GSWall: A Scalable Tiled-Display Wall

Submitted in partial fulfillment of
the requirements for the degree of

Master of Science (by Research)

in

Computer Science

by

Nirnimesh

<nirnimesh@research.iiit.ac.in>

http://students.iiit.ac.in/~nirnimesh

International Institute of Information Technology

Hyderabad, INDIA

August, 2006

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "*GSWall: A Scalable Tiled-Display Wall*" by Nirnimesh, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Advisor: Prof. P J Narayanan

To Ma, Papa, Abhishek and Madhuresh

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 large environment models. This thesis presents a tiled Display Wall setup based on a client-server architecture, with the server managing all the aspects leaving the clients with rendering as their sole responsibility. Our system uses off-the-shelf graphics hardware and standard ethernet network. High-level scene structure and hierarchy of a scene graph (OpenSceneGraph) is used by a central server to minimize network load. The view-frustums of the rendering nodes are treated hierarchically as well. A novel algorithm combines the object hierarchy of the scene graph with the hierarchy of frustums to determine the optimal process of unfolding the hierarchies so as to minimize the number of computations involved. Visible parts of the scene graph are transmitted and cached by the clients to take advantage of temporal coherence. The server, following a push-philosophy, is able to exploit the high degree of overlap in the computation space for each rendering node to avoid concurrent redundant computations. We use a multicast oriented protocol for data-transmission 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 be able 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. The display wall application is implemented as a library-intercept mechanism to seamlessly render any OpenSceneGraph-based graphics application to a tiled-display wall without the need of modification, recompilation or even relinking. This makes our display wall easy to use for several already existing applications. Our 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.

Contents

Chapter Page

1 Introduction . . . 1
1.1 Display Wall . . . 1
1.1.1 Focus plus Context Displays . . . 1
1.2 Cluster-based Tiled Display Wall . . . 2
1.3 Contributions of this Thesis . . . 4
1.4 Experimental Setup Used in the Thesis . . . 4
1.5 Overview of the Thesis . . . 5

2 Background and Related Work . . . 6
2.1 Display Wall Systems . . . 6
2.1.1 Specialized Hardware Setups . . . 6
2.1.2 Cluster-based Displays . . . 6
2.1.3 Master-Slave approach . . . 7
2.1.4 Client-Server approach . . . 7
2.1.5 Chromium . . . 8
2.2 Visibility Determination . . . 9
2.3 Scene Graph APIs . . . 11

3 Geometry-Served Display Wall . . . 12
3.1 Geometry Server . . . 12
3.2 Geometry-Served Display Wall . . . 13
3.2.1 Rendering Pipeline . . . 13
3.2.1.1 Server's Tasks . . . 15
3.2.1.2 Client's Tasks . . . 15
3.2.2 Pipelining . . . 17

4 Visibility Culling for Tiled Displays . . . 19
4.1 Object Hierarchy and Frustum Hierarchy . . . 19
4.2 Visibility Culling . . . 20
4.2.1 Preprocessing . . . 21
4.2.2 Traversal of Object and Frustum Hierarchies . . . 21
4.2.2.1 OH without FH . . . 22
4.2.2.2 FH without OH . . . 22
4.2.2.3 Adaptive OH and FH . . . 23
4.2.3 Experimental Results . . . 27

5 Data Transmission, Rendering and Display Synchronization 32
5.1 Data Transmission 32
5.1.1 Data Header: Meta Data 33
5.1.2 Multicast Data 33
5.1.3 Scene Graph Formation at Clients 34
5.2 Rendering and Display Synchronization 35
5.2.1 Load Distribution 35
5.2.2 Parallax 35
5.2.3 Display Tearing 36
5.3 Experimental Results 36

6 Transparent Rendering to a Tiled Display 40
6.1 Transparent Rendering Using GSWall 40
6.2 Limitations 43

7 Experiments and Results for the Complete System 45
7.1 Scalability 45
7.2 CPU Load 45
7.3 Cache 45
7.4 Optimal Cache-Size 47
7.5 Load Distribution 49
7.6 High-Performance Rendering Nodes 49
7.7 Comparison with Chromium 50

8 Conclusions and Future Work 52

Bibliography 54

List of Figures

Figure		Page
1.1	A 2×3 tiled display wall	3
2.1	Chromium in tilesort configuration	9
2.2	Chromium rendering Saturn	10
3.1	Geometry Server	12
3.2	Server-Client control flow	14
3.3	GSWall Pipelining	16
3.4	A 4x4 display wall showing UNC's power plant	18
4.1	Frustum Hierarchy	20
4.2	Object Hierarchy	26
4.3	Adaptive Culling	27
4.4	Culling performance comparison for various approaches	28
4.5	Experiment: Culling performance on Fatehpur Sikri	30
4.6	Experiment: Culling performance on power plant	31
5.1	Objects spanning across tiles	33
5.2	Header packing	34
5.3	Parallax due to gap between tiles	36
5.4	Display tearing	37
5.5	Experiment: Network usage	38
5.6	Experiment: Scalability with respect to network usage	39
6.1	Transparent rendering using Chromium vs. GSWall	41
6.2	Control flow for transparent rendering	42
7.1	Experiment: Scalability plot	46
7.2	Experiment: CPU usage graph	46
7.3	Experiment: 4300-frame walkthrough	47
7.4	Experiment: Varying cache sizes	48
7.5	Experiment: Load Distribution	48
7.6	Experiment: GSWall with Nvidia 6600GT rendering nodes	49
7.7	Fatehpur Sikri on a 4×4 display	51
7.8	Power plant on a 4×4 display	51

List of Tables

Table Page

4.1 Visibility Culling performance 29

5.1 Startup time 38

6.1 Operations while intercepting OSG functions 43

7.1 Chromium vs. GSWall 50

List of Algorithms

1 Preprocessing a scene graph 22
2 Hierarchical Frustum culling 24
3 Adaptive OH and FH culling algorithm 25

Abbreviations

AABB : Axis-Aligned Bounding Box
BSP : Binary Space Partitioning
FH : Frustum Hierarchy
FPS : Frames per Second
GSWall : Geometry-Served Display Wall (Chapter 3)
GLUT : The OpenGL Utility Toolkit
LRU : Least Recently Used
LAN : Local Area Network
MP : Mega Pixels
OBB : Oriented Bounding Box
OH : Object Hierarchy (referring to scene graphs)
OSG : OpenSceneGraph [10]
PC : Personal Computer (referring to commodity workstations)
SDL : Simple DirectMedia Layer
TCP : Transmission Control Protocol
UDP : User Datagram Protocol

Chapter 1

Introduction

1.1 Display Wall

A Display Wall is a large display system for scientific and medical visualization applications and for public displays. They are also useful for setting up immersive virtual-reality environments where the four walls and the ceiling are made of giant displays. Graphics accelerators on commodity systems cannot power such huge displays. A desktop workstation with high-end commodity graphics accelerator card can power resolutions up to 4 MP in twinview mode. Higher resolutions are either not possible or come at the cost of impaired refresh rate. This severely limits visualization capabilities for off-the-shelf workstations.

1.1.1 Focus plus Context Displays

Even though computing resources have been following Moore's law and graphics accelerators have been exceeding Moore's law with product cycles being barely 6 months, the display resolution for personal computers has grown very modestly over the past two decades. 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. While you can zoom in to view fine details, you lose the bigger picture. If you zoom out to get the bigger picture, you lose out on the minute details. Many professional computer users today work with visual documents that are large and detailed to fit on their screen. Examples can be found in chip design (blueprints of a semiconductor wafer), architecture, graphic design, geographic information systems, etc [21]. An orthopedic surgeon might want to view the detailed nature of a fracture while maintaining the contextual view to the body. Users of bioinformatics visualization applications want to view specific portions of large molecules in detail while preserving the context about how it is structurally connected to the molecule.

The most basic approach is zooming and panning [22]. Another approach, overview plus detail [46], uses two windows, i.e. one showing the entire view while the other showing the specific portion in detail. However, users must visually switch between windows and reorient themselves every time they do. Fisheye views [37] avoid using a second windows, but introduce distortion, which interferes with any task that requires judgment about scale, distance, or alignment. Baudisch et al. embedded high-

resolution portions in the screen while displaying low resolutions for the rest [20] in an effort to achieve focus plus context. Want et al. proposed a focus+context framework to magnify the features of interest [59]. 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 [26]. Large displays with high resolution are required to convey a feeling of being immersed in an environment in Virtual Reality applications.

Visual impact has a big role in the film industry. Cinemascope and VistaVision widened the 35mm film, multi-projector systems such as Cinerama and the IMAX format provide even wider presentations. The quest is for a large display that also has very high resolution. Multi-projector displays driven by a cluster of computers are attracting attention in recent times, driven by recent color-balance and seamless tiling technologies [26, 43, 44].

High-performance specialized rendering workstations are commercially available. They can power displays of resolutions much higher than commodity PCs. However, they are very expensive with the cost sometimes running into millions of dollars. Besides, they generally come marked for a particular upper resolution limit. Hence, they have severe scalability issues with respect to the display wall size. Upgrading the size of the wall generally means purchasing more specialized rendering workstations, which may not always be feasible. Alternatives to using dedicated high-performance rendering systems has been a hot area of research in visualization.

1.2 Cluster-based Tiled Display Wall

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. A tiled display (Figure 1.1) consists of a number of displays arranged together in a rectangular grid. Each display is called a tile. In such a tiled-display wall setup, each tile is powered by a node in the cluster, called a rendering node, also referred to as a client. One node acts as a server and it controls the other rendering nodes. The server may act as a rendering node as well. The viewer's view-frustum, referred to as primary view-frustum, is disintegrated into a rectangular grid of possibly non-symmetric sub-frustums, one for each tile in the display wall.

Network throughput and latency has always been the bottleneck in all cluster-based rendering systems. Chromium [40], a popular software API has been commonly used by most existing display wall setups [2, 3, 5, 7, 52]. These setups utilize a number of rendering nodes in a graphics cluster arranged in a rectangular grid to form the display wall. Myrinet, a low-latency high-throughput protocol requiring special hardware is used as the interconnect. However, Chromium is too wasteful of the network resources owing to its design. It deals with very low-level primitives due to which it cannot take advantage of higher level coherence in the graphics stream. Preprocessing load shoots up and graphics data has to be transmitted to the rendering nodes every frame even when there's no real visual change. Due to these factors, Chromium is practically ineffective in interactive visualization of scenes involving a

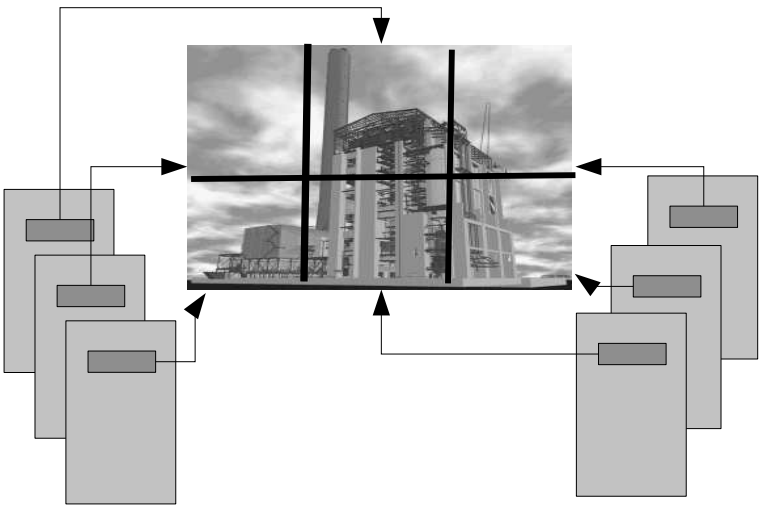


Figure 1.1 A 2×3 cluster-based tiled display wall

lot of heavy data, very common to walkthrough applications. especially with commodity interconnect technology. An advantage of Chromium is that any OpenGL-based graphics application can be used to render to a display wall. The application need not be aware of the wall. In fact, the application does not need recompilation or even relinking.

Our tiled display wall uses a client-server architecture. The server loads up the scene and controls most aspects of the geometry rendering pipeline using a geometry-push philosophy. In a sense, the server acts as a geometry server, a repository of large models. The server determines the visible portions of the scene for each of the rendering nodes and transmits only these portions. The rendering nodes cache the objects that are received once so that they need not be transmitted from the server. Once all the clients are finished with rendering the scene that is present in their view-frustums, The server coordinates the synchronization of the displays of all the rendering node so as to produce a coherent display. The viewpoint for all the rendering nodes is the same, that of the viewer. The rendering load is distributed across the rendering nodes. No individual node, including the server, needs to render the complete scene. Upgrading the display wall is as easy as adding more nodes in the cluster. Our display wall system can provide tiled display facility to any application built using the OpenSceneGraph (OSG) API automatically and transparently. The application does not need to modified, compiled or even relinked. This gives our display wall the ease of use of Chromium while improving on the performance at the same time.

The switch to a cluster based display wall as opposed to a dedicated high-performance rendering system comes at the cost of a few trade-offs. For instance, the tiles might have gaps between each other, especially if the display is to monitors or to LCD screens. Even with projectors, certain portions at the edge of the tiles is generally overlapped and needs to be treated specially. Besides, efficient network management algorithms are required to make optimal use of the network. This entails fast and

conservative visibility determination algorithms to be run at the server. Also, the displays need to be synchronized to each other to avoid display tearing.

1.3 Contributions of this Thesis

This thesis presents the design of a cluster-based tiled display wall that is built using commodity PCs and LAN. The fundamental difference with solutions like Chromium [40] is that the geometry is cached at the client nodes and coordinated by the server to exploit temporal and spatial coherence. This has been possible by working with high-level scene representation rather than primitives like lines and triangles as in Chromium. Our system is able to provide high-quality tiled display facility to any environment represented using the Open Scene Graph [10] API. Network load is minimized by client-side caching and multi-casting of objects. This makes our system scalable to a large number of tiles unlike existing solutions that get constrained by network bandwidth. A new algorithm for adaptively determining the visibility of objects to individual rendering nodes involves merging the hierarchies corresponding to the scene graph and the hierarchy of view frustums of rendering nodes. The implementation also includes an interceptor which can intercept library calls from any application written using the OpenSceneGraph API to render on a display wall. Not only is there no special coding required, the application need not even be recompiled or relinked, thereby making the implementation useful for a lot of already existing applications written using OpenSceneGraph.

1.4 Experimental Setup Used in the Thesis

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 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 3.4 shows the Powerplant model on our 4×4 display wall with the parallax error compensated.

1.5 Overview of the Thesis

Chapter 2 reviews the background literature on display walls and the underlying issues, the foremost being visibility determination. This chapter also describes the existing display wall techniques, especially Chromium. The design of our display wall setup is presented in Chapter 3 along with all the underlying speedup techniques utilized. The visibility determination stage of the display wall pipeline is described in great detail in Chapter 4. Chapter 5 discusses the issues relating to data transmission over the network and how we manage to keep the network load virtually independent of the display wall size along with the rendering and synchronization aspects of the system. Chapter 6 discusses the design of the transparent tiled rendering using the OpenSceneGraph API. Experimental results are also included along with individual descriptions wherever necessary. Results for the overall system are dealt with in Chapter 7. Some conclusions and future work are presented in Chapter 8.

