Distributed Massive Model Rendering

Revanth N R*
Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, India - 500032
revanth.nr@research.iiit.ac.in

P. J. Narayanan
Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, India - 500032
pjn@iiit.ac.in

ABSTRACT

Graphics models are getting increasingly bulkier with detailed geometry, textures, normal maps, etc. There is a lot of interest to model and navigate through detailed models of large monuments. Many monuments of interest have both rich detail and large spatial extent. Rendering them for navigation on a single workstation is practically impossible, even given the power of today's CPUs and GPUs. Many models may not fit the GPU memory, the CPU memory, or even the secondary storage of the CPU. Distributed rendering using a cluster of workstations is the only way to navigate through such models. In this paper, we present a design of a distributed rendering system intended for massive models. Our design has a server that holds the skeleton of the whole model, namely, its scenegraph with actual geometry replaced by bounding boxes at all levels. The server divides the screen space among a number of clients and sends them a list of objects they need to render using a frustum culling step. The clients use 2 GPUs with one devoted to visibility culling and the other to rendering. Frustum culling at the server, visibility culling on one GPU, and rendering on the second GPU form the stages of our distributed rendering pipeline. We describe the design and implementation of our system and demonstrate the results of rendering relatively large models using different clusters of clients in this paper.

Keywords

ICVGIP2012, Distributed rendering, Massive model rendering, Load balanced rendering, Parallel rendering

1. INTRODUCTION

Modern virtual reality applications have created geometric models of large and complex 3D environments. Recent initiatives on digitizing monuments of popular interest across the world are generating massive 3D models, preserving their minute details. These Virtual reality applications would like to use a user-steered interactive walkthrough of these models to engage lay users.

Conventional rendering architectures and techniques cannot cater to such massive environments. There is a serious limit on what can be stored and rendered even on the latest GPUs. When the geometry size crosses this limit, the GPU needs a secondary memory for the details. GPU manufacturers have devised methods to simultaneously use multiple GPUs connected through special channels, where each GPU processes a part of the geometry data. This increases the overall GPU memory of the system compared to single GPU driven system, but cannot scale to the massive models available today.

One way to overcome this limit is to use memory on CPU as the secondary memory to the GPU. The RAM on CPU is expandable and can hold much larger models than the GPU. Slow transfer to the GPU is inevitable under this arrangement, however. Situations can arise where even the CPU memory cannot hold the models due to their bulk. A second level of caching on the CPU hard disk can help to solve this problem, at further reductions in speed. The storage device on the system can provide buffering for the CPU memory with higher latency. These storage devices can provide buffering up to a few TBs of data. It is possible for the models to exceed this limit also. Distributed storage and rendering provide the only solution to this problem. This situation is already prevalent in the rendering of computer generated movies. They, however, do not have interactive performance requirements and break down the rendering problem into a series of computation tasks performed on a large render farm. This option is not available when interactive or near-interactive rendering is required to navigate through the space of a monument.

In this paper, we present a distributed rendering solution to this problem. We suggest using a client-server framework using a cluster of multi-GPU systems to render massive geometric models in a distributed manner. We assume that the model can reside in the secondary storage of each rendering client for this work. The clients share the rendering load by handling a rectangular portion of the screen independently. The server interacts with the user and partitions each rendering task among the clients based on the estimated rendering load. The server also estimates and sends the list of objects to be rendered by each client using a frustum culling step using bounding boxes of geometric objects. The clients ensure these objects are available and renders its portion of the scene. To reduce the rendering load, each client performs visibility culling using one of its GPUs, while

*Corresponding author

ICVGIP '12, December 16-19, 2012, Mumbai, India

the other GPU is used to render the previous frame. The sub-images are sent to the server, which assembles it into a common frame buffer. The rendering can be scaled by increasing the number of clients.

We present the design and implementation details of a distributed rendering system for massive models in this paper and also show quantitative results using a geometric environment constructed using multiple instances of the Power Plant model. Our studies show that the system can be scaled to handle truly massive models while achieving reasonable frame rates.

2. PRIOR WORK

Determining the visibility of a geometry from a given viewpoint is fundamental problem in computer graphics. Culling away the the geometry that is ultimately not visible avoids the load of rendering them. Many occlusion culling algorithms[7][13] have been designed for specific environments. Hardware occlusion queries[10] as supported by most of the GPUs allow us to identify the visible objects in a scene. This is done by rendering the geometry to the depth buffer and using the occlusion query to identify the number of pixels the geometry is rendered to. By having a threshold on the number of rendered pixels, we can mark a geometry as visible or invisible. The bottleneck of using occlusion queries is the CPU has to wait for the result of the queries before rendering the geometry. We will see in the later sections on how this can be overcome by using two parallel rendering pipelines. However, there are a few cases where occlusion queries can cause an overhead as each query adds an extra draw call.

