Scalable, Tiled Display Wall for Graphics using a Coordinated Cluster of PCs

Nirnimesh P J Narayanan
nirnimesh@students.iiit.ac.in pjn@iiit.ac.in
Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, India

Abstract

Tiled displays can provide high resolution and large display area. Cluster-based tiled displays are cost-effective and scalable. Chromium is a popular software API used to build such displays; Chromium based tiled displays tend to be network-limited affecting the scalability in the number of nodes and the ability to handle really large environment models. We present a tiled Display Wall setup based on a client-server architecture. Our system uses off-the-shelf graphics hardware and standard ethernet network. High-level scene structure and hierarchy of a scene graph is used by a central server to minimize network load. Visible parts of the scene graph are transmitted and cached by the clients to take advantage of temporal coherence. The server exploits the high degree of overlap in the computation space for each rendering node to avoid concurrent redundant computations. We use a broadcast oriented protocol to the clients making the system scalable. Geometry push philosophy from the server helps keep the clients in sync with one another and facilitates the pipelining of the constituent stages. Distributed rendering allows the display wall to render scenes which are otherwise too bulky for any of the individual rendering nodes. No node, including the server, needs to render the entire environment, making our system suitable for interactive rendering of massive models. We show performance measures for the different underlying aspects of our display wall consisting of up to 4×4 *tiles. Studies show that the server and network loads grow sub-linearly with the number of tiles. This makes our scheme suitable for the construction of very large-resolution displays.*

1. Introduction

The display resolution for personal computers has grown very modestly over the past two decades whereas computing resources have been following Moore's law. There is a trade off between resolution and display size on computer displays. Display resolution affects the visible detail and size affects the visual context. Baudisch et al. embedded high-resolution portions in the screen while displaying low resolutions for the rest [9] in an effort to achieve focus plus context. Want et al. proposed a focus+context framework to magnify the features of interest [27]. These methods assume that the viewer concentrates at a small region on the screen whereas people tend to move about and change viewpoints in a large display environment [13]. Large displays with high resolution are required to convey a feeling of being immersed in an environment in Virtual Reality applications. A tiled, Display Wall is a large display system for scientific and medical visualization applications and for public displays. General purpose systems with off-the-shelf graphics accelerators can be used in a cluster to provide a cost-effective and scalable alternative for setting up large tiled display walls.

In this paper, we present the design of a cluster-based display wall that is built using commodity PCs and LAN. The fundamental difference with solutions like Chromium [18] is that the geometry is cached at the client nodes and coordinated by the server to exploit temporal and spatial coherence. Our system is able to provide high-quality tiled display facility to any environment represented using the Open Scene Graph API. A special feature of our system is that even the server does not render the whole model. Network load is minimized by client-side caching and multicasting of objects. This makes our system scalable to a large number of tiles unlike existing solutions that get constrained by network bandwidth. We present the design of our system in Section 3 and experimental results from it in Section 4. Conclusions and future work are presented in Section 5.

2. Related Work

We present a critical review of the literature related to the construction of large displays divided into those using specialized hardware and those using PC clusters.

Specialized Hardware Setups: Large graphics display systems have been built using specialized hardware by companies like Silicon Graphics. High-end computer systems like the Onyx2, with multiple graphics pipelines and channels with each driving a projector, are often used for creating large displays for applications [5]. Such solutions are expensive and non-scalable.

Cluster-based Displays: Cluster-based solutions for creating large displays have gained a lot of interest recently [15, 18]. Such displays are constructed using a number of commodity PCs interconnected on a LAN or a low-latency network like Myrinet, as in [5, 18, 6]. Cluster-based displays are economical, scalable in performance and resolution and easy to maintain; the cluster can also be used for computational purposes. Li et al. reported techniques, software tools and applications that make high-resolution tiled displays scalable and easy to use, for the Princeton Scalable Display Wall project [19].

Two approaches are popular in cluster-based display setups: *master-slave* and *client-server* [15]. The dataset is mirrored across all the nodes in a master-slave setup and multiple instances of a program are run, one on each node and the execution is synchronized. Each node renders the entire scene but displays only a certain portion. The master-slave model is sub-classified as *System-level program synchronized (SSE)* or *Application-level program synchronized (APE)*. SSE attempts to synchronize transparently without requiring modification or even relinking of the source code. In Hypervisor [12], Bressoud et al. proposed a method that treats an actual software system as running on a virtual machine, which is close to the actual microprocessor architecture, resulting in severe slow down. With APE, the responsibility of synchronizing lies with the application. This approach has low network bandwidth requirements. However, since each node runs an instance of the application, there is a gain in display resolution only and no gain in performance. VR Juggler, a framework for virtual reality applications, falls under this category [10]. Net Juggler [8] is an open source library that turns a commodity component cluster running the VR Juggler [10] into a single VR Juggler image cluster. The master-slave approach assumes that each node in the cluster would be able to render the entire environment in its entirety. This runs counter to the motivation of load-balancing that is critical to cluster-based displays. It is also difficult to handle dynamic environments since the data is replicated. It is difficult to access real time data stream from a single external network source even if the data source is centralized.

