Sort-First, Distributed Memory Parallel Visualization and Rendering

E. Wes Bethel[a] Greg Humphreys[b] Brian Paul[c] J. Dean Brederson[d]

[a]ewbethel03@r3vis.com, R3vis Corporation, PO Box 979, Novato, CA 94948, USA and Lawrence Berkeley National Laboratory, Mail Stop 50F, Berkeley CA, 94720, USA.

[b]humper@cs.virginia.edu, Department of Computer Science, Universtiy of Virginia, PO Box 400740, Charlottesville VA, 22904, USA.

[c]brianp@tungstengraphics.com, Tungsten Graphics, 114 S. Prize Oaks Dr., Cedar Park TX, 78613, USA.

[d]jdb@cs.utah.edu, Scientific Computing and Imaging Institute, University of Utah, 50 S Central Campus Drive, Room 3490, Salt Lake City, UT 84112, USA

Abstract

While commodity computing and graphics hardware has increased in capacity and dropped in cost, it is still quite difficult to make effective use of such systems for general-purpose parallel visualization and graphics. We describe the results of a recent project that provides a software infrastructure suitable for general-purpose use by parallel visualization and graphics applications. Our work combines and extends two technologies: Chromium, a stream-oriented framework that implements the OpenGL programming interface; and OpenRM Scene Graph, a pipelined-parallel scene graph interface for graphics data management. Using this combination, we implement a sort-first, distributed memory, parallel volume rendering application. We describe the performance characteristics in terms of bandwidth requirements and highlight key algorithmic considerations needed to implement the sort-first system. We characterize system performance using a distributed memory parallel volume rendering application, and present performance gains realized by using scene specific knowledge to accelerate rendering by reducing network traffic. The contribution of this work is an exploration of general-purpose, sort-first architecture performance characteristics as applied to distributed memory, commodity hardware, along with a description of the algorithmic support needed to realize parallel, sort-first implementations.

CR Categories and Subject Descriptors: I.3.2 [Computer Graphics]: Graphics systems - Distributed/network graphics; C.2.4 [Computer-Communication Networks] Distributed systems – distributed applications.

Additional Keywords: distributed memory visualization, parallel visualization, parallel scene graph.

1. Introduction

In recent years, the increasingly favorable price to performance ratio of commodity computing and graphics hardware has provided an impetus for scalable visualization and rendering research. One of the themes common to such research has been techniques for realizing scalability of visualization and rendering algorithms on distributed memory platforms. The work we describe in this paper is similarly motivated and themed: use of commodity computing and graphics hardware to realize scalable visualization and rendering [Bartz et al. 2001; Law et al. 2001]. One emphasis in our approach is use of a sort-first architecture to leverage a cluster of commodity graphics systems. Another is the generality of the architecture to support a range of parallel visualization and rendering applications, especially those in which it is not possible for the entire renderable model to reside on each rendering node, as would be the case if the entire scene were simply replicated on all nodes.

Our sort-first architecture is realized through the extension and combination of two separate yet complementary technologies. The result is a highly flexible and scalable system. The rendering infrastructure for our architecture is provided by Chromium, a stream-oriented framework for manipulating streams of graphics API commands on parallel architectures, including shared and distributed memory systems [Humphreys et al. 2002]. Complementary to the graphics API is graphics and visualization data management, which is provided by OpenRM Scene Graph [Bethel et al. 2003; OpenRM 2000]. At the highest level of abstraction is the parallel application itself, which performs parallel I/O of scientific data, but which interfaces to OpenRM for data management and rendering. OpenRM, in turn, performs rendering by issuing OpenGL commands. Chromium then routes the OpenGL commands to one or more distributed memory rendering servers. The architecture of our implementation is depicted below in Figure 1.

There are two motivations for leveraging scene graph technology within the context of sort-first distributed memory parallel visualization. First, the scene graph provides a high-level interface for managing and rendering graphical data, freeing the application developer from the details of lower level graphics API and simplifying applications development. Second, the scene graph's processing infrastructure provides the opportunity to use problem- or scene-specific knowledge to implement and accelerate distributed memory rendering algorithms. Level-of-detail techniques are a good example: portions of the scene that are "far away" can be rendered using lower-resolution representations than those closer to the viewer, thereby reducing rendering load. As applied to scientific visualization, view-dependent processing holds promise to accelerate end-to-end system performance by allowing the application to avoid costly data I/O for regions that are outside the view frustum or that are "too far away" to be of

IEEE Symposium on Parallel and Large-Data Visualization and Graphics 2003,
October 20-21, 2003, Seattle, Washington, USA
0-7803-8120-3/03/$17.00 ©2003 IEEE

interest. Scene graph systems provide the infrastructure to implement such view-dependent processing in a general way.

The rest of this paper is organized as follows. First, we review previous work in parallel visualization, comparing and contrasting sort-last with sort-first approaches. Next, we describe our approach, including implementation details. These implementation details focus on the types of extensions needed for a scene graph system to function effectively in a distributed memory parallel environment, as well as a description of a two-stage sorting algorithm used to perform hardware accelerated volume rendering [Cabral et al. 1994] in a parallel environment. The performance of a volume rendering application is presented with a characterization of the data bandwidth requirements in the parallel environment. We conclude with discussion and comments about potential future directions.

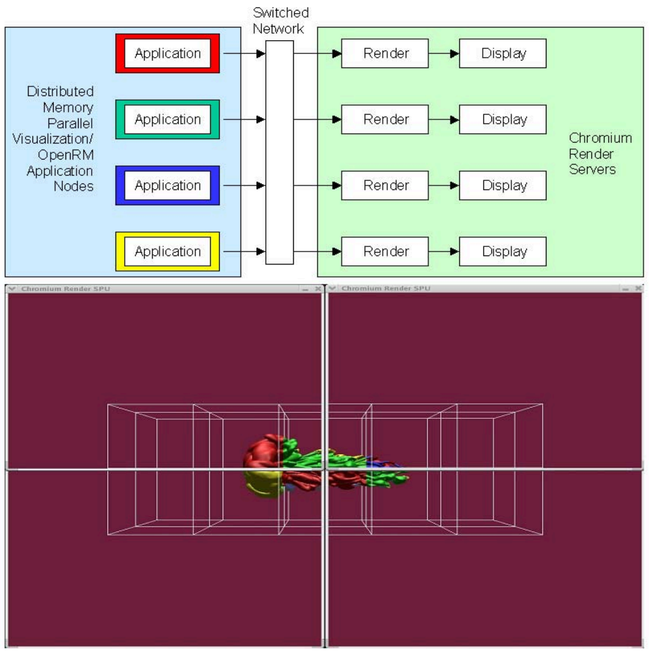

Figure 1. Distributed Memory Parallel Sort-First Application Architecture

2. Background and Related Work

The terms "sort-first," "sort-middle" and "sort-last" are used to describe where in the rendering pipeline graphics primitives are distributed amongst multiple, parallel renderers [Molnar et al. 1994]. "Sort first" refers to primitive redistribution in object coordinates, prior to transformation and conversion to Normalized Device Coordinate (NDC) space [Foley et al. 1990]. "Sort middle" refers to distribution after transformation to NDC space, but before rasterization. "Sort last" refers to distribution of pixels, and occurs after primitives have been rasterized. The advantages and disadvantages of each approach can be characterized in terms of bandwidth requirements, amount of duplicated work, and load balance amongst the parallel renderers.

