
*White Paper: Sort-First
Distributed Memory Parallel
Visualization and Rendering
with OpenRM Scene Graph
and Chromium*

E. Wes Bethel, R3vis Corporation

July, 2003

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

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



Sort-First, Distributed Memory Parallel Visualization and Rendering with OpenRM Scene Graph and Chromium

Introduction

In recent years, capacity and capabilities of commodity computing and graphics hardware has been growing at a rapid rate. Today, a commodity \$1500 workstation has computing and graphics capabilities that exceed those of a late 1990s \$500K graphics supercomputer. Somewhat orthogonally, some graphics and visualization research programs focus on high-resolution displays. There are numerous research examples of large display walls constructed from clusters of commodity graphics PCs, each of which drives a projector. Despite the favorable cost/performance ratio of commodity computing platforms, there remains little in the way of software technology to make use of such distributed memory parallel platforms. Without stable graphics and visualization libraries that are parallel-aware, application development is a difficult and time-consuming process.

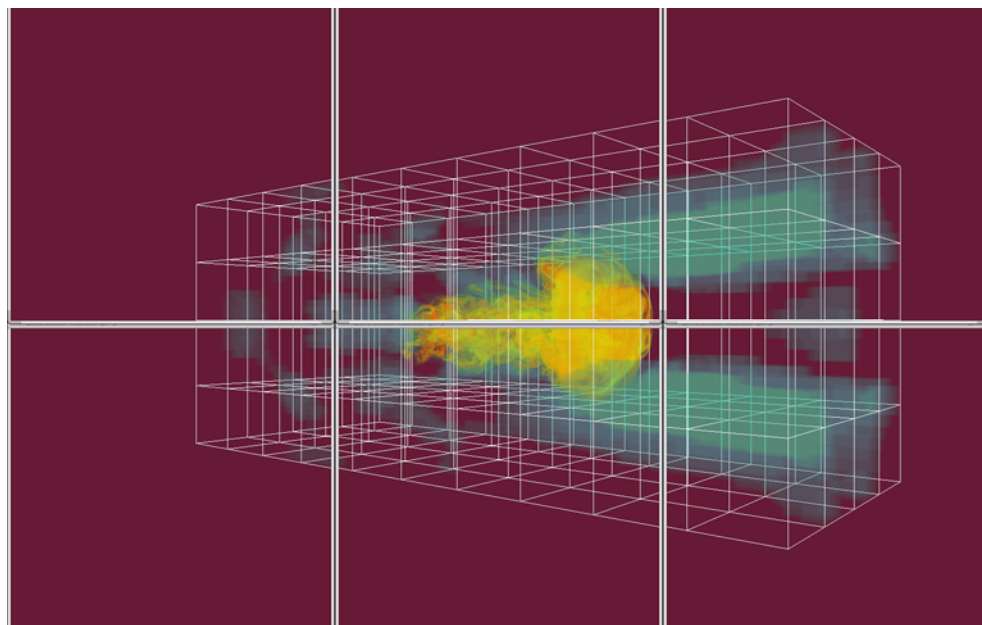
Our work is motivated by the desire to have the software tools necessary to effectively use distributed memory parallel computing platforms built from commodity components for the purposes of high-resolution graphics rendering and scientific visualization. The work we present in this white paper presents an overview of software technology that can be used to harness the power of parallel computing platforms for the purpose of high performance graphics rendering and scientific visualization. Our implementation allows any parallel graphics or visualization application to use parallel platforms for both computing and rendering. To achieve this objective, we have extended and combined two technologies producing a flexible and powerful software infrastructure.

Chromium is a stream-processing framework¹ that implements the OpenGL API². Simplistically, Chromium is intended to be used to drive multiple graphics servers as a single logical device. With Chromium, you can have a garden-variety

OpenGL program make use of a tiled array of projectors, where each of the different displays may be a single node in a PC cluster, or a different graphics pipe in a large SMP system.