The client-server models store the dataset at one central server. The data distribution can follow the sort-first strategy or the sort-last strategy [21, 20]. The required network bandwidth can be high when sending primitives to the appropriate rendering node. Samanta et al. investigated methods to improve load balancing by changing the tiling dynamically [22]. The server can also use a Distributed Data Management framework as in [17]. The server distributes appropriate data to each client node and performs the synchronization among the render nodes. One way to distribute the data transparently is to intercept function calls at the Graphics API level [18] or at the display manager level [1]. The latter can provide tiled display including all windowing features including menus, toolbars and decorations. The former provides the large display facility to any application using the API and is used by Chromium, which powers display walls such as the Hyperwall [23], Viswall [6], LionEyes Display Wall [3] and many others. Chromium can clusterize any application built over OpenGL transparently. It fails however to capture coherence of data across frames as each frame is treated independently. The network requirements are thus very high even when the scene is unchanged. It is also not able to take advantage of the high-level objects structure encoded in scene graphs due to its low-level focus.

Data can also be distributed at the 3D object level and not the primitive level. This allows the system to exploit the hierarchical structure, if any, in the dataset and take more informed decisions. This is the approach followed by Syzygy [24] and OpenSG [26] for display wall rendering. The Syzygy software library [24] consists of tools for programming VR applications on PC clusters. Syzygy includes two application frameworks: a distributed scene graph framework for rendering a single application's graphics database on multiple rendering clients and a master-slave framework for applications with multiple synchronized instances.

The rendering nodes in the cluster need to be synchronized to avoid display tearing effects during rendering. All nodes need to fulfill three requirements for locking: Genlock, Swap-lock and Data-lock. Genlock provides coherency to the display signals across all the nodes. Pure hardware solutions like Lightning2 [25] and Matrox's ASM [4] or software/hardware solutions like SoftGenLock [7] or WinSGL [28] are used for this purpose. Swap-Lock compensates for the differential rendering times in different nodes. Data-Lock refers to application-level coherency in the scene to be rendered.

Our approach of geometry management for display walls is closest to Syzygy's distributed scene graph approach. Our approach is more specific for scalable display walls and not necessarily for general VR environments. We exploit the coherence in computations required for each rendering node, rather than treating them individually. Consequently, we do not assume that each rendering node is powerful enough to manage the entire environment. We cache objects at the render nodes and evict objects out of the cache

to avoid overflows. This results in better utilization of the network and memory resources for improved scalability.

Geometry Server: Our Display Wall effort is an extension of an earlier work on a the geometry server, which is a high-performance, centralized storehouse for massive geometric data [16]. The server has a scene graph based representation of the virtual environment with multiple levels of detail. It can simultaneously serve a number of heterogeneous clients adaptively, ranging from a graphics workstation on the LAN to a PDA connected over a wireless network. Each client gets a visibility-limited portion of the model that is compatible with its rendering capabilities, computational resources and network characteristics, with an objective of providing consistent, interactive frame rates. The clients cache parts of the scene graph it encounters during its walk-through and employs a cache management policy that is based on potential visibility of cached objects. From the client's point of view, the remotely served geometry is yet another node in its scene graph and can be modified like a local model. Dynamic objects are handled consistently using a server-push for information and lazy-download for the geometry data. The system can optimally serve models loadable onto an Open Scene Graph [14] system on a wide range of clients [16] and finds ready applications in battlefield simulation, terrain visualization, etc. Our display wall system optimizes the philosophy of the geometry server for a large cluster-based display system. The geometry server just maintains independent contexts for each rendering node. In a display wall, the viewpoints of the rendering nodes are tied together, which allows concurrent computations to be applied as a whole rather than individually. Local caching of the geometry by the clients provides a way to exploit temporal coherency of the environment.

3. Geometry-Managed Display Wall

Our Display Wall system follows a client-server architecture. The server has the whole virtual environment as a scene graph in its memory. It also handles the viewpoint control. Out-of-core rendering techniques can be used at the server if the environment model demands it. Each tile is controlled by a client node that is connected to the server over a standard 10/100/1000 Mbps LAN. The server is a medium to high end machine but the clients are standard low to medium end machines.

Server Tasks: The server determines objects in the scene that are visible to the sub-frustum of each rendering node at each time instant. Conservative visibility using frustum culling of bounding boxes is preferred over real visibility as occluded objects may become visible in subsequent frames,

