Garuda: A Scalable, Tiled Display Wall Using Commodity PCs

Nirnimesh Pawan Harish P. J. Narayanan

Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, 500032 INDIA
{nirnimesh@research., harishpk@research., pjn@}iiit.ac.in

ABSTRACT

Cluster-based tiled display walls can provide cost-effective and scalable displays with high resolution and large display area. Software to drive them needs to scale too if arbitrarily large displays are to be built. Chromium is a popular software API used to construct such displays. Chromium transparently renders any OpenGL application to a tiled display by partitioning and sending individual OpenGL primitives to each client per frame. Visualization applications often deal with massive geometric data with millions of primitives. Transmitting them every frame results in huge network requirements that adversely affects the scalability of the system. In this paper, we present Garuda, a client-server based display wall framework that uses off-the-shelf hardware and a standard network. Garuda is scalable to large tile configurations and massive environments. It can transparently render any application built using the Open Scene Graph (OSG) API to a tiled display without any modification by the user. The Garuda server uses an object-based scene structure represented using a scene graph. The server determines the objects visible to each display-tile using a novel adaptive algorithm which culls the scenegraph to a hierarchy of frustums. Required parts of the scenegraph are transmitted to the clients, which cache them to exploit the inter-frame redundancy. A multicast-based protocol is used to transmit the geometry to exploit the spatial redundancy present in tiled display systems. A geometry push philosophy from the server helps keep the clients in sync with one another. Neither the server nor a client needs to render the entire scene, making the system suitable for interactive rendering of massive models. Transparent rendering is achieved by intercepting the cull, draw, and swap functions of OSG and replacing them with our own. We demonstrate the performance and scalability of the Garuda system for different configurations of display wall. We also show that the server and network loads grow sub-linearly with increase in number of tiles which makes our scheme suitable to construct very large displays.

Keywords: Parallel Visualization and Graphics Clusters, Visualization over Networks, Large scale displays.

1 INTRODUCTION

The size of a display in terms of the dimensions of the surface and the number of pixels are important for visualization applications. The available computing power has been following Moore's law over the past few decades. However, display resolutions have only grown modestly over the same. Public displays need large area for wider viewing but do not require high resolutions. A large screen lit by a projector serves such applications. Many visualization applications, however, require large display areas and high display resolutions simultaneously to provide both detail and context. Virtual Reality environments, scientific visualization, etc., are examples where large display size with high resolutions can be useful. Such displays can be built using specialized hardware or using a cluster of computers.

Hardware to drive large displays and display mechanisms like CRT and LCD are severely limited in the maximum achievable resolution. Costs increase prohibitively beyond a few mega pixels. Tiling multiple displays is an effective way to create arbitrarily large displays for visualization applications. 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 displays.

There is a trade off between resolution and display size in computer displays. The resolution of the display affects the visible detail and the size affects the visual context. Zooming in to view the fine details results in a loss of the bigger picture. Zooming out bring the big picture at the expense of the details. Baudisch et al. attempt a creative solution by embedding the high-resolution portions on a screen while displaying low resolutions for the rest in an effort to achieve focus and context simultaneously [12]. Wang et al. proposed a focus-plus-context framework to magnify the features of interest [42]. These methods, however, assume that the viewer concentrates on a small region of the screen; the focus doesn't extend to the entire screen. It is observed that in a large display environment people tend to move about and change viewpoints, and therefore might be distracting, especially at the edges of the resolution-change [15]. Displays with large size and high resolution solve the trade-off between focus and context. A few efforts are underway to build very large displays. Displays with 100-200 mega pixels have been built by multiple teams [2, 37, 22, 3]. These are mostly used to display static or slow-changing images. Driving such displays directly from a graphics program will facilitate their use in interactive visualization and virtual reality applications.

In this paper, we present the design of Garuda, a cluster-based tiled display wall for interactive graphics applications. Garuda system uses commodity PC hardware on Ethernet for driving individual tiles with a high-end server to coordinate them. We currently use low-end PCs on 100Mbps network to drive the tiles. Garuda's software system is built on the Open Scene Graph API [7]. Any application built on OSG can be rendered transparently on the display wall without any modification. Garuda caches and manages the transmitted geometry at the rendering clients to exploit temporal and spatial coherence in the scene. This sets it apart from solutions like Chromium [25] whose demands on the network are very high. Wallace et al. describe the design issues behind the Princeton display wall from the point of view of scalability to videos, images, graphics and audio [41]. A recent survey on large, high-resolution displays cite scalability to large tiles, high-performance rendering, and integration into a computing environment as among the top ten research challenges faced today [29]. The design of Garuda has addressed these issues directly.

The main contributions of the Garuda system for tiled-displays are two: (a) applicability to a large scenarios and (b) scalability of the hardware and architecture to very large displays. For wide applicability, we provide the ability to render any Open Scene Graph application to the display wall without modifications. Scalability in

hardware is achieved by the use of low-end PCs on commodity Ethernet as the rendering nodes, keeping costs low. Scalability in the architecture has multiple components. An adaptive algorithm at the server culls the object hierarchy to the frustum hierarchy optimally to determine the objects inside each tile's frustum. The caching of the partial scenegraph at each node facilitates the transmission of only the new objects in each frame, keeping the network requirements low. As the number of tiles increases, caching becomes more effective as fewer objects are visible to each tile. Lastly, the objects are sent to the rendering nodes using a multicast protocol to further reduce the network load. Since the server is on one end of all network traffic, multicasting can send an object to all clients in about the same time needed to send to one client. As the number of tiles increases, more objects will be visible to multiple tiles, bringing greater gains using a multicast protocol. Garuda system exploits the power of distributed rendering. No node in the system including the server needs to render the entire scene. This facilitates the rendering of very large scenes in a distributed manner. The server deals only with the bounding volumes and each client only renders the sub-frustum corresponding to its tile. As a result, a 2×2 system could render the full Powerplant model at about 50 fps and above at a resolution of 2048×1536. A single node of the same capacity achieved only 10-12 fps rendering a 1024×768 image. A 4×4 system using low-end machines with ATI Xpress 200 motherboard graphics achieved 10 fps on the same model, with a display resolution of 4096×3072.

This paper is organized as follows. The related literature is reviewed in Section 2. The design of Garuda system is given in Section 3. Transparent rendering for OSG applications appears in Section 4 and experimental results from it in Section 5. Discussions and conclusions are presented in Section 6.

2 Related Work

We present a review of the literature related to the construction of large displays in this section. See the recent survey on the hardware, software and human-factors aspects of large, high-resolution displays [29] to get a comprehensive review of the options for building such displays.

2.1 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 display unit, are often used for creating large displays. Implementations of such Power Walls exist at several places [8]. CAVE^{TM} can also be classified as such a system though it focuses on surround display [19]. Sepia by Heirich et al. , also falls under this category as it is a hardware approach to image combining; using additional hardware for image combining over a specialized high speed, high bandwidth network [27]. Metabuffer by Bajaj et al. is a software cum hardware approach to image combining using a special framebuffer for color combining and blending to a tiled display [45]. Though these systems achieve interactive framrates, they are expensive, have poor scalability to large sizes, and require expert-level maintenance and setup.

2.2 Cluster-based Displays

Cluster-based solutions for creating large displays have gained a lot of interest recently [16, 25]. Such displays have the potential to put high-performance visualization within the reach of more users. These systems consists of a number of commodity PCs that are interconnected over a LAN or via low-latency networks like the Myrinet [8, 25, 9]. Cluster-based displays are economical, scalable in performance and resolution, and are easy to maintain. The computing resources of a cluster can be used for running distributed tasks or doing parallel computations. Upgrading a cluster based display wall is as easy as upgrading the individual nodes in the cluster. Cluster-based displays have to address the concerns of color-balancing across displays. Recent advances in color-balance and seamless tiling technologies address this issue [15, 26]. Chen et al. classify cluster-based display setups into two approaches: *master-slave* and *client-server* [16]. We describe each briefly now.