Sort-last has been a promising avenue for parallel visualization research over the years. Earlier work in distributed memory volume rendering [Lombeyda et al. 2001; Moreland et al. 2001; Kniss et al. 2001; Heirich and Moll 1999; Neumann 1994] focuses upon the communication costs in sort-last image assembly. In these experiments, each application processing element (PE) is responsible for rendering a subset of data into an image. All such images are then combined into a final image. The data distribution model used by these sort-last systems scales well for visualization, especially when the visualization algorithms require minimal interprocessor communication. In addition to computational scalability, such parallel visualization algorithms typically scale well in terms of required I/O bandwidth by distributing the cost of expensive data read operations over many processors. Such I/O scalability is highly desirable when rendering large, time varying datasets.

Sort-last approaches have predictable and "well behaved" communication patterns and loads. The term "well behaved" means the bandwidth requirements and computational complexity are a function of the image size and not the complexity or size of the dataset or rendering technique. Communications patterns and loads are predictable in that they can be estimated as a function of *P*, the number of pixels in the image, and *N,* the number of participating processors. Bisection bandwidth rates vary depending upon the method being used, and represent the amount of pixel traffic exchanged during the image composition process. The total number of pixels transmitted in Binary Swap is $2.43N^{0.333}P$, as described in [Ma et al. 1994]. The implication is that sort-last performance is a direct function of both the size of the final image, well as the number of participating processors. Most importantly, it is a linear function of P, which will dominate in high-resolution display environments.

In our target environment, which consists of commodity computing, graphics and network components, sort-last approaches will encounter two difficulties. The first is that the high image resolutions of tiled display environments will have an increasingly adverse impact upon performance. The second, which is not surmountable, is that these graphics cards, which are intended for use on desktop platforms, do not singly provide the high resolution realized by arranging multiple display platforms into a single logical tile. Related is that fact that CPU resource requirements will grow linearly with image size when using software image compositors.

While the limitations of sort-last algorithms for high-resolution scalable displays are straightforward to identify and quantify, sort-first approaches present their own set of challenges. Among them is the fact that some visualization techniques (such as isosurface generation) can generate a substantial amount of geometric data, using more memory than either the original data or the final image. In other words, one of the motivations to explore sort-first approaches is to avoid sort-last limitations by transmitting geometry rather than pixel data. In some cases, the amount of geometry can grow quite large. The communication costs for sort-first algorithms are a function of the scene itself, rather than the final image size.

Another difficulty with sort-first concerns the non-uniform data transmission patterns resulting when transmitting geometry from application nodes to rendering servers. Sort-last exhibits relatively uniform communication patterns. In contrast, sort-first can produce highly non-uniform communication patterns depending upon the scene. Worse, the communication patterns and loads can vary from one frame to the next. This non-uniformity often manifests in uneven, jittery frame rates and has an adverse impact upon system usability. Earlier work in characterizing sort-first parallelism [Mueller 1995] provides estimates that indicate much less data is transmitted when redistributing primitives using sort-first than would be needed to transmit pixels when using a sort-last approach. [Mueller 1995] makes the observation that sort-first architectures will be most successful when retained-mode rendering models are used to reduce the amount of traffic. Our use of a scene graph as the basis for data management and rendering seeks to maximize use of retained mode structures wherever possible. An interesting compromise is to leverage the advantages of both sort-first and sort-last approaches with a hybrid sorting scheme that uses both image and data partitioning for load balancing [Garcia and Shen 2002].

More recently, [Samanta et al. 2001] describes a technique that repartitions models stored in a scene graph across multiple nodes in a PC cluster. The objective of this approach is to minimize geometry broadcast during rendering. In their example, a large, static 3D model is preprocessed to create a hierarchical, multiresolution model. Portions of the model are replicated across some, but not all, nodes to reduce potential communication bottlenecks associated with moving graphics data during interactive rendering. Like our work described here, the motivation is to use commodity clusters to render models that are too large to fit entirely on a single node. The approach described in [Samanta et al. 2001] is not completely sort-first, for they use a sort-last image compositing step to combine individual images into final images that are displayed to the user [Samanta et al. 2000]. The observation with this approach is that intra-frame communication of geometry data is expensive [Samanta et al. 1999], and can be avoided by pre-caching geometry data through limited model replication. The cost is model preprocessing and the partial replication of the model, similar to a sort-first approach for out-of-core large model rendering [Correa et al. 2002]. Such an approach works best for static scenes, which are not characteristic of time-varying scientific datasets. Their results show favorable speedups when compared to sort-last or sort-first approaches.

Balanced against sort-first's irregular data traffic patterns and loads is an inherent flexibility not possible with sort-last approaches. Sort-last algorithms combine images from separate renderings into one final image. Image composition requires strict ordering semantics, which implicitly places upstream restrictions on data distribution and the type of algorithm that can be used. Sort-first has no such implicit ordering constraints, and is therefore more widely applicable to many types of visualization and rendering algorithms. Our approach is strictly sort-first in order to realize the benefits of algorithmic flexibility, and is intended for deployment on PC clusters used to drive high-resolution, tiled displays [Schikore et al. 2000].

3. Architecture Overview

Our approach is based upon the architecture shown earlier in Figure 1. The parallel visualization application uses object-order task decomposition: each PE is responsible for reading, processing and rendering a subset of the dataset. After loading its subset of data, each PE generates a graphical representation of its subset and stores it in the local scene graph. Then, all PEs invoke the scene graph renderer in parallel. Each individual renderer performs a traversal of its local scene graph, and generates OpenGL rendering commands. Chromium intercepts the commands and routes them to the appropriate rendering server using a spatial sorting algorithm.

In our approach, graphics data is transmitted from an application node to a render server only if the graphics data intersects the viewing frustum managed by the rendering server. Chromium's *tilesort* Stream Processing Unit (SPU) is responsible for performing these spatial comparisons and routing graphics commands from the application to the rendering servers. Because OpenRM Scene Graph [OpenRM 2000; Bethel et al. 2002] makes extensive use of retained mode objects in OpenGL, graphical data is typically sent to a rendering server once and displayed by calling *glCallLists()*, which is very inexpensive in terms of network bandwidth. Furthermore, the scene graph tracks the spatial extents of the graphics data and passes that information to Chromium. Chromium then uses this information to accelerate the sorting process.

4. Distributed Memory Parallel Scene Graph Implementation

One of the key roles of scene graph technology in rendering applications is graphics data management. There are a number of possible approaches to extend scene graph algorithms for use in a distributed memory environment. One approach is to implement "parallel scene graph objects," which perform fundamental parallel operations. Examples include collective operations, such as scatter-gather amongst multiple PEs, and creation/destruction of parallel data structures on multiple PEs. There are many other considerations that are beyond the scope of this paper.

