Ninth Eurographics Workshop on Virtual Environments (2003)
A. Kunz, J. Deisinger (Editors)

A Survey and Performance Analysis of Software Platforms for Interactive Cluster-Based Multi-Screen Rendering

Oliver Staadt, Justin Walker, Christof Nuber, and Bernd Hamann

Center for Image Processing and Integrated Computing (CIPIC) and Department of Computer Science
University of California, Davis

Abstract

We present a survey of different software architectures designed to render on a tiled display. We provide an in-depth analysis of three selected systems, including their implementation of data distribution, sort-first rendering, and overall usability. We use various test cases to analyze the performance of these three systems.

Categories and Subject Descriptors (according to ACM CCS): C.4 [Performance of Systems]: Performance Attributes, I.3.2 [Computer Graphics]: Distributed/Network Graphics, I.3.4 [Computer Graphics]: Graphics Packages, I.3.7 [Computer Graphics]: Virtual Reality.

1. Introduction

Traditionally, multi-screen display environments have been driven primarily by powerful graphics supercomputers, such as SGI's Onyx systems. With features including shared-memory multi-processing and multiple synchronized graphics pipelines, they provided a stable and flexible development platform for high-performance virtual reality and visual simulation applications. Unfortunately, these features come at high cost. Hence, the use of multi-screen projection environments has been limited to a small number of users.

During the past several years, high-performance and feature-rich PC graphics interfaces have become available at low cost. This development enables us to build clusters of high-performance graphics PCs at reasonable cost. An important issue, however, is that the programming model for shared-memory systems and clusters differ significantly. In shared-memory graphics systems, the programmer does not have to worry about issues such as sharing data amongst different processors or distributing rendering information to different graphics engines. In cluster environments, it is necessary to deal with these issues explicitly. The use of clusters for computationally intensive simulations and applications has lead to the development of interface standards such as the *Message Passing Interface* (http://www.mpi-forum.org) and *OpenPBS* (http://www.openpbs.org). We focus on rendering in a multi-display environment. There are two important application areas where multi-display environments are used [14]:

- displaying images at very high resolutions exceeding those of available monitors and/or graphic cards and
- providing a larger field-of-view and better immersion into the scenery.

A larger image can be obtained by using special purpose video processors that split the incoming video signal and distribute it to the connected display systems. This approach, however, increases the area covered by a single pixel, which is not always desired. Increasing the resolution of a displayed image requires the combination of several display-devices into a single display-environment, providing a higher resolution by combining several images. In the past, high-performance computers, such as SGI's Infinite Reality with multiple graphics pipes have been used to drive multi-tiled displays. With the availability of affordable PC-based high-performance graphics cards like the NVidia GeForce- or the 3DLabs Wildcat-series, high-quality rendering is available at relatively low cost. Using a cluster-based approach requires the solution of problems like data-management and -distribution, output-synchronization and event handling. Solving these problems for an application can be very tedious, time-consuming and error-prone, so the usage of libraries providing the necessary support should be considered.

The design and development of platforms for cluster-based multi-screen rendering has become increasingly popular during the past few years [1,4,5,6,7,10,15,16], but a standard solution has yet to be found. Nevertheless, many developers are eager to port existing applications to cluster environ-

ments or to develop new ones. Although development of those platforms has only begun recently, various different architectures have been proposed, some are available as open-source software [1,7,10,15,16].

We provide a survey of different systems designed to render on a tiled display and discuss potential implications on the application development as well as advantages and disadvantages of these designs. We present detailed descriptions of three systems followed by a performance analysis. We defined a set of test-cases and conducted a quantitative evaluation of the usefulness of these systems for different kinds of application scenarios. Our goal is to help developers with the selection of a software platform that is appropriate for their particular application requirements. We do not discuss hardware-related issues, which are also important for building commodity clusters for rendering. We refer the reader to [19] for an overview of different hardware architectures.

The remainder of this paper is structured as follows: After discussing different applications for cluster-based rendering environments in Section 2, we will present the survey of different software platforms in Section 3. Three selected platforms will be evaluated in detail in Section 4. Section 5 contains the results and interpretations of our performance analysis, followed by conclusions in Section 6.

2. Cluster-based Rendering

Cluster-based rendering in general can be described as the use of a set of computers connected via a network for rendering purposes, ranging from distributed non-photorealistic volume rendering over ray tracing and radiosity-based rendering [17] to interactive rendering using application programming interfaces (APIs) like *OpenGL* [18] or *DirectX* [11]. We use the same terminology as is used by X-Windows. A *client* runs the application while the *server* renders on the local display.

Most of the recent research on cluster-based rendering focuses on different algorithms to distribute the rendering of polygonal geometry across the cluster. Molnar et al.[9] classified these algorithms into three general classes based on where the sorting of the primitives occurs in the transition from object to screen space. The three classes are

- sort-first,
- sort-middle, and
- sort-last.

In sort-first algorithms, the display is partitioned into discrete, disjoint tiles. Each rendering node of the cluster is then assigned one or more of these tiles and is responsible for the complete rendering of only those primitives that lie within one of its tiles. To accomplish this, primitives are usually pre-transformed to determine their screen space extents and then sent only to those tiles they overlap with. The required network bandwidth can be high when sending primitives to the appropriate render server, but utilizing knowledge of the frame-to-frame coherency of the primitives can reduce the amount of network traffic significantly. Sort-first algorithms suffer from load balancing due to primitive clustering. Samanta et al. [13] investigated methods to improve load balancing by dynamically changing the tiling. Sort-first algorithms also do not scale well when the number of nodes in the cluster increases. Every primitive that lies on the border of two tiles must be rendered by both tiles. As the number of tiles increases, the number of these primitives increases. Samanta et al. [12] solved this problem by using a hybrid sort-first, sort-last approach.