2.2.1 Master-Slave Approach

In the master-slave setup, the dataset is mirrored across all the nodes and multiple instances of a program run in parallel, one on each node. Their running is synchronized such that they assume identical behaviors at synchronization boundaries. Each node renders the entire scene but displays only a certain portion of it. The master-slave model can be sub-classified as *System-level program synchronized (SSE)* or *Application-level program synchronized (APE)* depending on the level at which the synchronization is carried out [16]. SSE attempts to synchronize transparently, i.e., without requiring modification or even relinking of the source and has been studied in the context of fault-tolerant computing. Bressoud et al. proposed Hypervisor, that treats an actual software system as running on a virtual machine, close to the actual microprocessor architecture [14]. Due to the low level of synchronization involved, the program slows down considerably and SSE mechanism is not guaranteed to work with arbitrary multi-threaded applications. With APE, the responsibility of synchronizing lies with the application. This approach has the lowest network bandwidth requirement. However, since each node runs an instance of the application, there's no performance gain with the cluster setup as compared to the performance without it; there is only a gain in the display resolution. VR Juggler, a framework for virtual reality applications, falls under this category [13]. Net Juggler is an open source library that turns a commodity cluster running the VR Juggler into a single image cluster [11].

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. Little performance gain is achieved from using a cluster setup. It is also difficult to handle dynamic environments since the data is replicated. Even if the data source is centralized, it is difficult to access real time data stream from a single external network source.

2.2.2 Client-Server Approach

The client-server models store the dataset on a central server. The server can also use a distributed data management framework, as in [23]. The server distributes appropriate data to each client node and performs the synchronization among the rendering nodes. The data distribution can follow the sort-first or the sort-last strategies or a hybrid k-ary distribution strategy [34, 33]. One way to distribute the data transparently is to intercept function calls at the graphics API level [25] or at the display manager level [1]. The former provides the large display facility to any application using the API. The latter can additionally provide all windowing applications to the large displays including menus, toolbars and decorations. This approach is close to image level splitting and may not exploit the rendering capabilities of the clients. Chromium intercepts and distributes OpenGL graphics primitives to a cluster of PCs for rendering [25]. It is used by display walls such as the Hyperwall [35], VisWall [9], LionEyes Display Wall [4], etc. The plus point of Chromium is that it can clusterize any application built over OpenGL transparently. However, it fails to capture the coherence of data across time 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 object structure encoded in scene graphs due to its low-level focus.

Another approach to the client-server architecture is to carry out the distribution at the 3D object level itself rather than at the primitives level. Virtual Graphics Platform (VGP), similar to Chromium, can handle 3D objects transparently for OpenGL based applications but is still high on network requirements. ModViz Renderizer software, based on VGP, can render any OpenGL Performer application to a tiled display with some code modification [6]. Though these systems render 3D data they do not exploit the spatial arrangement of objects in the scene. Spatial structure of a scene is captured well using a scenegraph representation. The server can determine the objects that need to be sent to each client using frustum culling. This helps reduce the network requirements significantly. This is the approach followed by Syzygy [36] and OpenSG [40] for display wall rendering. The Syzygy software library consists of tools for programming VR applications on PC clusters [36]. It includes two application frameworks: a distributed scenegraph framework for rendering a single application's graphics database on multiple rendering clients, and a master/slave framework for applications with multiple synchronized instances. Germans et al. presented a software environment Aura which has a multi-copy mode that uses a master-slave approach and a broadcast mode that uses a client-server approach [24, 39]. Corrêa et al. described iWalk, an out-of-core rendering scheme using visibility-based prefetching of data in massive models [18]. They also described their system for parallel rendering to tiled displays where the rendering nodes follow a pull-philosophy, which could be a limitation when dynamic objects are involved [17]. Blue-C Distributed scenegraph by Naef et al. uses a scenegraph node distribution approach, using OpenGL Performer, to distribute the scene across multiple clients with some user interaction [28]. The Garuda system follows a server-push philosophy which allows us to exploit temporal and spatial coherence during computations. The server-push philosophy also helps in handling dynamic objects in a scene. Garuda distributes the scenegraph nodes across multiple clients automatically in a manner such that all nodes common to all the clients are multicast only once and existing nodes at clients are not sent again, keeping the network load minimal.

2.3 Display Synchronization

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 [38], WinSGL [43] and Matrox's ASM [5] or software/hardware solutions like SoftGenLock [10] 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.

2.4 Multi-projector Displays

There has been a lot of interest in multi-projector displays recently. The emphasis has been on systems that can auto-configure themselves even when the projectors are placed together casually. Raskar et al. described a camera-based scheme for easily configuring multi-projector displays [31]. Yang et al. use a camera for closed-loop calibration of controllable projectors to automatically calibrate multiple screen configurations [44]. Majumder et al. presented a method to achieve luminance matching across all pixels of a multi-projector display that results in photometrically uniform displays [26].

In this paper, we concentrate on the culling and management of geometric representation for transparent rendering to a clustered, tiled display system, using CRT-based tiled displays. The issues of display alignment, color correction, etc, will be relevant if the tiled display is made using projectors. We do not concentrate on these aspects currently and restrict our attention to the algorithmic aspects of tiled displays.

2.5 Geometry Server

Garuda system grew from an earlier work from our group on building a Geometry Server for heterogeneous clients. We built a high-performance, centralized storehouse for massive geometric data that serves geometry to each client adaptively [21]. The Geometry Server stores and manages the scene's representation using multiple levels of detail (LoD). It can simultaneously serve a number of heterogeneous users adaptively, ranging from a graphics workstation on the LAN to a PDA connected over a wireless network. Each user gets a visibility-limited representation of the model at an LoD compatible with its rendering capabilities, computational resources, and network characteristics. The objective is to provide consistent, interactive frame rates to every user irrespective of the users capability and connectivity. The remotely served geometry appears as just another object in the client's environment and can be combined with other objects or modified like a local model. Dynamic objects are handled using a server-push for information and lazy-download for the geometry data. The server streaming philosophy was further extended for rendering of massive terrains in real-time [20]. The Garuda display wall system extends this philosophy for high-performance rendering to a cluster-based tiled display system. The Geometry Server treats each client as an independent user with independent viewpoint control, with very little coordination between them. The emphasis there is on the adaptation of the service to the network and computation capabilities of each client for an acceptable quality of service. Garuda, on the other hand, has a single user who controls the viewpoint. The emphasis is on the use of the tiled clients as a high-resolution display without the users even realizing its tiled nature. Coordinated culling, transmission, and rendering are the focus of the server in Garuda. The rendering nodes are uniform and connected using a reliable commodity network.

3 GARUDA: A GEOMETRY-MANAGED DISPLAY WALL

Two considerations guided the design of the Garuda system: scalability to large displays and usability to a number of applications. The system follows a client-server approach (see Figure 1), but additionally exploits the temporal and spatial coherence in the scene structure to minimize the network resource utilization. The Garuda system consists of a server that coordinates all activities and several rendering clients, currently one for each tile, that perform the rendering[1]. The server is a high-end machine while the clients are built using commodity computers. We currently use low-end PCs in our experiments; clients with better graphics and memory resources can directly enhance the performance of the overall system. No client nor the server needs to render the whole scene in our design. The client nodes and the server are connected using a standard 10/100 Mbps Ethernet.

The user application, built on an Open Scene Graph API, runs on the server machine. The interception mechanism takes the viewpoint from the running application at every frame in response to keyboard or mouse input. The server has a reference pointer to the entire scene loaded into memory by the running application as a scenegraph and determines the objects that are visible to each tile using an adaptive view frustum culling algorithm described later.

[1]The tiled-rendering literature refers to the rendering nodes as tile-servers and the node where the application runs as the tiling client [25, 39]. We use a traditional interpretation of server and client in this work and have multiple rendering clients and one tile server that coordinates them.

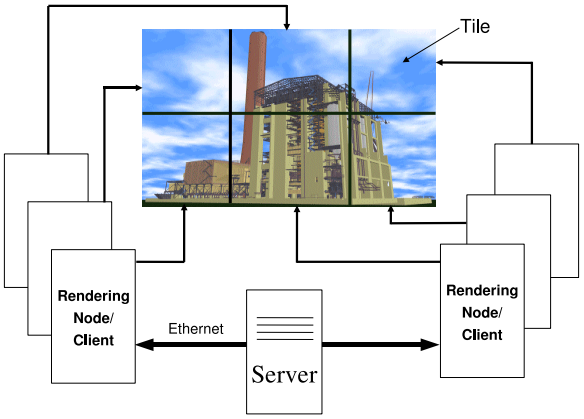

Figure 1: Schematic of the Garuda hardware system. The server runs the user application and controls the rendering and synchronous display of the rendering nodes of the clients, each of which draws a single tile. The server is connected to the clients over commodity ethernet network.