Chapter 2

Background and Related Work

This chapter reviews the background literature used for setting up display walls especially using a cluster of workstations. Existing display wall setups are also described. We also review the issue of visibility determination that is necessary for the interactive performance of such setups.

2.1 Display Wall Systems

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.

2.1.1 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 [11]. Such solutions are expensive and non-scalable. They are also difficult to construct and require expert-level maintainers. Typically, these setups come marked for a particular resolution. Upgrading entails purchasing replacing the setup of purchasing additional components which are costly and may not be feasible always.

2.1.2 Cluster-based Displays

Cluster-based solutions for creating large displays have gained a lot of interest recently [28,40]. Such displays are constructed using a number of commodity PCs interconnected on a LAN or a low-latency network like Myrinet, as in [11, 12, 40]. 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 [42].

Two approaches are popular in cluster-based display setups: *master-slave* and *client-server* [28]. They are discussed below.

2.1.3 Master-Slave approach

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 [25], 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 [23]. Net Juggler [17] is an open source library that turns a commodity component cluster running the VR Juggler [23] 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.

2.1.4 Client-Server approach

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 [49, 50]. 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 [51]. The server might use a distributed data management framework as followed by Gao et al. [38]. 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 as in Chromium [40] or at the display manager level as with DMX [1]. However, since the system is not able to exploit high-level scene structure, the network requirements are tremendous, often ineffective for rendering scenes with large amounts of geometry. The latter can provide tiled display including all windowing features including menus, toolbars and decorations. The former provides large display facility to any application using the OpenGL API, such as the Hyperwall [52], Viswall [12], LionEyes Display Wall [5] 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 [53] and OpenSG [58] for display wall rendering. The Syzygy software library [53] 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 [55] and Matrox's ASM [6] or software/hardware solutions like SoftGenLock [16] or WinSGL [60] 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.

2.1.5 Chromium

Chromium [40] is a software API used in most existing cluster-based display walls and therefore it needs special attention. Chromium is a system for interactive rendering on clusters of graphics workstations. Various parallel rendering techniques such as sort-first and sort-last may be implemented with Chromium. Furthermore, Chromium allows filtering and manipulation of OpenGL command streams for non-invasive rendering algorithms.

Among Chromium's features are

- Sort-first (tiled) rendering - the frame buffer is subdivided into rectangular tiles which may be rendered in parallel by the hosts of a rendering cluster.

- Sort-last (Z-compositing) rendering - the 3D dataset is broken into N parts which are rendered in parallel by N processors. The resulting images are composited together according to their Z buffers to form the final image.

- Hybrid parallel rendering - sort-first and sort-last rendering may be combined into a hybrid configuration.

- OpenGL command stream filtering - OpenGL command streams may be intercepted and modified by a stream processing unit (SPU) to implement non-photorealistic rendering (NPR) effects, etc.

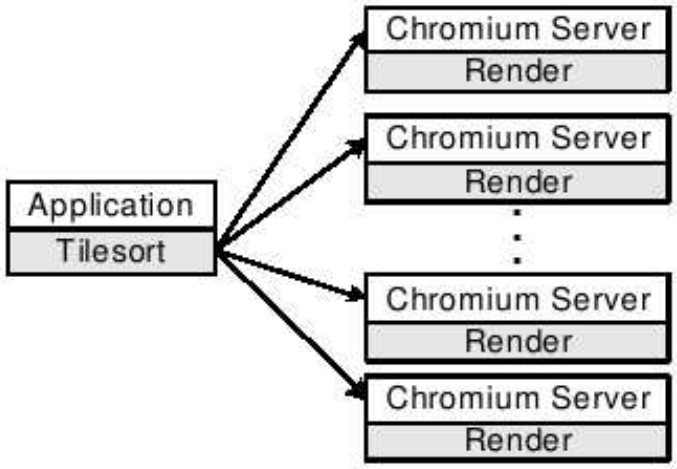


Figure 2.1 Chromium in tilesort configuration for rendering to a tiled display. Courtesy [40]

- Many OpenGL programs can be used with Chromium without modification.

Conceptually, for display wall rendering, Chromium is a stream processor which performs sorting of the graphics primitives to a tiled-display as shown in Figure 2.1. A significant positive aspect of Chromium is that it does not require a graphics application to be modified or even recompiled for rendering to a display wall (Figure 2.2). However, Chromium has several weaknesses in its use for display wall rendering. First, since it works on very low-level primitives (lines, points, triangles) by intercepting function calls from an OpenGL graphics stream, it cannot exploit higher-level scene structure out of the sense; no relation between the stream elements can be figured out. Therefore, Chromium has to send all graphics data to the rendering nodes every frame; it cannot figure out whether there has been a real visual change or not. This puts a tremendous load on the network, so much so that Chromium becomes inefficient for interactive display wall rendering of moderately-heavy scene. Several hacks have been attempted to perform stream caching but no comprehensive solution has been worked out.

2.2 Visibility Determination

Visibility determination has been a fundamental problem in computer graphics [56] since the scene is typically much larger than the graphics rendering capabilities. Cláudio et al. [30] and Durand et al. [34] have presented comprehensive visibility surveys. View-frustum culling algorithms avoid rendering geometry that is outside the viewing frustum. Hierarchical techniques have been developed [29], as well as other optimizations [19,41]. Funkhouser et al. [36] described a system that could support models larger than main memory, based on the from-region visibility algorithm of Teller and Sequin [57]. Aliaga et

Figure 2.2 Chromium rendering a low-geometry model of Saturn over a 4×4 tiled display wall.

al. [15] described MMR, the first system to handle models with tens of millions of polygons at interactive frame rates, although it did require an expensive high-end multi-processor graphics workstation.

Assarsson et al. [19] presented several optimizations for fast view frustum culling, using different kinds of bounding boxes and bounding sphere. For their octant test, they split the view frustum in half along each axes, resulting in eight parts, like the first subdivision of an octree. Using bounding sphere for objects, it is sufficient to test for culling against the outer three planes of the octant in which the center of the bounding sphere lies. This can be extended to general bounding volumes as well [18]. Our frustum hierarchy approach is inspired by this idea of subdividing the view-frustum into octants. However, Assarsson et al. divide the view-frustum only once, whereas we complete this procedure to construct a full frustum hierarchy. Bittner et al. [24] used hardware occlusion query techniques to exploit temporal coherence and reduce CPU-GPU stalls for occlusion culling. Since the occlusion culling information holds good for all frustums for our specific case of tiled display walls, separate occlusion culling for each frustum is not necessary.

Another way to look at occlusion relationships is to use the fact that a viewer cannot see the occludee if it is inside the shadow generated by the occluder. Hudson et al. [39] proposed an approach based on dynamically choosing a set of occluders, and computing their shadow frustums, which is used for culling the bounding boxes of a hierarchy of objects. Bittner et al. [61] improved this method by combining the shadow frustums of the occluders into an occlusion tree. This method has an advantage over Hudson et al. as the comparison in the latter is done on a single tree as opposed to each of the m frustums individually, hence improving the time complexity from O(m) to O($\log m$). Our approach of

constructing a tree of view frustums resembles this technique of handling frustums. We go even further to combine the frustum hierarchy with object hierarchy.

2.3 Scene Graph APIs

A scene graph is a general data structure commonly used in games. It is an object-oriented structure that arranges the logical and often (but not necessarily) spatial representation of a graphical scene They provide higher level functions for managing the representation of the scene as well as for interacting with the graphics hardware. Scene graphs are ideal for modern games using 3D graphics and increasingly large worlds or levels. In such applications, nodes in a scene-graph (generally) represent entities or objects in the scene. For instance, a game might define a logical relationship between a knight and a horse so that the knight is considered an extension to the horse. The scene graph would have a 'horse' node with a 'knight' node attached to it. As well as describing the logical relationship, the scene-graph may also describe the spatial relationship of the various entities: the knight moves through 3D space as the horse moves. Commonly used scene graphs include OpenGL Performer [48], Open Inventor [9], Nvidia Scene Graph [8] and OpenSceneGraph (OSG) [10, 27].

OpenSceneGraph is an open source implementation of a scene graph. It provides an object oriented framework on top of OpenGL freeing the developer from implementing and optimizing low level graphics calls, and provides many additional utilities for rapid development of graphics applications. Like any other scene graph, OpenSceneGraph maintains a spatial bounding-volume arrangement of a scene. All geometry resides at the terminal nodes, known as Geodes, while the internal nodes, known as Groups, maintain the spatial hierarchy. The bounding volume, computed for each node when the scene is loaded, is such that the bounding volume of any internal node completely contains the bounding volumes of all its children. We chose the OpenSceneGraph API for our experiments since it is Open-source, cross platform, free and in wide use by a large number of graphics and visualization developers.

Chapter 3

Geometry-Served Display Wall

The architecture of our tiled-display wall (GSWall) utilizes a Geometry Server. This chapter first describes the geometry server in general and then the adaptations of the geometry server for the specific application of the display wall.

3.1 Geometry Server

Our Display Wall effort is an extension of an earlier work from our group [33] on a geometry server. A geometry server is a high-performance, centralized storehouse for massive geometric data. The server has a scene graph based representation of the virtual environment with multiple levels of detail. Generally, the server has sufficient amount of memory to be able to accommodate the entire model. Alternatively, the server might store just a skeleton of the scene in the memory, thereby fetching data from a secondary storage when required. In fact, it's common to have a hybrid of both. This allows the geometry server to handle massive models which are larger than its memory, or perhaps models which have procedural data.

The geometry server 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 (Figure 3.1). Each client gets a visibility-limited portion of the model that is compatible with its rendering capabil-

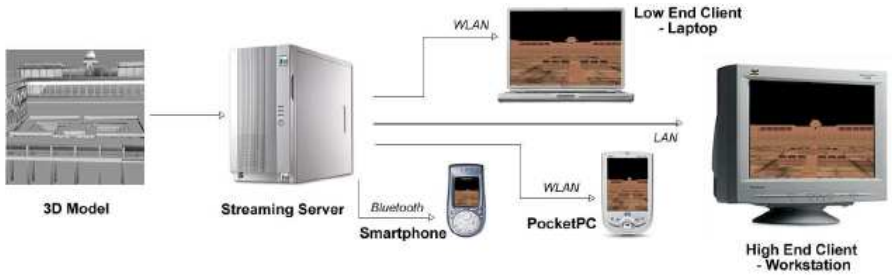


Figure 3.1 Geometry Server catering to a variety of clients. Client applications can access the model from the server transparently. Figure courtesy [33]

.

ities, computational resources and network characteristics, with an objective of providing consistent, interactive frame rates. Client caches 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. 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 system on a wide range of clients and finds ready applications in battlefield simulation, terrain visualization, etc. 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. A client module facilitates this transparency of use. Note that the model operates in a pull-philosophy. The clients initiate view-point change and hence effect potential data transmission. Salient features of the system include:

- *Visibility based representation:* Only visible portions of the scene are transmitted by the server
- *Compression of transmitted data:* Data is compressed before transmission. Clients uncompress while unpacking the data
- *Client-side prediction of motion:* Client module predicts viewpoint-motion so as to pre-fetch data.
- *Caching by clients;* Clients cache data locally for subsequent reuse. Cache size is limited. An LRU scheme is used to evict objects from the cache when it's full.

Adaptations for GSWall: Our display wall system optimizes the philosophy of the geometry server for a large cluster-based display systems. The geometry server just maintains independent contexts for each rendering node, essentially to keep track of the status of each rendering node and the contents of their cache. 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. There's a shift in philosophy from client-pull in geometry server to server-push in GSWall; this allows for further pipelining of individual stages of our display wall and also enables the server to perform synchronization of all displays.

3.2 Geometry-Served Display Wall

The design of our Geometry-Served Display Wall system [45], known as GSWall, is presented in this section.

3.2.1 Rendering Pipeline

Our Display Wall system follows a client-server architecture, as opposed to a master-slave one. The server loads the whole virtual environment as a scene graph in its memory and performs some preprocessing. This preprocessing includes the computations for obtaining the bounding boxes for all

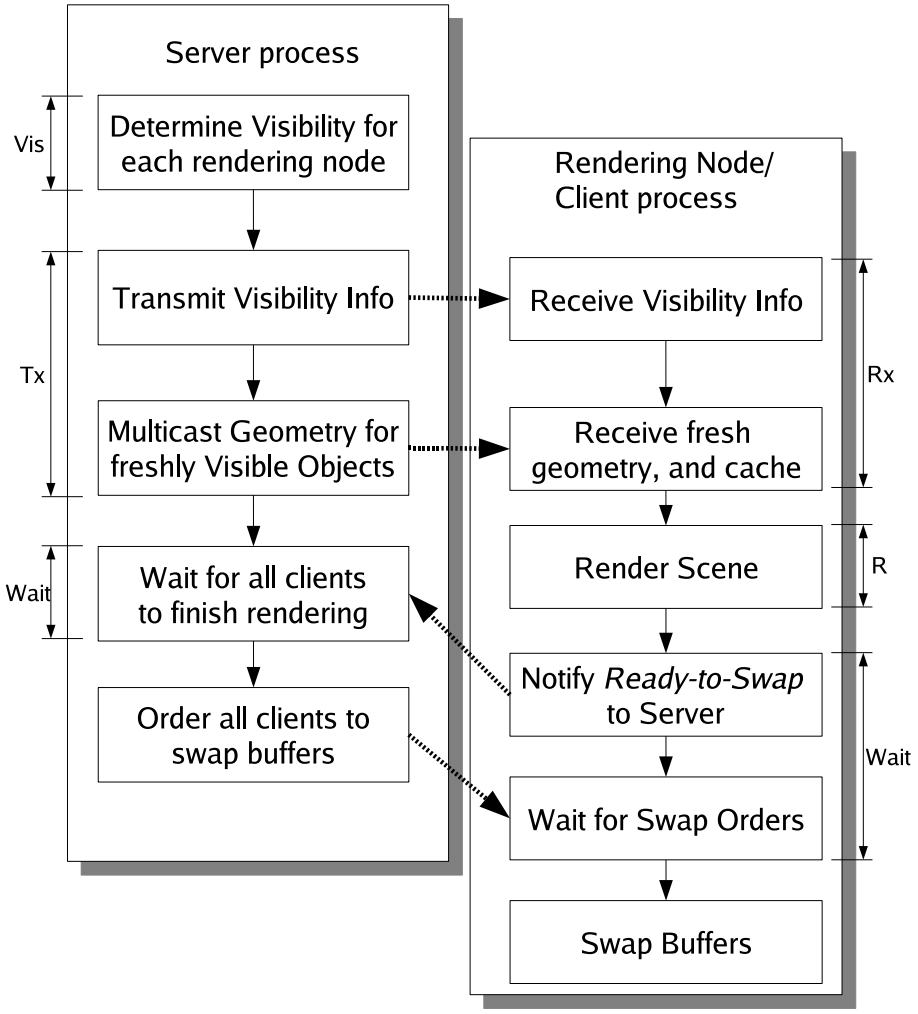


Figure 3.2 Server-Client control flow

the internal as well as terminal nodes in the scene graph so that later visibility computations can use it. A master node controls the viewpoint typically in response to keyboard/mouse inputs. It notifies the server of a viewpoint change. Any of the rendering nodes or even the server can act as the master node too. The server performs visibility culling to determine the objects that are visible to each of the tiles. If these objects are not present already, they are transmitted. The clients cache these objects so as to be reused subsequently. If a client's cache is full, an LRU-scheme is used to evict old objects. Once the clients receive data, they start rendering but don't swap buffers until the server allows it. They instead inform the server about their readiness to swap. When the server determines that all rendering nodes are done with rendering and are ready to swap, it orders a swap for all the clients, at which point all the rendering nodes swap their buffers. This produces a coherent display.