Embarking upon parallelization of any code requires "commitment" to a particular parallel processing framework and memory model. A scene graph system that has been modified for use on parallel machines using one framework is likely compatible only with applications built using the same framework. Scene graph systems should place the least possible number of limitations on applications, including selection of a parallel processing framework. At a minimum, the scene graph should be threadsafe to support development on a cluster [Voss et al. 2002; Reiners et al. 2002]. OpenRM is both threadsafe and capable of pipelined-parallel rendering within a given instance of a scene graph, similar to the multi-threaded approach used by Performer [Rohlf and Helman 1994].

In order to minimize the number of constraints on applications developers, OpenRM's "distributed memory" parallel implementation contains no "parallel scene graph objects." Referring back to Figure 1, each application PE reads in a subset of a large scientific dataset and transforms it into a graphical representation, with the results being stored in a local scene graph. Instead of providing explicit "parallel scene graph objects," the parallel application must adhere to a few simple guidelines to ensure consistency in the face of parallelism. The most important of these guidelines is that all application PEs must create scene graphs with one scene graph element that contains a synchronization construct, as will be explained below.

When it is time to render a frame, each application PE invokes the scene graph's frame-based renderer in parallel. Each of these renderers performs a depth-first traversal of its scene graph, and generates OpenGL graphics commands that are dispatched to Chromium. Chromium then routes the commands to the appropriate rendering server, where an image is rendered. During execution of the graphics commands from parallel streams, certain operations are subject to ordering requirements. Chromium provides facilities for synchronizing multiple streams of graphics commands. These operations consist of semaphores and barriers [Humphreys et al. 2001; Igehy et al. 1998]. Note that the Chromium's barriers and semaphores are implemented in the rendering servers and do not block PE program execution. Parallel applications must provide their own execution synchronization.

For example, MPI-based applications can use *MPI_Barrier()* to create an execution rendezvous point.

In the sections that follow, we describe the implementation of the synchronization operations needed to enable parallel rendering using parallel scene graph operations with Chromium. The discussion focuses upon the elements of scene graph and application infrastructure needed to support distributed memory rendering with Chromium, with the objective being sort-first parallelism.

4.1 Synchronizing Global Operations

Framebuffer clear operations, which initialize the color and depth planes of the framebuffer, typically occur prior to any other rendering. In serial applications, a *glClear()* command that is dispatched to OpenGL prior to any other graphics commands will be executed prior to later commands. Such ordering is not guaranteed when multiple applications are issuing parallel graphics streams. OpenRM uses a Chromium barrier to ensure that the framebuffer clear completes on all render servers before any render server begins executing draw commands.

OpenRM supports a number of framebuffer clear operations in a "framebuffer clear" "scene parameter," which is a scene graph construct. These operations include: (1) filling the color planes with a solid color; (2) filling the color planes with an image, perhaps tiling the image to fill the framebuffer; (3) filling the depth planes with a single value; and (4) filling the depth planes with a depth image, perhaps tiling the image to fill the depth buffer All such framebuffer clear operations are synchronized in the sort-first parallel implementation using a single Chromium barrier, which is managed by OpenRM itself from the framebuffer clear scene parameter. Each application PE must therefore include such framebuffer clear operations as part of its scene graph so that all graphics streams are synchronized. The number of participants in the Chromium barrier is specified to Chromium by the scene graph itself, and the application must specify the number of application PEs that will be dispatching graphics commands in parallel to OpenRM as part of the scene graph initialization process.

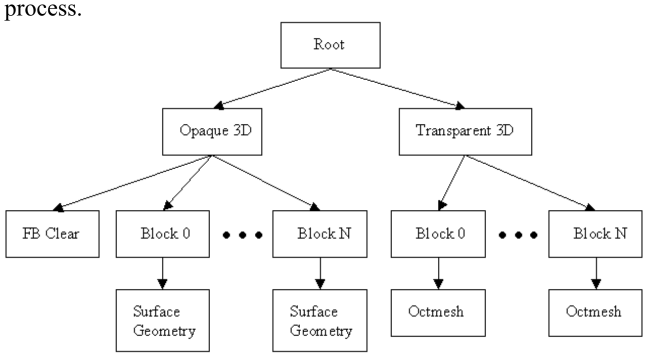

Figure 2. Scene Graph Topology

Figure 2 shows the scene graph topology created by each application PE. During rendering, the OpenRM renderer performs a depth first, left-to-right traversal of the scene graph, issuing graphics commands. OpenRM's multi-pass rendering traversal first processes the opaque 3D objects (which appear in the left part of Figure 2) followed by the transparent 3D objects (which appear in the right half of Figure 2). Note that the framebuffer clear appears as the first item processed during the first rendering pass.

In addition to framebuffer clears, we must also synchronize execution of the Swapbuffers command. Swapbuffers, like the framebuffer clear, is an operation that has global impact. Without Swapbuffers synchronization, one rendering server might execute a Swapbuffers call before the graphics commands from all application PEs have been executed. Whereas the framebuffer clear must be specified by the application through the use of a scene graph construct, the Swapbuffers call is internal to the scene graph renderer itself, and requires no explicit application action.

4.2 Synchronizing Draw Operations

In addition to Swapbuffers and framebuffer clears, other types of drawing operations are also subject to ordering constraints in the sort-first parallel architecture. When rendering transparent objects with "over" compositing, the primitives must be drawn in back-to-front order to produce the correct result. Volume rendering is a good example of an application that requires ordering of many transparent primitives.

OpenRM provides a volume rendering primitive known as an *octmesh*, which is a 3D version of a quadmesh. The octmesh is a procedural primitive that generates geometry in immediate mode, and uses the 3D texturing capabilities of OpenGL to achieve hardware accelerated volume rendering. Each octmesh primitive generates its geometry in the correct back-to-front order, using the current model and view transformations to determine the rendering order of its geometry. While correct geometry order is guaranteed within a single octmesh primitive, there is no such ordering guarantee amongst multiple octmeshes on a single PE, or amongst all octmeshes on all PEs. The need for correct transparency ordering is not specific to OpenRM – it plagues any scene graph implementation.

We can solve the first problem – correct render order of all octmeshes on a single PE – by using an OpenRM callback function. The *render order* callback, invoked during the view-stage traversal during rendering, is used to specify the rendering order for all first-generation children of a given scene graph node. When each PE creates its scene graph nodes, it will create one node per block of data for each rendering pass (as shown in Figure 2.) Inside the render order callback, each PE first computes the correct depth order of its blocks, then returns a list indicating the order in which the children should be rendered. The combination of the render order callback and the correct depth ordering created by the octmesh primitive results in the proper depth ordering of all transparent volume rendering primitives within a single application PE. Because of the block decomposition used to partition the large data into smaller units, there exist *many* such octmeshes on a given PE, and they must be rendered in sorted order.

While the render order callback solves the ordering problem on a single PE, the scope of ordering requirements spans the total set of graphics commands from all application PEs. In the case of parallel volume rendering, which uses ordered OpenGL blending, it is possible for the correctly ordered individual streams to be executed out of order on the rendering servers. Figure 3 shows what happens when this occurs.