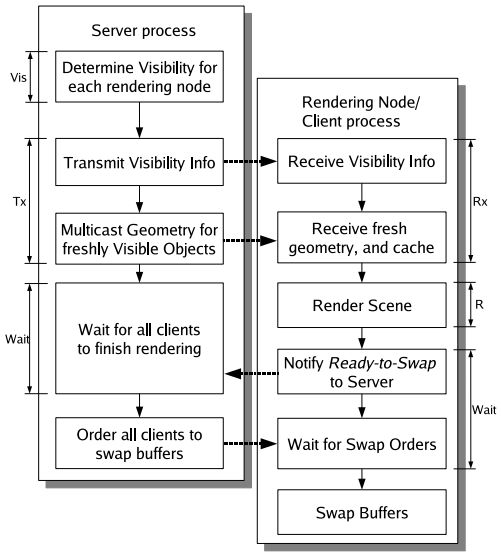

Figure 2: Server and client side processing and control flow for each frame. *Vis* refers to the visibility determination stage, *Tx* and *Rx* refer to the transmission stage, and *R* denotes the rendering stage. Pipelining of these stages is not shown here.

(One of the easy extensions to the system is the use of out-of-core rendering at the server to handle truly massive environments.) The objects not already present in the cache of the clients are sent to each using a multicast approach. Multicasting can exploit the spatial coherence of objects that intersect multiple tiles. The clients cache these objects so as to reuse them subsequently. This exploits the temporal coherence in the visible models in each tile. The clients start rendering a frame immediately after receiving all necessary data and inform the server when they are ready to swap the buffers. The server synchronizes the displays by ordering a common swap after receiving the ready-to-swap messages from all clients. Garuda facilitates transparent tiled display of any Open Scene Graph-based application without the user having to modify anything. This is accomplished by replacing the cull-draw-swap actions of the Open Scene Graph system using Garuda's cull, distributed draw, and synchronized swap mechanisms described later.

Figure 2 shows the tasks undertaken at the server and client along with the flow of control between them for each frame rendered using our system. We now explain the different steps involved in Garuda's operations.

3.1 System Startup and Initialization

The Garuda library is loaded before running any OSG based user application at the server machine. The init function in the Garuda library initializes all the client connections and pre-processes the scenegraph before entering the rendering loop. The user application loads the scenegraph into memory whose reference pointer is stored with the server. The server computes the oriented bounding boxes (OBB) for all nodes in the scenegraph using a simple PCA approach. OBBs are used for view-frustum culling by the server at each frame in the rendering loop. A generic OSG-based viewer application is fired up at each client node. A simple client-daemon enables the server to initiate this process over the network. The client application receives scenegraph data and viewing parameters from the sever incrementally and performs the steps described below.

Data transmission starts when the first frame is rendered. The cull traversal of the first frame by OSG will result in the Garuda server determining the objects to be sent to each tile. A partial scenegraph is constructed at each client as objects are received from the server. The package of data sent to the clients contain the geometry nodes for the visible objects as well as the entire path from the scenegraph root to them. Transmission, rendering, and synchronization of the first frame is handled like all subsequent frames as explained in Section 3.2. Rendering the first frame involves large data transmission due to the starting up issues; subsequent frames take less time as only the incremental portions of the scenegraph needs to be sent.

3.2 Rendering Pipeline

The user or the application program need not be aware of the tiled display when using Garuda system as any Open Scene Graph based application can be displayed on it without any modification. The system expands the user's view volume – called the *primary view frustum* – to fill the complete tiled display, creating internal sub-frustums for each tile. The primary view frustum is divided into equal-sized tiles in accordance to the arrangement of tiles in the display wall. The user controls the viewpoint and can modify the scenegraph. The cull and draw operations of the user program initiate special processes that render the scene to the tiled display. Garuda's rendering pipeline can be divided into three stages: Visibility determination, data transmission, and rendering and synchronization. These are the operations that determine the overall performance of the system and are pipelined to maximize the rendering performance. We describe each stage now.

3.2.1 Visibility Determination

Visibility needs to be calculated for each tile's view frustum for tiled rendering. We use an adaptive view frustum culling algorithm to do the tile-sorting. This algorithm culls the objects of the scenegraph to each tile's sub-frustum in every frame. We do not use visibility culling as the temporal coherence is better exploited with frustum culling than true occlusion culling for a tiled display. This is because the occluded objects in the view frustum can become visible in subsequent frames. It is thus safe to send such objects to the tiles in the first place. The tile-clients can do occlusion culling to reduce rendering load. Our philosophy of using low-end clients for scalability, however, makes this difficult.

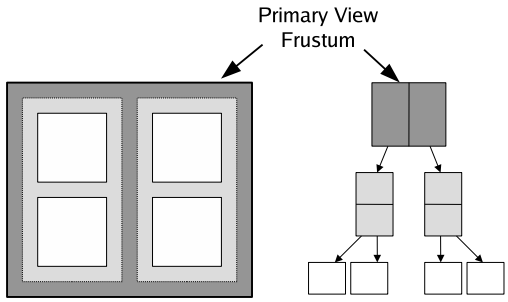

Figure 3: Frustum hierarchy is built by dividing the primary view frustum using horizontal and vertical planes successively. Each division creates an additional level in the hierarchy. The near and far planes are common and do not play any role.

Garuda's visibility algorithm deals with two hierarchies: the Object Hierarchy (OH) of the scenegraph and the Frustum Hierarchy (FH) of the primary and sub-frustums. The Frustum Hierarchy is obtained by bisecting the primary view-frustum recursively – alternately using vertical and horizontal planes – until each individual tile's view-frustum is reached (see Figure 3). The bisection plane used at each level is saved at the internal nodes in the FH. The algorithm traverses the OH and the FH adaptively to minimize the number of frustum-box intersection tests. The process begins by marking all objects visible to any tile using conventional culling of the scenegraph to the primary view frustum. The `adaptive_OHandFH_Cull(OH_Node, FH_Node)` algorithm is invoked with the root `OH_Node` of the culled scenegraph and the root `FH_node` of the frustum hierarchy.