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.

3.2.1.1 Server's 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, 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 3.2 shows these steps in a flow diagram.

3.2.1.2 Client's Tasks

Clients receive lists of objects from server for each frame. These are the objects that are expected to be transmitted. Each client 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. Note that this has been possible by treating the scene in terms of higher-level objects instead of low-level primitives. 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 neither the server nor any of the clients has to render the whole model. It computes the per tile visibility based on the object's bounding boxes using a novel algorithm described later. Client-side caching reduces network bandwidth by exploiting temporal coherence. Additionally, multicasting the actual objects minimizes 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:

1. Visibility Determination

2. Data transmission

3. Rendering and Synchronization

These are the critical operations that directly determine the overall performance. These three stages are therefore discussed in detail in later chapters.

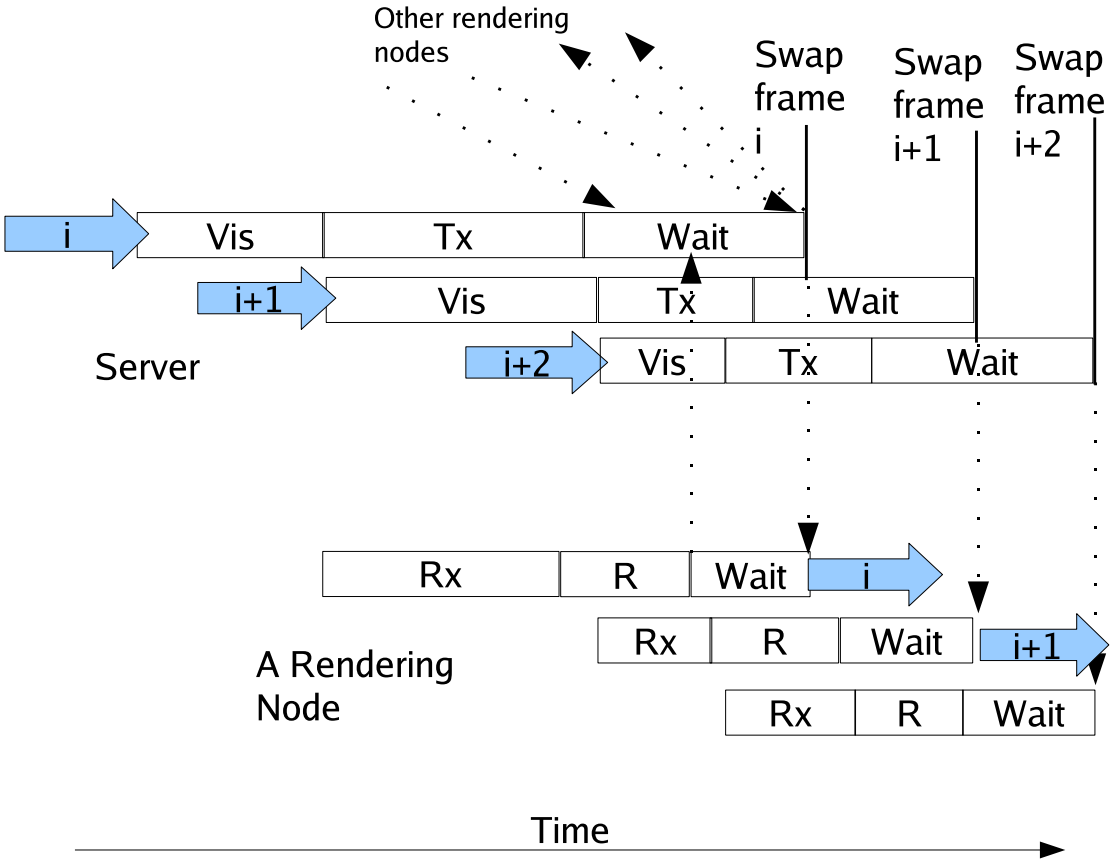


Figure 3.3 Inter-frame pipelining of the 3 stages of Display Wall rendering. *Vis* denotes the Visibility Determination Stage, *Tx* the geometry transmission stage, *Rx* the geometry reception stage and *R* the rendering stage. The effective FPS of the system gets increased due to this interleaving.

3.2.2 Pipelining

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 3.3 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. Pipelining is especially useful if the systems have multiple CPUs so that while a CPU is waiting for I/O, another thread can be working.

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.

Figure 3.4 A 4×4 display wall rendering of UNC's Powerplant. The combined resolution is 12 MPixels. Efficient rendering to a display wall requires fast visibility culling of the scene to all the frustums. Adaptive culling by merging of the object and frustum hierarchies makes this possible for even bigger tile configurations

Chapter 4

Visibility Culling for Tiled Displays

Visibility culling of a scene is central to any interactive graphics application. The idea is to limit the geometry sent down the rendering pipeline to only the geometry with a fair chance of eventually becoming visible and appearing on the display. It is important for the culling stage to be fast for it to be effective; otherwise the performance gain achieved will be overshadowed. Hierarchical scene structures are commonly used to speed up the process. Hierarchical culling of bounding boxes to a view frustum is fast and sufficient in most applications. Assarsson et al. [19] described several optimizations for view frustum culling. Bittner et al. [24] exploited temporal coherence to minimize the number of occlusion queries for occlusion culling to a view frustum.

Fast frustum culling is particularly crucial for rendering to multiple frustums simultaneously. (1) CAVE [32] is a multi-display virtual-reality environment which requires visibility culling to multiple frustums. (2) Another application using multiple frustums involves occlusion culling of a scene by eliminating objects in the frustum shadows formed by all principal occluders, as proposed by Hudson et al. [39]. (3) Cluster-based tiled displays require fast culling to multiple frustums corresponding to each tile in the display (Figure 3.4). (4) Multi-projector display systems [47] use several overlapping frustums corresponding to each of the projectors. (5) Multiple frustums are also required to compute visibility for architectural environments [13, 14, 57].

4.1 Object Hierarchy and Frustum Hierarchy

We work with two hierarchies in our culling technique. The first is the spatial hierarchy of the scene, represented using a scene graph (OpenGL Performer [48], Open Scene Graph [27]). In the Object Hierarchy (OH), each node has a bounding volume such that the bounding volume of an internal node entirely encloses the bounding volumes of all its children. Only leaf nodes contain actual geometry. A well-formed scene graph would have compact geometry nodes so that bounding volumes can be used to provide accurate visibility tests.

The second hierarchy we deal with is that of view frustums (Figure 4.1). Our Frustum Hierarchy (FH) is analogous to a BSP-like division. In the most general scenario, a number of independent view frustums in 3D are grouped together hierarchically. Every internal node's bounding volume encloses

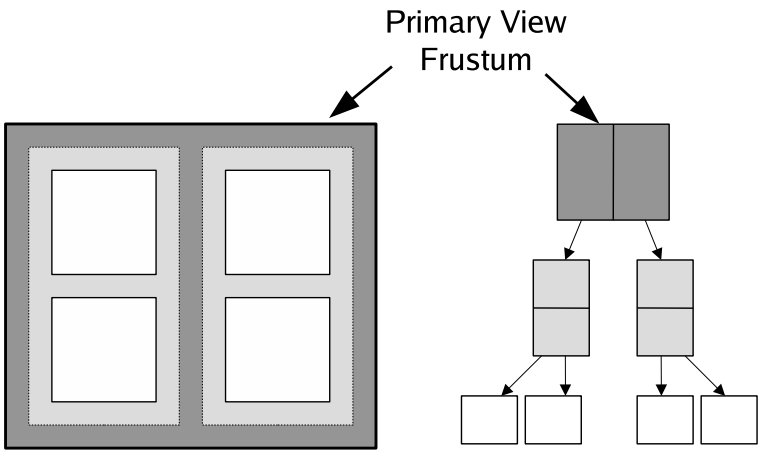


Figure 4.1 Frustum Hierarchy (FH). White boxes represent view frustums. Their hierarchical grouping for 3 levels is shown on the right. The bisection plane at each internal node is also shown. Note that near and far planes are not shown

that of its children. A plane bisects each internal node's volume into half-spaces containing its children. The leaf nodes in the hierarchy correspond to individual view frustums. Each rendering node corresponds to one view-frustum for a tiled display wall application. The root node in the frustum hierarchy corresponds to the primary view-frustum (shown in Figure 4.1). The case of overlapping view frustums, commonly used for multi-projector displays, is easily handled by treating the overlapping regions as additional independent frustums.

4.2 Visibility Culling

In this section, we use a hierarchical representations of scene structure and that of view frustums. We discuss various alternatives that we experimented with and then lead the discussion to our method which adaptively merges the two hierarchies – the scene hierarchy and the frustum hierarchy – for visibility culling. To this end, we present an algorithm which determines which hierarchy to traverse down and when. To our knowledge, this is the first work which considers this decision to be important and effective for coherent culling to multiple frustums. Here, we address the specific problem of culling an object hierarchy to a frustum hierarchy for a tiled display wall (Figure 3.4). Our tiled display wall system uses a number of commodity systems in a cluster, each powering a tile. The system uses a scene graph (OpenSceneGraph representation of a massive scene. The network resources limit the amount of data that can be transmitted, thereby making efficient visibility culling an important requirement. The individual frustums in the display wall have a fixed arrangement with respect to each other and have a common viewpoint. Such a tight arrangement of frustums motivates our visibility culling algorithm to perform coherent computations which are both fast and scalable. We are able to bring down the culling

time for a hierarchical version of UNC's power plant model for a 4×4 tiled display from about 14 ms using the traditional approach to about 5 ms using our adaptive algorithm.

Our visibility culling approach performs *from-point* visibility as opposed to *from-region* visibility performed by several other culling techniques [15, 35]. As discussed by Correa et al. [31], from-point visibility has several advantages over from-region visibility. First, from-region methods typically require long preprocessing times (tens of hours), while our from-point method requires little preprocessing. Second, the set of nodes visible from a single point is typically much smaller than the set of nodes visible from any point in a region. Third, some from-region methods require that cells coincide with axis-aligned polygons in the model. Our from-point method imposes no restriction on the model's geometry. Finally, the nodes visible from a cell may be very different from from the nodes visible from a neighbor of that cell. Thus, a from-region method may cause bursts of disk activity when the user crosses cell boundaries, while a from-point method can better exploit frame-to-frame coherence. Thus, choosing to perform from-point visibility enables handling of dynamic scenes. Besides, our culling approach is conservative, as opposed to other probabilistic or approximate culling techniques [31, 41, 54], which can lead to serious rendering artifacts. This is critical for the kind of applications in which the multiple frustums come into use. For a cluster-based tiled display wall, for instance, the load on the network needs to be minimized and the interactivity needs to be preserved. Culling determines the geometry that will be cached on the rendering nodes. Approximate culling techniques lead to probabilistic prefetching, often leading to freezes during rendering.

4.2.1 Preprocessing

Oriented bounding boxes (OBB) give a compact approximation of an object's geometry and orientation in space. It is desirable that culling be performed to OBBs as opposed to the whole geometry since it is fast and conservative. During the preprocessing stage, the scene graph is loaded into the main memory. For a set of 3D points, their eigenvectors represent their orientation. Therefore, at the leaf nodes of OH, the eigenvectors of the geometry points provide oriented bounding boxes. However, at the internal nodes, we compute the eigenvectors using just the children's bounding box vertices. This is a fast approximation of an oriented bounding box for the internal node.

A Frustum Hierarchy FH is constructed, with each internal node having a bisection plane. Beginning at the root node, `root` in OH, call Algorithm 1 as `preprocess(root)`. Preprocessing takes place in a bottom-up fashion. The bounding box information thereby computed is stored with each of the nodes in the scene graph.

4.2.2 Traversal of Object and Frustum Hierarchies

Ideal traversal through OH and FH is crucial for optimal performance. Our culling procedure for determining scene visibility for a tiled display wall involves a first level culling to the primary view frustum so as to eliminate objects completely outside the view. The next step involves classifying these n objects to m view frustums. A *naive approach* involves testing each of these objects with all the

Algorithm 1 preprocess(OH_Node)

```
1: G ⇐ φ
2: if not leaf(OH_Node) then
3:    for all child c of OH_Node do
4:       preprocess(c)
5:       G ⇐ G+ bounding box vertices of c
6:    end for
7: else
8:    G ⇐ G+ OH_Node.getGeometry()
9: end if
10: e ⇐ compute eigenvectors of G
11: BBOX ⇐ compute OBB from e
12: OH_Node.save_bbox(BBOX)
```

view frustums. The expected time complexity for this approach is $O(mn)$. We now discuss several hierarchical variations to this approach, followed by our adaptive algorithm.

4.2.2.1 OH without FH

This is a commonly used approach, wherein the scene graph is culled to all the view frustums one by one. Bounding volume at each node in the object hierarchy is intersected with the view-frustum to determine visibility. The bounding volumes could be axis-aligned or object-oriented or even bounding spheres. This method does not take advantage of the high-level arrangement of view frustums, and therefore performs poorly in applications with large number of view frustums. This approach has a average time complexity of $O(m \log n)$. A typical graphics application that uses multiple view frustums typically has a high-level arrangement of view frustums. This must be exploited to perform coherent calculations for efficiency.

4.2.2.2 FH without OH

It is very common to deal with scenes which do not have any spatial hierarchy. In this section, we describe how we perform their visibility determination.

First, the visibility of different objects in the scene with respect to the primary 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 frustums 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 frustums 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 frustums 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 2 gives

the pseudo-code of the hierarchical frustum culling algorithm. Later, we describe an improvement over this algorithm which adaptively determines the visibility by combining the scene's object hierarchy and the frustum hierarchy of the scene.

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 frustums.
3. Call *FHonly_Cull(*0*,* N*,* 0*,* N*,* O*, true)*

After this procedure, an object O needs to be transmitted only to the rendering nodes corresponding to the frustums 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 [4] 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 [24]. We show results without hardware occlusion queries (Table 4.1), since the amount of useful work that can be interleaved during CPU-wait is subjective to an application,.

The hierarchy of frustums used in Algorithm 2 exploits the high degree of overlap in the computation space of visibility determination for all the rendering nodes. Algorithm 2 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 FHonly_Cull(), in contrast with naive view-frustum collision algorithms where all the 6 planes need to be considered.