The Gigawalk project[12][6] presents a parallel occlusion culling method using 2 GPUs each running a different graphics rasterization pipeline and one or more CPU processors. This system runs three processes in parallel. Occluder Rendering (OR) process uses all visible geometry from previous frame as the occluder set renders that set into a depth buffer. This runs the first graphics pipeline. Scene Traversal, Culling and LOD selecting (STC) process computes the hierarchical z-buffer using the depth buffer computed by OR. It traverses the scene graph, computes the visible geometry and selects appropriate LODs. The visible geometry is used by RVG for the current frame and OR for the next frame. This runs on one or more processors. Rendering Visible scene Geometry (RVG) process renders the visible scene geometry computed by STC. This uses the second graphics pipeline.

A number of parallel rendering approaches [11] [4] have been introduced before that are based on multiple rendering pipelines. These parallel rendering concepts can be classified mainly as object-parallel, frame-parallel or screen-space-parallel. Specific examples include rendering every Nth frame or distributing primitives to different pipelines based on screen space. Some of these systems and APIs like Chromium[8], Garuda[9], NetJuggler[2], CGLX[3] and Equalizer[5] provide scalable rendering on shared memory systems.

The Garuda system[9] provides a scalable, geometry managed display wall. This is a cluster-based tiled display wall which uses an Adaptive Culling algorithm to determine the objects visible to each display-tile. A multicast-based protocol is used to transmit the geometry exploiting the spatial redundancy present especially on large tiled displays. A geometry push philosophy from the server helps keep the tiles

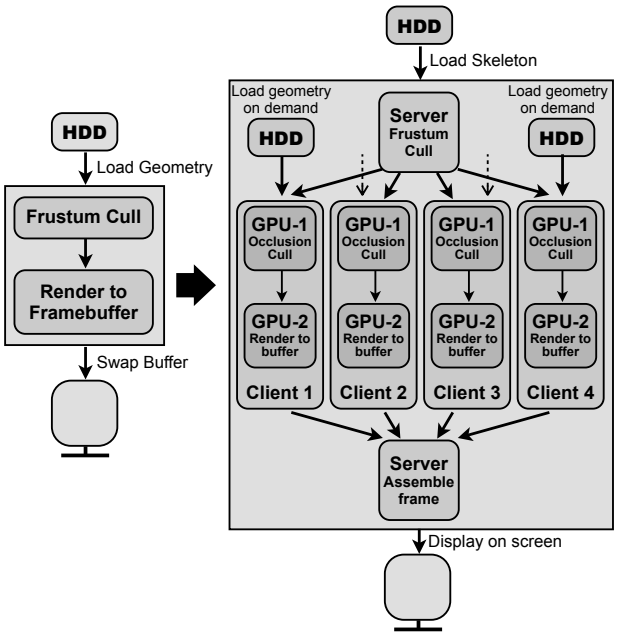

Figure 1: Regular graphics pipeline (left) and DMMR's parallel rendering pipeline (right)

in sync with one another.

Works like Gigawalk address rendering a large model interactively on a single system while CGLX, Garuda and Equalizer systems present methods to render large models on multiheaded displays for high resolution. We look beyond it focussing on massive models that are spatially large and detailed, with a lot of occlusion and can't fit into any single system.

3. DISTRIBUTED MASSIVE MODEL RENDERING

We present a distributed rendering technique to handle massive models. These models consist of spatially large and complex 3D environments for monuments. We distribute the rendering process among a cluster of computers where each system is pushed to its limits for rendering its sub-scene. This process uses a load based distribution so as to distribute the work evenly among the client nodes in the cluster. Since we handle the rendering in a distributed fashion there could be no upper limit on the size of the model as long as the cluster is scaled appropriately.

3.1 Parallel Rendering Framework

The distributed system is setup as a cluster of multi-GPU systems connected over network. This cluster consists of a server and multiple clients. The server and the clients communicate using MPI. These three modules run in parallel to each other.

- Server : The Server is a system with a CUDA enabled GPU. The Server runs the load balancing module, where the frustum is divided into sub-frustums and each client is assigned a sub-frustum. It also assembles the rendered sub-frames from clients into the final frame.

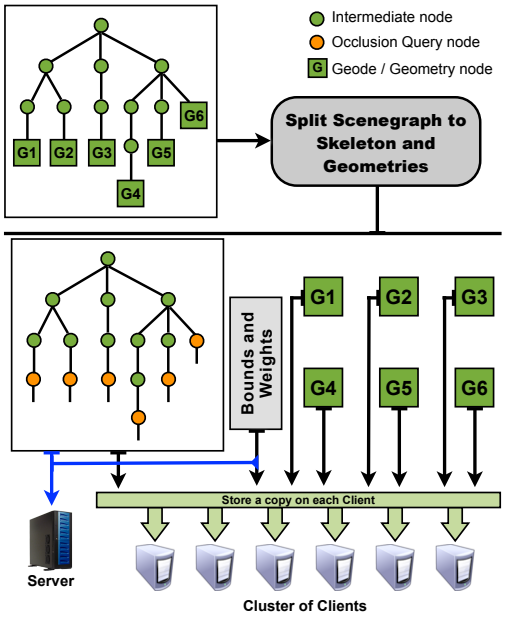

Figure 2: Preprocessing of Scenegraph

- Client : Each Client is a multi-GPU system with the first GPU supporting Hardware Occlusion queries with decent memory while the second GPU optimized for rendering with higher memory. Each client runs two different modules, visibility determination module and sub-frame rendering module, with each module following its own pipeline. The visibility determination module runs on the first GPU to determine the visible objects in the sub-frustum. The sub-frame rendering module runs on the second GPU which loads and renders the visible objects in the sub-frustum.