Only when "global" ordering constraints are enforced amongst all streams will correct rendering occur, as shown in Figure 4. The scope of inter-PE ordering requires global knowledge of all data blocks and requires an implementation that synchronizes the rendering order of all blocks of data on all PEs. In our prototype, each application PE has information about all grid blocks, resulting in an N-way replication of block metadata, where N is the number of application PEs. While each application PE has a copy of the block decomposition metadata, the underlying data are not replicated across all nodes. The size of the metadata in our application is only a few integers per block. The metadata describe each block's size and position in space. Referring back to Figure 2, the scene graph nodes labeled "Block 0," "Block 1" and so forth in the transparent 3D subtree will be present on all application PEs. However, not all such nodes will contain renderable children. The presence of children under each of the "Block N" nodes is a function of whether or not an application PE is responsible for visualization and rendering of that particular block in the block decomposition.

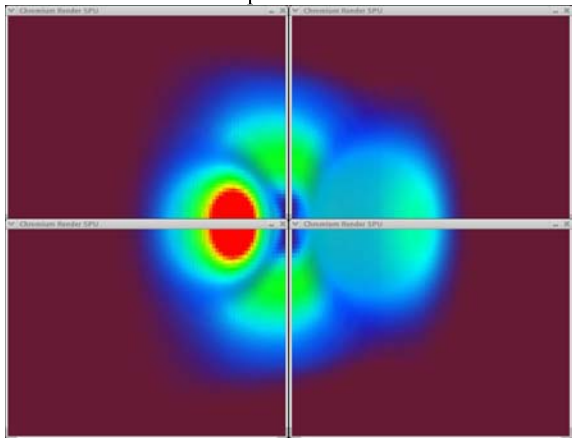
Figure 3. Volume Rendering with no Interprocessor Ordering

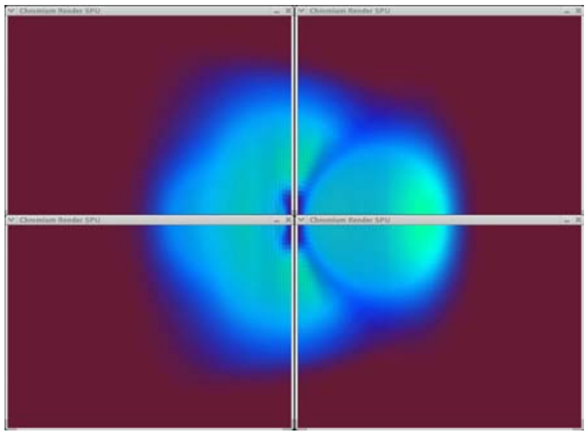
Figure 4. Volume Rendering with Interprocessor Ordering

The amount of block metadata, and the number of data blocks, is entirely problem-specific. The example dataset we used for these tests consists of a 640x256x256 grid that was decomposed into 10x4x4 sub-blocks. Such a decomposition ratio is not uncommon, and reflects the desire to achieve good load balancing in parallel visualization. The sort-first approach benefits from smaller block sizes in the form of reduced data duplication on each graphics server, as will be discussed in the next section.

Equipped with a complete set of block metadata, each application PE determines the correct global order in which all blocks will be rendered. Synchronization is enforced by Chromium barriers executed from the application using the OpenRM render-order callback. OpenRM provides "wrapper functions" that simplify creation and use of Chromium barriers for just this purpose.

Our implementation, which uses OpenRM for scene graph services and Chromium for distributed rendering, is a very flexible and widely applicable system. OpenRM's "parallelization" consists of rendering synchronization using Chromium barriers. OpenRM does not depend upon features of any parallel processing environment except for Chromium barriers for rendering synchronization. Given the absence of any parallel processing framework dependency, the OpenRM and Chromium combination may be used by all parallel programs regardless of the parallel programming environment[1].

5. Results

In this section, we present performance profiling of a prototype sort-first, distributed memory parallel volume rendering application built using OpenRM and Chromium. We focus on the data traffic patterns and loads required by the application in varying configurations of rendering servers and application PEs. We are interested in showing the impact on data traffic patterns and loads resulting from a change in the number of application PEs and/or number of display nodes. We are also interested in showing the reductions in data traffic resulting from use of scene- and view-specific information.

The scientific data we used for these tests is the results of a turbulent flow simulation, and consists of floating point fluid density values on a 640x256x256 grid. The full grid is decomposed into 64x64x64 blocks arranged into a 10x4x4 block grid. Blocks are assigned to each application PE on a round-robin basis, regardless of the number of application PEs. Round-robin assignment typically results in a block distribution with little spatial coherency between blocks on a PE, but also tends to result in favorable load balancing characteristics. Because there is no duplication of raw scientific data, and because we are using a distributed memory parallel implementation, one of the primary benefits of our approach is the ability to perform renderings of data that far exceed the RAM size of any single processor. A typical rendering is shown in Figure 5.

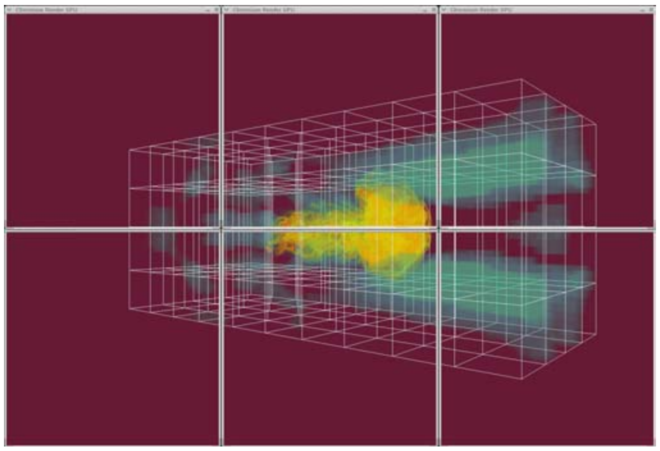
Figure 5. Sort-first parallel volume rendering of Turbulence Simulation Data using OpenRM Scene Graph and Chromium. This image was created using six rendering servers and thirty-two application PEs.

[1] Certain combinations present known caveats: our examples use MPI for the application parallelization, and each PE invokes OpenRM's multithreaded renderer. MPI programs can spawn multiple threads, but the converse is not true.

5.1 Performance Metrics

The performance numbers we present below show the amount of 3D texture data moving between the application PEs and the rendering servers. The 3D texture data is used for hardware accelerated volume rendering. The volumetric model is rotated 360 degrees about the Y-axis over the course of eighteen frames. We measured the amount of data sent from all application PEs to the Chromium rendering servers using Chromium's *perf* SPU, which reports the size and number of objects flowing through a stream per frame.

Our tests are designed such that we report the amount of 3D texture traffic inbound to each of the parallel rendering servers. Keep in mind that Chromium's *tilesort* SPU, which is effectively resident on each application PE and receives directives from OpenRM, is responsible for routing graphics commands from the application PE to the rendering servers. The amount of 3D texture data outbound from all application PEs is equivalent to the amount of 3D texture data inbound to all rendering servers, so we report only the amount of inbound 3D texture data at each frame.