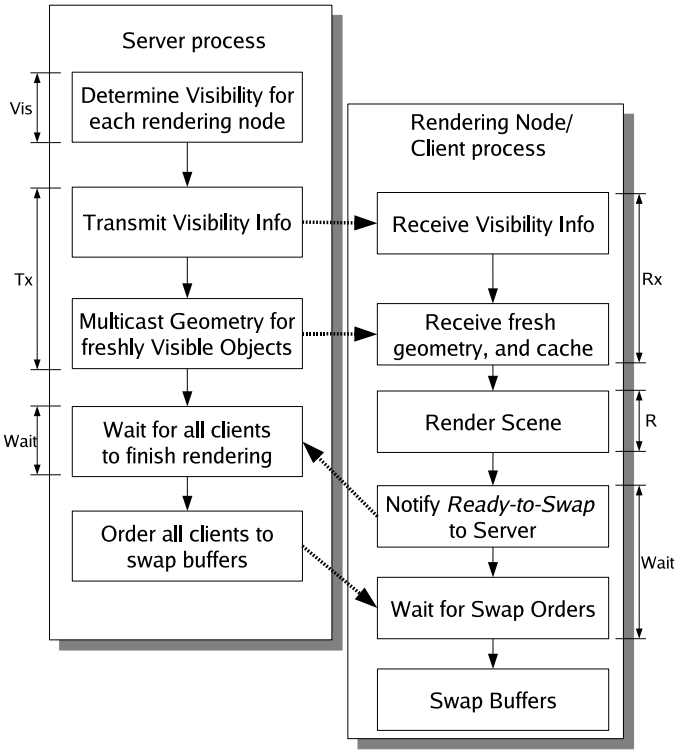

Figure 1. Server-Client control flow

deriving benefit from client-side caching. The server knows which objects are already with the client and determines the new objects to be sent to each client node per frame. The server first sends the results of culling to each node and the list of new objects. Then it sends the new objects to the clients using a multicast mode. Multicasting optimizes the use of network bandwidth – which is a critical resource in such clusters – as several objects might be needed by multiple clients at the same time. The server also ensures synchronized rendering for simultaneous update of each display. Figure 1 shows these steps in a flow diagram.

Client Tasks: Clients receive lists of objects from server for each frame. Each server listens on a multicast port and picks all objects it needs for the next frame. Client places the objects in a local cache to exploit temporal coherence of objects to reduce network bandwidth. It then renders all visible objects and informs the server about its readiness to swap the buffers. The buffer is swapped on getting a go ahead from the server.

A distinguishing feature of our system is that the server does not have to render the whole model. It computes the per tile visibility based on the object's bounding boxes using a novel algorithm described below. Client-side caching reduces network bandwidth by exploiting temporal coherence. Additionally, multicasting the actual objects mini-

mizes the network load and increases the scalability in terms of the number of tiles. The system also achieves better frame rates on bulky models by effective use of parallel rendering. The rendering pipeline of our Display Wall can be broken into three stages: Visibility determination, data transmission and rendering and sync. These are the critical operations that directly determine the overall performance.

3.1. Visibility Determination

It is necessary to determine the visibility of each object with respect to each frustum so as to minimize the network load and also to reduce the geometry sent down the rendering node's GPU pipeline. First, the visibility of different objects in the scene with respect to the overall view frustum is determined to eliminate objects that are totally non-visible. Subsequently, we use a hierarchical frustum culling approach to determine visibility of each object with respect to the frusta of each rendering node or tile. This second step uses an approach similar to quad-tree decomposition using frustum planes. The combined view frustum for the display wall consists of an $M \times N$ arrangement of identical frusta consisting of (M+1) horizontal planes and (N+1) vertical planes. Each plane partitions the frustum into two half-spaces. If an object is found to be entirely on one side of a plane, it cannot be on the other side. Hence, the visibility tests for all frusta on the other side can be safely eliminated. If an object intersects the plane, the process needs to repeated for both the half-spaces. Algorithm 1 gives the pseudo-code of the hierarchical frustum culling algorithm.

With $N \times N$ tiles where $N = 2^k$, the horizontal planes are h_0, h_1 ... h_N and the vertical planes are v_0, v_1 ... v_N. The hierarchical culling can be performed using the following steps.

1. Eliminate objects outside the outer frustum.
2. For each object O, mark it to be visible in all frusta.
3. Call *cullFrust(*0, N, 0, N, O, *true)*

After this procedure, an object O needs to be transmitted only to the rendering nodes corresponding to the frusta that are still marked as visible.

This two-stage visibility algorithm is fast and efficient. We can cull a Power plant scene of 1185 objects to a 4×4 configuration at 827 times per second using the above algorithm. We also experimented with hardware occlusion queries [2] on the server's GPU. The same scene could be culled only 670 times per second using this approach, primarily due to the CPU wait and GPU stalls introduced by the occlusion queries. Occlusion query can free the CPU for other tasks and therefore might actually be preferred in situations where the data-generation process is CPU-limited or where the CPU wait time can be further utilized to narrow down the search space [11]. We show results without hardware occlusion queries (Table 2), since the amount of useful work that can be interleaved during CPU-wait is subjective to an application,.

Algorithm 1 cullFrust(i, j, m, n, O, flag)

```
1:  if Frustum cannot be subdivided then
2:      return
3:  end if
4:  if flag then
5:      Intersect O with v_k where k = (i + j)/2
6:      if O on left of v_k then
7:          Eliminate all frustums to right of vk
8:          if any frustum remaining then
9:              call cullFrust(i, k, m, n, O, false)
10:         end if
11:     else if O on right then
12:         Eliminate all frustums to left of v_k
13:         if any frustum remaining then
14:             call cullFrust(k, j, m, n, O, false)
15:         end if
16:     else
17:         call cullFrust(i, k, m, n, O, false)
18:         call cullFrust(k, j, m, n, O, false)
19:     end if
20: else
21:     Intersect O with h_k where k = (m + n)/2
22:     if O on top of h_k then
23:         Eliminate all frustums below h_k
24:         if any frustum remaining then
25:             call cullFrust(i, j, m, k, O, true)
26:         end if
27:     else if O on bottom then
28:         Eliminate all frustums above h_k
29:         if any frustum remaining then
30:             call cullFrust(i, j, k, n, O, true)
31:         end if
32:     else
33:         call cullFrust(i, j, m, k, O, true)
34:         call cullFrust(i, j, k, n, O, true)
35:     end if
36: end if
```