Complementary to Chromium is OpenRM Scene Graph, which provides scene data management capabilities that simplify and streamline application development. The scene graph API provides features and capabilities not present in low-level graphics APIs, such as semi-procedural primitives, encapsulation of complex graphics state information, parallel and multipass rendering, view-dependent processing, and so forth. The scene graph is not a replacement of the graphics API, but instead is a software layer that simplifies access to and extends the capabilities of complex graphics APIs. Both OpenRM and Chromium are freely available as Open Source projects.

FIGURE 1. Parallel Volume Rendering Using Six Graphics Servers



1. chromium.sourceforge.net
2. OpenGL is a registered trademark of Silicon Graphics, Inc. See www.opengl.org.

The main motivation for the OpenRM+Chromium work is to provide the means to have a parallel application make use of a scene graph system that is capable of leveraging parallel rendering resources in an efficient and easy-to-use fashion. While it is possible to take nearly any serial OpenGL program and have it drive an arbitrary number of graphics servers using Chromium, it is still a serial graphics application. In many cases, it is desirable to parallelize the graphics application itself in order to realize performance improvements. Scientific visualization is one such example, where large datasets are loaded and processed in parallel. Not only is I/O bandwidth amortized across many processors, but CPU load and memory use is also amortized. Using parallelization, it is possible to tackle problems that are much too large to fit on a single machine. Having a portable and efficient infrastructure for processing and visualization of such large problems is the primary motivation of our work.

One troublesome issue when considering support for general purpose parallel programming is the notion of the parallel programming model. There are several different widely-used parallel programming models: the Message Passing Interface, also known as MPI³, the Parallel Virtual Machine, also known as PVM⁴, and OpenMP⁵. Generally speaking, applications written to use one parallel programming model must be ported to use a different one. One of our primary design objectives in the OpenRM+Chromium work was to minimize, if not eliminate, the dependency upon any parallel programming environment in order to achieve maximum portability.

Our implementation of "parallel OpenRM" realizes those goals, and is not dependent upon any parallel programming framework. To realize that objective, we made design decisions based upon two realizations. The first is that parallel rendering, in general, requires interprocessor synchronization in certain key areas. Such synchronization is accomplished through the use of constructs provided by Chromium. Like OpenRM, Chromium may be used by parallel applications using any parallel processing framework. Chromium's synchronization mechanisms consist of semaphores and barriers. Therefore, the rendering synchronization in a parallel OpenRM/Chromium application occurs during rendering, not in the appli-

3. <http://www-unix.mcs.anl.gov/mpi>

4. http://www.csm.ornl.gov/pvm/pvm_home.html

5. <http://www.openmp.org/>

cation - the application is never aware that any synchronization is being performed. In other words, the kind of synchronization needed to realize parallel rendering is implemented downstream from and independent of the application's parallel programming environment.

Second, the Chromium-enabled implementation of OpenRM does not provide any explicitly parallel scene graph operations. Instead, it is the application developer's responsibility to implement application-level parallelism. Each application process creates its own private scene graph based upon whatever form of spatial data decomposition it deems best for the task at hand. Also, each application process will invoke the OpenRM frame-based renderer, which causes parallel rendering to happen as each application process will generate, through OpenRM, parallel streams of graphics commands. Sorting and routing of graphics commands from application process space to one or more graphics servers is handled by Chromium transparent to the application. Any application-level parallelism must be created by the application developer. This approach produces an elegant, compact and extremely flexible scene graph implementation.

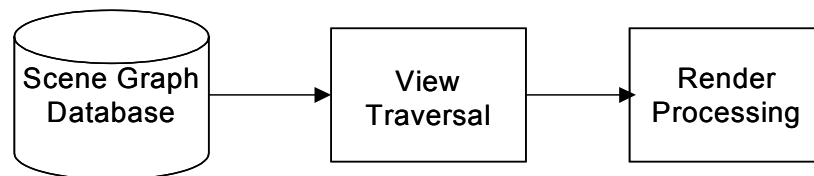
In addition to being able to take advantage of parallel platforms for increased application processing and graphics rendering performance, all the "usual" scene graph tricks work in the parallel environment. Each different application process can use view-dependent operations, like LOD-based model switching and view frustum culling, along with useful OpenRM capabilities like multipass rendering and a rich set of application callbacks to assist in the implementation of application-specific features.

