is implemented in many computer games and flight simulators. Like other commercial scene graph products, this interface makes assumptions about the underlying coordinate system, where the x- and y-axes are left and forward, respectively, and the z-axis is up.

## 6.0 Visualization Tools

The visualization toolkit included with RMSG provides numerous tools that convert raw data into renderable geometry. A subroutine call to one of the functions in the RMV library returns an RMSG node or subtree that may be inserted directly into the application's scene graph. Full source for the RMV library is included with the RM distribution so that developers may extend the library, as well as have access to a large body of code that demonstrates how RM can be used. Applications that do not need or want visualization tools may effectively ignore the RM visualization tools.

The subject of interfaces to visualization tools is complex and spans many diverse subjects. In addition to the visualization algorithms themselves are issues related to data management and data modeling. The RM visualization library, called *RMV*, implements a base set of visualization tools that operate on data of the form  $y=f(x)$ ,  $z=f(x,y)$  and  $w=f(x,y,z)$  producing visualizations that are 2D, 2.5D or 3D. These tools generate common "plot types" such as bar graphs, plots using glyphs or markers, surface or mesh plots, isosurfaces and contours. The 2D and 2.5D tools provide a unique set of parameters that can be used to "stack" plots, thereby increasing the amount of information in a single display.

Another difficult topic in visualization is the notion of grid types and data structures to accommodate a diverse array of different grids. All of the RMV tools are designed to index through the data in a sequential fashion, and rely upon an application-supplied callback to return grid coordinates and data for a given index. One dimensional data, such as unstructured data, may be embedded in two or three dimensional space. Similarly, two or three dimensional data refers to those datasets that can be indexed using two or three variables. These two or three dimensional data sets may be embedded in a two or three dimensional space. All grid coordinates are assumed to be on a Euclidean grid, as this maps directly to the viewing model used by OpenGL. Translation from alternate grids, such as polar to euclidean, must be performed by the application either at visualization time or a preprocessing step.

The visualization tools in RMV are a good starting point from which one can build significantly more complex visualization tools. Similar to applications that have a substantial investment in user interface and window system code, many applications have a similar substantial investment in visualization tools. One of the RMV design goals is to provide a flexible framework with little overhead that facilitates integration of existing visualization tools.

## 7.0 Future Work and OpenRM

RM represents a significant body of work that implements a stable and general purpose framework for developing platform-independent graphics applications. Building from this stable base, a number of improvements and extensions are in the works, including an Open Source version of RM Scene Graph called OpenRM.

### 7.1 Upcoming Features for RM Scene Graph

In order to support the needs of the visual simulation community, plans are in place to extend RM to support parallel-pipelined rendering. This type of parallelization is present in Performer, a popular scene graph API from SGI. The fundamental approach in this parallelization model is to have separate processes that manage traversal of the scene graph, culling and drawing. Such a stratification results in an effective two-frame delay; changes made to

the scene graph are rendered two frames later. However, since the goal is to produce a steady frame rate of, say, 60Hz, the effect of the delay is insignificant. The greatest challenge at this time is to ensure consistent results across all supported platforms, including Unix/Linux and Win32.

Another significant optimization technique is to better match groups of renderable data to the cache line sizes of the various types of graphics hardware. For example, a long triangle strip may be more quickly rendered if broken into a number of smaller-sized triangle strips. We intend to implement this type of optimization through the use of compile-time striding through OpenGL vertex arrays.

The use of the RM visualization colormap is a powerful technique for providing better control over how image data appears on-screen, as well as for minimizing the amount of data managed by the application, RM and OpenGL. Extending this concept to the colorization of all primitives is a high priority. The result will be that a single visualization colormap may be applied to divergent types of geometric data, such as triangle and line strips.

## 7.2 OpenRM, an Open Source Scene Graph API

In order to better serve the graphics and visualization community, R3vis Corporation has launched OpenRM, an Open Source variant of RM Scene Graph. The goal of this project is to promote a higher level of graphics technology available to all developers. We feel that central to the future of graphics applications to come will be the notion of a powerful set of graphics services that are ubiquitous across all platforms. With such ubiquitous services, it becomes possible to implement a broad array of network-based applications that transmit a small amount of data that then blossoms into a compelling and enjoyable experience for the user. This notion builds upon the quip of “just add data, and you have an application.” The OpenRM project is being distributed under the Mozilla Public License with the intent of being equally friendly to developers and commercial entities alike.

## 8.0 Conclusion

RM Scene Graph is application-extensible scene graph middleware that enables developers to quickly create high performance, portable, 3D graphics and visualization software. Central to RM is a scene graph design that emphasizes flexibility and performance. A broad set of rendering design goals results in seamless support for a mixture of 2D and 3D objects based upon geometry, vector and image-based data, along with direct support for stereoscopy. Rendering performance is accelerated through an aggressive use of retained mode rendering tactics. The data modeling burden is shared with the application, rather than being dictated by the scene graph. RMSG inherits many of the positive characteristics of OpenGL, yet provides valuable and much-needed high level primitives along with scene and render state management tools.

Future work in the rendering and modeling subsystems of RMSG will include expansion of RMSG’s multi-pipe/multi-processing model to include distributed-memory architectures and support for more high level services, such as dynamic texture paging, and collision detection. The RM rendering pipeline will be parallel-pipelined to increase overall performance. Advanced rendering features will include support for shadows. Additional visualization tools will be incorporated into RMSG as the system evolves. “Loaders” and “converters” to and from other formats, such as VRML, will be made available at a later date.

We believe that many developers will find RMSG to be a highly useful and beneficial tool, and will serve as the foundation for a new class of visualization tools that are capable of meeting the challenges posed by today’s computational environments.

## 9.0 Recommended Reading

[1] “The Design of the OpenGL Graphics Interface,” M. Segal and K. Akeley. [http://trant.sgi.com/opengl/docs/white\\_papers/opengl/index.html](http://trant.sgi.com/opengl/docs/white_papers/opengl/index.html).

[2] “OpenGL Programming Guide,” M. Woo, J. Neider, T. Davis, Addison-Wesley Developers Press, 1997 (The “Red Book”).