Algorithm 1 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
```

In Algorithm 1, if `FH_Node` is a leaf, it represents a tile, and the corresponding `OH_Node` is marked to be visible to it (line 2). If `OH_Node` is a leaf, `FH_Node` needs to be unfolded to its sub-frustums. However, if neither `FH_Node` nor `OH_Node` is a leaf, a decision needs to be taken to determine which of the two hierarchies to unfold next. Line 5 uses `ClassfiyLCR`, an auxiliary function which groups the children of `OH_Node` into three sets: L (Left), C (Cuts) and R (Right), with respect to `FH_Node`'s bisection plane. Classification to these groups is done using OBB of the `OH_Node` instead of its actual geometry as it is a compact and conservative representation for it. Note that the sets L, C and R are disjoint. So, objects in set L need to be tested further with the left sub-frustum only (lines 10–12), whereas those in set R need to be tested with the right sub-frustum only (lines 13–15). Objects in set C cut the bisection-plane and hence must to be tested with both the sub-frustums. In a typical scene, a majority of objects fall in sets L and R, thereby potentially reducing the computations by half. The algorithm adapts to the scene structure and the viewpoint, following an optimal route to assign objects to a tile using the shortest path. If N is the number of objects in the primary view frustum and M the number of tiles, the algorithm has a worst-case running time of $O(MN)$ when all objects intersect all tiles and a best-case running time of $O(\log M)$ when all objects are in one tile. The average running time depends on the branching factor of the scenegraph and the number of nodes that intersects the frustum plane in each step. For good hierarchical scenes the running time varies in practice as $O(\min(N\log M, M\log N))$. The culling time for a hierarchical version of the Powerplant model for a 4×4 tiled display is under 4 milliseconds (see Section 5). More analysis and details on the culling algorithm including comparison with other approaches can be found in [30].

3.2.2 Transmission

The visibility determination stage identifies the OSG nodes present in each tile's view-frustum. Garuda server needs to send these nodes to the respective clients and coordinate their rendering for every frame. Transmission over the network is a serious limiting factor in all cluster-based display systems. The performance of this stage is determined by the network bandwidth and latency. The server updates the visibility information at each client by sending it the list of objects in its frustum for the next frame. This information in the Garuda system is less than 512 bytes per frame for each client. The visible objects that are not present already in each client's cache are subsequently send to it next. The server keeps track of the cache state of each client and has the necessary information for this. These nodes to be sent are serialized into memory buffers using OSG's file-writing mechanism; this data can optionally be compressed. The serialized data is then multicast over the network to a multicast group that includes all clients. The client nodes gather the necessary data from this multicast, deserialize the geometry, add it to the scenegraph and cache it for later reuse. The server keeps track of the objects that have been transmitted to each client to avoid retransmission in later frames. A simple handshake mechanism ensures that the data is transmitted reliably using the UDP multicast. The protocol overheads are low as packet losses are insignificant even on the UDP as the system is built as a tight LAN sub-network.

The alternative to multicasting is the use of unicast using a more reliable TCP protocol. Multicasting ensures that the sever transmits data only once even when an object is needed by multiple clients. This situation is common for tiled displays, especially as the number of tiles increase and the tile size shrink. This is a critical aspect for the scalability of the Garuda system. Multicasting thus ensures that the network requirements don't scale linearly with the number of tiles. The network requirements remain practically constant for different tile configurations. A demonstration of this for the initial startup of the display wall – when the network requirements are the highest – is given in Section 5. A unicast-based scheme needs to send each object separately to each client that needs it, increasing the network requirements.

All communications between the server and the clients are executed in separate threads or processes in the respective computers. This ensures that the communication doesn't hold up other activities. Integrity of the data is checked before the received data is used, to make sure all of it is available.

3.2.3 Rendering and Synchronization

The clients start rendering the scene for a frame when all objects for it are available. The time taken for this step is proportional to the geometry inside the view frustum. This can vary from client

to client. A server-coordinated swapping of the frame buffers is, therefore, used for synchronizing the display to ensure simultaneous change. Each client sends a *ready-to-swap* signal to the server after rendering to the back buffer is completed. The server sends a *swap* message to all clients after receiving ready-to-swap from all of them. Network latency is crucial for this simple synchronization procedure. Our experiments show that a commodity Ethernet can synchronize satisfactorily at interactive frame-rates.

3.3 Scenegraph Management

Data sent to the client contain the position of an object with respect to the root node in the scenegraph along with its geometry. The client creates the tree structure from the data received and inserts each node at the appropriate location in its scenegraph. As more and more objects are sent to a client, its scenegraph begins to resemble the overall scenegraph present at the server.

The client also caches the geometric objects received. The cache enables our system to exploit the inter-frame coherence of data. Each client has a fixed cache and uses an LRU algorithm to remove objects from it when the cache gets full. Only the geometry nodes are cached and removed; the internal nodes of the scenegraph are retained. Eventually, the structure of the server's scenegraph will be present at every client, but with only fewer leaf nodes. Such a replication of scenegraph structure helps for consistent and incremental scenegraph management by the server.

The caching and the exploitation of temporal coherence is another important step towards scalability to a large number of tiles. As the number of tiles increases, each client needs to render less and can hold more data in its cache. The server-push philosophy reduces the client's load and hence low-end clients can be used.

3.4 Pipelining

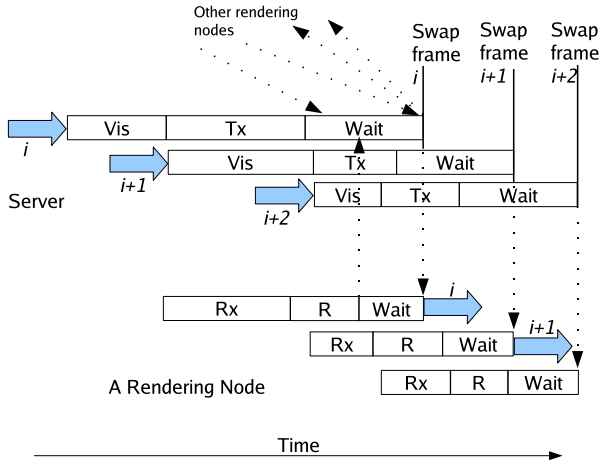

Figure 4: Inter-frame pipelining of the different stages. Vis denotes the visibility determination, Tx the geometry transmission, Rx geometry receiving and R the rendering. Here *i* denotes the current frame number. The effective FPS of the system gets increased due to this interleaving.

The three stages of Garuda's rendering pipeline (Section 3.2) can be pipelined. While the server is waiting to order swap for frame *i*, it can perform the visibility computation and transmission for the next frame. The client, on the other hand, can receive data for the next frames when frame *i* is being rendered. Figure 4 illustrates this pipelining between the various stages of the server and a client. We implement pipelining by taking advantage of the pipelined App-Cull-Draw of the Open Scene Graph system.

3.5 Handling Dynamic Objects

Handling dynamic objects properly is a big challenge in any master-slave framework. Yet, the possibility of some objects changing their positions or orientations is critical to many applications. The Garuda system handles this using the dynamic transformation nodes of Open Scene Graph. Selected intermediate transformation nodes are monitored every frame for any change in their parameters. If the parameters change, the composite transformation matrix at that node is extracted and sent to all clients that have the node using a special notification mechanism. The client will replace the matrix with a new one and the object will appear transformed when the next frame is drawn. This mechanism can handle all dynamic and articulated objects which are the most common form of dynamic objects. Dynamic objects could also include those that change their shape or appearance. Since the geometry itself changes in this case, the server can order the deletion of the corresponding nodes. The new object added will automatically be identified and transmitted using the normal visibility mechanism.