Parallelism on Shared Memory Systems

Generally speaking, there are a number of potential avenues to increase rendering performance using parallelism. One approach is to divide the process of rendering into many subunits that are arranged into an "assembly line" organization. In the assembly line model, "workers" at different "stations" take their input from upstream stations, and produce finished work that is consumed by downstream stations. OpenRM implements this type of parallelism using a two-stage rendering process. The first stage is called the "view tra-

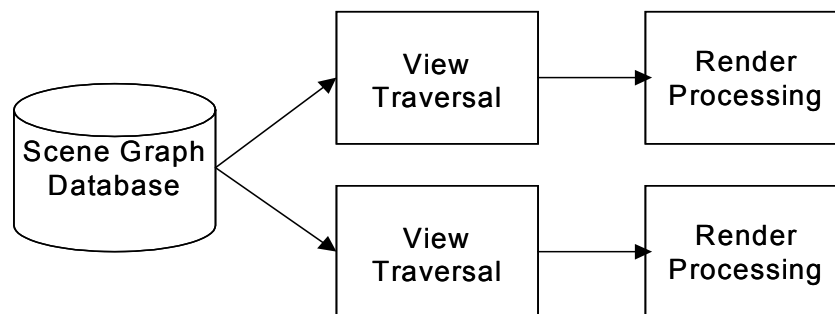
versal,” and consists of scene graph processing to determine which portions of the graphics database are actually visible. The second stage, called the “render traversal,” uses input from the view traversal and issues graphics commands that cause the scene to be rendered. This type of parallelism was implemented in OpenRM in late 2000, with the first public release of the source code early in 2001.

FIGURE 2. Multistage View/Render Pipeline



Parallelism through pipelining, which is analogous to an “assembly” line, represents a single flow of work. In other words, work is performed more quickly and efficiently, but there is only one assembly line. On shared memory computers, it is possible to run two or more simultaneous instances of the OpenRM renderer pipeline on a single machine, in effect using multiple assembly lines in parallel. In such a configuration, each renders the contents of single or multiple graphics databases to multiple windows in a simultaneous fashion. However, all such processes are constrained by a single address space.

FIGURE 3. Multiple, Simultaneous Pipes Using a Single Data Source



Migrating to distributed memory computers, such clusters of commodity PCs, requires substantially more effort due to the challenges presented by the distributed memory architecture. The basic objective is to harness the power of low-cost

but high performance PC computing and graphics hardware to drive extremely high resolution displays. For example, the maximum resolution that may be obtained from modern graphics systems is a function of the amount of frame-buffer memory and the constraints imposed by the windowing system. Frame-buffer memory size dictates the limits of the maximum image size that may be rendered. A framebuffer with 256MB of memory could theoretically support a final rendered image size of 8K by 8K RGBA pixels with 8 bits of resolution per color channel. The window system, which “brokers” the window or offscreen rendering area setup between the application and OpenGL, typically has limits that are much smaller than framebuffer memory. Window system limits reflect a design bias towards screen display, and typically don’t support “very large” pixel buffers.

The most advanced shared memory systems can support up to eight simultaneous displays using special hardware (“display generators”). On the other hand, shared memory systems built from commodity components can typically support only two such displays from a single graphics accelerator. Some specialized systems exist that use many commodity graphics accelerators to drive more than two displays on a single system. These systems have performance limits, which reflect the bandwidth bottleneck encountered when attempting to drive many graphics accelerators from a single memory/processor bus.

Despite these architectural limitations, SMP platforms provide an attractive environment for creating high performance, parallel visualization and rendering applications. The most alluring feature of these platforms is the ability for multiple processes to share data by using the same memory. No explicit transmission of data from one process to another is required, which thereby simplifies programming and speeds execution.