The hierarchy of frusta used in Algorithm 1 exploits the high degree of overlap in the computation space of visibility determination for all the rendering nodes. Algorithm 1 can further be optimized to take advantage of object hierarchy, if present. It can also extend to potentially infinite environments where objects are generated on discovery. Also note that the algorithm needs to perform object-intersection-test with only one plane (line: 5 or 21) for each invocation of cullFrust(), in contrast with naive view-frustum collision algorithms where all the 6 planes need to be considered.

3.2. Transmission

This stage involves the transmission of objects to the rendering nodes. The network transmission is almost always the limiting factor in all cluster-based rendering systems. The performance of this stage is determined by the network bandwidth and latency. The server optionally compresses the geometry before transmitting. This data is packed into datagrams and assigned a sequence identifier before being multicasted. The clients collect objects that are destined for them and acknowledge the server. The clients also cache the objects received. This involves evicting objects out of the cache if it tends to overflow. We use LRU eviction to free space for new objects. The server keeps track of the objects in the cache of the clients in order to avoid retransmission. The datagrams for which the server doesn't receive an acknowledgment from the client are re-sent after a pause. This pause time is scaled incrementally. Multicasting ensures that the network requirements do not scale linearly with the number of nodes in the cluster. In fact, the network requirements remain practically constant for variations in the number of nodes as will be demonstrated by the experiments reported in the next section.

3.3. Rendering and Sync

The clients start rendering the scene when all objects for the next frame are received. The time taken for this step is proportional to the geometry size within the view frustum. This varies from client to client and hence the swapping of the render buffers is synchronized by the server. Network latency is crucial for the server to be able perform this synchronization. Our experiments show that a standard ethernet network is sufficient for synchronization at interactive frame-rates.

The clients also cache the objects received from the server. The cache enables our system to exploit inter-frame coherence in data. Each client has a fixed cache and uses an LRU algorithm to remove objects from it when cache gets full. Note that the rendering nodes will able to render environments of sizes well beyond their capability due to the cache replacement strategy. This has been possible as a result of maintaining higher-level object information with the help of scenegraphs at the server. This is an improvement over systems that store the entire scene on each client. The push philosophy adopted by the server frees the clients of much load and hence even low-end clients can be used. The server manages most aspects of the system, leaving the clients free for data reception and rendering.

The three stages mentioned above can also be pipelined. For instance, while the server is waiting for ordering the swap because some clients haven't finished rendering, it can carry out the visibility determination or transmission for subsequent frames. The client, on the other hand, can receive data for subsequent frames, while rendering one. This essentially means that the framerate for rendering that wall would be determined by the slowest stage of the sequentially ordered 3 stage pipeline rather than the sum of the times of individual stages. Figure 2 illustrates this pipelining between the various stages for the server and one rendering node. Interleaving of the stages for consecutive frames allows the visibility determination of frame $i+1$ to start just after the visibility determination for the i^{th} frame has completed, without waiting for it to be transmitted and rendered. The clients signal back to the server when they are ready to render. Meanwhile, they start rendering the next frame. The server orders a swap of buffers for all rendering nodes when it receives *I'm Ready* from all of them. All the rendering nodes synchronize their display at this point. This same process is carried out for all the frames (the figure shows this only for frame *i*). Pipelining may increase the latency but the gain in framerate more than compensates for it.

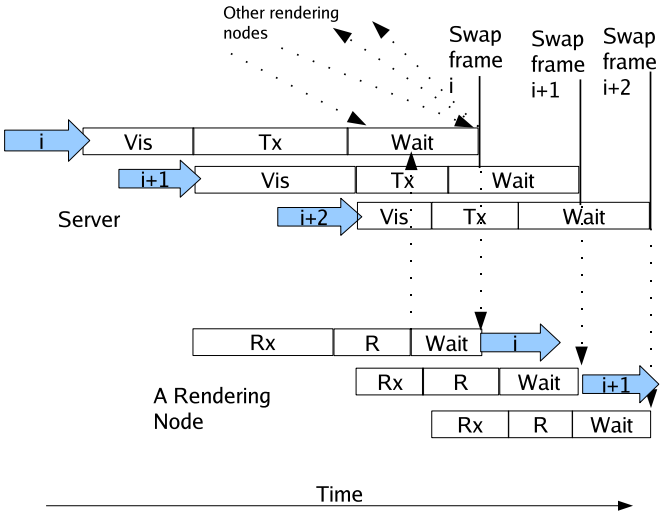

Figure 2. Inter-frame pipelining of the 3 stages of Display Wall rendering. *Vis* denotes the Visibility Determination Stage, *Tx* the geometry transmission stage, *Rx* the geometry reception stage and *R* the rendering stage. The effective FPS of the system gets increased due to this interleaving.