3.6 Garuda System: Summary

In summary, the Garuda server performs the following for each frame.

1. Receive new viewpoint from the application
2. Determine the visible objects for each tile
3. Transmit visibility information to each rendering node
4. Identify objects not in the cache and the dynamic objects that were modified for each client.
5. Serialize these objects from the scenegraph nodes and multicast them to the clients
6. Wait for each client to complete its rendering and send it the *Ready to Swap* signal
7. Send *Swap* order to all rendering clients

Each client performs the following for each frame.

1. Receive the visibility information from the server
2. Receive the newly visible object nodes and attach to the scenegraph.
3. Render the scenegraph, no culling required
4. Send *Ready to Swap* to the server
5. Wait for *Swap* order from server
6. Swap buffers

4 Rendering Transparently to a Tiled Display

A central issue concerning a tiled display wall system is its universal applicability. One would not want to specially modify an application for rendering it to a tiled display, as it would severely restrict the applicability of such a system. The utility of a tiled display system is high only if any application developed using a standard graphics/visualization API can be rendered automatically to it. Chromium system manages to do this for any OpenGL application by intercepting all OpenGL calls and sending the primitives to the clients. However, it does not cache the data easily at the rendering nodes. This increases the network requirements heavily, especially for scenes with huge geometry.

The Garuda system is designed to automatically render any application built on the Open Scene Graph API [7], without modifying the application in any way. A scenegraph structure enables the caching of objects at the clients to exploit the inter-frame coherence

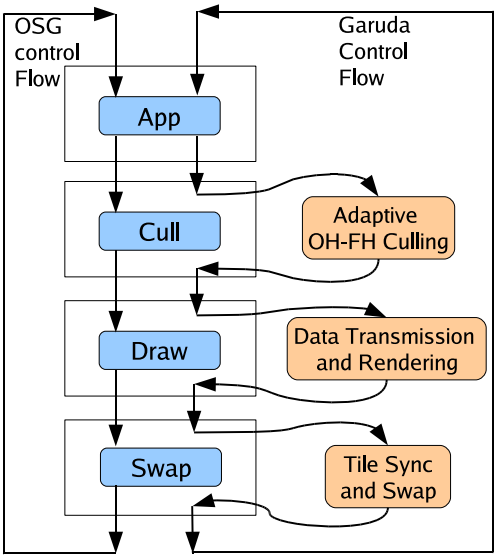

Figure 5: The cull, draw and swap steps of OSG's default control flow (left) are intercepted and replaced in the control flow (right) so the user program doesn't need to be modified for tiled rendering.

of data and helps reduce the network traffic. The scenegraph API also provides the mechanism to pipeline the app-cull-draw stages to enhance the throughput of the system. A standard API like the OSG has a wide user-base which can benefit from transparent tiled rendering. Other scenegraph APIs like OpenGL Performer [32] can also be used. We selected OSG for its rising popularity and the open-source development model. Transparent, tiled rendering is provided by intercepting relevant calls of the OSG API and replacing them with our own. Since OSG maintains high level objects as nodes, the intercept mechanism is less computation-intensive and it is possible to optimize the usage of network bandwidth by caching OSG nodes at the clients.

An OSG application typically loops through three main stages: App, Cull, and Draw. For transparent rendering, the Garuda system intercepts the calls to these at the OSG API level and performs the culling, drawing and synchronization steps needed for the tiled display. Figure 5 shows the normal control flow for OSG and the control flow adopted by our system. Garuda's actions in each stage are described below.

1. **App:** The application stage includes keyboard/mouse event handling, scenegraph manipulations, animations, etc. This is how the user controls the viewpoint and manipulates the environment. Garuda does not alter this stage and leaves it under the user program's total control.

2. **Cull:** The cull stage performs visibility culling of the scene graph in preparation for rendering. Garuda overrides OSG's native culling to perform the adaptive OH and FH culling as discussed in Section 3.2.1. The results of the culling are the list of visible objects for each tile and the list of objects to be sent to each client.

3. **Draw:** The draw stage renders the visible portions of a scene graph. Garuda replaces OSG drawing stage with rendering on the tiles. All scenegraph data required for rendering is transmitted in this stage (Section 3.2.2). The clients draw their scenes when all data is received but postpone the buffer swap until notified by the server.

These stages are followed by a call to swap the display buffers, available with the windowing API. Garuda intercepts the swap call at the server (which is strictly not part of OSG) and implements the last part of the synchronization described earlier. The server makes sure all ready-to-swap messages have come and orders all the clients to swap, thereby rendering a frame. The overhead involved is kept low by performing the communication between the client and the server in different threads and sometimes processes.

Garuda's ability to transparently render any OSG-based application brings clustered, tiled-display to a large number of applications. The user does not need to learn a new API or modify their programs in anyway. All aspects are handled implicitly if OSG is used in a standard manner. Garuda today supports OSG-based applications that use groups, geodes and static and dynamic transformation nodes. Support for textures, normals and lighting is also present. Advanced features such as animation, FX effects, particles, etc., are not currently handled.

5 Experimental Results

The Garuda system was evaluated for its performance on typical complex scenes, and on its scalability to large displays. We want to achieve good frame rates on challenging applications. We also want to provide a scalable design such that very large display walls can be built using commodity hardware.

Test Setup: The test setup consists of 16 low-end systems with AMD Athlon 64 3000+ CPU, 512 MB RAM, and an ATI Radeon Xpress 200 on-board graphics. The GPU on these systems uses 64 MB of system RAM as the video memory. These machines act as rendering nodes in the cluster. For all experiments, each tile renders its frustum to a 1024×768 display in all configurations. The 4×4 display wall thus has a total resolution of 4096×3072, or well over 12 Million pixels. The server has an AMD Athlon 64 3200+ system with 3GB RAM. It has an Nvidia 6600GT GPU, but the graphics capabilities of the server are not important. Our server also doubles as a rendering node in our cluster. The low-end systems serve our objective of building a display wall using low-end hardware. An increase in the rendering capabilities at the clients results in a direct performance improvement for our system, as shown by the experiments below. Unless mentioned otherwise, the experiments are carried out over a 100 Mbps Ethernet network. We present results of tests for scalability with various tile-configurations (2×2, 2×3, 2×4, 3×3, 3×4, 4×4).

Two hierarchical OSG models were used for the experiments. The first is the model of Fatehpur Sikri[2], an architectural monument with a scenegraph hierarchy with 530,000 triangles spread over 1113 geometry nodes with 331 internal nodes. Its average branching factor is 4.36. The second is a hierarchical version of the UNC Powerplant model. It consists of about 13 million triangles spread over 19666 geometry nodes with 4760 internal nodes. Its average branching factor is 5.13. The spatial hierarchy was created from the original non-hierarchical models which resulted better scene management at the cost of increase in the number of triangles. Another synthetic, hierarchical teapot model was also employed in testing certain aspects of the system as described later.

The server reads and initializes large amounts of data and sends a lot of it to the clients on system start up as described before. Table 1 compares the startup transmission time when using UDP multicast with the TCP unicast approach, for the Powerplant model. This includes the preprocessing of the scenegraph and the preparation and transmission of the data for the first frame. The startup time of our system is virtually independent of the tile-configuration when us-

[2]Fatehpur Sikri is a world heritage site located near Agra. It was constructed by the Moghul emperor Akbar in the late 16^{th} century. The graphics model was created by NCST/CDAC, India.

Tile configuration	Time for TCP unicast (seconds)	Time for UDP multicast (seconds)
2×2	7.24	11.95
2×3	10.35	11.76
2×4	13.33	11.53
3×3	15.07	11.87
3×4	19.38	11.52
4×4	25.09	11.58