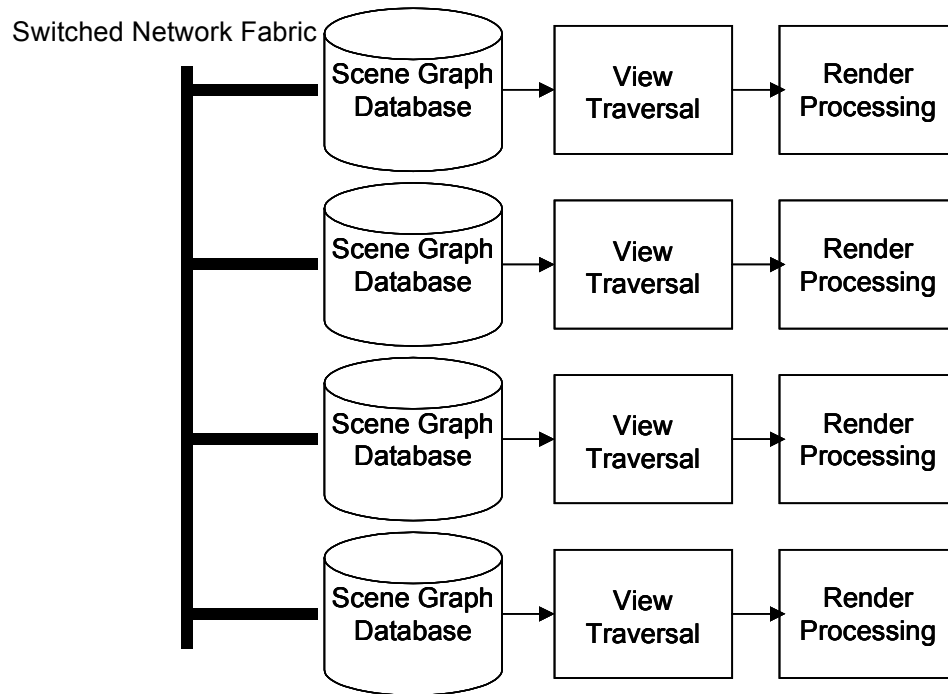
Parallelism on Distributed Memory Platforms

Historically, shared memory systems have been expensive since they were built using non-commodity components. Specialized shared memory systems, which implement Cache Coherent Non-Uniform Memory Access (CC-NUMA) using multiple memory subsystems, are very expensive in comparison to systems based upon commodity hardware that don’t implement cache coherency.

While expensive, the specialized SMP systems are architecturally capable of scaling to 64 CPUs and beyond. The economic allure of combining many low-cost, commodity-based systems into a large system has provided the impetus for a significant amount of research and development within the graphics and visualization communities over the past decade. One of the challenges in doing so is the lack of system-level software and application-usable APIs to take advantage of parallel system resources.

Early work in the area of “hooking up multiple machines” to implement distributed memory rendering systems dates to the early 1990s. In this work, which was based upon the CAVE Library from the University of Illinois, the scene data was replicated on each computer explicitly by the application (no scene graph library), and transformation data was broadcast over an IP LAN to each node. Each separate computer would then render the scene using a different viewpoint onto a different display device. This early work was motivated primarily by the limits of early SMP systems, which supported only four displays per platform. Later, when SMP systems supported more graphics pipes per platform, developers tended to prefer using a single large SMP because of the simplified development environment and better system performance.

FIGURE 4. Parallel Distributed Memory Rendering Architecture



More recently, research efforts have produced solutions that simplify development of parallel graphics and visualization applications on distributed memory architectures. These solutions were designed with commodity hardware in mind. The resulting technology is suitable for use in realizing scalable and portable applications that are portable across systems, and which are capable of realizing very high performance rates.

The solution we describe in this paper fills the gap by providing a stable infrastructure layer that is capable of delivering high performance rendering, visualization, and scene management services on either shared- or distributed-memory systems.

What is Chromium?