Since the frustum hierarchy only is utilized, each object has to be tested against it, beginning from the root. For every internal node in the FH, if an object is present entirely on one side of its bisection plane, its visibility can be safely eliminated from all frustums lying in the other half-space. Therefore, we can potentially eliminate half the number of frustums at each node in the hierarchy. Hence, the average case time complexity is $O(n \log m)$.

4.2.2.3 Adaptive OH and FH

In both the above cases, the two hierarchies (OH and FH) are used independent of each other, i.e. when the OH is traversed, frustums are treated non-hierarchically and when FH is traversed, objects are treated non-hierarchically. Hence, adaptive merging of the two hierarchies leads to substantial reduction in computations. Consider the different cases:

Algorithm 2 FHonly_Cull(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 FHonly_Cull(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 FHonly_Cull(k, j, m, n, O, false)
15:      end if
16:   else
17:      call FHonly_Cull(i, k, m, n, O, false)
18:      call FHonly_Cull(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 FHonly_Cull(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 FHonly_Cull(i, j, k, n, O, true)
31:      end if
32:   else
33:      call FHonly_Cull(i, j, m, k, O, true)
34:      call FHonly_Cull(i, j, k, n, O, true)
35:   end if
36: end if
```

- At leaf nodes in OH, only FH traversal remains.
- At leaf nodes in FH, only OH traversal remains.
- At all internal nodes, decide whether to further traverse FH or OH.

The precomputed data stored during the preprocessing stage (Section 4.2.1) is utilized to arrive at the above decision. If an OH-node is not intersected by an FH-node's bisection plane, the frustum hierarchy should be unfolded further, keeping the OH intact. Unfolding OH here leads to a large number of OH-nodes to deal with in the next iteration. If the bisection plane of an FH-node intersects the bounding box of an OH-node, OH should be unfolded, thereby breaking the object to its constituents. We classify the children into three groups (L=Left, C=Cuts, R=Right) depending on their position with respect to the FH-node's bisection plane. The group L contains all the children lying completely in negative half-space, R contains those lying in the positive and C contains the rest (the objects that cut the plane).

Algorithm 3 adaptive_OHandFH_Cull(OH_Node, FH_Node)