4. Experimental Results

We tested our system for scalability with respect to several metrics for up to 4×4 nodes. The sub-linear growth of the network and computation requirements indicates that

Table 1. Time taken to start a 4×4 display wall. This involves sending of the initial data to the clients.

Tile Configuration	Time taken (in seconds) with TCP	with Multicast
2×2	26.44	26.00
2×3	32.02	27.29
2×4	40.54	27.60
3×3	44.03	27.71
3×4	56.38	27.72
4×4	72.10	27.74

Table 2. Time taken for Visibility Determination using Hierarchical View Frustum Culling (Algorithm 1) on UNC's Powerplant model with 1185 objects.

Config	Time taken (ms)
2×2	2.99
2×3	2.83
2×4	2.90
3×3	2.77
3×4	3.44
4×4	3.81

the system can be used to set-up gigantic display walls from a cluster of low-end systems. Our test setup consists of 15 low-end systems with AMD Athlon64 3000+ systems with 512MB memory and an on-board ATI Radeon Xpress 200 graphics. The GPU uses 64MB of the system memory as the video memory. These machines act as rendering nodes in the cluster. The server is an AMD Athlon 64 3200+ system with 3GB RAM and an Nvidia 6600GT graphics accelerator. The server machine also hosts one rendering node. The performance gain achieved by using better rendering nodes would also translate naturally to our system as well. The 16 systems are connected using a separate 100Mbps ethernet switch. Some experiments were performed with higher or lower speed networks, as mentioned. We present results of tests for scalability with various tile-configurations (2×2, 2×3, 2×4, 3×3, 3×4, 4×4). We currently use a tiled arrangement of monitors with no special attention paid to their alignment for the display. Using monitors causes visual distraction due to the gaps between the adjacent tiles: straight lines don't appear straight anymore due to the parallax. We corrected this by adjusting the view frustum for each tile. The view frustum of each tile is clipped a little along the edges. Figure 11 shows the Powerplant model on our 4×4 display wall with the parallax error compensated.

The server has to read and initialize large amounts of data and send some of it to the clients to start the wall. Table 1 compares the startup time when using TCP vs. UDP multicast for the Powerplant model. The startup time is virtually independent of the tile-configuration when using multicast over UDP but increases linearly with TCP based network transmission.

Table 2 shows the time taken to perform the Visibility Determination using Algorithm 1 on UNC's Powerplant model with 483M of geometry in 1185 objects.

In Figure 3, we show the variation of the frame rate in fps for different tile-configurations during a walkthrough. The scene consists of 5.5 million vertices in 100 objects scattered around with an average object size of $2.03M$. We steer the walkthrough in such a way that fresh objects become visible to all the tiles at once, bringing high network loads at the same time. Note that there's almost no variation in fps with different tile-configurations of the wall. The sharp trenches are seen when fresh objects become visible to all the tiles at once, where the display freezes for an instant. The fps reduces to about 12 due to the visibility of more objects until some are culled out. The fps then reaches near 20. The pattern of the walkthrough is repetitive and so is the fps variation, demonstrating the efficiency of cache management.

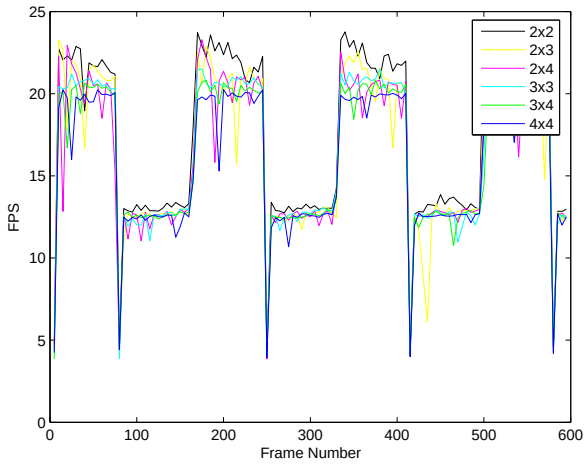

Figure 3. Showing system scalability with respect to the number of nodes. Note that the performance of the system remains almost unchanged for different tile-configurations

Caching at the clients eliminates the dependence of the performance on network bandwidth. Figure 4 shows the nature of the network utilization for the above walkthrough.

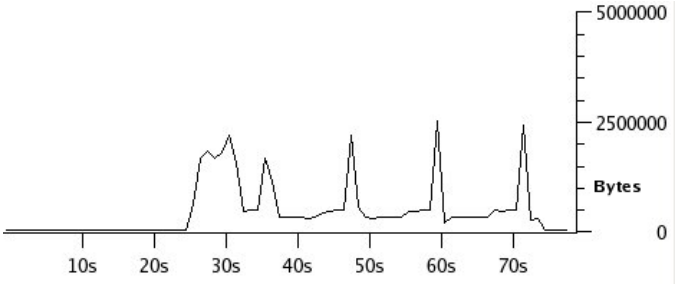

Figure 4. Network usage vs. time for a 600-frames walkthrough. The walkthrough is such that objects appear in multiple tiles at once. Multicast mitigates this effect. Due to caching of objects by the client, the network is used only when fresh objects are to fetched.