Sort-middle algorithms begin by distributing each graphics primitive to exactly one *processor*† for geometry processing. After the primitive has been transformed into screen space, it is forwarded to another processor for rendering. Similar to the sort-first approach the screen space is divided into tiles, but each processor is only responsible for rasterization of primitives within that tile. This approach requires a separation of rasterization engine and rendering pipeline, so that primitives can be redistributed. Currently this approach can only be implemented using specialized hardware, such as SGI's InfiniteReality engine.

In sort-last approaches, each primitive is sent to exactly one node for rendering. After all primitives have been rendered, the nodes must composite the images to form the final image. This usually requires a large amount of bandwidth because each node must send the entire image to a compositor.

Tiled displays lead naturally toward a sort-first approach. The screen is already partitioned into tiles, with each tile being driven by a single cluster node. Other approaches require to distribute the primitives to the rendering nodes and distributing the final image to the nodes responsible for tile-rendering. For these reasons the majority of software systems designed for rendering on a tiled display implements a sort-first algorithm.

3. Systems Survey

In our survey we analyzed systems designed to support rendering on a tiled display. All systems evaluated implement to some extent a sort-first method. They vary widely with respect to the way data is distributed among the cluster nodes. Chen et al. [2,3] first looked at the problem of data distribution. Two general models have emerged:

- client–server and
- master–slave.

In the client–server model a user interacts with a single instance of the application that runs on a client node. This client is responsible for generating the geometry and distributing it to the render servers (see Figure 1a). We can distinguish between two rendering modes - immediate mode and retained mode. In immediate mode the client sends the primitives over the network every frame. In retained mode each render server stores primitives it has already been sent to locally for re-use. The client then needs to send only changes to the geometry. This method is usually accomplished through the use of a scene graph.

†The term *processor* is used in a more general sense, not restricted to CPUs or GPUs.

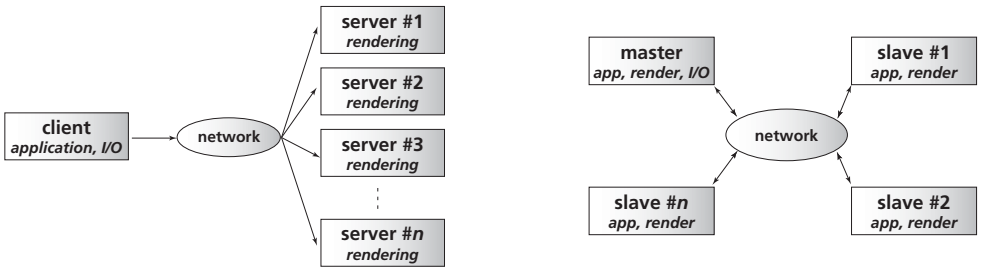

Figure 1: *Different setups for interactive rendering cluster: a) client–server setup. b) master–slave.*

In the master–slave model the application executes on every cluster node. Execution of the application on all nodes must be synchronized to insure consistency among all application instances. Typically, a master node handles all user interaction and synchronizes state changes between all other nodes (see Figure 1b).

The master–slave approach usually requires the least amount of bandwidth. The results of user interactions and other state changes are sporadic and relatively simple to transmit over a network. This approach, however, is not transparent as everything affecting program execution must be considered as input. Timers, random number generation, system calls, or any variables influencing program execution need to be distributed and synchronized among the nodes. This limits the types of inputs a programmer is allowed to use.

The client–server approach is usually fairly transparent to the programmer. The program can be implemented as if it were running on a single machine and the system will handle the rest.

In the next section, we will present software systems from each of the two classes. We will describe their intended uses, how they implement the sort-first algorithm, and potential performance impacts.

3.1. Client–Server

Aura (Broadcast). Aura [16] is a multi-platform API designed for scientific visualization on a tiled display. In Broadcast-mode it implements a client-server model. It provides the user with a scene graph interface to take advantage of frame to frame coherency. In addition to Broadcast-mode, Aura also provides a master-slave configuration called *Multiple Copies*; see Section 3.2 for details. The Broadcast implementation replicates the scene graph on all cluster nodes. Any changes to the scene graph are then broadcast to every node to insure consistency across the nodes. Aura uses MPI for communication between nodes.

Syzygy (Scene Graph). Syzygy [15] is a VR library designed specifically to run on a cluster. It provides support for sound and input device handling. Syzygy provides two programming interfaces, a scene graph API and a master-slave framework; see Section 3.2 for details. The scene graph is implemented as a distributed database that is modified using Syzygy's own messaging protocol, allowing Syzygy to run as a multi-threaded applications. The user may add and remove render servers during application execution. It is also possible to reuse existing clients when switching applications.

Parallel iWalk. iWalk [4] is a system for visualizing extremely large models for Linux or Windows. It uses an out-of-core algorithm and storage scheme to visualize models that are too large to be loaded into main memory. A client application handles user interaction; each rendering server executes the basic iWalk code which uses the prioritized-layered projection algorithm [8] to determine a set of nodes visible only to that server.

For and in-depth analysis of OpenSG [10] and Chromium [8] see Section 4.1 and Section 4.2, respectively.

3.2. Master–Slave

Aura (Multiple Copies). Like its broadcast counter part, the multiple copies version of Aura [16] also provides a scene graph API. Following the master–slave model all cluster nodes run an instance of the application. All supported user interactions are broadcast to every slave and frame-buffer swaps are synchronized.