```
1:  if leaf(FH_Node) then
2:     Mark OH_Node as visible to FH_Node
3:     return
4:  end if
5:  [L, C, R] ⇐ ClassifyLCR(OH_Node, FH_Node.plane)
6:  for all c in set C do
7:     adaptive_OHandFH_Cull(c, FH_Node.neg)
8:     adaptive_OHandFH_Cull(c, FH_Node.pos)
9:  end for
10: for all l in set L do
11:    adaptive_OHandFH_Cull(l, FH_Node.neg)
12: end for
13: for all r in set R do
14:    adaptive_OHandFH_Cull(r, FH_Node.pos)
15: end for
```

A call to `adaptive_OHandFH_Cull(OH_Node, FH_root)` (Algorithm 3) is performed for each node `OH_Node` determined to be visible in the primary frustum, where `FH_root` is the FH root. `ClassifyLCR()` is an accessory function which categorizes `OH_Node`'s children into the sets L, C and R according to their position with respect to an FH node's current bisection plane. The algorithm recurses for the members in L and R to the corresponding child-frustum (Algorithm 3: lines 11, 14) while the members of C is recursed for both the child-frustums (Algorithm 3: lines 7–8). When the FH is exhausted, the remaining objects are marked to be visible in the corresponding view frustum. The objects in set C need to be checked with both the half-spaces. However, the number of frustums under consideration get reduced by half for objects in set L and R, thereby potentially halving the computations. The number of children to deal with might increase because `classifyLCR()` breaks up an OH node into its children. At this stage, there are two options. We can carry on with each

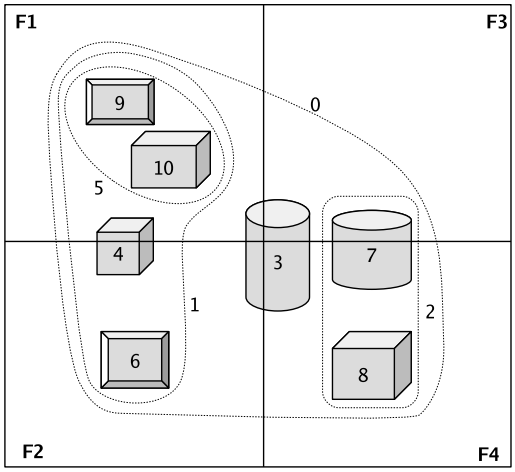


Figure 4.2 Hierarchy of objects as visible to a 2×2 tiled arrangement of view frustums. The grouping of objects is shown. F1, F2, F3 and F4 represent view frustums. Their adaptive culling is shown in Figure 4.3

object independently or can regroup the objects in sets L and R into pseudo-groups. This involves recomputing the bounding box for the pseudo-group. Pseudo-groups do not really exist in the scene graph but can reduce the computations required for further stages. Our experiments show that this regrouping is advantageous only for scene graphs with very high branching factor. Otherwise, the overhead of forming the pseudo-group overshadow the gain achieved. Note that pseudo regrouping is not shown in Algorithm 3.

Our algorithm follows an $O(m\, n^{\log_p q} + (p-q)\log m)$ time complexity, on a quick analysis, where p is the average branching factor of OH and q is the average number of nodes in set C. Exact analysis is difficult as it depends on the goodness of the branching and the spatial separation of child nodes at each level. Hence, p and q depend heavily on the scene structure, the view frustums arrangement and the viewpoint. The average case time complexity follows a sub-linear pattern. In the worst case, the complexity becomes $O(m\, n)$ when $q = p$, when all OH leaves fall in all FH leaves, and the hierarchy is inconsequential. In the best cast, the complexity is $O(\log m)$ when $q = 0$. This is the situation when only one FH leaf contains the entire OH. In practice, the algorithm is able to adapt to variations in complexity of the visible scene, which is very common during interactive walkthroughs.

Line 2 of Algorithm 3 marks `OH_node` as visible to the `FH_node`. Line 5 performs the classification of the object node to L, C and R. Lines 7, 8 recurses for every child in C, the set of objects cut by the plane. Lines 11 and 14 recurse to the next stage of FH.

The algorithm can easily deal with dynamic scenes as well, since the preprocessing stage involves cheap eigenvector calculations only. The visibility determination remains unchanged. Besides, very little extra data is stored for the algorithm execution. Note that bisection using a plane is possible in Algorithm 3 for an application such as tiled display walls because the tile sizes are uniform and the frustum space ultimately divides to form the individual tile frustums. This might not be true for non-

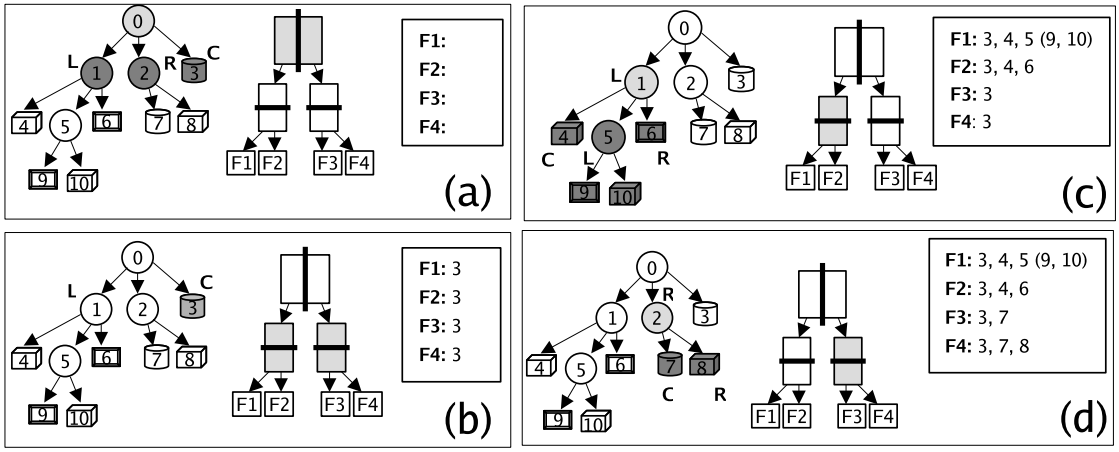


Figure 4.3 Adaptive culling of the scene structure in Figure 4.2. The object and frustum hierarchies are shown along with already determined visibility list. Working nodes are shown as light-gray. Dark gray objects are the ones that need to be recursed further. (a) OH root is classified as per the bisection plane of the FH root. L, C, R classification is shown. (b) Continuing culling for set C. (c) Continuing culling for set L. (d) Continuing culling for set R

uniform frustums. However, a hierarchy of frustums can still be built. Only, in such a case, the terminal frustum in line 2 of Algorithm 3 will further involve a check for visibility before marking an object as visible.

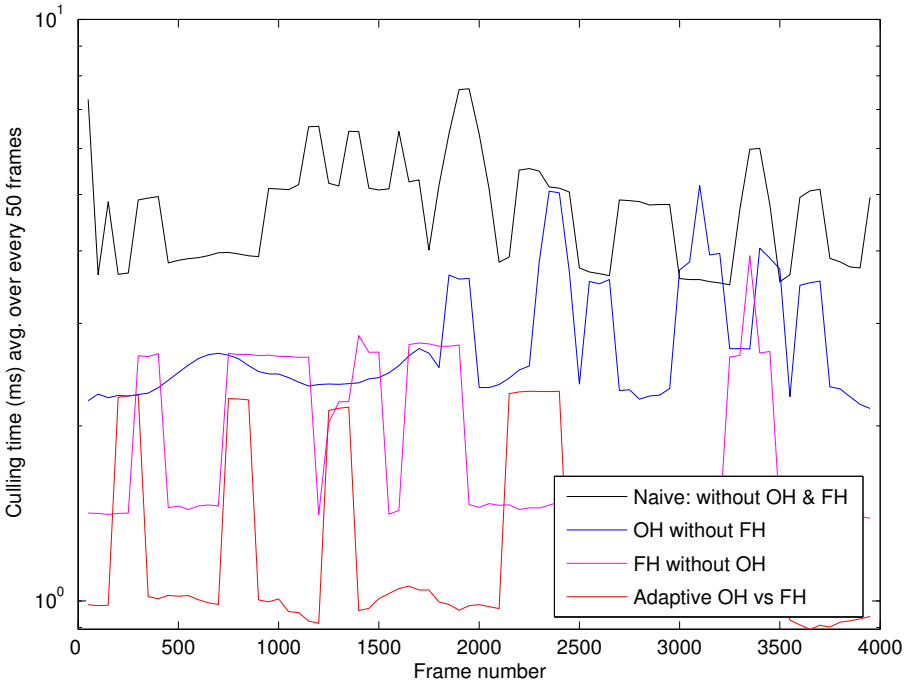
4.2.3 Experimental Results

We present experimental results from our visibility culling algorithm for the Fatehpur Sikri model and UNC's Power plant model. We have focused only on fast culling to multiple frustums, and have therefore not discussed the later stages in the rendering pipeline. We compare the results with different variants for culling to multiple frustums. We also investigate the performance of our culling technique for a dynamic scene, when many objects change position. This involves additional overheads in updating the bounding boxes at many nodes before the culling can be performed.

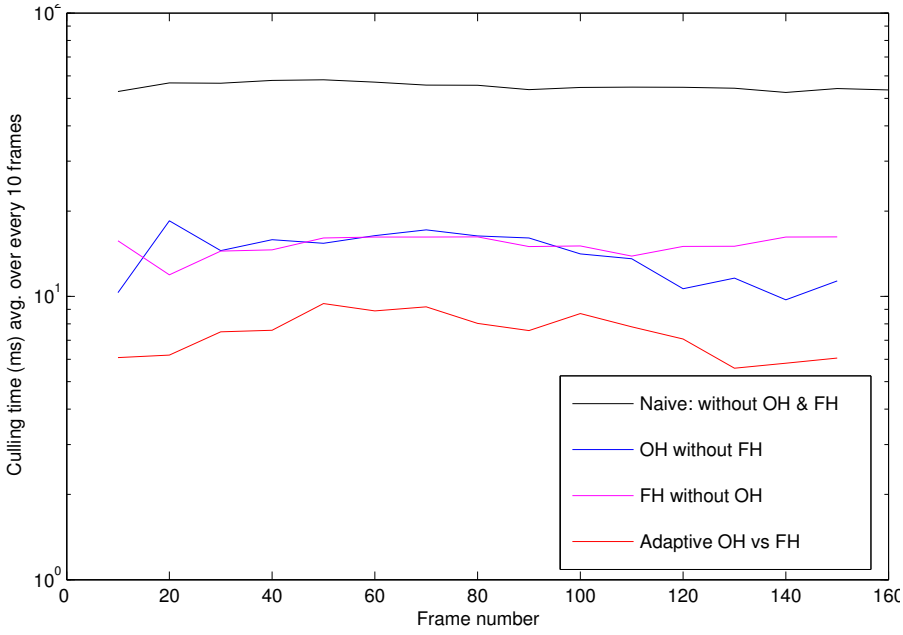
Table 4.1 shows the time taken to perform the Visibility Determination using Algorithm 2 on UNC's Powerplant model with 483M of geometry in 1185 objects.

We then perform several walkthrough experiments on models of different scene complexities to test the performance of the adaptive algorithm. We used a hierarchical model of Fatehpur Sikri, which has 1.6 million triangles spread over 770 nodes (288 internal $+$ 482 leaf), with an average branching factor of 2.67. We also used a hierarchical model of UNC's power plant, which has geometry spread over 5037 nodes (1118 internal $+$ 3919 leaf), with an average branching factor of 4.5.

Figure 4.4(a) shows a logarithmic plot of culling time taken by various algorithms discussed in Section 4.2.2 for a 4000 frame walkthrough on the Fatehpur model. The walkthrough is such that the entire scene is visible. This is a worst-case situation; typical walkthroughs perform better. Our adaptive



(a) A 4000 frame walkthrough on the Fatehpur Sikri model



(b) Walkthrough on the power plant model

Figure 4.4 Culling performance for various approaches. The lower the time, the better. Our adaptive algorithm outperforms others almost throughout the walkthrough

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

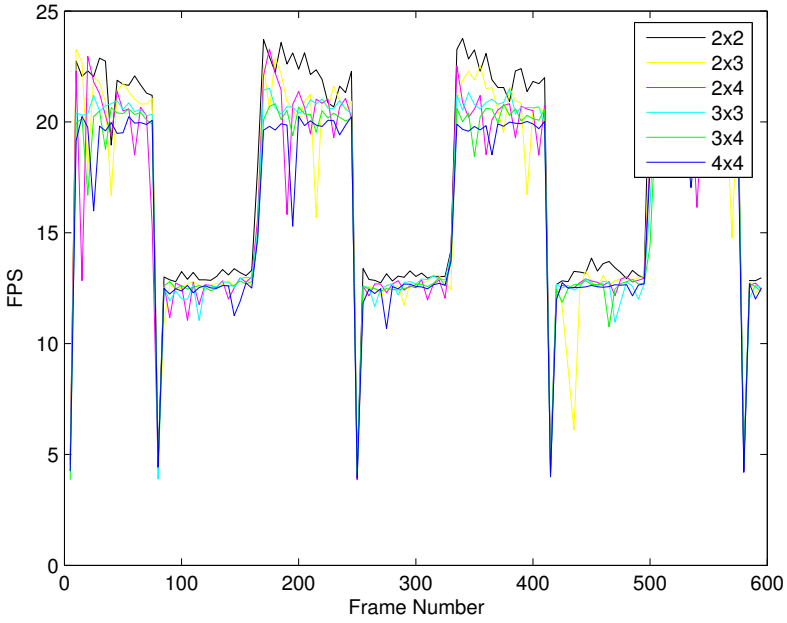
Table 4.1 Time taken for visibility culling using FH without OH (Algorithm 2) on UNC's Powerplant model with 1185 objects without a hierarchy.

algorithm (Algorithm 3) takes the least time almost throughout the walkthrough. This is followed by the *FH without OH* approach. The *OH without FH* approach performs worse than both these two.

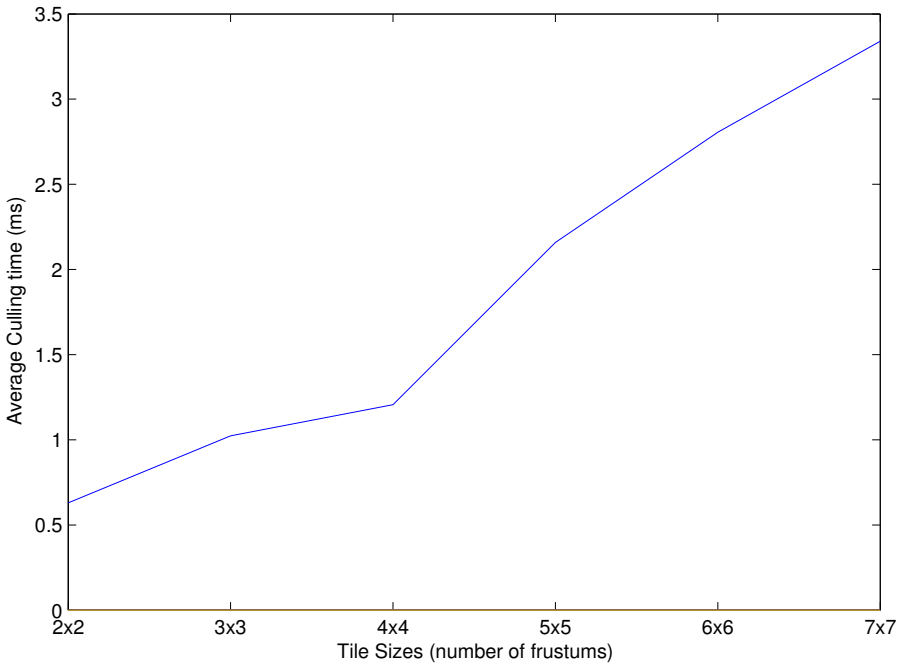
Figure 4.4(b) shows the culling time for a walkthrough on the power plant model. The plots for *OH without FH* and *FH without OH* approaches coincide, as opposed to lagging performance by *OH without FH* approach in Figure 4.4(a). This is because the high branching factor in OH makes the *OH without FH* approach more significant. However, the adaptive algorithm performs significantly better than all the other approaches.

We conducted scalability tests with respect to the tile size for different configurations of tiled displays. Figure 4.5 shows the plots for our adaptive algorithm for an 11000 frame walkthrough of the Fatehpur Sikri model. The algorithm takes about 11 ms, on an average, for culling to an 8×8 configuration, thereby making the culling applicable for setting up display walls of such configuration. Otherwise culling time limits the overall frame rate achievable on a server-managed display wall such as ours (Figure 3.4), where the rendering is done by client machines and data-transmission can be performed in parallel with the culling of the next frame.

Figure 4.6 shows the performance of our adaptive algorithm for a dynamic scene. Different percentages of the scene is changed prior to every update. Dynamic scenes have objects moving in space. The bounding boxes of these objects and their parents till the OH root need to be recomputed. Fast OBB computations (Section 4.2.1) permit dynamic scenes to be culled at interactive frame rates. Although speed can be further increased with axis-aligned bounding boxes (AABB), it comes at the cost of poor visibility culling. In an optimal situation, a hybrid of both OBBs and AABBs should be used. It is beneficial to compute AABBs for dynamic portions of the scene. Note that the percentage of dynamic objects shown in Figure 4.6 are extreme case situations. In practice, the scenes are less dynamic and so the adaptive algorithm performs even better.

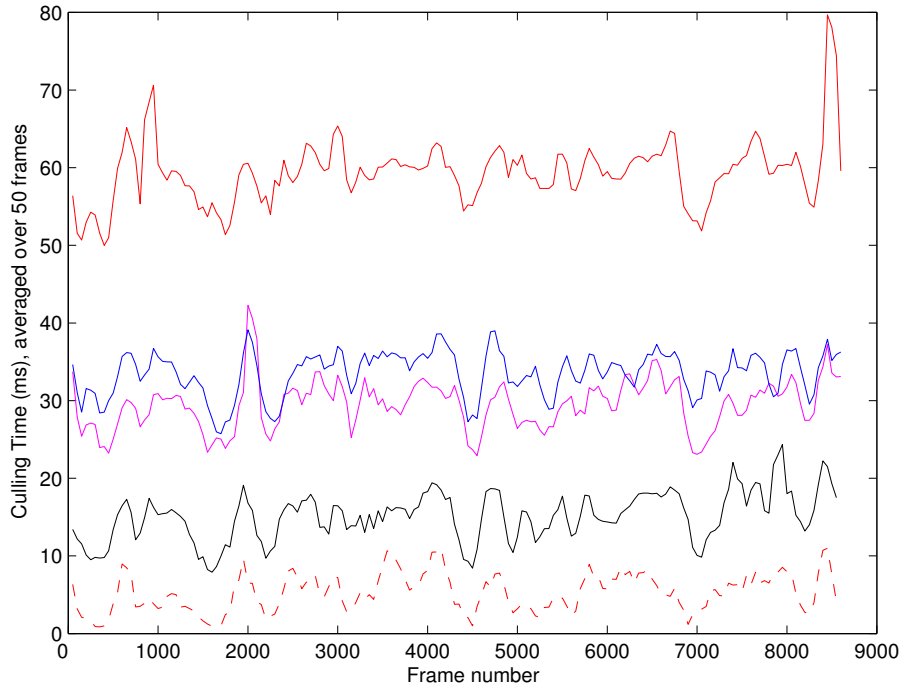


(a) Culling scalability plot for a 11000 frame walkthrough

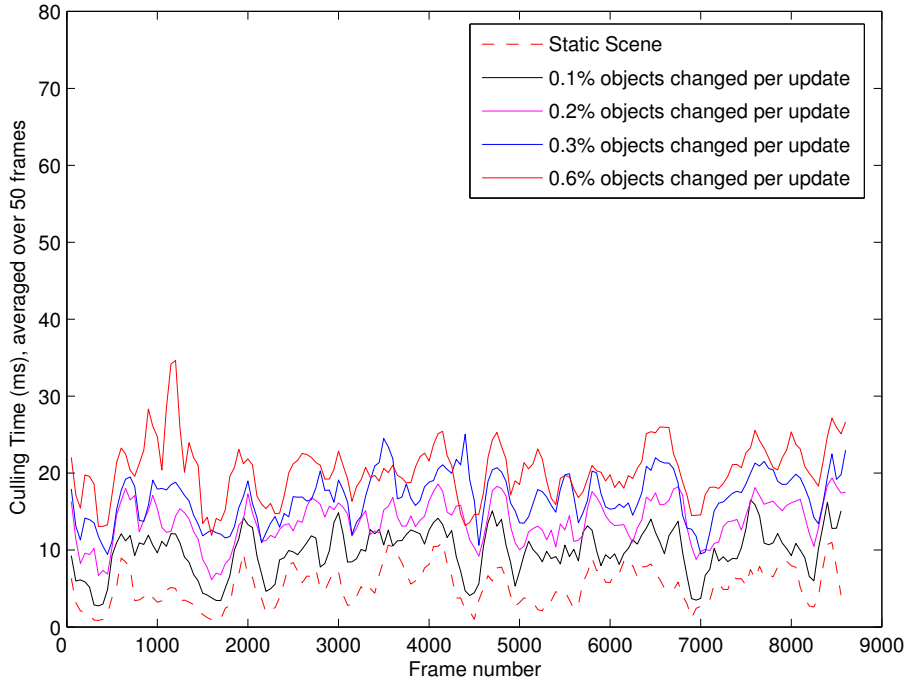


(b) Plot showing average culling time against different tile configurations

Figure 4.5 Culling scalability performance on the Fatehpur Sikri model



(a) OBBs used at dynamic nodes



(b) AABBs used at dynamic nodes

Figure 4.6 Culling performance on the power plant scene for different percentages of dynamic objects. The model has a total of 5037 objects. The performance with AABB is better than with OBB but at the cost of over-conservative visibility culling

Chapter 5

Data Transmission, Rendering and Display Synchronization

The visibility culling stage of the display wall provides the server with information about which objects are present in which tile's view-frustum and are hence visible to it. These objects need to be transmitted to them, if they are not present in their cache. This chapter discusses the issues pertaining to transmission. Later, rendering and synchronization to produce the final tiled display wall is discussed along with the issues involved.

5.1 Data Transmission

This stage involves the transmission of objects to the rendering nodes. The network transmission is very often the limiting factor in all cluster-based rendering systems. As the viewpoint changes, new objects become visible and they need to be transmitted. The objects that are visible but are not freshly visible need not be retransmitted if they are still in the clients' cache. The performance of this transmission stage is determined by the network bandwidth and latency. The amount of objects to be transmitted varies from scene to scene, being very huge for dense scenes. Dense scenes involve a lot of data transmission with each update of the viewpoint, leading to freezes. The situation is similar when the viewpoint has moved a lot since there are then a lot of new objects. This is in contrast to typical walkthrough applications in which the viewpoint changes gradually and so new objects become visible gradually rather than in large bursts.