Chromium is a stream-based framework that implements the OpenGL API. The basic idea is that OpenGL graphics commands are intercepted by Chromium, then sent through a one or more stream processing units (SPUs) that can operate on the graphics stream. The graphics commands eventually make their way to one or more rendering servers, where they are transformed into images using a native or software OpenGL implementation. The predecessor to Chromium was known as WireGL, which was used primarily to redistribute graphics commands from a single stream to multiple rendering platforms. Chromium builds on WireGL's capabilities by permitting multiple simultaneous streams of graphics commands to be processed by parallel rendering servers, and by the SPU idea that implements serial filters operating on streams of graphics commands.

Chromium is an Open Source project that is hosted at SourceForge, and the Chromium project is run by Dr. Greg Humphreys of the University of Virginia. Chromium receives support from a wide number of sponsors within the high performance computing and rendering community, including the U. S. Department of Energy.

What is OpenRM Scene Graph?

OpenRM Scene Graph is an Open Source scene graph technology designed for use by applications that demand the utmost in rendering performance and flexibility. OpenRM's scene graph technology provides graphics data management to applications, as well as a flexible and extensible processing framework. One of the primary benefits of scene graph technology is that applications developers can focus on scene content and efficient processing rather than the detailed semantics of the underlying graphics API.

OpenRM uses two-stage pipelined parallelism to accelerate rendering: the first processing stage performs scene graph traversal and implements view dependent operations, such as frustum culling and distance-based model switching (LOD). The second stage dispatches graphics commands to OpenGL in an optimized fashion to achieve very high rendering rates. OpenRM's unique thread-safe

design permits multiple simultaneous renderers to create images from a single scene graph database into multiple display windows, as well as support for multiple, simultaneous loaders to populate a single scene graph.

OpenRM is also an Open Source project hosted at SourceForge, and is managed by E. Wes Bethel of R3vis Corporation and Lawrence Berkeley National Laboratory. OpenRM is supported by donations from the community, and through research grants provided by the Office of Science in the U. S. Department of Energy. OpenRM is one of the most mature Open Source scene graph packages, derived from R3vis Corporation's proprietary scene graph technology.

Sort-First Parallel, Distributed Memory Visualization and Rendering Architecture Overview

In sort-first parallel rendering, graphics primitives are distributed among N different computers. Such distribution occurs prior to the transformation and lighting stage of the rendering pipeline. In contrast, sort-middle redistributes primitives after transformation and lighting, but before rasterization. Sort-last redistribution occurs after rasterization, and involves image pixels, not graphics primitives.

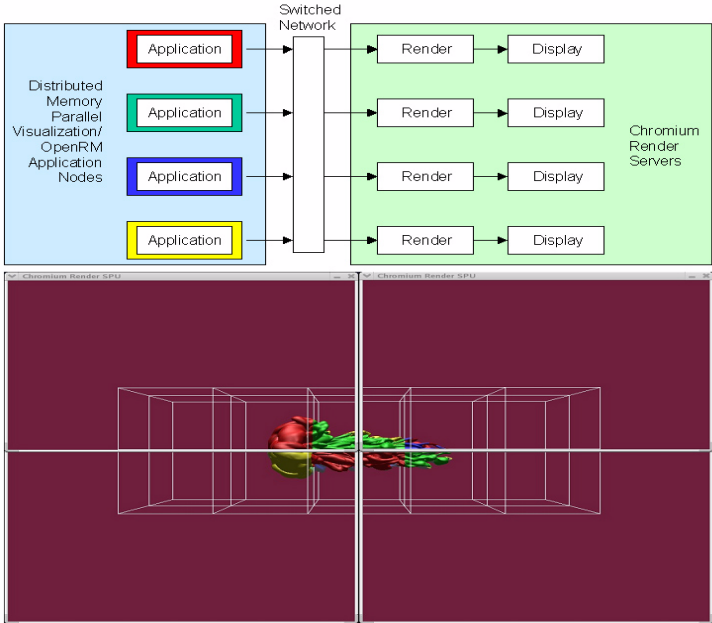
Sort-first approaches have not been widely explored for use in parallel visualization and rendering architectures. One of the reasons is because sort-last approaches, which involve compositing images using orderly communications patterns, exhibit favorable scaling characteristics that are more sensitive to the final image resolution, not the size of the input model. Increasing the size of the display will have an adverse impact on sort-last algorithms. The performance of sort-first approaches, on the other hand, is largely independent of display resolution, but suffers from duplication of model data that renders onto more than one image tile. Previous studies on the subject of sort-first methods have focused on minimizing the amount of such duplication.