5.2 3D Texture Traffic in Sort-First Parallel Volume Rendering

The first example shows the total amount of 3D texture data inbound to all rendering servers on each frame. In this example, there are six rendering servers, but a varying number of application PEs during each run. Figure 6 shows that the amount of 3D texture traffic inbound at all parallel servers for a given dataset is a function of the number of displays, not the number of application PEs. For the rest of our performance tests, we use the observation shown by Figure 6 that the total data transmitted for a given scene does not vary with the number of application nodes, but varies only with the number of displays.

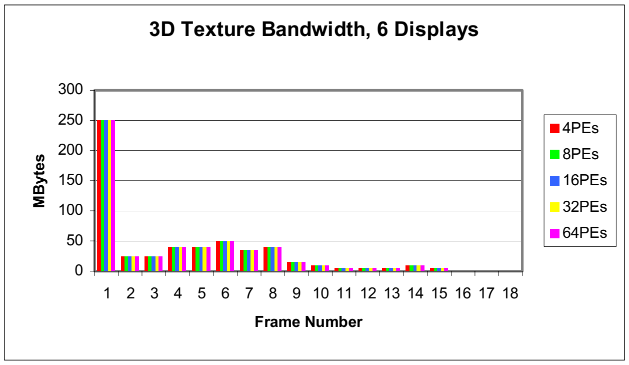

Figure 6. Amount of 3D texture data inbound to all rendering servers on each frame, varying the number of application PEs.

The first frame is the most expensive because data are being sent from all application PEs to all rendering servers. The amount of 3D texture data sent from all PEs totals 160MB, but the amount of 3D texture data inbound at all rendering servers totals 250MB. This overhead represents the fact that some blocks of 3D texture data are sent to more than one rendering server, since one 3D volume block may project onto more than one display tile. As the model rotates and the primitives move from tile to tile, the 3D texture data must then be sent to a different rendering server.

The amount of traffic generated during incremental sends is much less than the cost of the initial send. In the six-display configuration for this particular dataset, no additional data transfer is required after about frame number 15: as we continue to rotate the model about the Y-axis, there is no additional 3D texture data traffic. There is some additional traffic required to transmit the immediate mode geometry and other graphics calls, which total about 9K per block in this particular example, or a total of about 1.5MB for all blocks. We are treating this additional traffic as a constant, and are ignoring its effects in this discussion.

The sort-first overhead in the first frame is the result of needing to duplicate 3D texture data that covers more than one tile. In this case, the source data, which is 160MB in size, produces 250MB of traffic, representing an efficiency of about 65% in this particular configuration of parallel rendering servers and data block decomposition. The level of efficiency is dependent upon these two factors, which are configuration and dataset dependent.

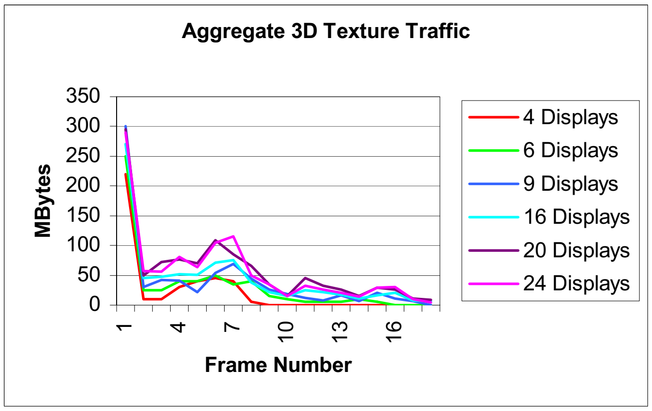

Figure 7. Per-frame 3D texture data inbound to Chromium rendering servers during a parallel sort-first volume rendering transformation sequence.

Figure 7 shows the amount of 3D texture traffic inbound to a varying number of graphics servers from six application nodes. Increasing the number of rendering servers increases the amount of total traffic. Such increases are apparent in all frames throughout the transformation sequence, not just at the first frame. The increase reflects the additional overhead incurred by the sort-first approach. The additional costs that occur with more rendering servers reflects the fact that a given block covers an increasing number of tiles, and must therefore be sent to an increasing number of rendering servers. The exact amount of overhead increase is dependent upon projected block size and the number of rendering servers. Smaller block sizes will result in fewer overlaps, and decreased duplication.

In Figure 8, we see the amount of 3D texture data inbound to each of the six parallel rendering servers in a six-display configuration. The bandwidth requirements in a switched network are effectively the maximum of each inbound data streams for each of the six servers. In the first frame, Server-1 consumes the most bandwidth at about 60MB. Later in the run, Server-1 peaks at 20MB of bandwidth at about frame number four.

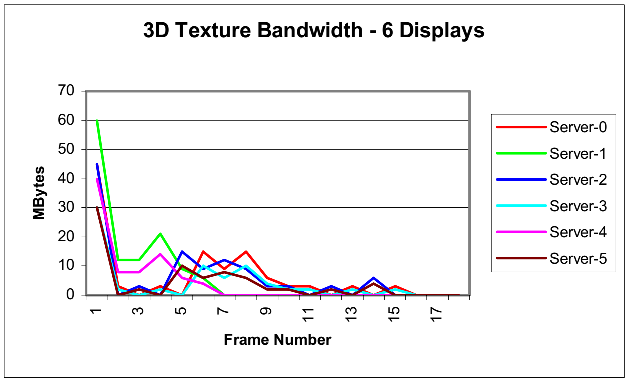

Figure 8. Per-frame 3D texture bandwidth for six parallel rendering servers.

In contrast to Figure 8, Figure 9 shows the maximum bandwidth requirement for each of the configurations of parallel rendering servers we tested in our runs. If the application were to rotate the model about the Y-axis a second time, there would be very little additional traffic since the 3D textures needed are already loaded onto the rendering servers. A different transformation sequence, such as rotating the model about the X-axis, would generate additional 3D texture data traffic. Generally speaking, the bandwidth requirements drop as the number of servers is increased in a switched network environment, even though the aggregate amount of data transferred increases. Our 9-display configuration goes contrary to this observation, but it reflects a worst-case block-to-renderer mapping for this particular problem.

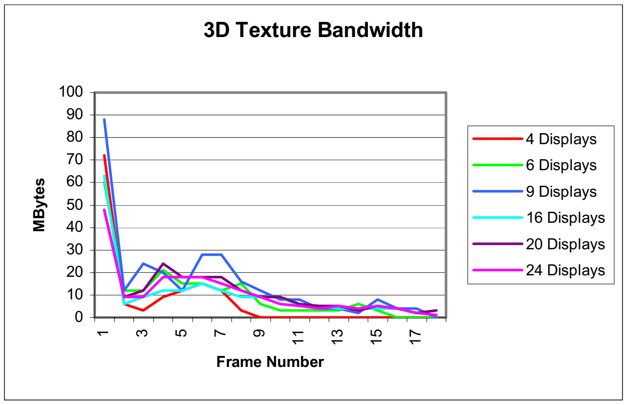

Figure 9. Per-frame 3D texture bandwidth for several configurations of parallel rendering servers.