After the visibility determination stage computes the objects that need to be transmitted to each client, a approach would unicast data to each over TCP. However, it is commonly observed that an object doesn't fall completely inside a tile, but might span across several tiles, and, in extreme cases, all the tiles (Figure 5.1). The number of objects spanning across multiple tiles is significant in typical scenes. The unicast approach certainly becomes a limiting factor even with tile-configurations as small as 2×2, since it transmits data serially.

The server maintains 3 channels for interacting with each client. The first is a control connection (control-channel) maintained over TCP. The second is UDP connection for data transmission (data-channel). The third is a TCP connection for clients to acknowledge the receipt of multicasted packets

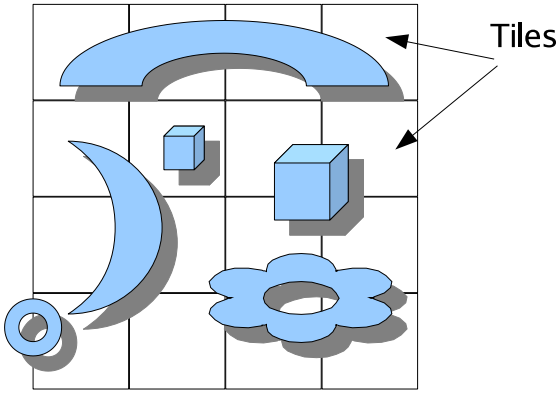


Figure 5.1 Scene objects spanning across several tiles in a 4×4 tiled-display. These objects need to be transmitted to all these tiles

(ack-channel). Even though the control-channel and the ack-channel could have been merged into one, they have been kept separate to facilitate pipelining of operations.

5.1.1 Data Header: Meta Data

To begin with, the server packs all visible objects that needs to be sent to at least one of the clients, to a transmission list. This involves serialization of nodes in the visible OSG tree of the geometry server. The OSG API provides a function which serializes an OSG tree to a file. Thereafter, header data is prepared which contains information about the count of these objects, their size and position identifier of each of these objects in the scene graph so that the client can place it at the correct location in its scene graph. Intermediate nodes corresponding to a node's parents might also need to be transmitted if they don't exist already in a client's OSG tree. In Figure 5.2, the shaded nodes are the ones that need to be transmitted but their parents will also be packed since they are required to to build an identical scene graph at the client. The header data is transmitted to each of the clients over the control-channel.

5.1.2 Multicast Data

Multicasting offloads the task of sending data to the clients to the ethernet switch, rather than being managed actively by the server as in the case of unicast. Nodes to be transmitted are packed into datagrams and assigned a sequence identifier before being multicasted over the data-channel. For purposes of efficiency, multiple small nodes might be clubbed together to form one larger datagram packet. The clients collect all objects that are destined for them and acknowledge to the server over the ack-channel. Acknowledgment is necessary because UDP being an unreliable protocol, transmission is not guaranteed. On controlled LAN environments, however, we have experimentally observed that most datagrams are received without the need for retransmission; there's no significant loss of datagrams. The clients, on receiving data, unpack the datagrams into objects and put them into the cache. This might involve evicting objects out of the cache if it tends to overflow. The client then informs the server about the

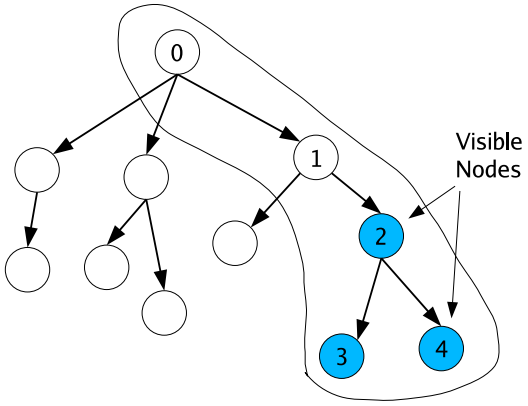


Figure 5.2 Showing header packing. The scene graph at the server is shown. The colored nodes are the ones that need to be transmitted (2, 3, 4). However, their parents (0, 1) will also be packed for transmission since they will be required by the client to form an identical scene graph

objects that it needs to evict. 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, the pause duration being scaled every time. 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 at the end of this chapter.

In a slightly modified approach, the server can compress data before transmission. Correspondingly, the clients will need to uncompress it during the unpacking step. The need for this should be experimentally justified as it might vary from situation to situation. For instance, if the server is heavily loaded while the network is relatively free, it is better to send the data without any compression.

Multicasting of objects and caching by the clients has been made possible by the use of higher-level primitives in the scene, rather than lines and points as in Chromium. Besides, multicast suitably tackles the problem of an object spanning across a number of tiles – there's virtually no overhead. The clients might be carrying out other operations in a pipelined fashion while waiting for data from the server.

5.1.3 Scene Graph Formation at Clients

As more and more nodes are transmitted from the server to the clients, the scene graph at the clients keeps getting formed. This scene graph is a subset of the one at the server. During eviction from the cache, only the geodes (terminal nodes) are thrown; the internal nodes, and therefore the skeleton, is preserved. Eventually therefore, the client's scene graph will start resembling the scene graph at the server, except that it will have fewer geodes.

This scheme of partial and on-demand formation of scene graph at the client ensures that scene's data is distributed optimally across the cluster. Therefore, no rendering node needs to keep the entire scene graph at any time. This facilitates load distribution for rendering of huge models.

5.2 Rendering and Display Synchronization

Rendering of the scene is performed by the clients. They (clients) start rendering the scene as soon as all objects, if any, for the frame are received. The client does not need to perform any culling since this has already been performed for it by the server. Thus, the server also transmits information about the object that are actually visible. The clients render only these objects from its cache. The time taken for rendering is proportional to the amount of visible geometry within the view frustum. This geometry is only a small fraction of the entire scene and varies from client to client. An object which span across several tiles are rendered by all the rendering nodes to which it is visible, even though it might be visible only partly – this is an overhead inherent of tiled-display systems.

Once the rendering nodes are done with rendering, they intimate to the server with an *"Ready-to-Swap"* message indicating their readiness to swap. When the server receives this message from all the rendering nodes, it orders a swap. Thus all displays are synchronized together. 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.

Since the server can initiate synchronization only when all the clients are done rendering, the time taken to render one frame is determined by the slowest performing client. That is, if one of the tiles has heavy geometry, other tiles will have to wait until it completes its rendering before the server can issue the go-ahead signal for swapping buffers.

5.2.1 Load Distribution

A positive aspect of using a cluster-based display wall system is to distribute load amongst the systems in the cluster. The greater the number of nodes in the cluster, the lesser the amount that each node has to render. In practice, there is some overhead involved too, but the general load distribution phenomenon speeds up the rendering performance. Due to this, even if one does not crave for high resolutions, one might use a cluster-based renderer just so as to be able to distribute the load, especially when the model size is enormous. The server does not need to perform any rendering but it should still be able to hold the model in memory for fast transmission to the clients.

5.2.2 Parallax

A display parallax artifact is observed when rendering to tiled-displays which have intervening gaps between the tiles in the rectangular grid. Straight lines do not appear straight any more due to the parallax effect, as illustrated in Figure 5.3. Fortunately, this artifact is easily cured by clipping the

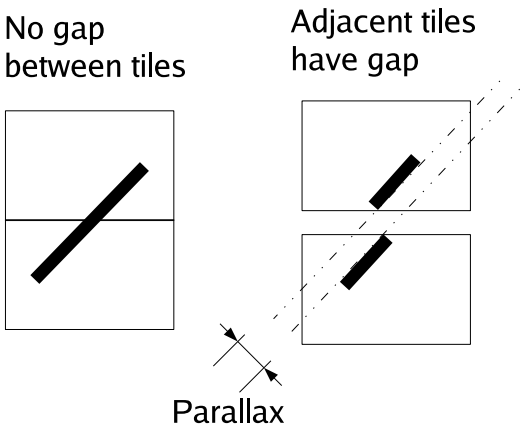


Figure 5.3 Parallax error due to gap between tiles. Straight lines don't appear straight anymore on the display

view-frustums appropriately while rendering. The amount of clipping is proportional to the gap-size. In a display, therefore, even though some objects might get clipped off, but the parallax effect is not observed. The display looks more realistic with parallax removed. Figure 3.4 shows a 4×4 tiled-display of UNC's power plant model with the parallax effect cured.

5.2.3 Display Tearing

It is noteworthy that a rendering node can physically swap buffers only at the next vertical refresh of the display even though it might have received the go-ahead from the server much earlier. Even if all the displays have identical vertical refresh frequency, they might be out of sync, which can lead to perceptible but reasonable lags between displays on tiles (referred to as tearing). In Figure 5.4, two displays have the same vertical refresh frequency (60 Hz) but they are out of sync. Even when the server orders *"Swap"* at the same time to both the displays, they end up actually refreshing at different times, leading to display tearing. Hardware solutions like Genlock [16, 60] overcome this by synchronizing the vertical refresh signal for all the displays.

5.3 Experimental Results

The server has to read and initialize large amounts of data and send some of it to the clients to start the wall. Table 5.1 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.

Caching at the clients eliminates the dependence of the performance on network bandwidth. Figure 5.5 shows the nature of the network utilization for a 600-frame walkthrough through a scene with 5.5 million vertices in 100 objects scattered around with an average object size of 2.03 MB. The initial

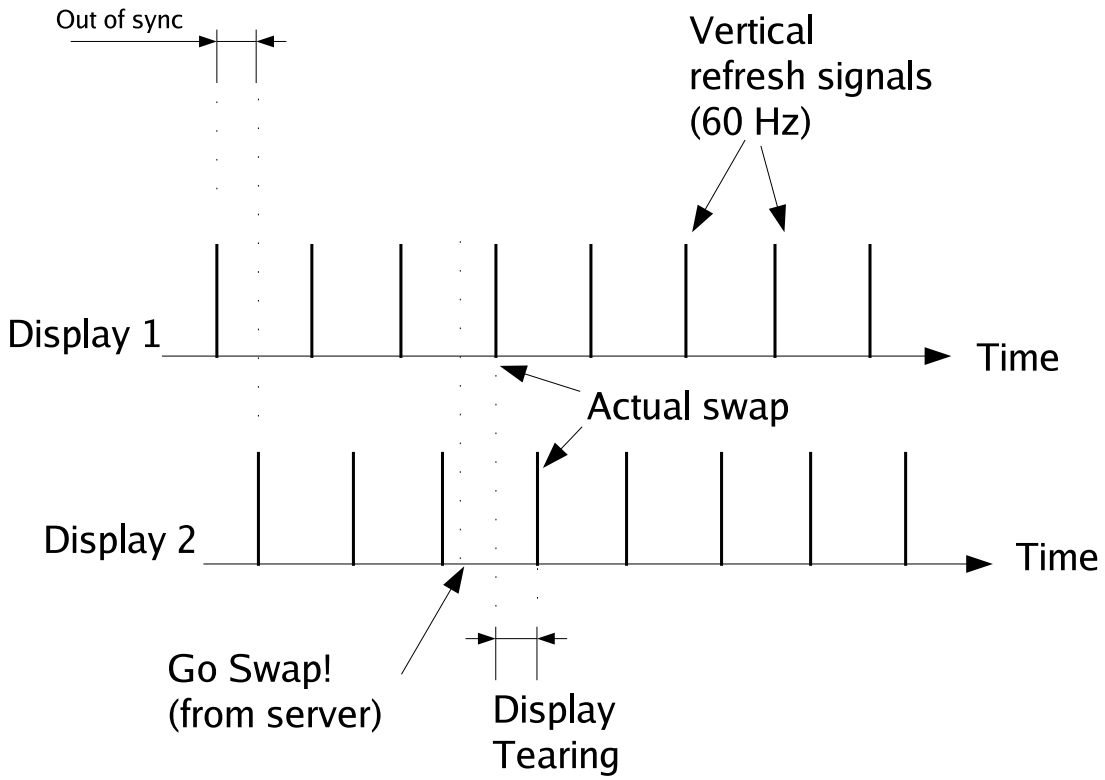


Figure 5.4 Display tearing takes place due to out of sync vertical refresh signals for the different displays

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.

In Figure 5.6, 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.

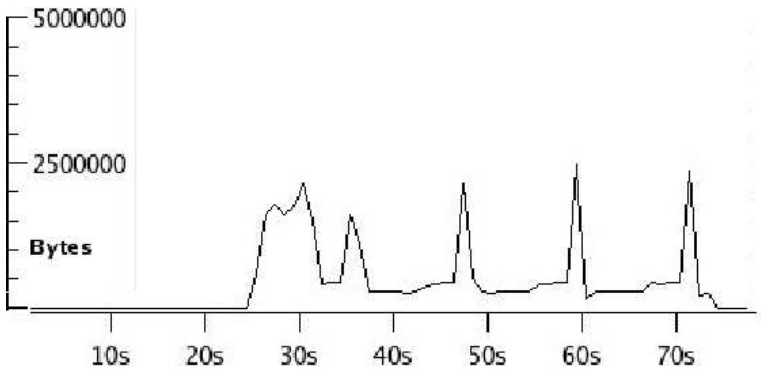


Figure 5.5 Network usage vs. time for a 600-frames walkthrough. The walkthrough is such that objects appear in multiple tiles at once. Multicast mitigates this effect. Due to caching of objects by the client, the network is used only when fresh objects are to fetched.

Tile Configuration	Time taken (seconds) with unicast	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 5.1 Time taken to start a 4×4 display wall. This involves sending of the initial data to the clients.

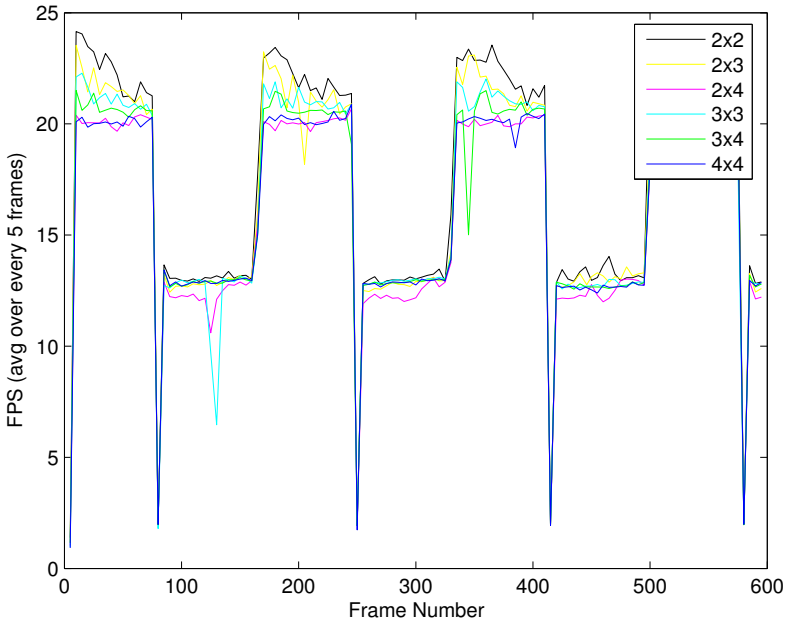


Figure 5.6 Showing system scalability with respect to the number of nodes with network bandwidth capped at 10 Mbps. Due to caching, the network requirement doesn't degrade the performance except at times of data-fetch.

Chapter 6

Transparent Rendering to a Tiled Display

A central issue concerning tiled display wall systems is their universal applicability. Does the large display application have to be written specially for a setup? This will severely restrict the applicability of such a system. Can any application developed using a standard graphics/visualization API be rendered automatically and transparently on the display wall? This will boost the utility of the display wall system greatly.