Syzygy (Master–Slave). Syzygy [15] presents an alternate framework for when the scene graph approach is inappropriate. The master-slave framework provides automatic sharing of certain data across multiple instances such as user input, timestamps, random number seeds, and a viewing matrix. Syzygy also provides a mechanism for sharing other types of data as well.

VRJuggler [1] will be discussed in Section 4.3.

4. Systems Evaluation

We chose three systems for an in depth comparison and performance analysis. The systems were chosen for the following reasons:

- They were designed to execute on a multi-platform, heterogeneous cluster.
- They were designed to be used with little or no modification by the user.
- They were all being widely used by members of the graphics community.
- They were all open source.

We discuss how the system implements the sort-first algorithm and potential performance ramifications. We also

discuss ease of use and flexibility, i.e., whether the system supports trapezoidal or overlapping tiles that are useful for tiled displays in which the projector's tiles are trapezoidal or overlapping projection tiles. Finally we discuss how the three systems compare to each other.

4.1. OpenSG

OpenSG [10] is a scene graph API, similar to *Open Inventor* and *OpenGL Performer*, designed to support simultaneous multithreading in a scene graph. Whenever the user makes changes to the scene graph it must explicitly be stated which nodes and fields have been changed. Each of the threads can then use this information to update their local copies of the scene graph. The mechanisms used to support simultaneous multi-threading are easily extended to support rendering on a cluster. OpenSG uses a client-server setup with the scene graph replicated on every node. The user interacts with the client, manipulating the scene graph. These changes are stored in a change list and broadcast to every cluster node every frame.

OpenSG provides a sort-first algorithm to use a cluster to display an image on a single display, and an algorithm for rendering on a tiled display. The sort-first algorithm dynamically changes the tile extents to balance the rendering load across all cluster nodes. When rendering on a tiled display, OpenSG divides the screen into M x N evenly spaced rectangular tiles with a render server assigned to each tile. OpenSG also supports trapezoidal or different sized tiles, but the configuration of these tiles is difficult. It is also difficult for OpenSG to support a cluster that does not easily fit into a M x N configuration.

Rendering is performed by the cluster servers. Each server is assigned a single tile of the display and adjusts its viewing frustum accordingly. Bounding boxes are calculated for each geometry node and view frustum culling is performed on a per-node basis. This can lead to problems when the complete geometry is in a single node, e.g., when rendering an isosurface. In this case every server must push the entire geometry through the pipeline and rely on OpenGL to perform clipping. This results in a big performance loss when servers need to process and render geometry that do not lie in their tile.

By storing the scene graph on each server and transmitting only changes to the scene graph, OpenSG does not require a high bandwidth for static scene graphs. The cost to transmit a few transformation matrices is small when compared to the cost of transmitting the geometry every frame. A highly dynamic scene graph with changing geometry and textures results in a large amount of information being sent every frame, putting a strain on networking resources.

4.2. Chromium

Chromium [7] is the successor to WireGL [5,6], a system to support OpenGL applications on a cluster. Chromium replaces the system's OpenGL library with its own, directly operating on the stream of OpenGL graphics commands. Chromium provides *Stream Processing Units* (SPU). Each SPU has as its input a stream of graphics commands, performs some operation on these commands, and passes them on. SPUs can be chained together to perform combined operations. Some basic SPUs include *render* which passes the stream to the system's local implementation, *pack* which packs the stream into a buffer for transmission to cluster servers, and *print* which outputs the stream in a human readable format.

By replacing the OpenGL library Chromium can theoretically run any application using OpenGL. In the current version Chromium does not completely implement all features of the OpenGL 1.2 specifications. Chromium does not implement OpenGL 1.2 imaging functions related to histogram, min/max, convolution and colortables, and display lists are not completely conformant (Chromium tracks changes to the OpenGL state, and changes to the state within a display list are not visible to the state tracker). Any OpenGL extensions the user uses must also be implemented by Chromium.

Chromium supports rendering on a tiled display via the *tilesort* SPU. The user can specify rectangular tiles of different sizes, overlapping tiles, and tiles that do not lie on a grid. It is also possible to specify a geometric transformation in the form of a matrix to support trapezoidal and sheared tiles.

To render geometry Chromium pre-transforms each vertex and maintains a screen-space bounding box of all geometries. When the send-buffer is filled, Chromium determines which tiles the bounding box overlaps with and sends the data to those tiles. The send-buffer can be explicitly flushed using glFlush. Chromium assumes that geometry which lies together temporally also lies together spatially. This is a reasonable assumption when rendering an isosurface or other types of mesh data. In the worst case, the bounding box of the geometry covers all tiles, requiring the data be transmitted to all tiles. Since Chromium transmits the geometry every frame, it is possible that substantial network traffic is generated.

4.3. VRJuggler

VRJuggler is a framework for virtual reality applications that handles window and viewport management, user interactions via various input devices, sound, and cluster support[1].

VRJuggler implements a master-slave model for cluster support. The application is started on each cluster node; one of the nodes is designated as a master and waits for all slaves to connect. The master node is responsible for synchronizing execution of all slave nodes, user input can occur at any node. The input is then broadcast to all other nodes in order to maintain a consistent application state. VRJuggler provides support for a large number of commercial interaction devices, such as trackers by Ascension, Polyhemus, and InterSense. It does not provide inherent support for random number generators or system calls. For any input device that is not supported by VRJuggler, an input device interface must be provided. Currently three types of input devices are supported: digital, analog, and positional.