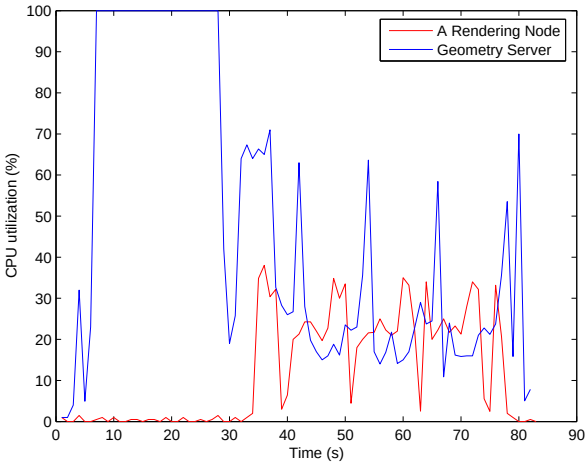

Figure 5. CPU Usage by the Geometry Server and a Rendering Node during a 600 Frame walkthrough.

The initial low network utilization corresponds to the server starting up. This is followed by a high-utilization phase when the server is sending all the startup data to the clients. Figure 5 shows the CPU utilization of the server and a client. The initial high utilization for the server is due to the preprocessing. For the rest of the walkthrough, however, the CPU remains only moderately loaded and is available for other computations.

In Figure 6, we show the above walkthrough but with the network bandwidth capped to 10Mbps. The performance is not degraded at all with the reduced bandwidth due to geometry caching and the use of multicasting. The freeze times of the display due to the transmission of new objects are longer due to the lower network bandwidth.

In another series of experiments, we perform a 4300-frames long walkthrough in a scene containing 205 objects totalling to 416MB where each object has an average size of 2MB. Figure 7 shows the walkthrough for various tile-configurations of the wall. The clients need to fetch a lot of data in the initial phases of the traversal as they discover new objects causing the huge variation in FPS shown in the figure during the initial stages (till 80s). The walkthrough almost retraces its path for the next 70 seconds, in which case the graph shows a sustained frame-rate for all tile-configurations. The walkthrough starts discovering new objects (for about 30 seconds) near the 150 seconds mark. After this point, a lot of the scene has been cached. FPS persists at 23 FPS. Also note that the plots corresponding to the various tile-configurations are closely similar. This shows the low dependence of the system on the tile-configuration, hence improved scalability.

Figure 8 shows the performance of the 4×4 configuration with 3 different cache sizes. The correct cache size depends on the density of objects in an environment and varies from scene to scene. In this experiment, a 100MB cache size is too large and remains underutilized while a 30MB cache is too small and causes frequent re-fetches resulting in the large disturbance of framerate. The 60MB cache size seems optimal.

It should be noted that the above experiments were carried out on worst-case scenes where a lot of objects are introduced at the same time. In real-life situations, the viewpoint changes slowly and the inter-frame coherence of geometry at every client is high. The geometry management strategy will give consistent and high performance in such situations as was our experience with normal scenes and interactive walkthroughs.

Further, to demonstrate the load distribution capabilities of our display wall rendering cluster, we chose an environment with a lot of small-in-size but heavy-on-rendering objects. In Figure 9, the 4×4 configuration outperformed both 3×3 and 2×2. The rendering load is heavy with fewer number of nodes in the cluster but better distributed with more nodes as in 4*x*4. With less number of nodes in the cluster, the rendering load is heavy, but with more nodes, the load is better distributed, as in 4×4.

Even though we have shown the above results on rendering nodes with low-end graphics, our system scales well to perform load distribution and high-performance rendering on systems with good graphics as well. Figure 10 shows the performance of our system on a cluster of 4 systems with the same configuration as our server. They have an Nvidia 6600GT graphics accelerator each. We perform the above walkthrough on the Powerplant model from UNC, with 13 million triangles spread over 1185 objects. A 2×2 cluster

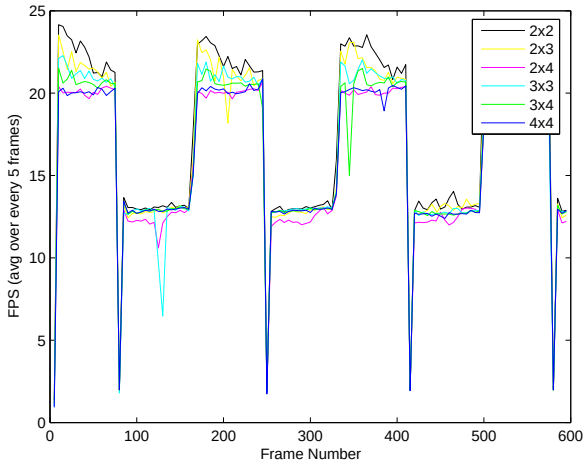

Figure 6. Showing system scalability with respect to the number of nodes with network bandwidth capped at 10 Mbps. Due to caching, the network requirement doesn't degrade the performance except at times of data-fetch.