One positive aspect of Chromium is that it can transparently render any OpenGL application to a tiled display. The application need not be aware of the cluster. In fact, there's no need of modifying, compiling or even relinking the application. In a general case, Chromium is a stream processor. It intercepts the OpenGL stream containing all OpenGL primitives (vertices, lines, triangles, etc), textures, material properties, etc., and can perform operations on them. Tilesort is one stream processing unit which is used to render to a tiled display. The stream processor places OpenGL primitives into small bins for sorting to the correct tile. When the bin gets filled, the bounding box of the primitives in the bin is used to determine the tiles to which it needs to be transmitted. No higher level object structure can be derived from the primitives in the bin. Hence, tilesort needs to perform these calculations for each frame. Besides, these primitives need to be transmitted to each tile every frame, irrespective of whether or not the display has really undergone a visual change. Due to these reasons, tilesort is not only computationally expensive, it causes severe load on the network, making it an ineffective choice for rendering large scenes to a tiled display.

Since the basic philosophy of our system is to transmit and cache objects at the rendering nodes, we designed our system to automatically and transparently clusterize any application developed using a standard scene graph API.

6.1 Transparent Rendering Using GSWall

We chose the OSG API to provide transparency features are provided by our display wall system. Transparency is provided by intercepting function calls to the OSG API. Whereas Chromium intercepts all calls to the OpenGL API, GSWall intercepts only selected calls only (see Figure 6.1). The application

need not be aware that it is being rendered to a display wall. All features of our display wall as discussed in earlier chapters are inherited along with this flexibility of seamless rendering.

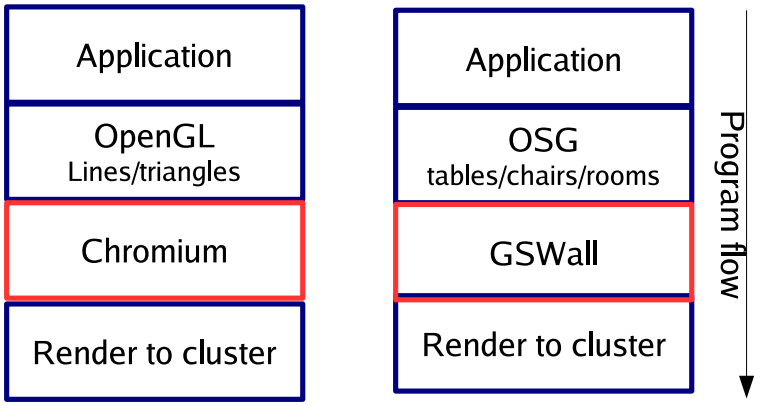


Figure 6.1 Comparison between transparent rendering using Chromium and that using our GSWall. Since Chromium works at a very low level (lines/triangles) while GSWall works with objects (tables/chairs)

Figure 6.1 shows the comparison between transparent rendering to a display wall using Chromium and that using our display wall system. Since Chromium intercepts low-level OpenGL calls, it has to deal with a huge volume of data. Moreover it cannot perform any caching. These issues are solved with GSWall since it works on higher-level objects.

Any scene graph based graphics application performs three function calls in an unending loop. These are:

- *App:* This includes the keyboard/mouse event-handling, scene graph manipulation, animations, transformations, etc. User-defined operations on scene graphs can also be carried out in this stage. As a result of this stage, several nodes in the scene graph might be marked dirty so that their bounding boxes is recalculated before being used.

- *Cull:* This stage performs visibility culling on the scene graph to prepare a smaller scene graph for rendering. Optionally, sometimes optimization and state-sorting is performed as well. The output of this stage is a scene graph (called Render Graph in OSG) which is ready for being rendered.

- *Draw:* This stage involves the actual drawing of primitives by issuing the actual OpenGL calls corresponding to rendering each node.

A scene graph API generally implements these three stages in different threads or processes with shared memory segments (for pipelining). In either case, our interception mechanism traps the calls to these three stages and embeds GSWall operations. The actual steps carried out while intercepting these functions are shown in Table 6.1. These three stages are followed by a *Swap* stage where the actually swapping of the display buffers takes place thereby putting things to the display.

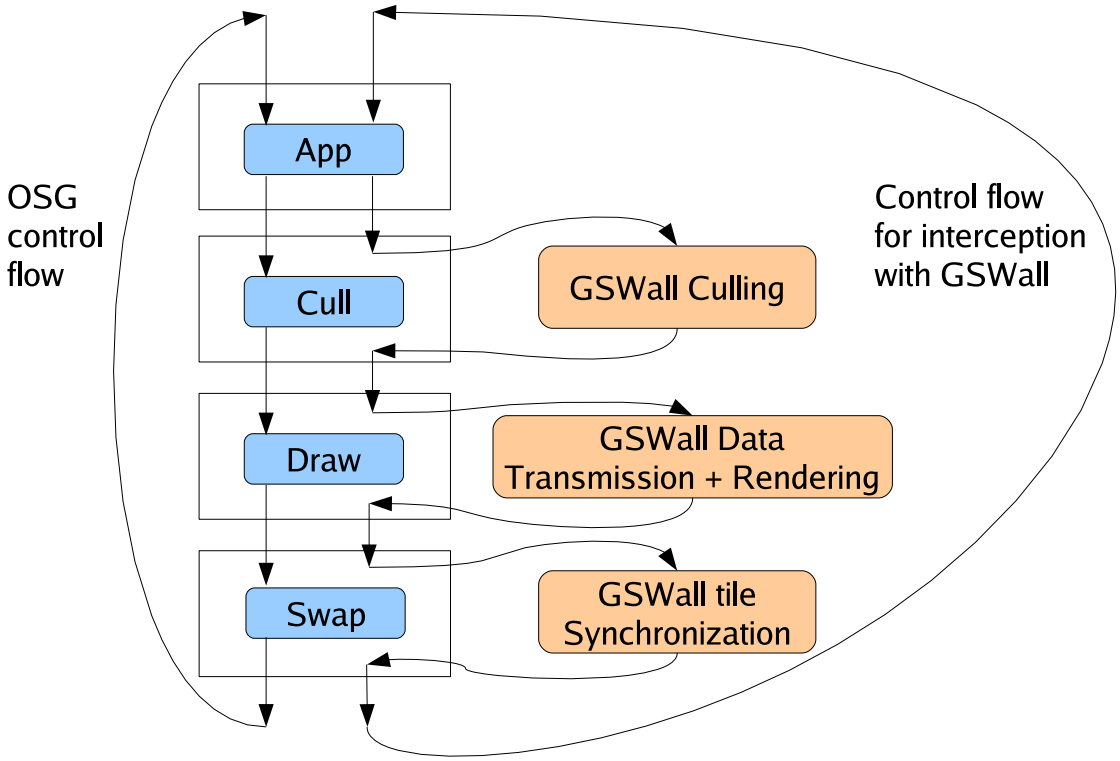


Figure 6.2 Control-flow for intercepting calls to the OpenSceneGraph API for transparent rendering to a display wall. The line-loop on the left shows the normal control-flow for an OpenSceneGraph application going through the calls to *App*, *Cull* and *Draw* followed by the call to *Swap* of display buffers. With GSWall, calls to *Cull*, *Draw* and *Swap* are intercepted to perform corresponding functions as indicated by the line-loop on the right. Note that call to OSG's *App* is not intercepted (or intercepted to perform the same operation). Also note that OSG's *Cull*, *Draw* and *Swap* are not called by the intercept procedures (shown on right). The OSG application is unaware of being rendered to a tiled display wall.

OSG function call	Actual job performed
App()	– Call original App (OSG's App())
Cull()	– Save view-point for this frame – Perform GSWall visibility culling
Draw()	– Begin data transmission in a separate thread – Clients start rendering (after data has been received)
Swap buffers	– Wait for all clients to inform *"Ready-to-Swap"* – Issue *"Swap"* to all clients

Table 6.1 OSG function calls are intercepted to embed calls to GSWall operations. The actual jobs carried out in each step is shown. Swap Buffers is not really an OSG operation but is included here for the sake of completeness.

OSG's *App* stage is either not intercepted at all or is intercepted but ends up just calling back the original function from the OSG library. OSG's call to *Cull* is intercepted to replace it with the adaptive culling algorithm, as already discussed earlier in Section 4.2.2.3. This results in the marking of the nodes as per visibility to all the tiles. The original operations in the OSG's *Cull* function is ignored. The *Draw* function call is intercepted to perform data transmission, as discussed in detail in Chapter 5. This includes sending the header to the clients followed by multicasting the packed geometry data. As soon as the clients have received all geometry, they start rendering. This stage generally is carried out in a separate thread. Again, the original operations in the OSG library are ignored. The *Swap* call is intercepted to perform display synchronization in GSWall, as discussed in Chapter 5 earlier. The server waits to until all the clients are done with rendering the scene in their view-frustums. Thereafter, the server issues a *"Swap"* order to all the clients at which time all of them swap their buffers. The operations in the original *Swap* call are bypassed. Figure 6.2 shows the control flow for transparent rendering of an OSG application. The line-loop on the left shows the control flow for a typical, non-intercepted run of the application. The *App*, *Cull*, *Draw* and *Sync* stages are called in an unending loop. The line-loop on the right shows how the interception mechanism embeds GSWall operations into the normal flow.

Although *App*, *Cull* and *Draw* functions are part of the OSG API, the *Swap* operation is not. It is provided by the windowing API (like GLUT, OSG Producer, SDL). GSWall provides interception mechanism for *Swap buffers* by intercepting corresponding calls to the windowing API.

6.2 Limitations

The transparent rendering system cannot be 100% fool-proof and hence it might not work in certain special situations. This is because our system works at OSG API level and relies on the assumption that the application to be run will not perform direct operations to the scene graph. For instance, since our GSWall stores its required data in OSG nodes itself using the `SetUserData` OSG operation, if

an OSG application uses this function, GSWall's data will be lost. Additionally, OSG provides several special-purpose nodes to specify multiple cameras, fancy animations, particle-effects, etc. Handling them transparently is not trivial. Efforts to tackle this limitation would involve tracking a lot of objects at a much lower level thereby losing the performance advantages of moving to a higher level.

Our serialization mechanism relies on an OSG function `writeNodeFile` which writes an OSG tree to a file. However, when the serialization is performed to a binary format, it embeds the texture data, material information, shaders, etc. Incremental updates to the client's scene graph can lead to multiple instances of this data being loaded rather than just a single one.

Since the function call for swapping the display buffers is strictly not part of the scene graph API but that of the windowing API (OSG Producer, GLUT, SDL, etc), the intercepting mechanism needs to a provide a collection of functions, one for each windowing API, though only one will be used in an application.

Chapter 7

Experiments and Results for the Complete System

We present several experimental results to demonstrate the various aspects of our complete display wall system. Some experiments are designed such that critical resources such as network bandwidth, computation load, etc are strained so as to better demonstrate the effectiveness of our system. Experimental results for some individual stages of the system have already been presented earlier.

7.1 Scalability

In Figure 7.1, 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.

7.2 CPU Load

Figure 7.2 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.

7.3 Cache

In another series of experiments, we perform a 4300-frames long walkthrough in a scene containing 205 objects totaling to 416MB where each object has an average size of 2MB. Figure 7.3 shows the walkthrough for various tile-configurations of the wall. The clients need to fetch a lot of data in the initial

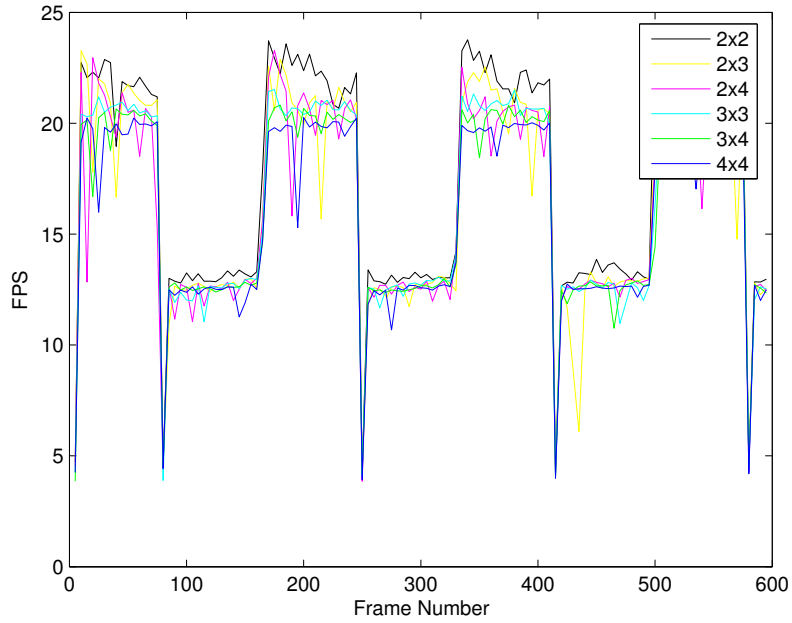


Figure 7.1 Showing system scalability with respect to the number of nodes. The scene consists of 5.5 million vertices scattered over 100 objects. Note that the performance of the system remains almost unchanged for different tile-configurations

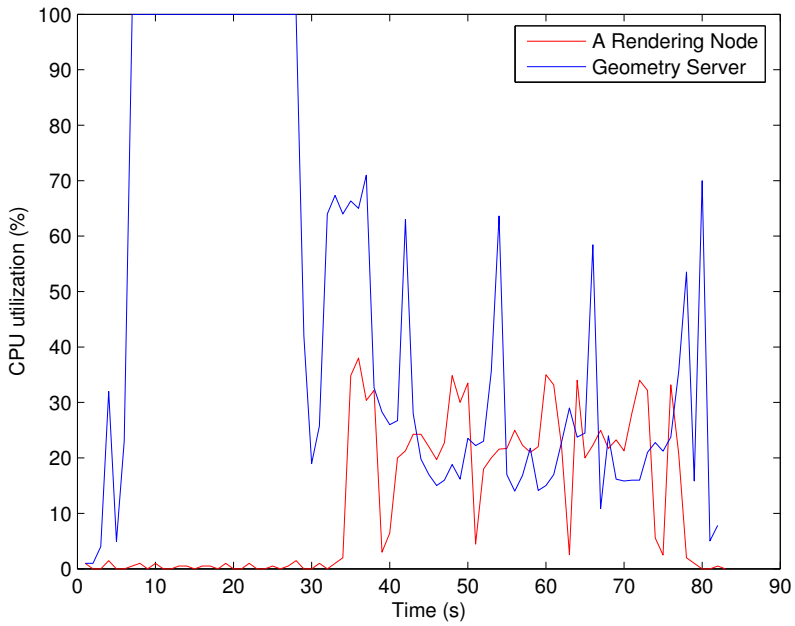


Figure 7.2 CPU Usage by the Geometry Server and a Rendering Node during a 600 frame walkthrough. The scene consists of 5.5 million vertices scattered over 100 objects

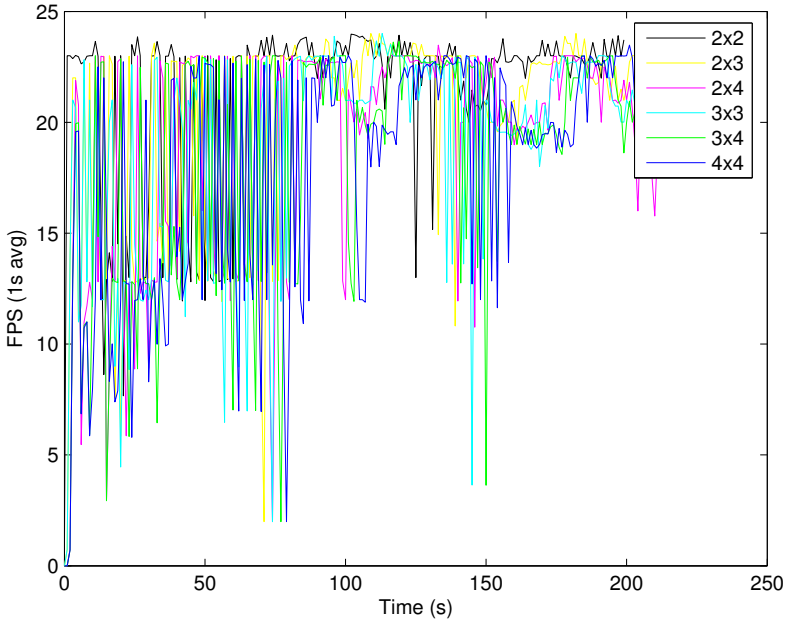


Figure 7.3 A 4300-frames walkthrough in a jungle of 416 MB scene spread over 205 objects.

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.

7.4 Optimal Cache-Size

Figure 7.4 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.

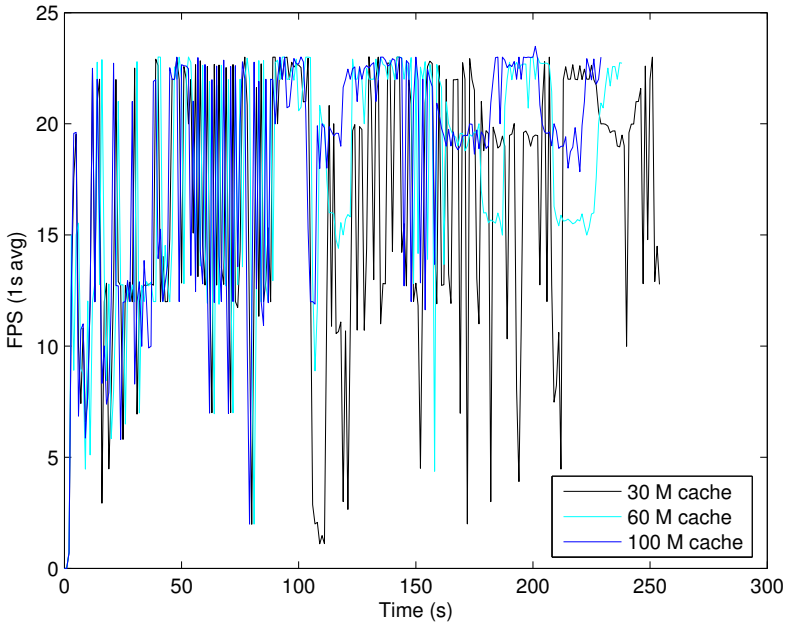


Figure 7.4 The performance of our 4×4 system with varying cache sizes in a jungle of 416 MB spread over 205 objects.

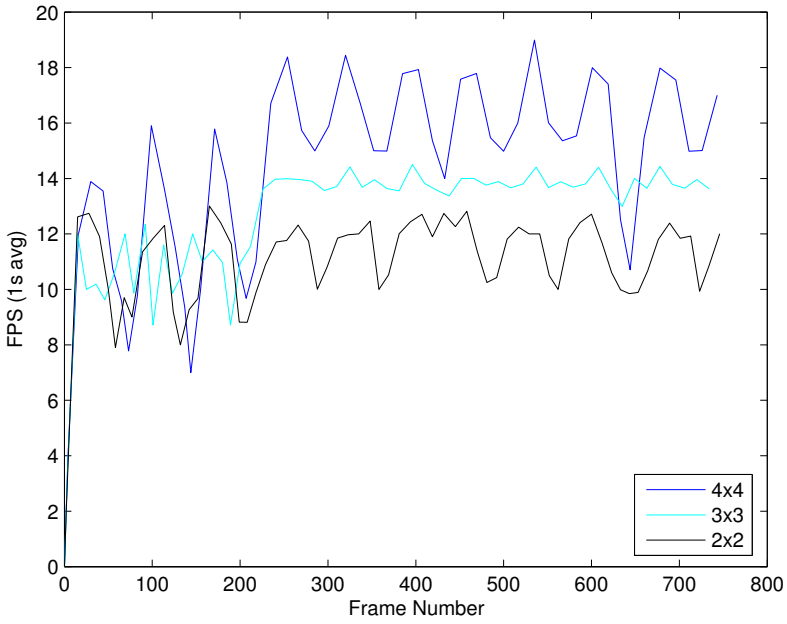


Figure 7.5 The rendering capability of our display wall increases with increase in the number of nodes in the cluster. More number of nodes distribute the load in the visible frustum.

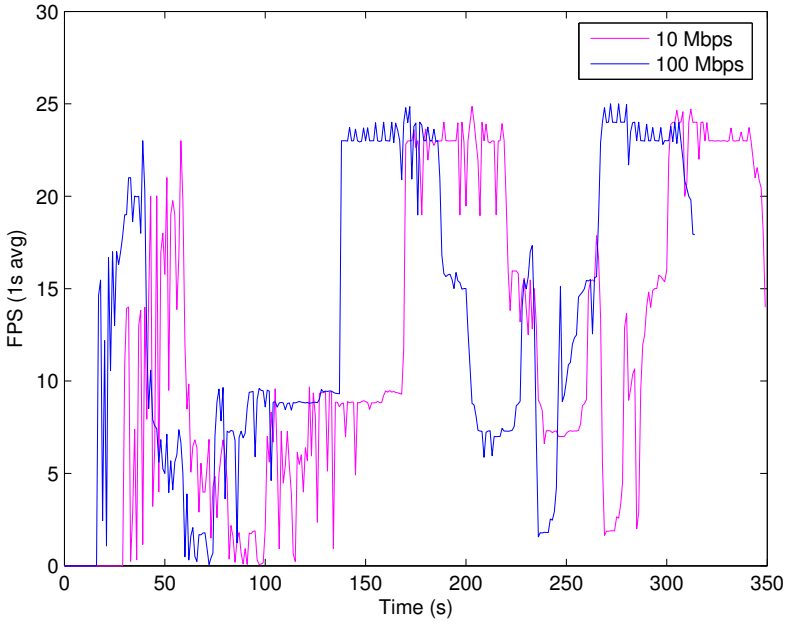


Figure 7.6 The performance of our 2×2 system with Nvidia 6600 GT on each rendering node. The network is still the bottleneck at times of huge data fetch.

7.5 Load Distribution

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 7.5, 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 $4x4$. 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.

7.6 High-Performance Rendering Nodes

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 7.6 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 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.

Characteristic	Display Wall using Chromium	GSWall
Caching?	No caching	Rendering nodes cache locally
Working	Works on low-level OpenGL primitives (lines, triangles)	Works on high-level objects (tables, chairs)