Each cluster node can drive one or more tiles. The four corners of each tile are specified by the user, allowing the use of overlapping tiles and trapezoidal tiles.

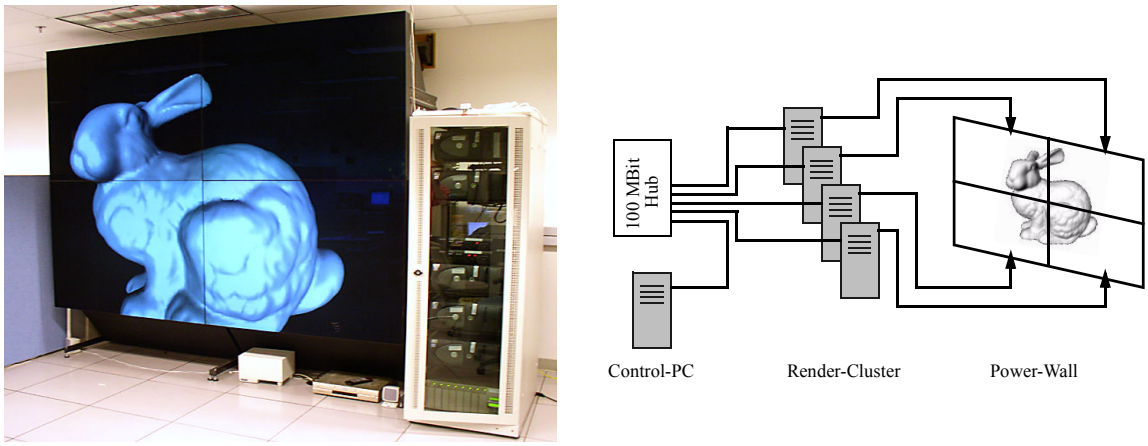

Figure 2: *Setup of our test bed showing the two-by-two tiled display wall and the rendering cluster.*

VRJuggler does not perform any geometry culling; it simply lets OpenGL clip to the window borders. This means that running VRJuggler on a cluster will show little to no performance improvement. Each tile must still send the complete geometry through the pipeline.

4.4. Comparison

Of the three systems compared VRJuggler is expected to be running at the most consistent frame rates, running at about the same rate as running on a single machine, not faster and not much slower. Broadcasting user interactions requires very little bandwidth and should not affect the performance of VRJuggler at all. VRJuggler's lack of geometry culling results in a lot of duplication and unnecessary work being done. Tiles render geometry that does not lie within their viewing volume.

OpenSG performs well when rendering static geometry that is distributed among several nodes. Nodes will only be rendered by tiles their bounding boxes overlap with. With increasing bounding box size, more tiles will be forced to render that node and more redundant work must be done. In the worst case the bounding box covers all tiles and it takes as long to render on a cluster as it does on a single machine. If the geometry changes, these changes must be broadcast to all cluster nodes. Frequent changes can significantly slow down system performance.

Since Chromium transmits the geometry every frame it requires a large amount of bandwidth. By creating bounding boxes for the geometry and transmitting it to only those nodes it overlaps with, Chromium attempts to speed up rendering and reduce the amount of data sent over the network. This only works if the primitives lie close together spatially and temporally. If they do not the data must be sent to all servers.

5. Performance Analysis

We split our analysis in two parts. The first part covers system behavior and performance with a static number of polygons (Section 5.2). For the second part we used a highly dynamic environment with objects being created and deleted every frame (Section 5.3).

5.1. Test Environment

Our test-environment comprises 5 Linux-PCs, each running RedHat 7.3 with a GeForce3 graphics card, a 2GHz Intel Pentium Processor, and 512 MByte of memory. The machines were connected to a 100 MBit switch. While one PC was used to control the application, the remaining four were used to drive a two-by-two tiled display wall (see Figure 2), with each tile driven at a resolution of 1280 x 1024.

For each system we used OpenSG as the underlying rendering API. This is possible because OpenSG uses OpenGL as its underlying graphics API. Since Chromium replaces the OpenGL library, Chromium is able to support OpenSG applications. VRJuggler also supports OpenSG as a possible graphics API. Using OpenSG on all systems insures that any performance differences are the same on all systems. For instance, when OpenSG loads certain geometric model files it attempts to optimize the geometry by turning individual triangles into triangle strips. Since each of the test systems use OpenSG to load the model, all of them make use of the geometry optimization.

Figure 3 shows frame rates for each system running locally on a single machine when rendering a single static object that overlaps every tile. The results show that there is virtually no difference in performance between each of the three systems when rendering on a local machine, independent of the number of triangles per object. It can be seen that Chromium is slightly slower on the smaller models. A possible reason for this slight performance hit on programs with high frame rates is the fact that Chromium must monitor all changes to the OpenGL state. This overhead for monitoring state changes is mitigated by longer rendering times on high-resolution models. However, it becomes significant

when rendering times become shorter and a relatively larger amount of time is spent monitoring the state and not doing rendering.

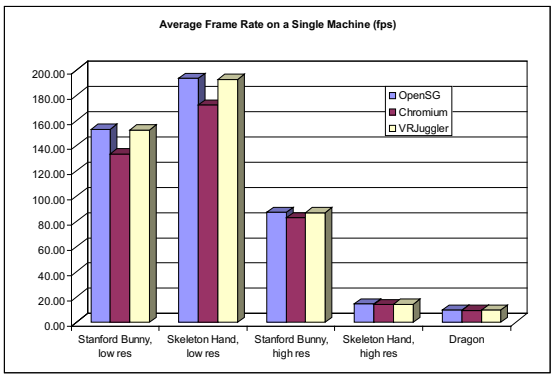

Figure 3: *A baseline comparison of the three test systems. The table shows the average frames per second for each system rendering a single static model on a single machine.*

5.2. Tests with Static Numbers of Objects

These tests are used to analyze how the systems' performance and network utilization scale when the number of polygons increases, how they handle synchronization between cluster nodes during runtime, and how long it takes to load and initialize all cluster nodes.