Our target deployment environment is clusters of computers equipped with high performance graphics accelerators that drive large tiled display systems.

Such an environment poses a significant challenge to sort-last approaches, and presents unique opportunities that are especially well-suited for use by sort-first methods. Each PE in a sort-last configuration needs to generate a full-sized image, then all such full-sized images are composited together into a final image that is displayed to the user. Unfortunately, there are no graphics systems that are capable of generating images that are suitably large enough for use in scalable, high resolution display environments. The opportunity for sort-first approaches stem from the fact that modern graphics processing units are capable of rendering upwards of 100 million triangles per second, each driving a single tile of a multi-tile mural.

Parallel visualization, visual simulation and rendering applications that use Chromium and OpenRM employ an architecture similar to that shown below in Figure 5. Each of the application processing elements (PEs) loads a subset of scientific data or graphics database in parallel, thus amortizing the cost of data input over several processors and disk or network interfaces

FIGURE 5. OpenRM/Chromium Sort-First Parallel Application Architecture.



Each application PE stores its visualization results or graphics data using OpenRM's data management facilities. Then, each PE invokes the OpenRM renderer, which dispatches graphics commands to Chromium. Chromium then routes the graphics commands to the appropriate rendering server. The rendering servers shown in Figure 5 are located on a different set of hosts connected to the application PEs via a switched network. It is possible for application PEs and rendering servers to execute on the same set of hosts, but the switch network is required.

In Figure 5, each of the application PEs loads in a subset of a scientific dataset, and performs an isosurface calculation in parallel using OpenRM's visualization tools, and stores the results into an OpenRM scene graph. The isosurface generated by each PE is color coded according to PE number. Figure 5 shows that the isosurface model generated by any given PE may be rendered by one or more parallel rendering servers.

Using OpenRM in a Distributed Memory Parallel Environment

Among the primary consideration affecting parallel application design and implementation is the choice of parallel processing framework that is used to manage memory and synchronize process execution. Typically, when algorithms are ported to one such parallel processing framework, they are not usable in other parallel processing frameworks. This is due to semantic differences between frameworks, as well as fundamental differences in how memory variables are treated: are they plural variables present on all processors, or local to a single processor? One approach in parallel "toolkits" is to provide objects that are plural in nature. In other words, a data object created by one PE will be created across all PEs. This approach hides the complexity of parallel processing from the application developer. However, it can also bind the toolkit to a particular parallel processing model. In many instances, such a binding is completely appropriate. The restriction to a single parallel processing framework is inappropriate for parallel graphics and visualization toolkits, however. A hallmark of a "good" graphics or visualization toolkit is one that provides the maximum amount of flexibility and portability.

Our approach to OpenRM parallelization is designed to minimize dependence upon any single data model or parallel processing framework. As such, parallelism will be specified and implemented by the application, which is really the best place for such activities. Otherwise, the scene graph system would be dictating parallel architectural constraints to the application. It is our intent to have the scene graph system be a flexible tool to be used by applications, rather than having the scene graph system dictate fundamental architectural design constraints to the application. The scene graph should be akin to “duct tape” - useful in ways never imagined by its creators. ***To minimize the impact upon parallel architecture design, our approach assumes that parallelism is dictated by the application, and that OpenRM in turn relies upon features specific to Chromium to realize correct rendering operation in parallel environments.***

OpenRM Capabilities to Use Chromium in Distributed Memory Parallel Environments

As indicated before, certain types of rendering synchronization are required in parallel environments in which multiple processes generate parallel graphics streams that are consumed and rendered by parallel graphics servers. The types of operations that require synchronization are those that are “global” in nature. In other words, operations that affect the state of the framebuffer in a general way must be synchronized across all display tiles.