5.3 Performance Gains from Scene-Specific Knowledge

One of the motivations in our work is to measure the performance gains realized when using scene-specific knowledge to accelerate rendering operations in sort-first parallel architectures. The parallel sort-first volume rendering application used to generate performance numbers in the previous section was extended to use distance-based, level-of-detail (LOD) model switching. For brevity, we refer to such view-dependent model selection simply as "LOD" in the remainder of this discussion. The basic idea is that objects that are "far away" from the viewer are rendered using lower resolution models, and those "close to" the viewer are rendered using higher resolution models. In contrast to previous work in the areas of volume visualization using multiresolution textures [LaMar et al. 1999, Weiler at al. 2000], our emphasis is upon view-dependent LOD selection in a distributed memory parallel environment, rather than focusing on optimizing use of limited texture memory on a single resource, or on methods for creating optimal LOD textures.

In the case of this particular application, a full resolution model is the 3D texture produced by conversion of the source scientific data into a block of RGBA voxels. The low resolution model is a 3D texture that is 1/64 the size of the original texture, and is created by a two-pass bilinear voxel interpolation that reduces the original texture's size by a factor of four in each of the three texture dimensions. The expectation is that use of LOD will reduce the amount of 3D texture data traffic in the parallel application. Figures 10 and 11 below show close-up views of renderings when using full resolution and LOD textures, respectively. Opacity values in LOD textures are modified to produce visual consistency, as described in [Laur and Hanrahan 1991].

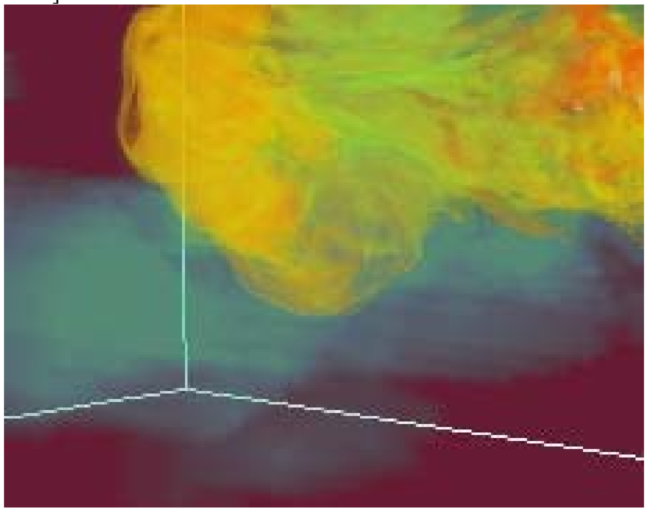

Figure 10. Sort-first parallel volume rendering using full-resolution 3D textures.

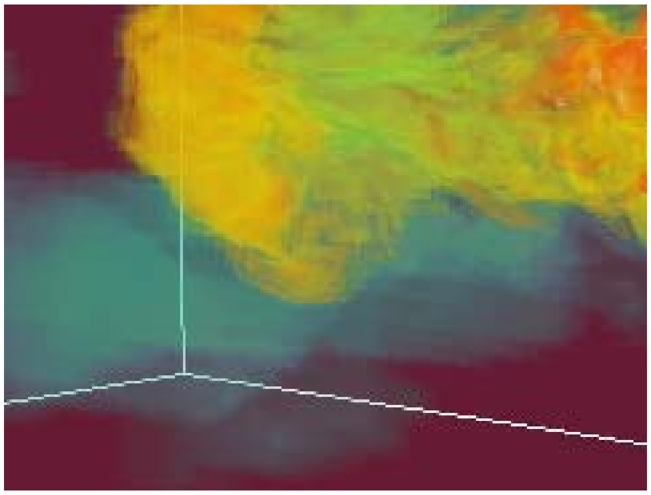

Figure 11. Sort-first parallel volume rendering using LOD to select between full- and reduced-resolution textures.

The 3D texture data traffic profile when using LOD is indeed different than when using only full resolution textures. Figure 12 shows the total amount of 3D texture traffic on each frame for a varying number of parallel rendering servers.

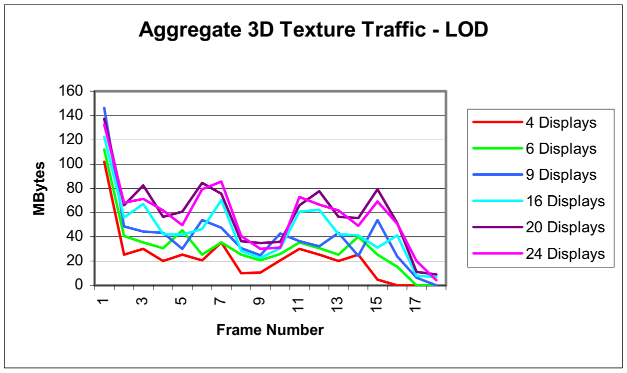

Figure 12. Per-frame 3D texture data inbound to Chromium rendering servers during a parallel sort-first volume rendering transformation sequence using LOD.

These results show a dramatic reduction in 3D texture traffic in the first frame, but an increased amount of traffic in later frames. A direct comparison for a six-display configuration is shown in Figure 13. With our testing protocol, we see that LOD consumes less bandwidth for most of the transformation sequence, but in the end consumes slightly more. During the initial frames of the LOD transformation sequence, some of the 3D textures are sent using full resolution, and others are sent using reduced resolution. As the model rotates over time, the application will have sent all 3D textures in both full and reduced resolution models. This fact accounts for the LOD method sending slightly more 3D texture data over the course of the entire application. In cases where only a few views are required, LOD approaches result in substantial gains in terms of 3D texture traffic.

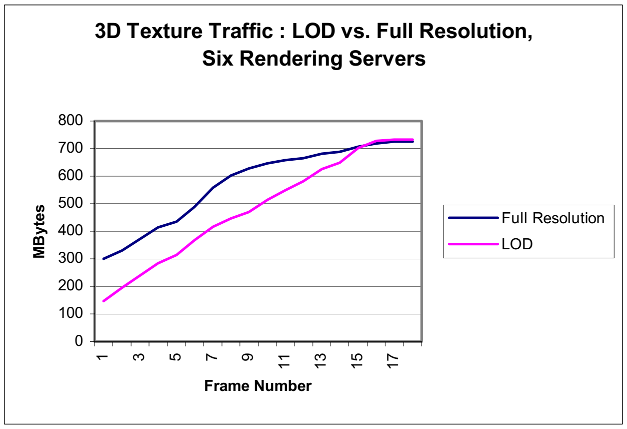

Figure 13. Comparison of amount of 3D texture data moved when using LOD and full resolution textures.

Peak bandwidth rates when using LOD are shown below in Figure 14. Again, we see a dramatic reduction in bandwidth requirements in Frame 1 as compared to the full resolution sends. In later frames, LOD has occasional bandwidth requirements that exceed those of full resolution sends as duplicated data lands on one or more parallel rendering servers.

5.4 Discussion

The amount of traffic generated by sort-first architectures is influenced by many factors. While sort-first is comparatively immune to sort-last's sensitivity to final image size, increasing the number of displays will increase the amount of transmitted data as objects cover an increasing number of tiles, and must be sent to multiple rendering servers. When renderable objects are sent to more than one rendering server, data duplication results. Such duplication is the overhead inherent in the sort-first approach. The limit of such duplication in the first rendered frame in our examples varies from about 1.4 in the four-display case to about 1.8 in the 24-display case, as shown below in Figure 15. The increased number at nine displays reflects an unfavorable block-to-tile mapping that happens to occur in that particular configuration. More tests are needed to examine how this number grows as more parallel rendering servers are added to the system.