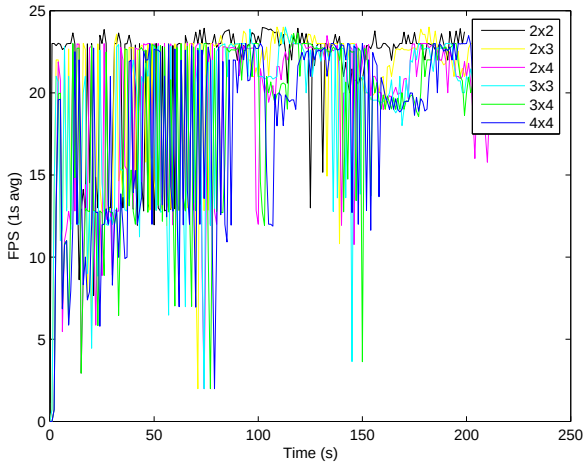

Figure 7. A 4300-frames walkthrough in a jungle of 416M scene spread over 205 objects.

renders it at 23 fps. The trench observed at near 160s is due to huge data fetches as new parts of the scene is being discovered, at which time the network is still the bottleneck. The FPS stabilizes back to 23 FPS thereafter. There is a natural startup-time lag due to the network factor. The powerplant model runs at 1.5 FPS on our 4×4 cluster where the poor rendering capability of our rendering nodes is the limiting factor.

Comparison with Chromium: Chromium is very network-intensive and sends the OpenGL primitives for each frame independently. Use of display lists with Chromium has issues. The full Powerplant model runs at about 0.5 FPS on the server machine (with Nvidia 6600GT and 3GB of RAM). Sending it to the display wall using Chromium further slows down the rendering. The application node experiences bursts of high CPU activity, followed by high network transmission, for every frame. The performance with Chromium worsens as the number of tiles in the cluster increases. With our geometry management techniques, we achieve a frame rate of 23 on a 2×2 cluster (each with Nvidia 6600 GT). The powerplant performs at 1.5 FPS on our 4×4 display wall, which is due to the poor rendering capability of the rendering nodes used. The load distribution over 16 rendering nodes however makes this configuration perform even better than a single system with Nvidia 6600 GT, as reported above.

5. Conclusions and Future Work

In this paper, we presented a geometry-managed, tiled-display system that uses commodity computers. The system uses a client-server architecture that works with even modest clients. The local caching of the geometry and the multicast mode of transmission of geometry keeps the network requirements moderate and provide excellent scalability of the architecture in contrast with prior work. We also demonstrated the scalability and parallel rendering capabilities of our system on the Powerplant model from UNC.

We are currently working on combining the the display wall with an out-of-core rendering system to render extremely large resolutions. Viewpoints can be predicted at the server in such systems and data can be sent to the clients speculatively.

Swap-lock and gen-lock of the display is an important area that needs further research. Since the use of commodity graphics cards is a design goal of our system, hardware gen-lock is not an option. We are working on a camera based scheme to bring the displays into perfect synchronization. This is similar to the camera-based approaches for color-balancing and geometry correction followed by multi-projector display systems.

References

[1] Distributed Multihead X Project (DMX). `http://dmx.sourceforge.net`.

[2] GPU based Occlusion Query, `http://oss.sgi.com/projects/ogl-sample/registry/ARB/occlusion_query.txt` .

Figure 11. A 4×4 rendering of UNC's Powerplant. The combined resolution is 12 MPixels. The gaps between monitors introduce parallax. Frusta have been adjusted to compensate for this so that straight lines appear straight.

[3] LionEyes Display Wall. Penn State University. `viz.aset.psu.edu/ga5in/DisplayWall.html`.

[4] Matrox Advanced Synchronization Module. `www.matrox.com/mga/products/asm/home.cfm`.

[5] PowerWall. University of Minnesota. `www.lcse.umn.edu /research/powerwall/powerwall.html`.

[6] VisWall High Resolution Display Wall. `http://www.visbox.com/wallMain.html`.

[7] J. Allard, V. Gouranton, G. Lamarque, E. Melin, and B. Raffi n. Softgenlock: Active Stereo and Genlock for PC Cluster. In *Proceedings of the Joint IPT/EGVE'03 Workshop*, 2003.

[8] J. Allard, V. Gouranton, L. Lecointre, E. Melin, and B. Raffi n. Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In *IEEE Virtual Reality Conference*, pages 273–274, 2002.

[9] P. Baudisch, N. Good, and P. Stewart. Focus plus context screens: Combining display technology with visualization techniques. In *ACM Symposium on User Interface Software and Technology*, 2001.

[10] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *VR*, pages 89–96, 2001.

[11] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Comput. Graph. Forum*, 23(3):615–624, 2004.

[12] T. C. Bressoud and B. B. Schneider. Hypervisor-based fault-tolerance. In *Fifteenth ACM Symposium on Operating System Principles (ASPLOS V)*, pages 1–11, Copper Mountain Resort. Colorado, 1995. ACM.

[13] M. S. Brown and A. Majumder. SIGGRAPH 2003 Course Notes: Large-Scale Displays for the Masses, 2003.

[14] D. Burns and R. Osfi eld. Open Scene Graph A: Introduction, B: Examples and Applications. In *VR*, page 265, 2004.

[15] H. Chen, D. W. Clark, Z. Liu, G. Wallace, K. Li, and Y. Chen. Software Environments For Cluster-Based Display Systems. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 202–211, 2001.