W chose three different models, two of them at different resolutions, giving us 5 different objects with different complexity to use. This way we could see how system-performance correlated to model size and complexity. Table 1 contains an overview of the number of vertices and triangles of the static models used during performance analysis.

Model	# Vertices	# Triangles
Skeleton Hand, low res	5356	2178
Stanford Bunny, low res	6135	3155
Stanford Bunny, high res	81539	74457
Skeleton Hand, high res	808654	719594
Dragon	1120192	981038

Table 1: *Models used for performance analysis, sorted by number of triangles (models are available at Georgia Institute of Technology, Large Geometric Models Archive (http://www.gatech.edu/projects/large_models).*

We carried out five different tests for each model, each test lasting 500 frames. The tests can be described as follows (see Figure 4 for illustrations):

1a: Rendering a static model overlapping every tile, providing us with a good baseline.

1b: Rendering a single model overlapping every tile, this time rotating around a central axis. Rotating the model allows us to detect synchronization problems between cluster nodes and balances the rendering load by constantly changing the number of polygons each cluster node has to render.

1c: Rendering a single object, initially centered in one of the tiles. The model is then moved from tile to tile in straight lines. This indicates how the systems handle geometry distribution.

1d: Rendering four identical objects, each instance is initially centered in a different tile; the objects move from tile to tile, following straight lines. This tests geometry distribution, but requires four times the number of polygons to be rendered.

1e: Rendering four identical objects, each instance centered in a different tile; each instance rotates in place. This comparison determines the impact of objects lying on the boundary of two tiles.

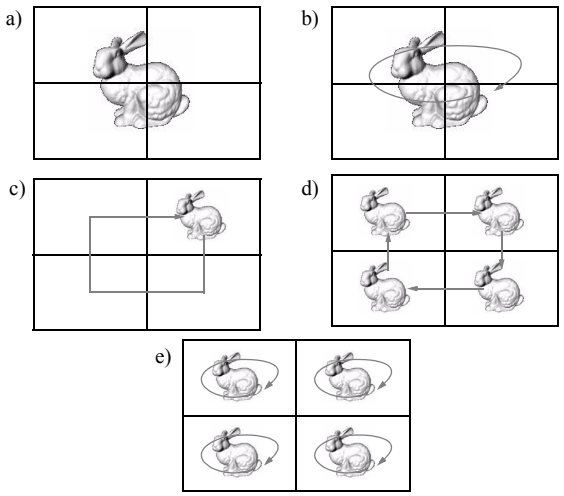

Figure 4: *Tests performed for each model: a) static, b) rotating, c) moving from tile to tile, d) moving between tiles, e) rotating in each tile.*

Figure 5 shows the frames-per-second measured for each system and every model when rendering a single static model in cluster mode. We noticed a large performance drop of Chromium when compared to the other two systems. OpenSG and VRJuggler perform almost identically, but Chromium runs 91%–98% slower. All three systems showed similar results for all five tests, with OpenSG and VRJuggler achieving the same frame rate and Chromium clocking in at over 90% slower.

The primitives in the model are not organized spatially. As a result Chromium must transmit every primitive to every cluster node every frame. This fact can be verified by enabling rendering of Chromium's bounding boxes. Network traffic accounts for 100% of the slowdown in Chromium.

Chromium's inability to handle certain function calls within display lists was the main motivation for not using them in our tests. Chromium however is the most likely system to benefit from the use of display lists. In retained mode applications such as rendering a high resolution model, creating a display list on each cluster node can speed up rendering. For Chromium a display list could be generated on each node during the first frame; for each subsequent frame only the view transformations would then need to be sent to

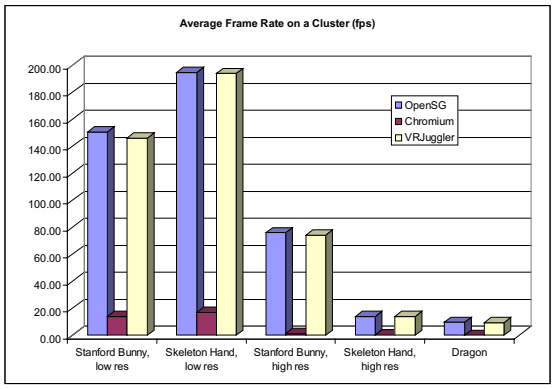

Figure 5: *Average number of frames-per-second for each system rendering a single static model on the tiled display.*

each node, saving a large portion of the frame time. In OpenSG and VRJuggler the display lists would also be created on each cluster node. The only performance gain would be the typical gain achieved when using a display list as opposed to issuing all commands every frame.

When comparing the results of test 1a on the cluster (Figure 5) to the results on a single machine (Figure 3), there is no performance gain for tests run on a cluster. In an ideal situation with n cluster nodes, each cluster node should render $1/n$th of the scene and the entire rendering time should be n times faster. Since all of the geometry for a single model is located inside a single scene graph node, none of the geometry is culled out by OpenSG. As we discussed earlier, since the geometry is not organized spatially, Chromium also does not experience any speedup when using a cluster.