Table 1: Startup transmission times using the unicast and multicast approaches for the Powerplant model, including the transmission of the initial data to the clients. The time for multicast remains constant practically while the unicast time grows linearly with the number of tiles.

Tile configuration	Powerplant Culling time (ms)	Fatehpur Sikri Culling time (ms)
2×2	0.81	1.05
2×3	1.20	1.55
2×4	1.44	1.77
3×3	1.68	2.22
3×4	2.05	2.48
4×4	2.36	2.60

Table 2: Visibility determination time using the adaptive visibility culling algorithm on the full Powerplant and Fatehpur Sikri models.

ing multicast but increases linearly when using TCP unicast since it ends up sending the same object multiple times to each node that needs it. It should be noted that as the number of tiles go up, the frustum of each tile shrinks, and the number of objects that are visible to multiple tiles increases. This results in greater gains for the multicast approach. This makes the Garuda architecture scalable to large display sizes.

The adaptive visibility algorithm is an important step of the display wall. Table 2 shows the time taken to perform the visibility determination on the Powerplant and Fatehpur Sikri models for various tile-configurations using Algorithm 1. The culling time increases sub-linearly as the number of tiles increases, owing to the hierarchal treatment of scene and frustums in our algorithm. This implies that the algorithm scales well for large display configurations. Sub-linear increase in the culling time improves the scalability of the system to a large number of tiles. The table shows the culling time averaged over a 3000 and 2700 frame typical walkthrough for the Powerplant and Fatehpur Sikri models respectively.

Figure 6 shows the performance of Garuda on a 2700-frame walkthrough on the Fatehpur Sikri model, that goes over dense and sparse areas of the model. In the dense regions, the performance is about 10 frames per second (fps) due to the limited rendering capabilities of the clients. Framerate increases to about 40 when less geometry is visible (frames 1000–1400 and 2100–2350). Note that the framerate remains constant for different configurations as the resolution increases four-folds from 2×2 to 4×4. The multicast communication scheme and the server's data-push philosophy make it possible by facilitating concurrent transmission to multiple clients. The network requirements are kept to a minimum with the use of caching of objects at the clients.

The performance of the Garuda system is determined by the worst-performing rendering node. Better system performance can be obtained if better rendering performance is available at each tile. Figures 7 shows the results using high-end client nodes with 1GB RAM and Nvidia 6600GT GPU. The walkthrough used is more challenging than the previous experiment with viewpoints that see

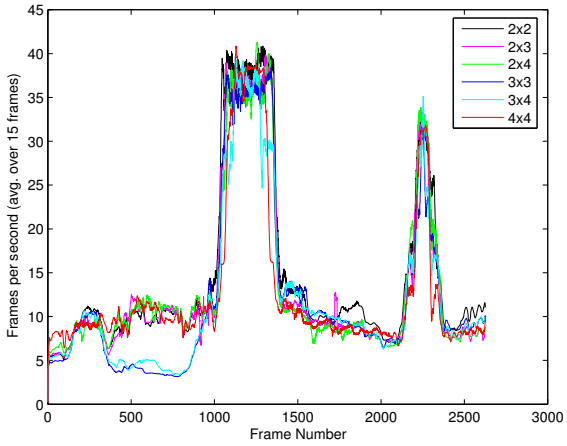

Figure 6: Framerate achieved during a 2700 frame walkthrough of the Fatehpur Sikri model. The performance remains almost unchanged for different tile-configurations though the display resolution increases four-fold.

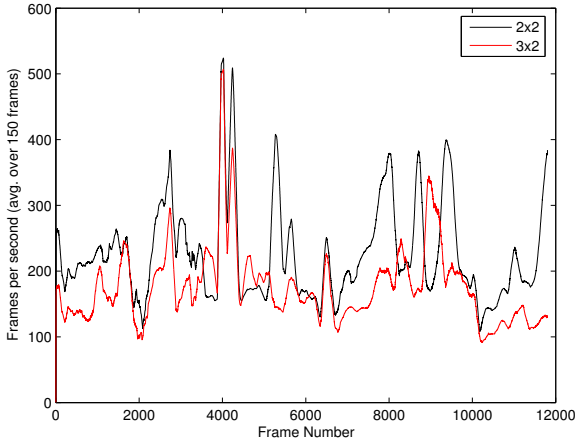

Figure 7: Framerate for a 12000 frame walkthrough of the Fatehpur Sikri model using high-end rendering nodes with Nvidia 6600GT graphics cards. The system achieves a rendering performance as high as 200 fps due to the improved graphics capabilities.

more of the model at all times. Frame rates of 200 fps and above were obtained, confirming that the experiment shown on Figure 6 is rendering limited on the low-end clients. Note the improvement in performance even using smaller tile configurations.

The fps does not increase with the number of tiles in Figures 6 and 7 because of the poor load distribution. A lot of the model geometry is concentrated in the lower quarter of the display (refer Figure 15) around the ground region. The ground region also contains models with large triangles which do not get frustum-culled effectively. The worst-performing rendering node ends up drawing about the same amount of scene even after the tile size reduces. Better load distribution can be observed for scenes with granularity such that the geometry is distributed evenly over the entire scene. Figure 8 shows a synthetic environment created to test this. The scene has 417 teapots with a total of 1.5 million triangles, that are distributed randomly in space. The figure shows near-linear increase in the frame rates when rendering to tile configurations of 2×2, 3×3, and 4×4. Based on these experiments, we can recom-

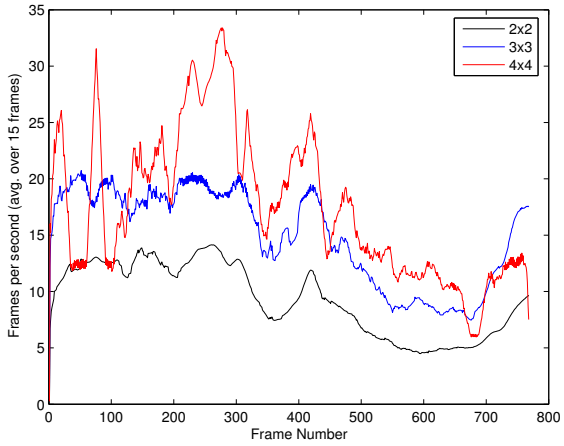

Figure 8: Rendering performance of Garuda on a synthetic environment with 417 randomly distributed teapots and 1.5 million triangles. The distributed rendering results in increased fps as the number of tiles increases on low-end clients.

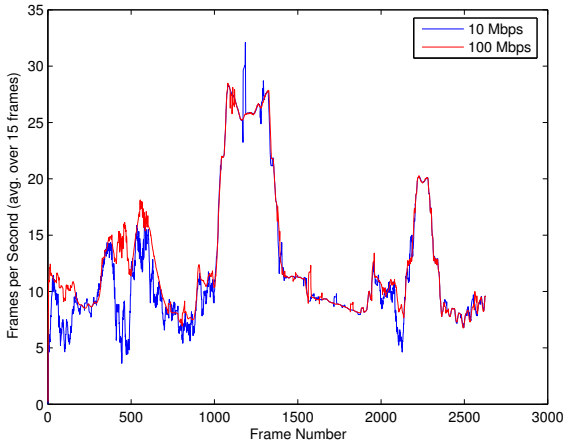

Figure 10: Framerates for the 2700 frame walkthrough of the Fatehpur Sikri model on a 4×4 wall with 10 Mbps and 100 Mbps networks. Caching at clients significantly lowers the network dependence, both networks have near-constant fps after the initial startup time.

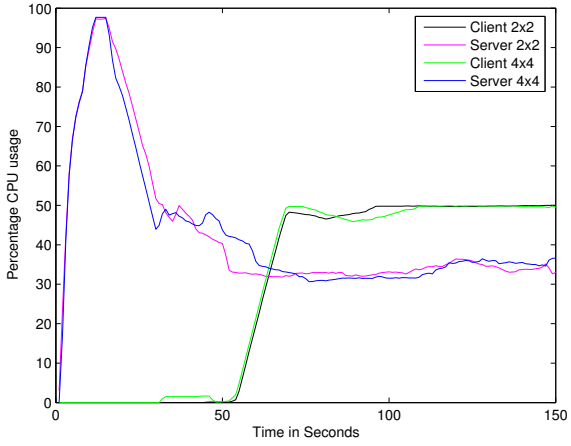

Figure 9: The CPU load for different tile sizes on the server and the client for the Powerplant model. CPU usage is nearly constant for different tile configurations resulting in excellent scalability.

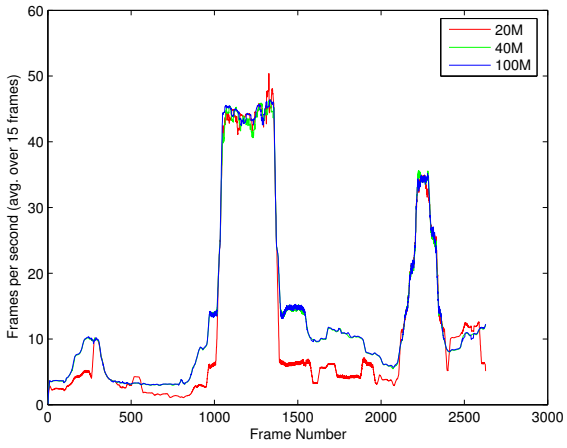

Figure 11: Framerates for a 2×2 system for three cache sizes on the Fatehpur Sikri model for the 2700 frame walkthrough. 20 MB cache has lesser frame rate because of retransmission of geometry data.

mend that the the lower portions of a tiled display should perhaps be powered by nodes with high-end graphics for better performance on architectural models. Using different levels of detail (LoD) for objects is another option to ensure more even load distribution. This has to be done for the whole scene as all tiles should have the same LoD for an object. The LoD switching cannot easily be performed without the user application being aware of it. Since applicability to any OSG-based application was in initial goal, we did not pursue this approach further.

Figure 9 shows the CPU usage for different tile configurations at the server and a typical client node for Garuda. The startup phase is compute-intensive on the server, but its load settles to a comfortable 40% of CPU usage very quickly. The client's CPU usage picks up slowly since it only has rendering to perform. The client node's usage settles at a 50% level to manage the scenegraph, the cache and the rendering. The CPU usage is practically constant for a $2\times$ 2 configuration and a 4×4 configuration. Please note that the 50 seconds of startup time includes the time to load the models from files, to pre-process the loaded scenegraph, to compute the visibility for the first frame and to send the data to the clients. The loading time was not shown in Table 1. This again establishes that our system architecture is scalable to a large number of tiles.

Figure 10 demonstrates the low dependence of the system on the network bandwidth for a 4×4 configuration. The faster network performs better initially, as more and more scenegraph is discovered and fresh objects are transmitted to the clients (till frame number 900). Both 10 Mbps and 100 Mbps networks perform equally well thereafter. This is due to the caching of geometry at the tiles, which ensures minimal transmission of bulky geometry data for subsequent frames. These results suggest that the Garuda architecture can comfortably scale to display walls that have several dozens of tiles using commodity LANs of 100 or 1000 Mbps.

We show the impact of caching for different cache sizes in Figure 11 for a 2×2 display wall. A 2×2 wall was used here as cache size is more critical for smaller tile configurations. The experiment shows the same walkthrough as in Figure 6 for three different cache sizes. The optimal cache size depends on the density of objects in an environment and can vary from scene to scene. It can be seen that

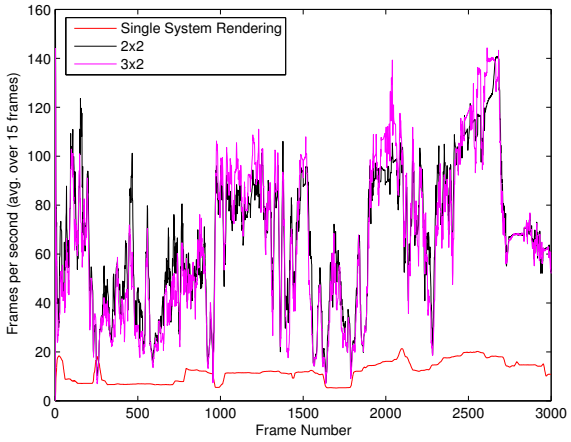

Figure 12: Framerate for an 800 frame walkthrough of the Powerplant model using rendering nodes with Nvidia 6600GT graphics. The system performs at more than 40 fps for most part of the walkthrough. The 3×2 configuration performs better due to lower rendering load on individual tiles.

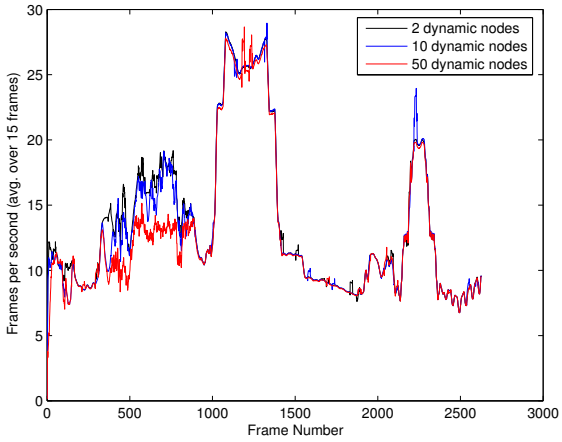

Figure 13: Fatehpur Sikri walkthrough for a 4×4 system with dynamic Open Scene Graph transformation nodes. The scene has different number of rotating cow models (each with 5800 triangles) added to it. The transformation nodes are updated every frame. The fps remains constant practically. Low-end clients were used for this test.