[16] S. Deb and P. J. Narayanan. Design of a Geometry Streaming System. In *Indian Conference on Computer Vision, Graphics and Image Processing*, 2004.

[17] J. Gao, J. Huang, C. R. Johnson, S. Atchley, and J. A. Kohl. Distributed Data Management for Large Volume Visualization. In *IEEE Visualization*, page 24, 2005.

[18] 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. In *SIGGRAPH '02*, pages 693–702. ACM Press, 2002.

[19] K. Li, M. Hibbs, G. Wallace, and O. G. Troyanskaya. Dynamic Scalable Visualization for Collaborative Scientifi c Applications. In *International Parallel and Distributed Processing Symposium*, 2005.

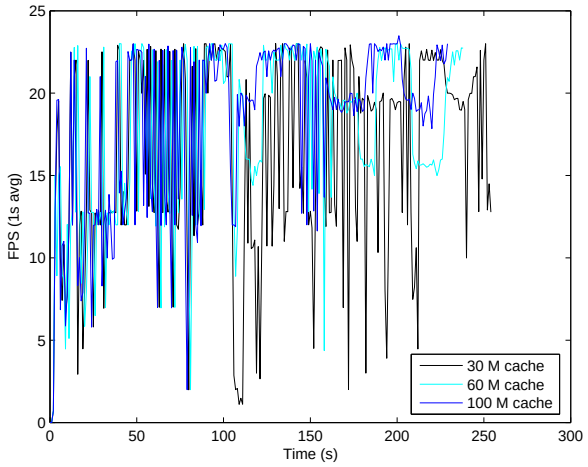

Figure 8. The performance of our 4×4 system with varying cache sizes in a jungle of 416M spread over 205 objects.

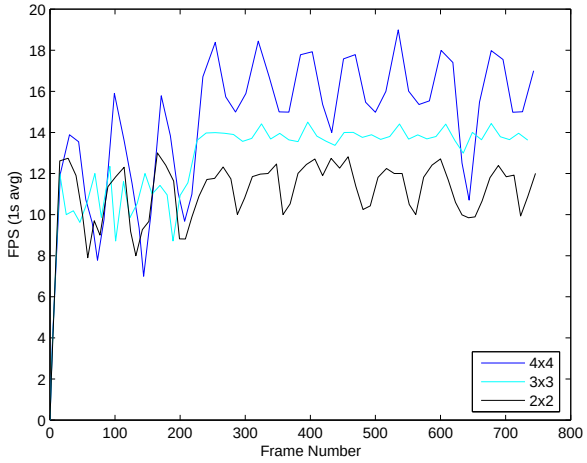

Figure 9. The rendering capability of our display wall increases with increase in the number of nodes in the cluster. More number of nodes distribute the load in the visible frustum.

[20] R. Samanta, T. Funkhouser, and K. Li. Parallel rendering with k-way replication. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 75–84, 2001.

[21] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 97–108, 2000.

[22] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. Load balancing for multi-projector rendering systems. In *ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 107–116. ACM Press, 1999.

[23] T. A. Sandstrom, C. Henze, and C. Levit. The hyperwall. pages 124–133. International Conference on Coordinated and Multiple Views in Exploratory Visualization, 2003.

[24] B. Schaeffer and C. Goudeseune. Syzygy: Native PC Cluster VR. In *VR '03: Proceedings of the IEEE Virtual Reality 2003*, page 15. IEEE Computer Society, 2003.

[25] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2: a high-performance display subsystem for PC clusters. In *SIGGRAPH '01*, pages 141–148, 2001.

[26] G. Voss, J. Behr, D. Reiners, and M. Roth. A multi-thread safe foundation for scene graphs and its extension to clusters. In *Eurographics Workshop on Parallel Graphics and Visualization*, pages 33–37, 2002.

[27] L. Wang, Y. Zhao, K. Mueller, and A. E. Kaufman. The Magic Volume Lens: An Interactive Focus+Context Technique for Volume Rendering. In *IEEE Visualization*, page 47, 2005.

[28] M. Waschbüsch, D. Cotting, M. Duller, and M. Gross. WinSGL: Software Genlocking for Cost-Effective Display Synchronization under Microsoft Windows. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2006.

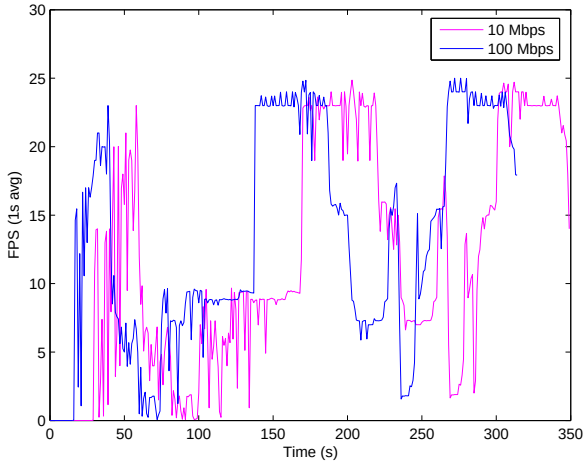

Figure 10. The performance of our 2×2 system with Nvidia 6600 GT on each rendering node. The network is still the bottleneck at times of huge data fetch.