Test 1b showed no different results than test 1a. When comparing test 1c to test 1a, there was no remarkable difference for OpenSG and VRJuggler, whereas Chromium showed an average speedup of a factor of three, with a speedup of four when the object was in one tile only, and a speedup of two when the object was crossing tile boundaries.

Tests 1d and 1e were designed to measure each systems' ability to render multiple objects in different tiles. Figure 6 shows the speedup for each system running test 1e on a cluster when compared to running it on a local machine. In this case each node of the cluster is responsible for rendering exactly 1/4th of the entire scene. OpenSG and VRJuggler show the expected speedups of a factor close to four, especially for high polygon models. The speedup from VRJuggler is due completely to using OpenSG as its underlying API. Chromium, which is still hindered by network performance, does not show an improvement when compared to running on a single machine; network bandwidth is still the limiting factor. Comparing the results of test 1d with the results of test 1e shows that Chromium performs bucketing correctly. In test 1d, when the bunnies overlap tiles, rendering slows down to approximately 50%.

Figure 7 shows the average number of kilobytes sent per frame for test 1d. OpenSG and VRJuggler both show a constant amount of network traffic for all models. OpenSG only has to update four transformation matrices every

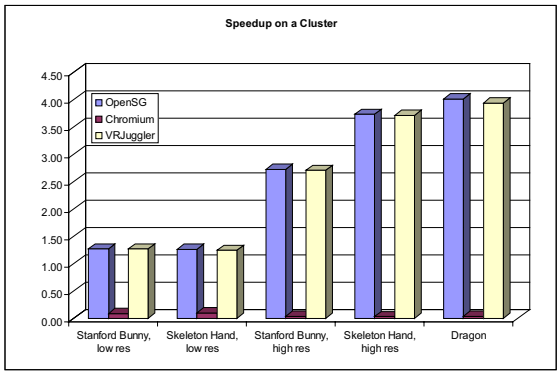

Figure 6: *The speedup for each system rendering four rotating models on the tiled display over a single machine (test 1e).*

frame, and VRJuggler only has to synchronize execution of the program. Both operations are model-size independent.

Chromium, as explained earlier, has to transmit the geometry every frame. In this test it sends only the geometry for each object to one tile, except during times when the object spans two tiles. In this case it must send the geometry for each object to both tiles.

Table 2 shows the average number of kilobytes per second sent from the client to the cluster nodes during test 1d. Chromium is close or over the theoretical limit for a 100 Mbps Ethernet network. The two low-resolution models are slightly below the limit, but all three high resolution models completely saturate the network. OpenSG and VRJuggler both show a decreasing number of bytes per frame as the rendering time for a single frame is increasing, while the amount of network data being sent remains constant.

By multiplying the average number of bytes per frame required by Chromium with the theoretically achievable frame rate of OpenSG, we can calculate the required network bandwidth in order to eliminate the network as a bottleneck. The smallest bandwidth required to run any of the five tests and models is 36 Mbytes per second. This is required to render a single, low-resolution hand model

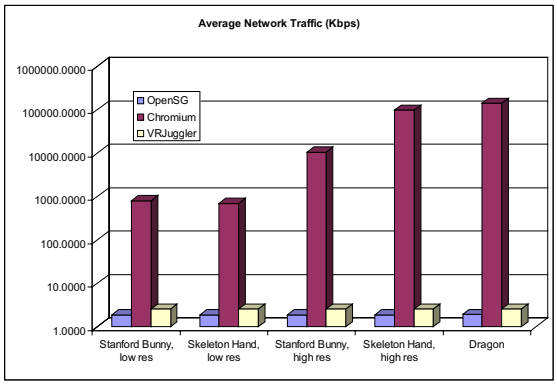

Figure 7: *The average number of Kbytes per frame sent over the network when rendering four moving objects on the tiled display.*

Model	OpenSG Kbytes / sec	Chromium Kbytes / sec	VRJuggler Kbytes / sec
Skeleton Hand, low res	115	10206	1266
Stanford Bunny, low res	89	10480	955
Stanford Bunny, high res	45	11324	485
Skeleton Hand, high res	8	11506	91
Dragon	6	11517	62

Table 2: *The average number of bytes per second sent over the network for each system rendering a single static model on the tiled display.*

moving from tile to tile. The bandwidth required to display a single, static, low res hand in full screen is 110 Mbytes per second, barely below the capacity of gigabit Ethernet. In order to display the dragon in full screen, a network capable of 894 Mbytes per second is required.

We also tried to determine system initialization time, but measuring this value is very difficult. Each system requires starting up each of the render servers and the application client. All synchronization for Chromium is performed before the application begins executing its first line of code. OpenSG uses a function call within the application to connect to each of the cluster nodes, but the scene graph is not synchronized between all of the nodes until the first frame is rendered. For VRJuggler, each of the slave nodes synchronize with the master before any execution of application code, while the scene graph is synchronized after executing some of the application's code.

We decided to measure the amount of time it takes from executing the first line of code in the application until the first frame has been rendered. All render servers are started first, then the application client is started. This method of measuring startup time insures that we include the scene graph synchronizations of OpenSG and Chromium.

Figure 8 shows the times required by each system to synchronize for test 1d. These numbers were calculated by first measuring the time it takes from the first line of code until the end of rendering the first frame for both cluster and a single machine. In order to eliminate the time it takes to render the scene we subtracted the average time per frame from the startup time for the cluster. Finally, we used the difference between that number and the startup time for a single machine to eliminate the time it takes to load and process the model files.

Chromium performs no synchronization of data, and therefore shows no extra start-up time. Any extra time appearing in Figure 8 is due to random variations in loading the file, optimizing the geometry and rendering.