The most obvious operation requiring synchronization is required is SwapBuffers. Generally speaking, applications want all the tiles to invoke SwapBuffers at the same time. Another area where synchronization is required is for framebuffer clear operations. Unlike SwapBuffers, in which we want all the tiles to execute the command at the same time, framebuffer clear operations on each tile don't necessarily need to happen at the same time. However, they do need to occur before any rendering takes place. Therefore, the synchronization required for framebuffer clears is not temporal in nature. Instead, the requirement is one of ordering: we want the framebuffer clear to occur prior to any rendering.

OpenRM implements both types of synchronization using constructs provided by Chromium: Chromium barriers and semaphores. Such synchronization is com-

pletely transparent to the application, and does not rely on the synchronization mechanisms of any parallel programming environment. In the case of Swap-Buffers, OpenRM provides the synchronization using Chromium barriers “under the hood,” and no action is required on the part of the application.

The case of framebuffer clears is slightly more complex. Generally speaking, applications will specify to OpenRM that the framebuffer will be cleared using one of the four different framebuffer clear scene parameters (solid color, background image tile, constant depth buffer value, depth buffer image tile) in a scene graph node. OpenRM implements a Chromium barrier that is associated with all framebuffer clear scene parameters. Because there is no “parallel scene graph object” in OpenRM (doing so would marry “parallel OpenRM” to a particular parallel processing environment, which we want to avoid), application processes each create a local scene graph that holds part of the overall scene spread across all application processes. While the contents of the scene graph will vary from application process to process, all such scene graphs must have at least one trait in common: they all have the same number of framebuffer clear operations specified as scene parameters. The number of such operations in each scene graph is arbitrary, but the overall number must be constant throughout all scene graphs on all application processes. Your application may choose to not clear the framebuffer at all, in which case no scene graph node on any processor may contain a framebuffer clear scene parameter. Or, your application may specify multiple views/multiple viewports within a single scene graph. In that case, all scene graphs on all processors must contain the same number of framebuffer clear operations.

The bottom line is that the Chromium-enabled OpenRM implementation is compact and extremely flexible. You can use it to create parallel graphics and visualization applications on virtually any parallel platform. ***The design means that you - the application developer - decide how to parallelize your application, rather than having such a choice dictated to you by your scene graph library or graphics API.***

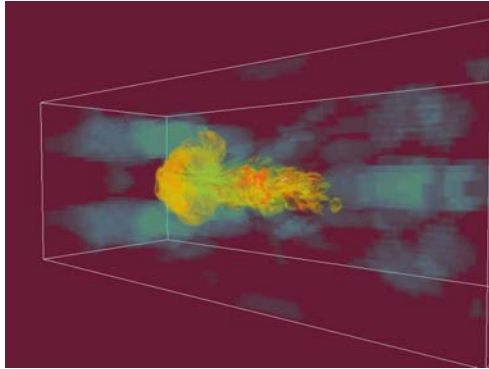
Parallel View-Dependent Operations

In the parallel environment, all the usual scene graph “tricks” are available to the application. A complete survey of possibilities is beyond the scope of this document. However, a single example - parallel Level-of-Detail (LOD) based model switching - demonstrates the power and flexibility available to developers.

The basic idea behind LOD-based model switching is to use low-resolution of models that are “far away” from the viewer, and use high-resolution models that are “close to” the viewer. The motivation for using LOD-based model switching is to reduce the rendering load by sending fewer graphics primitives down the pipe for rendering. In a parallel environment of networked resources, reducing network traffic will have a significant impact on overall parallel rendering performance. The ideal of LOD-based model switching extends to the parallel environment without loss of generality, and, more importantly, without any changes to the serial OpenRM application code.

In our 2003 Parallel Visualization and Graphics Symposium paper (see the R3vis.com website), we describe the performance improvements realized from using scene-specific knowledge to accelerate rendering in a parallel environment. When performing parallel volume rendering, the portions of the model that are “close to” the viewer are rendered in full resolution, while those portions “far from” the viewer are rendered at reduced resolution. In static scenes, the cost savings is substantial: a 50% savings in network traffic is realized by using scene specific knowledge. The cost savings decreases as the model is interactively transformed, causing full resolution subsets of the model to be sent to tiles.