Hierarchy	No hierarchy	Both frustum and scene treated hierarchically
Network	Uses TCP unicast	Uses UDP multicast
Data Transmission	Done every frame	Done incrementally and only when necessary
Performance for powerplant model on a 2x2 wall	0.5 FPS (13 mn triangles)	23 FPS (1185 objects)

Table 7.1 Comparison between Tiled-Display Wall using Chromium and that using GSWall

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.

7.7 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. Table 7.7 summarizes the comparison between tiled-display wall using Chromium and that using GSWall.

Figure 7.7 Fatehpur Sikri on a 4×4 tiled display.

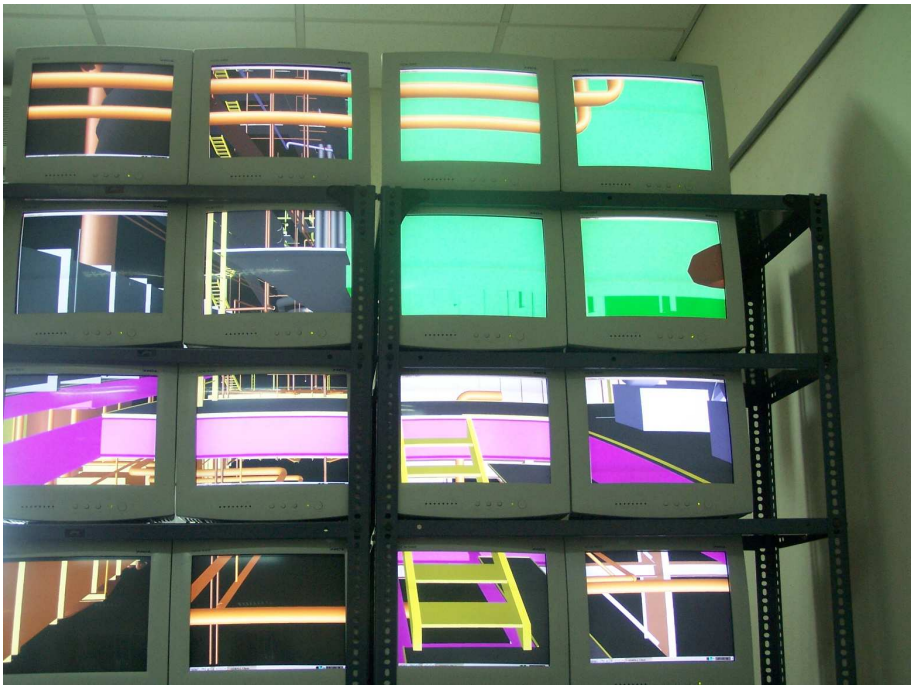


Figure 7.8 Power plant on a 4×4 tiled display wall.

Chapter 8

Conclusions and Future Work

In this thesis, we presented a geometry-managed, tiled-display system that uses commodity computers. The system uses a client-server architecture that works with modest clients. Local caching of scene's geometry (represented using OpenSceneGraph) and the multicast mode of transmission keeps the network requirements moderate and provide excellent scalability compared to prior work. For the first time, results for display walls using rendering nodes with moderate rendering power is shown. Our system scales well to perform rendering for massive models such as UNC's power plant model as well.

We also presented a conservative, from-point visibility culling approach for culling a large scene to a hierarchy of view-frustums using a novel adaptive algorithm which determines the optimal path, merging object and frustum hierarchies. The algorithm performs logarithmically in practice. Due to this, it can scale well for large number of frustums which is critical for the application scenario of a tiled display wall. The performance gain by our algorithm is shown on several walkthrough experiments. The algorithm makes culling for very large tile display setups feasible. Huge models can be handled at interactive frame rates. We also showed that the adaptive algorithm is applicable for dynamic scenes as well.

Though we showed the performance of our adaptive visibility algorithm on a two-dimensional, tight-fit array of frustums, the results can be extended to other hierarchies of frustums. We are currently extending it to other typical situations like a 2D array of frustums with small overlap used in multi-projector displays. We are also working on culling to general frustum hierarchies needed for applications like shadow volume computations. A BSP-tree like partitioning of the frustums, very similar to our current approach, will be needed in such cases.

We showed a thorough analysis and experimental results of the underlying system as well as the system as a whole. The experimental results assert the scalability of our system and indicate that exploiting high-level scene structures is the way to setup extremely large but cost-effective display walls, as is used in our system, aiming towards giga pixels.

A mechanism to automatically and transparently render any OpenSceneGraph application without modifying the source code or even recompiling it, provides huge applicability advantages for our work. For the first time, a mechanism for interactively rendering huge scenes to display walls is possible.

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.

Bibliography

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

[2] Gears High Resolution Tiled Display Wall
http://gears.aset.psu.edu/viz/facilities/displaywall.

[3] GigaPixel Project, Virginia Tech
http://infovis.cs.vt.edu/gigapixel/index.html.

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

[5] LionEyes Display Wall. Penn State University.
http://viz.aset.psu.edu/ga5in/DisplayWall.html.

[6] Matrox Advanced Synchronization Module.
http://www.matrox.com/mga/products/asm/home.cfm.

[7] NCSA Pixel Blaster.
http://brighton.ncsa.uiuc.edu/~prajlich/wall/ppb.html.

[8] NVIDIA Scene Graph Software Development Kit.
http://developer.nvidia.com/object/nvsg_home.html.

[9] Open Inventor.
http://oss.sgi.com/projects/inventor.

[10] OSG: OpenSceneGraph.
http://www.openscenegraph.org.

[11] PowerWall. University of Minnesota.
http://www.lcse.umn.edu/research/powerwall/powerwall.html.

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

[13] John M. Airey, John H. Rohlf, and Jr. Frederick P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D graphics*, 1990.

[14] John Milligan Airey. *Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations*. PhD thesis, 1990. Director-Frederick P. Brooks, Jr.

[15] Daniel G. Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Kenneth E. Hoff III, Tom Hudson, Wolfgang Stürzlinger, Rui Bastos, Mary C. Whitton, Frederick P. Brooks Jr., and Dinesh Manocha. MMR: an interactive massive model rendering system using geometric and image-based acceleration. In *Symposium on Interactive 3D Graphics*, 1999.

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

[17] Jérémie Allard, Valérie Gouranton, Loïck Lecointre, Emmanuel Melin, and Bruno Raffin. Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In *IEEE Virtual Reality Conference*, pages 273–274, 2002.

[18] Ulf Assarsson and Tomas Möller. Optimized View Frustum Culling Algorithms. *Technical Report 99–3, Department of Computer Engineering, Chalmers University of Technology*, 1999.

[19] Ulf Assarsson and Tomas Möller. Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools: JGT*, 5(1):9–22, 2000.

[20] 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.

[21] Patrick Baudisch, Nathaniel Good, Victoria Bellotti, and Pamela Schraedley. Keeping things in context: a comparative evaluation of focus plus context screens, overviews, and zooming. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 259–266, New York, NY, USA, 2002. ACM Press.

[22] Benjamin B. Bederson, James D. Hollan, Ken Perlin, Jonathan Meyer, David Bacon, and George W. Furnas. Pad++: A zoomable graphical sketchpad for exploring alternate interface physics. *Journal of Visual Languages and Computing*, 7(1):3–32, 1996.

[23] Allen Bierbaum, Christopher Just, Patrick Hartling, Kevin Meinert, Albert Baker, and Carolina Cruz-Neira. VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *VR*, pages 89–96, 2001.

[24] Jirí Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Comput. Graph. Forum*, 23(3):615–624, 2004.

[25] Thomas C. Bressoud and Bruce 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.

[26] Michael S Brown and Aditi Majumder. SIGGRAPH 2003 Course Notes: Large-Scale Displays for the Masses, 2003.

[27] Don Burns and Robert Osfield. Open Scene Graph A: Introduction, B: Examples and Applications. In *VR*, page 265, 2004.

[28] Han Chen, Douglas W. Clark, Zhiyan Liu, Grant Wallace, Kai Li, and Yuqun Chen. Software environments for cluster-based display systems. In *CCGRID*, 2001.

[29] James H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[30] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Trans. Vis. Comput. Graph.*, 9(3):412–431, 2003.

[31] Wagner Toledo Corrêa, James T. Klosowski, and Cláudio T. Silva. Visibility-Based Prefetching for Interactive Out-Of-Core Rendering. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 2003.

[32] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality: the design and implementation of the CAVE. In *SIGGRAPH*, 1993.

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

[34] Frédo Durand. *3D Visibility: analytical study and applications*. PhD thesis, Université Joseph Fourier, Grenoble I, July 1999.

[35] Thomas A. Funkhouser. Database Management for Interactive Display of Large Architectural Models. In *Graphics Interface*, 1996.

[36] Thomas A. Funkhouser, Carlo H. Séquin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Symposium on Interactive 3D Graphics*, 1992.

[37] G. W. Furnas. Generalized fisheye views. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 16–23, New York, NY, USA, 1986. ACM Press.

[38] Jinzhu Gao, Jian Huang, C. Ryan Johnson, Scott Atchley, and James Arthur Kohl. Distributed Data Management for Large Volume Visualization. In *IEEE Visualization*, page 24, 2005.

[39] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In *Symposium on Computational geometry*, 1997.

[40] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02*, pages 693–702. ACM Press, 2002.

[41] James T. Klosowski and Cláudio T. Silva. The Prioritized-Layered Projection Algorithm for Visible Set Estimation. *IEEE Trans. Vis. Comput. Graph.*, 6(2):108–123, 2000.

[42] Kai Li, Matthew Hibbs, Grant Wallace, and Olga G. Troyanskaya. Dynamic Scalable Visualization for Collaborative Scientific Applications. In *IPDPS*, 2005.

[43] A. Majumder. Properties of Color Variation across Multi-Projector Displays. Presented at Eurodisplay, Nice, France, 2002.

[44] A. Majumder and R. Stevens. Color nonuniformity in projection based displays: Analysis and solutions. 2003.

[45] Nirnimesh and P. J. Narayanan. Scalable, Tiled Display Wall for Graphics using a Coordinated Cluster of PCs. In *Pacific Graphics (to appear)*, 2006.

[46] Catherine Plaisant, David Carr, and Ben Shneiderman. Image-browser taxonomy and guidelines for designers. *IEEE Softw.*, 12(2):21–32, 1995.

[47] Ramesh Raskar, Michael S. Brown, Ruigang Yang, Wei-Chao Chen, Greg Welch, Herman Towles, W. Brent Seales, and Henry Fuchs. Multi-projector displays using camera-based registration. In *IEEE Visualization*, pages 161–168, 1999.

[48] John Rohlf and James Helman. Iris performer: a high performance multiprocessing toolkit for real-time 3D graphics. In *SIGGRAPH*, 1994.

[49] Rudrajit Samanta, Thomas Funkhouser, and Kai Li. Parallel rendering with k-way replication. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 75–84, 2001.

[50] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal 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.

[51] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load balancing for multi-projector rendering systems. In *ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 107–116. ACM Press, 1999.

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

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

[54] Mel Slater and Yiorgos Chrysanthou. View volume culling using a probabilistic caching scheme. In *Virtual Reality Software and Technology*, 1997.

[55] Gordon Stoll, Matthew Eldridge, Dan Patterson, Art Webb, Steven Berman, Richard Levy, Chris Caywood, Milton Taveira, Stephen Hunt, and Pat Hanrahan. Lightning-2: a high-performance display subsystem for PC clusters. In *SIGGRAPH*, 2001.

[56] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, 1974.

[57] Seth J. Teller and Carlo H. Sequin. Visibility preprocessing for interactive walkthroughs. In *SIGGRAPH*, 1991.

[58] Gerrit Voss, Johannes Behr, Dirk Reiners, and Marcus Roth. A multi-thread safe foundation for scene graphs and its extension to clusters. In *EGPGV*, pages 33–37, 2002.

[59] Lujin Wang, Ye Zhao, Klaus Mueller, and Arie E. Kaufman. The Magic Volume Lens: An Interactive Focus+Context Technique for Volume Rendering. In *IEEE Visualization*, page 47, 2005.

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

[61] Hansong Zhang, Dinesh Manocha, Tom Hudson, and III Kenneth E. Hoff. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH*, 1997.