the framerates are less with a 20 MB cache than with 40 MB or 100 MB cache. This is due to the retransmission of objects necessitated by their eviction when cache fills up. For the regions with higher framerates (frames 1000–1200 and 2100–2300) there is lesser geometry to render in the walkthrough. No retransmission is required even with smaller caches. The framerates for the 40 MB and 100 MB are almost identical, which suggests that a 40 MB cache is sufficient for this model. It is interesting to consider the case when the cache is sufficient to hold the whole scene at each rendering client. The performance of Garuda's client-server architecture will approach that of the master-slave architecture [16], but with better handling of dynamic objects. The server needn't perform elaborate visibility computations. This, however, will not be compatible with our initial design goal of using an inexpensive processor as the rendering client.

Figure 12 shows Garuda's performance on a walkthrough of the Powerplant model. All tiles are powered by high-end systems with an Nvidia 6600GT card in each. The framerate is maintained above 40 fps for most of the walkthrough even on such a challenging model. For comparison, we setup a 2×2 configuration involving the high-end machines using Chromium for tiled rendering. The system was able to achieve only about 0.2 fps due to the high network requirements. The same walkthrough performs at about 10 fps on a single machine of similar configuration. The display wall achieves a framerate 4-5 times higher in spite of increasing the pixel resolution 4 to 6 times. The high variability in the fps is due to the inherent randomness associated with network communications while the overall improvement in fps is due to the cluster rendering.

A gigabit Ethernet will be able to provide better predictability and lower variations in performance. Levels of detail (LoDs) can be used to reduce the network load, if high. This requires that the user's model be available in multiple detail, which runs counter to the objectives of transparent rendering. The same LoD needs to be used on all clients for visual continuity. This may involve additional data management between the server and the clients. Clients can exchange data among themselves when that is more efficient. This adds to the complexity of the clients and makes it harder for the server to keep track of the state of the clients. True visibility or occlusion culling could be employed to spread the network load over time. This will require very fast occlusion culling at the server using the graphics processing unit. Viewpoint prediction and a second level of cache can also be used to reduce variability at the expense of complexity of the server.

The same walkthrough runs on a 4×4 configuration using the low-end clients (ATI Xpress 200 with 64 MB shared video memory) at about 3 fps. The application fails to startup due to poor resources on a single system of same capabilities. This demonstrates that the Garuda system is able to exploit the power of distributed rendering to render challenging models using modest computers.

Dynamic scenes can also be rendered at interactive frame rates using the Garuda system. Figure 13 shows the system's performance for a scene with many dynamic objects. The Fatehpur Sikri model was modified by adding 2, 10 and 50 Open Scene graph dynamic transformation nodes, each with a cow model of 5800 triangles. Data for these nodes changes every frame and hence must be sent every frame, adding to the the overhead of rendering dynamic environments. Culling at each frame needs to recompute the transformed OBB for each node for proper results. The system is able to maintain more than 10 FPS throughout the walkthrough even with large number of moving objects in the scene. The graph also shows that Garuda is scalable with respect to number of dynamic nodes in the scene; as the reduction in fps is very low even for a large increase in dynamic nodes.

We also check to see to what extent a single server can be pushed to power a tiled display. Figure 14 shows a single server culling time on the Fatehpur Sikri model for the 2700 frame walkthrough for various tile sizes. The culling time increases sub-linearly with the increase in tile size which adds to scalability of the Garuda system to large configurations. Although the increase is sub-linear, for an 8×8 tile configuration a single server takes around 6.5 ms for culling only which leaves around 9 ms for other overheads and data transmission if the required 60 fps is expected. Clearly for a larger display system a hierarchy of servers will be need, consisting of culling stages. First stage culls to a bigger sub-frustum and sends the information to next stage which in turn culls to the tile frustums inside its sub-frustum. The second stage is done in parallel and hence much culling time is saved. A single server hence becomes a bottleneck after a 7×7 tile configuration.

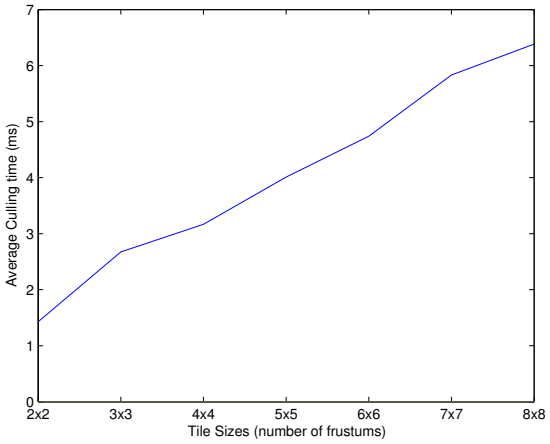

Figure 14: Fatehpur Sikri 2700 frame walkthrough's average culling times. The increase in culling time is sub-linear, adding to the scalability of the Garuda system. Also showing that the culling becomes a bottleneck after 7×7 tile configuration for a single server system.

Figure 15: A 4×4 display wall showing the Fatehpur Sikri model. The combined resolution is 12 mega pixels.

Figure 16: A 4×4 display wall showing the Powerplant model. The combined resolution is 12 mega pixels.

6 Conclusions, Discussions and Future Work

In this paper, we presented Garuda, a geometry-managed, tiled display system built using commodity computers that can transparently render any application built using the Open Scene Graph API. The design choices made with respect to the hardware and software make Garuda scalable to large displays. It can also find wide applications due to its ability to render any OSG-based application without modifying it. Local caching of the geometry and the multicast mode of transmission keeps the network requirements moderate and the system scalable. Garuda was shown to be scalable for different tile geometry using commodity components. Spatial and temporal coherence are exploited by using a push-philosophy adopted by the server. This also allows for consistent handling of dynamic objects in the scenegraph.

Scalability is critically important to the design of a cluster-based tiled display system [29]. Can the system be scaled to a billion pixels driven by a single interactive graphics application? We believe that the Garuda architecture can scale to those levels. We discuss the design issues for such a large display.

Display: The display is best built using LCDs. CRTs can also be used but are more bulky and will be discontinuous at the borders because of larger casing. Projectors are not suitable as they are more expensive for the same resolution and are more cumbersome to handle.

Clients: A giga-pixel display will need 500 tiles if each tile is of 2 mega pixel resolution. It is important to keep the cost of each low for the sake of scalability. The rendering load per tile will be low for most reasonable scenes as the tiles are likely to be small. The pixel fill-rate may be the bottleneck for such a system. Game consoles like Microsoft Xbox 360 or Sony PS-3 are ideal for the rendering nodes. They have very good CPU power, excellent graphics capabilities, and cost about the same as the low-end servers we use. The Sony PS-3 may be able to render two tiles simultaneously going by its specs. A giga-pixel display can be built using 250 clients controlling the 500 displays with a significant savings in costs.

Network: Geometry caching and multicast transmission produce better gains when the number of tiles increases as described earlier. The network activity for administrative aspects like communicating visibility and synchronization increases linearly but the bulky data transmission activity remains more or less constant due to multicasting. Commodity 1000 Mbps Ethernet will be able to handle the network load comfortably under such an architecture. Their higher performance will be able to reduce the high variability we witness with the current system.