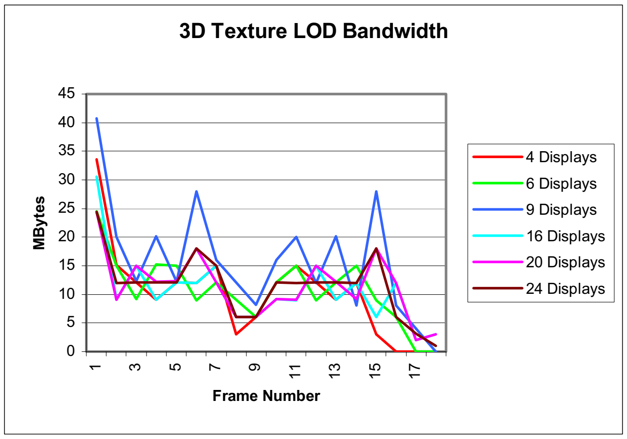

Figure 14. 3D texture bandwidth when using LOD to accelerate sort-first parallel volume rendering.

The amount of data duplication is also dependent upon the spatial partitioning of the original data. For a given set of view and model transformation parameters, large data blocks will produce greater duplication than smaller blocks: larger blocks are more likely to appear on multiple displays than smaller blocks. Similarly, blocks with a compact shape, such as cubes, are likely to produce less duplication than narrow, long blocks for the same reason. The data decomposition strategies that favor reduced duplication may be in contention with conditions that result in better application performance: parallel visualization applications may perform better when executing fewer I/O operations while loading data. A thorough study that explores this relationship is beyond the scope of this paper.

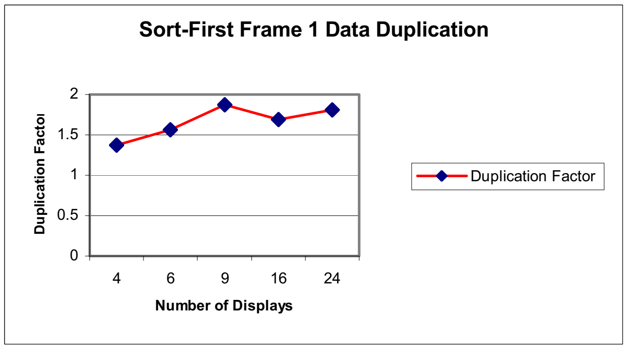

Figure 15. The amount of data duplicated in sort-first increases as the number of rendering servers increases.

6. Future Work

In the work we have described, the parallel application and scene graph infrastructure contain very little knowledge that they are in fact running in parallel, or using a distributed memory tiled display system. The main benefits of such an approach are a high degree of portability and ease of development. While our parallel applications were written using the MPI programming model, the scene graph system is completely independent of MPI or any other application-level parallel processing framework. The scene graph system only uses Chromium barriers to enforce synchronization at key points during the rendering process. As a result, the OpenRM/Chromium combination can be used in any parallel application to implement sort-first, distributed memory parallel rendering.

Were we to take advantage of per-display information, we may be able to realize improved use of resources. For example, during geometric transformation, retained mode objects (display lists, textures) are automatically broadcast by the tilesort SPU to a new rendering server when needed. Once sent, the retained mode objects are "immortal" in the sense that they persist until the end of the application run: they still take up space on the rendering server, even if not used. Adding the ability to "age" retained mode objects and to schedule them for removal when no longer needed would make more efficient use of resources. It is not clear if such a modification should occur in Chromium, which reflects the OpenGL API, or in the scene graph system, which manages display list creation and usage. Placing such functionality into Chromium represents a departure from the OpenGL specification. Placing the functionality into the scene graph breaks the metaphor we have presented: neither the application nor the scene graph system have any awareness that they are in fact using a tiled display system built from multiple rendering servers. Ideally, a least-recently used (LRU) algorithm or a time-critical method [Li and Shen 2002] would manage the available graphics hardware resources and maximize their use downstream from the application code. The present system has no such per-display knowledge, but the combination of OpenRM and Chromium provides a number of opportunities for such optimizations.

Our examples show use of LOD methods to switch between models of varying resolution to improve rendering performance in a distributed memory parallel context. The performance gain, while both substantial and measurable, depends on several problem-specific factors. The concept of reducing graphics load by model switching is not new [Rohlf and Helman 1994], and there is opportunity to more fully exploit this notion in future projects. View and model transformations can be used to reduce bandwidth into the application by directing data I/O mechanisms to load reduced resolution models of data for far away objects, thereby reducing memory and I/O requirements on the application PEs, as well as to accelerate application execution. Data-specific knowledge may also be used to reduce texture bandwidth requirements [Li and Kaufman 2002]. On the rendering side, use of scene specific knowledge can be used to honor "render budgets" that are a function of frame rate, bandwidth, or other factors [Rohlf and Helman 1994; Li and Shen 2002].

Comparing sort-first and sort-last in terms of bandwidth requirements is difficult. Sort-last bandwidth requirements are a function of the final image display size, which will be prohibitively large in high-resolution tiled display environments, as well as the number of processors participating in the compositing process. In contrast, sort-first bandwidth requirements are a function of contents of the scene, the partitioning of scene data across application processors, and the number of graphics servers. Sort-first bandwidth requirements are independent of the final image size. More work is needed to fully explore the effect on sort-first bandwidth requirements of data and scene partitioning strategies. One approach to such a study is to compare the sort-first bandwidth requirements using different data partitioning strategies.

7. Conclusion

We have presented the results of a project intended to demonstrate the performance characteristics of a distributed memory parallel visualization application that uses a sort-first rendering architecture. The sort-first infrastructure is created through the combination of the OpenRM Scene Graph, a scene graph API designed for use by high performance applications, and Chromium, a stream-oriented framework that implements parallel and distributed memory OpenGL. Our work describes how the scene graph, which is used in a distributed memory context, is augmented to use synchronization operations within Chromium to enable distributed memory parallel operation. One of our demonstration applications shows a novel use of the scene graph infrastructure to implement a distributed memory sorting algorithm, which is needed to perform correct, view-dependent parallel rendering in a sort-first architecture. Our application, which uses round-robin block-to-PE assignment in block-decomposed parallel visualization, would not have been possible using a sort-last approach. Our approach shows performance characteristics that scale well, but which are also susceptible to jitter resulting from variance in network bandwidth requirements between the parallel application and the parallel rendering servers. Performance of our system, as a function of data transmission requirements, is sublinear with respect to number of parallel rendering nodes. In high resolution display configurations, our examples show less bandwidth requirements than needed for sort-last approaches. The sort-first bandwidth requirements are further reduced using scene specific knowledge to accelerate rendering.

8. Acknowledgement

This work was supported by the U. S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under SBIR grant DE-FE03-02ER83443. The authors wish to thank Randall Frank of the ASCI/VIEWS program at Lawrence Livermore National Laboratory and the Scientific Computing and Imaging Institute at the University of Utah for use of computing facilities during the course of this research. The Argon shock bubble dataset was provided courtesy of John Bell and Vince Beckner at the Center for Computational Sciences and Engineering, Lawrence Berkeley National Laboratory.