OpenSG requires virtually no additional time when loading the low resolution models, but it does require a significant amount of time to transmit the high resolution models to each of the cluster nodes. OpenSG, however, is able to compress its internal data structures for transmission over the network, allowing OpenSG to send almost four million triangles to four different machines in under forty seconds.

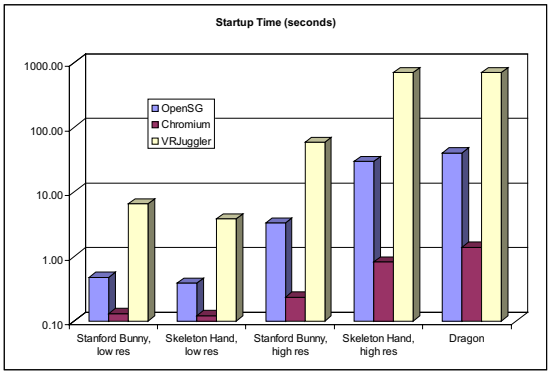

Figure 8: *The amount of time (in seconds) required to connect to and synchronize each of the cluster nodes.*

5.3. Tests with Dynamic Number of Polygons

This test is designed to measure how well the systems handle a dynamically changing environment and how they synchronize these geometry changes among the cluster nodes. The emphasis here is on runtime performance and network usage and less on initialization.

We decided to implement Conway's "Game of Life" [20]. The "Game of Life" is a simple, but highly dynamic simulation of objects populating a screen. The simulation starts with a set of seed-objects, and after each frame objects are generated and removed according to the algorithm shown below.

```
for each square do
   if (object in square)
   then
      if (2 or 3 objects in neighborhood)
      then
         object remains
      else
         object dies
      fi
   else
      if (3 entities in neighborhood)
      then
         create new object
      fi
   fi
done
```

Figure 9: *"Game-of-Life"-algorithm*

Depending on the starting configuration the simulation stabilizes after a relatively small number of steps. A stable world is defined as a configuration where the number of objects remains constant. For our simulation we used the R-pentomino configuration which is known to stabilize after 1103 steps. Figure 10 shows the initial configuration and the result after running two iterations. We let the simulation run for 1200 frames with one step per frame.

The "Game of Life" was run on a 100 x 100 grid. The scene graph consists of a root node with 10 children. Each child is a transformation node that forms the top row of the grid. Each of these 10 children has a daisy chain of children

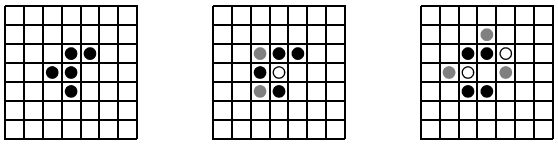

Figure 10: *Configurations for '"Game of Life"-simulation: a) start-configuration, b) step 1, c) step 2. Deleted objects shown with white circles, created objects shown in grey.*

forming the columns of the grid. Pointers to each grid cell are stored in an array for direct access. Every time a new sphere is created in the simulation, the geometry for the sphere is regenerated and stored as a new geometry node. This node is then attached to the appropriate transformation node. Every time a sphere is removed in the simulation it is detached from the transformation node and its memory is deallocated. This requires all changes to the scene graph be sent over the network.

Figure 11 shows the average frame rate for each system running the simulation on the cluster. Once again Chromium is the slowest, but this time OpenSG is much slower than VRJuggler.

Chromium once again must transmit all geometry information every frame. The spheres are each contained within a single scene-graph node which means that Chromium performs a screen space bounding box test per sphere. Chromium sends each sphere only to the node responsible for rendering that sphere. Despite this efficiency there is still too much data sent over the network for Chromium to avoid network bandwidth as the bottleneck.

Since we decided to create and delete spheres instead of reusing geometry or sharing geometry nodes, OpenSG is forced to send a lot of data over the network. OpenSG must send the geometry for all spheres that are going to be added to all cluster nodes. It must also send the ID of all spheres that are going to be deleted. Since each sphere is contained within its own node, frustum culling allows each of the render server to render only the spheres within its tile. This results in a speedup on rendering, however the geometry must still be sent to every cluster node.

Since the application is deterministic there is no data VRJuggler needs to synchronize between cluster nodes other than synchronizing the frames. If this program were based on random numbers then the random number would be declared as an input device and synchronized between each of the nodes. Synchronizing a single random number however requires much less network traffic than sending large amounts of geometry.

6. Conclusions

We have presented a survey of software systems designed to render on a tiled display. We have provided an in-depth analysis of three of the more popular systems. For each of these systems we have ran a series of performance analysis tests to measure their performance in different circumstances.

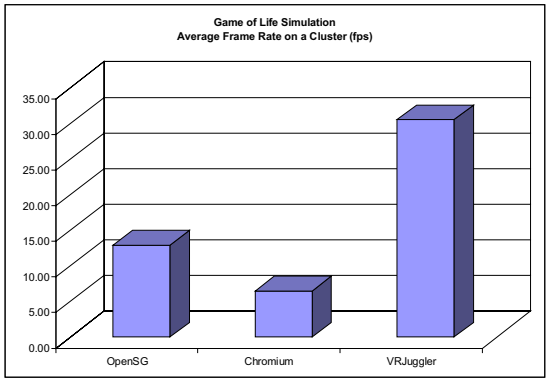

Figure 11: *The average number of frames rendered per second by each system running the "Game of Life."*

VRJuggler produced the fastest frame rates on all tests by keeping network traffic to a minimum. However VRJuggler was only able to produce frame rates consistent with OpenSG on tests 1d and 1e because VRJuggler was able to make use of OpenSG's node culling abilities. Without OpenSG as an underlying API VRJuggler's frame rates will not improve as the number of cluster nodes increases. VRJuggler also took an order of magnitude longer than OpenSG, and two orders of magnitude longer than Chromium to initialize the cluster.