Server: The visibility computation load increases with the number of tiles but the adaptive algorithm keeps the increase logarithmic. A multilevel server hierarchy may be required to handle the additional load when there are 500 tiles. The top-level server performs tile-sorting to 25 large tiles and sends geometry to as many second-level servers. Each of those perform the tile sorting to 20 tiles each and send the data to the actual rendering clients. The user's application process runs on the primary server, which is also responsible for synchronization. The network performance can also be improved by using dedicated switches at each of these servers. This arrangement can handle large tile configurations, perhaps at a slightly increased latency.

Architecture: The basic tile-sorting philosophy can be modified if the tile processors are as powerful as those advertised for

the game consoles. The server can cull the scene to the primary view frustum and multicast all surviving objects to all tiles. The tile-clients then cull the scene to its sub-frustum before rendering. Since the second cull step is happening in parallel, the overall visibility computation time could be smaller.

Swap-lock and gen-lock of the display system need 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. A high-speed camera and a calibration pattern that appears on all tiles in the same frame can together estimate the delay between each tile's display from a reference tile. It is possible to adjust the low-level video parameters of the display card till the delay is minimized. This may be performed in an iterative manner till the genlocks are sufficiently synchronized. This needs further experimentation.

Acknowledgments: We thank Microsoft Research for partly funding this research through their university relations, India. We also thank the National Center for Software Technology (NCST) (now renamed CDAC), Mumbai for providing the Fatehpur Sikri model and the University of North Carolina for the Powerplant model.

REFERENCES

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

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

[3] HIPerWall, `http://cg.calit2.uci.edu/mediawiki/index.php`.

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

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

[6] ModViz Renderizer, `http://www.modviz.com/products/renderizer.asp`.

[7] OSG: OpenSceneGraph. `http://www.openscenegraph.org`.

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

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

[10] Jeremie Allard, Valerie Gouranton, Guy Lamarque, Emmanuel Melin, and Bruno Raffin. SoftGenLock: Active Stereo and Genlock for PC Cluster. In *Proceedings of the Joint Immersive Projection Technology / Eurographics Virtual Environments'03 Workshop*, 2003.

[11] Jeremie Allard, Valerie Gouranton, Loick 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.

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

[13] 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 '01: Proceedings of the Virtual Reality 2001 Conference (VR'01)*, pages 89–96, Washington, DC, USA, 2001. IEEE Computer Society.

[14] Thomas C. Bressoud and Fred 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.

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

[16] Han Chen, Douglas W. Clark, Zhiyan Liu, Grant Wallace, Kai Li, and Yuqun Chen. Software environments for cluster-based display systems. In *IEEE International Symposium on Cluster Computing and the Grid*, 2001.

[17] Wagner T. Corrêa, James T. Klosowski, and Cláudio T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 89–96, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[18] Wagner T. 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.

[19] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the CAVE. In *Proceedings of ACM SIGGRAPH*, volume 27, pages 135–142. ACM, August 1993.

[20] Soumyajit Deb, Shiben Bhattacharjee, Suryakant Patidar, and P. J. Narayanan. Real-time streaming and rendering of terrains. In *Indian Conference on Computer Vision, Graphics and Image Processing*, volume 4338 of *Lecture Notes in Computer Science*, pages 276–288. Springer, 2006.

[21] Soumyajit Deb and P. J. Narayanan. Design of a geometry streaming system. In *Indian Conference on Computer Vision, Graphics and Image Processing*, pages 296–301. Allied Publishers Private Limited, 2004.

[22] Michael Deering and David Naegle. The SAGE graphics architecture. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 683–692, New York, NY, USA, 2002. ACM Press.

[23] 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.

[24] Desmond Germans, Hans J.W. Spoelder, Luc Renambot, and Henri E. Bal. VIRPI: A High-Level toolkit for interactive scientific visualization in virtual reality. In *Immersive Projection Technology / Eurographics Virtual Environments Workshop*, pages 109–120, 2001.

[25] 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: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 693–702, New York, NY, USA, 2002. ACM Press.

[26] Aditi Majumder and Rick Stevens. Color nonuniformity in projection-based displays: Analysis and solutions. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):177–188, 2004.

[27] Laurent Moll, Mark Shand, and Alan Heirich. Sepia: Scalable 3d compositing using pci pamette. In *FCCM '99: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 146–155, Washington, DC, USA, 1999. IEEE Computer Society.

[28] Martin Naef, Edouard Lamboray, Oliver Staadt, and Markus Gross. The Blue-C distributed scene graph. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003*, pages 125–133, New York, NY, USA, 2003. ACM Press.

[29] Tao Ni, Greg S. Schmidt, Oliver G. Staadt, Mark A. Livingston, Robert Ball, and Richard May. A Survey of Large High-Resolution Display Technologies, Techniques, and Applications. In *Virtual Reality*, 2006.

[30] Nirnimesh, Pawan Harish, and P. J. Narayanan. Culling an object hierarchy to a frustum hierarchy. In *Indian Conference on Computer Vision, Graphics and Image Processing*, volume 4338 of *Lecture Notes in Computer Science*, pages 252–263. Springer, 2006.

[31] Ramesh Raskar, Michael S. Brown, Ruigang Yang, Wei-Chao Chen, Greg Welch, Herman Towles, Brent Seales, and Henry Fuchs. Multi-projector displays using camera-based registration. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 161–168, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[32] John Rohlf and James Helman. Iris performer: a high performance multiprocessing toolkit for real-time 3d graphics. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 381–394, New York, NY, USA, 1994.

ACM Press.

[33] 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.

[34] 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.

[35] Timothy A. Sandstrom, Chris Henze, and Creon Levit. The Hyperwall. In *CMV '03: Proceedings of the conference on Coordinated and Multiple Views In Exploratory Visualization*, pages 124–133, Washington, DC, USA, 2003. IEEE Computer Society.

[36] Benjamin Schaeffer and Camille Goudeseune. Syzygy: Native PC Cluster VR. In *VR '03: Proceedings of the IEEE Virtual Reality 2003*, pages 15–22. IEEE Computer Society, 2003.

[37] Rajvikram Singh, Byungil Jeong, Luc Renambot, Andrew E. Johnson, and Jason Leigh. Teravision: a distributed, scalable, high resolution graphics streaming system. In *CLUSTER*, pages 391–400, 2004.

[38] 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 '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 141–148, New York, NY, USA, 2001. ACM Press.

[39] Tom van der Schaaf, Luc Renambot, Desmond Germans, Hans Spoelder, and Henri Bal. Retained mode parallel rendering for scalable tiled displays. In *Immersive Projection Technology (IPT) Symposium*, 2002.

[40] Gerrit Voss, Johannes Behr, Dirk Reiners, and Marcus Roth. A multi-thread safe foundation for scene graphs and itsextension to clusters. In *Eurographics Workshop on Parallel Graphics and Visualization*, pages 33–37, 2002.

[41] Grant Wallace, Otto J. Anshus, Peng Bi, Han Chen, Yuqun Chen, Douglas Clark, Perry Cook, Adam Finkelstein, Thomas Funkhouser, Anoop Gupta, Matthew Hibbs, Kai Li, Zhiyan Liu, Rudrajit Samanta, Rahul Sukthankar, and Olga Troyanskaya. Tools and applications for large-scale display walls. *IEEE Comput. Graph. Appl.*, 25(4):24–33, 2005.

[42] 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.

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

[44] Ruigang Yang, David Gotz, Justin Hensley, Herman Towles, and Michael S. Brown. Pixelflex: a reconfigurable multi-projector display system. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 167–174, Washington, DC, USA, 2001. IEEE Computer Society.

[45] Xiaoyu Zhang, Chandrajit Bajaj, and William Blanke. Scalable isosurface visualization of massive datasets on COTS clusters. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 51–58, Piscataway, NJ, USA, 2001. IEEE Press.