References

OPENRM SCENE GRAPH. 2000. http://openrm.sourceforge.net/, http://www.r3vis.com.

D. BARTZ, D. STANEKER, W. STRASSER, B. CRIPE, T. GASKINS, K. ORTON, M. CARTER, A. JOHANNSEN, AND J. TROM. 2001. Jupiter: A Toolkit for Interactive Large Model Visualization. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 129–134, San Diego, CA.

E. W. BETHEL, R. J. FRANK, AND J. D. BREDERSON. 2002. Combining a Mulithreaded Scene Graph System with a Tiled Display Environment. In *Proc. SPIE Stereoscopic Displays and Virtual Reality Systems*, volume 4660, pp. 430–436, San Jose, CA.

B. CABRAL, N. CAM, AND J. FORAN. 1994. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proc. IEEE Symposium on Volume Visualization*, pp. 91–98, Washington, D.C.

W. CORREA, J. T. KLOSOWSKI, AND C. SILVA. 2002. Out-of-Core Sort-First Parallel Rendering for Cluster-Based Tiled Displays. In *Proc. Eurographics Workshop on Parallel Graphics and Visualization*, pp. 89–96, Blaubeuren, Germany.

J. FOLEY, A. VAN DAM, S. FEINER, AND J. HUGHES. 1990. Computer Graphics, Principles and Practice (2nd Edition). Addison-Wesley.

A. GARCIA AND H.-W. SHEN. 2002. An Interleaved Parallel Volume Renderer with PC-clusters. In *Proc. Eurographics Workshop on Parallel Graphics and Visualization*, pp. 51–59, Blaubeuren, Germany.

A. HEIRICH AND L. MOLL. 1999. Scalable Distributed Visualization Using Off-the-Shelf Components. In *Proc. IEEE Parallel Visualization and Graphics Symposium*, pp. 55–59, San Francisco, CA.

G. HUMPHREYS, M. ELDRIDGE, I. BUCK, G. STOLL, M. EVERETT, AND P. HANRAHAN. 2001. WireGL: A Scalable Graphics System for Clusters. In *Proceedings of ACM SIGGRAPH 2001,* ACM Press/ACM SIGGRAPH, New York. E Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, pp. 129-140.

G. HUMPHREYS, M. HOUSTON, R. NG, R. FRANK, S. AHERN, P. D. KIRCHNER, AND J. T. KLOSOWSKI. 2002. Chromium: A Stream Processing Framework for Interactive Rendering on Clusters. In *Proc. ACM SIGGRAPH*, pp. 693–702, San Antonio, TX.

H. IGEHY, G. STOLL, AND P. HANRAHAN. 1998. The Design of a Parallel Graphics Interface. In *Proc. ACM SIGGRAPH*, pp. 141–150, Orlando, FL.

J. M. KNISS, P. MCCORMICK, A. MCPHERSON, J. AHRENS, J. S. PAINTER, A. KEAHEY, AND C. D. HANSEN. 2001. TRex, Texture-based Volume Rendering for Extremely Large Datasets. *IEEE Computer Graphics and Applications*, 21(4): 52–61.

E. LAMAR, B. HAMANN AND K. JOY. 1999. Multiresoultion Techniques for Interactive Texture-based Volume Visualization. In Proceedings of IEEE Visualization 1999, Computer Society Press, pp 355-361.

D. LAUR AND P. HANRAHAN. 1991. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. In *Proc. ACM SIGGRAPH*, pp. 285–288, Las Vegas, NV.

C. LAW, A. HENDERSON, AND J. AHRENS. 2001. An Application Architecture for Large Data Visualization: A Case Study. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 125–128, San Diego, CA.

W. LI AND A. KAUFMAN. 2002. Accelerating Volume Rendering with Bounded Textures. In *Proc. IEEE Volume Visualization and Graphics Symposium*, pp. 115–122, Boston, MA.

X. LI AND H.-W. SHEN. 2002. Time-Critical Multiresolution Volume Rendering using 3D Texture Hardware. In *Proc. IEEE Volume Visualization and Graphics Symposium*, pp. 29–36, Boston, MA.

S. LOMBEYDA, L. MOLL, M. SHAND, D. BREEN, AND A. HEIRICH. 2001. Scalable Interactive Volume Rendering Using Off-the-Shelf Components. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 115–121, San Diego, CA.

K.-L. MA, J. S. PAINTER, C. D. HANSEN, AND M. F. KROGH. 1994. Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68.

S. MOLNAR, M. COX, D. ELLSWORTH, AND H. FUCHS. 1994. A Sorting Classification of Parallel Rendering. IEEE Computer Graphics and Applications, (14)4, pp. 23-32.

K. MORELAND, B. WYLIE, AND C. PAVLAKOS. 2001. Sort-Last Parallel Rendering for Viewing Extremely Large Data Sets on Tile Displays. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 85–92, San Diego, CA.

C. MUELLER. 1995. The Sort-First Rendering Architecture for High-Performance Graphics. In *Proc. ACM Symposiumon Interactive 3D Graphics*, pp. 75–83, Monterey, CA.

U. NEUMANN. 1994. Communication Costs for Parallel Volume Rendering Algorithms. *IEEE Computer Graphic sand Applications*, 14(4):49–58.

D. REINERS, G. VOSS, AND J. BEHR. 2002. OpenSG: Basic Concepts.http://www.opensg.org/OpenSGPLUS/symposium/Papers2002/, http://www.opensg.org/.

J. ROHLF AND J. HELMAN. 1994. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Proc. ACM SIGGRAPH*, pp. 381–394.

R. SAMANTA, T. FUNKHOUSER, AND K. LI. 2001. Parallel Rendering with K-Way Replication. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 75–84, San Diego, CA.

R. SAMANTA, T. FUNKHOUSER, K. LI, AND J. P. SINGH. 2000. Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 97–108, Interlaken, Switzerland.

R. SAMANTA, J. ZHENG, T. FUNKHOUSER, K. LI, AND J. P. SINGH. 1999. Load Balancing for Multi-Projector Rendering Systems. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 107–116, Los Angeles, CA.

D. R. SCHIKORE, R. A. FISCHER, R. J. FRANK, R. GAUNT, J. HOBSON, AND B. WHITLOCK. 2000. High-Resolution Multiprojector Display Walls. *IEEE Computer Graphics and Applications*, 14(4): 38–44.

G. VOSS, J. BEHR, D. REINERS, AND M. ROTH. 2002. A Multi-Thread Safe Foundation for Scene Graphs and its Extension to Clusters. In *Proc. Eurographics Workshop on Parallel Graphics and Visualization*, pp. 33–37, Blaubeuren, Germany.

M. WEILER, R. WESTERMANN, C. HANSEN, K. ZIMMERMAN, T. ERTL. 2000. Level-of-Detail Volume Rendering via 3D Textures. In Proceedings of 2000 IEEE Symposium on Volume Visualization, Salt Lake City, Utah, United States, pp 7-13.