FIGURE 6. Sort-First Parallel Volume Rendering with LOD Control.



The result is view-dependent reduction in graphics rendering load in a distributed memory parallel environment that is completely portable across architectures.

The left image shows volume rendering using full-resolution data. The right image uses reduced-resolution data for blocks that are “far” from the viewer, and full resolution data for blocks “close to” the viewer.

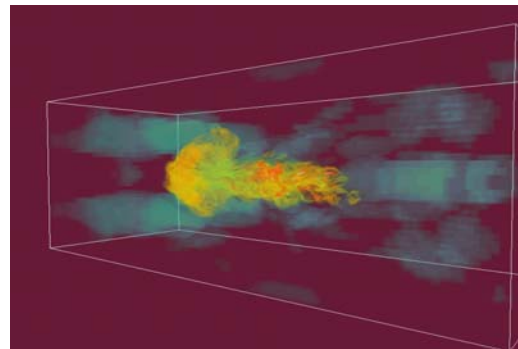
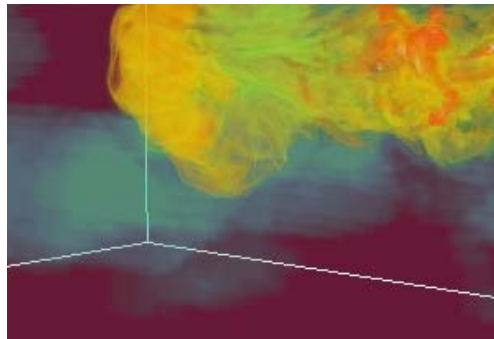
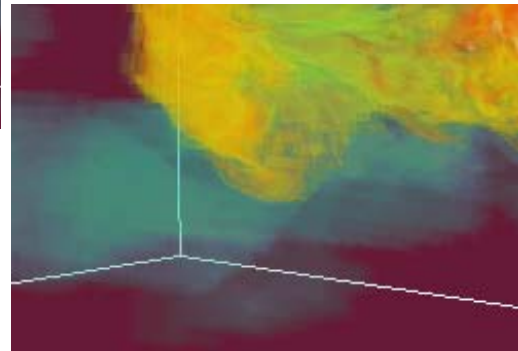


FIGURE 7. Sort-First Parallel Volume Rendering with LOD Control.



The image on the right is rendered using LOD-based model switching. Fine details present in the full resolution image are “blurred” when “far from” the viewer.

These images show close-ups of the images from Figure 5. The image on the left is rendered at full resolution, with no LOD-based model switching.



Summary and Conclusions

The combination of OpenRM and Chromium provides the capabilities needed to create portable, high performance parallel visualization and graphics applications. Chromium provides the ability to render on multiple graphics servers, and OpenRM provides the means for your application to efficiently manage scene data and to quickly harness the power of multiple rendering services. Chromium implements its own parallelism using custom methods, which are leveraged by OpenRM to implement certain types of synchronization operations required in the context of parallel rendering. Since OpenRM is independent of parallel processing framework, it can be used by any parallel application framework.

Further Reading

1. G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Koslowski, "Chromium: A Stream Processing Framework for Interactive Rendering on Clusters." In *Proceedings of ACM Siggraph*, pp. 693-702, San Antonio, TX, July 2002.
2. G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett and P. Hanrahan, "WireGL: A Scalable Graphics System for Clusters." In *Proceedings of ACM Siggraph 2001*, pp. 129-140, July 2001, Los Angeles, CA.
3. E. W. Bethel, G. Humphreys, B. Paul, J. D. Brederson. "Sort-First, Distributed Memory Parallel Visualization and Rendering." *To Appear in the 2003 Symposium on Parallel Visualization and Graphics, IEEE Visualization 2003, Seattle WA, October 2003.*