OpenSG produced the second best frame rates of the three systems tested. Frame rates for tests with static geometry equal those produced by VRJuggler, and the frame rates scale with the number of cluster nodes provided the screen space size of the bounding boxes of the geometry is comparable to the size of a single tile. The frame rate of applications with dynamically changing geometry were much lower than that of VRJuggler, and the network became saturated. OpenSG required an order of magnitude longer than Chromium to initialize the cluster.

Chromium was able to run the OpenSG application with very few modifications, and it's startup times were the best of all systems. Chromium produced the slowest frame rates of all systems tested. In all cases, the available bandwidth was too small to support the network traffic generated by Chromium, and the network became a severe bottleneck.

Both, VRJuggler and OpenSG are well suited for applications with complex and large geometries, while Chromium can only be used for smaller models or in connection with a high-speed network. While VRJuggler performed best, it requires similar machines driving the tiles, as the applications run fully duplicated. For applications where large datasets need to be transmitted on a regular basis or computing power is not available, OpenSG would be the best solution in connection with a high-performance node running as client and several servers running the render-engine only.

7. Acknowledgments

This work was supported by the National Science Foundation under contract ACI 9624034 (CAREER Award), through the Large Scientific and Software Data Set Visual-

ization (LSSDSV) program under contract ACI 9982251, through the National Partnership for Advanced Computational Infrastructure (NPACI), and a large Information Technology Research (ITR) grant. We thank the members of the Visualization and Graphics Research Group at the Center for Image Processing and Integrated Computing (CIPIC) at the University of California, Davis.

The geometric models of the bunny, dragon, and hand are courtesy of the Graphics Group at Stanford University.

8. References

1. A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. "VR Juggler: A Virtual Platform for Virtual Reality Application Development." *IEEE Virtual Reality*, 89-96, 2001.
2. H. Chen, Y. Chen, A. Finkelstein, T. Funkhouser, K. Li, Z. Liu, R. Samanta, and G. Wallace. "Data Distribution Strategies for High Resolution Displays." *Computers and Graphics Vol. 25*, 811-818, 2001.
3. Y. Chen, H. Chen, D. W. Clark, Z. Liu, G. Wallace, and K. Li. "Software Environments For Cluster-based Display Systems." *IEEE Symposium on Cluster Computing and the Grid*, 202-210, 2001.
4. W. T. Correa, J. T. Klosowski, and C. T. Silva. "Out-Of-Core Sort-First Parallel Rendering for Cluster-Based Tiled Displays." *EGPGV*, 2002.
5. G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan. "Distributed Rendering for Scalable Displays." *IEEE Supercomputing,* 2000.
6. G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. "WireGL: A Scalable Graphics System for Cluster." *SIGGRAPH*, 129-140, 2001.
7. G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. "Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters." *SIGGRAPH*, 693-702, 2002.
8. J. T. Klosowski and C. T. Silva. "The Prioritized-layered Projection Algorithm for Visible Set Estimation." *IEEE Transactions on Visualization and Computer Graphics*, 365-379, 2000.
9. S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. "A Sorting Classification of Parallel Rendering." *IEEE Computer Graphics and Applications*, 23-32, 1994.
10. G. Voss, J. Behr, D. Reiners, and M. Roth. "A Multi-thread Safe Foundation for Scene Graphs and its Extension to Clusters." *Eurographics Workshop on Parallel Graphics and Visualization*, 33-38, 2002.
11. R. Samanta, T. Funkhouser, K. Li, and J. Singh. "Sort-First Parallel Rendering with a Cluster of PCs." *SIGGRAPH 2000 Technical Sketches*, 2000.
12. R. Samanta, T. Funkhouser, K. Li, and J. Singh. "Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs." *Eurographics/SIGGRAPH workshop on Graphics hardware*, 99-108, 2000.
13. R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. "Load Balancing for Multi-Projector Rendering Systems." *Eurographics/SIGGRAPH workshop on Graphics hardware*, 107-116, 1999.
14. D. R. Schikore, R. A. Fischer, R. Frank, R. Gaunt, J. Hobson, and B. Whitlock. "High-Resolution Multiprojector Display Walls." *IEEE Computer Graphics and Applications*, 38-44, 2000.
15. B. Schaeffer and C. Goudeseune. "Syzygy: Native PC Cluster VR." To appear in *IEEE Virtual Reality,* 2003.
16. T. van der Schaaf, L. Renambot, D. Germans, H. Spoelder, and H. Bal. "Retained Mode Parallel Rendering for Scalable Tiled Displays." *Immersive Projection Technologies Symposium*, 2002.
17. I. Wald, P. Slusallek, and C. Benthin. "Interactive Distributed Ray Tracing of Highly Complex Models." *Eurographics Workshop on Rendering*, 277-288, 2001.
18. M. Woo, J. Neider, and T. Davis. *Open GL Programming Guide*, Second Edition, 1998. Addison Wesley, ISBN 0-201-46138-2.
19. M. Knorich Zuffo. SIGGRAPH 2002 Course 47: Commodity Clusters for Immersive Projection Environments. *SIGGRAPH*, 2002.
20. M. Gardner. "Mathematical Games: The Fantastic Combinations of John Conway's New Solitarire Game 'Life'." *Scientific American*, Volume (4)223, 120-123, Oct. 1970