3.2 Scene Representation and Preprocessing

The entire digital model is represented as a scenegraph using the OpenSceneGraph API[1]. This is an object-based scene structure where the geometries are grouped as objects. The scene represents a collection of these objects that are spatially aligned. Some of these objects are marked as potential occluders. The geometry nodes are represented as Geodes in the OSG API.

The 3D model is preprocessed through a series of steps to be prepared for the distributed rendering process (as illustrated in Figure 2). The given model is converted to scenegraph and all the leaf nodes (geometry nodes/geodes) are indexed. For each geode the bounding sphere is computed. A bounding box is also computed if it occupies lesser volume compared to the bounding sphere. The weight of the geode is also computed based on the complexity of the geometry. Presently, we consider the triangle count of the geometry as its weight. All the bounding spheres, bounding boxes and the weights are saved in a bounds file. Each geode is saved as an independent geometry file. The geodes in the scenegraph are replaced with occlusion query nodes which also store the index of the geode it replaced. This geode stripped scenegraph is saved as a skeleton file. A list of occluder node indices is saved to occluders file.

Each client has a copy of these bounds, occluders, skeleton and geometry files residing on their local storage. The server stores a copy of bounds, occluders and skeleton files.

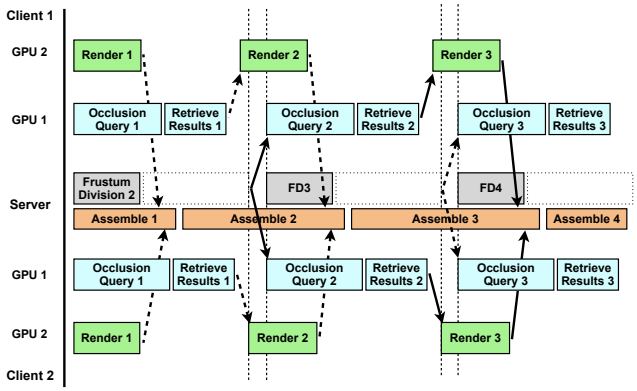

Figure 3: Distributed rendering pipeline

3.3 Distributed Rendering Pipelining

The regular graphics pipeline does the frustum culling followed by rendering. However, we split this pipeline into four parallel modules (Figure 1) where each module is assigned to a different GPU.

1. Load balanced frustum division : The server handles this module. Here the the scene is frustum culled and the frustum is divided into sub-frustums based on the objects in the frustum using a load balanced frustum division algorithm.

2. Visibility determination : This module runs on the first GPU of each client. Hardware Occlusion queries are used to identify the visible objects in each client's sub-frustum.

3. Sub-frame rendering : This module runs on the second GPU of each client. The visible objects identified by the previous module are rendered to a frame buffer with the corresponding sub-frustum parameters.

4. Assemble Frame : The sub-frames rendered in the previous module are transferred back to the server to be assembled into a complete frame buffer.

All the clients operate asynchronously and the three modules execute in parallel as illustrated in Figure 3. While the frustum culling of i^{th} frame is run on the server, the occlusion culling of $i\text{-}1^{th}$ frame is run on the first GPU of each client and the rendering of $i\text{-}2^{th}$ frame is done on the second GPU of each client, thereby executing the three modules in parallel.

3.4 Load Balanced Frustum Division

The server handles the frustum culling and the sub-frustum division of the scene (Figure 4). We load the skeleton scenegraph and the bounding spheres to identify the objects in the current frustum. Each object in the current frustum is assigned a precomputed weight which represents the work load for rendering the object with the net weight of the frustum being the total weight of all the objects in the frustum. This list of objects is then processed through a load balanced frustum division algorithm which identifies the split

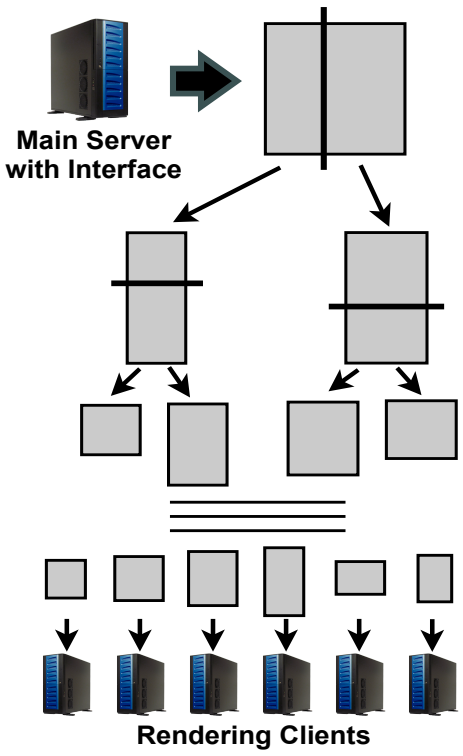

Figure 4: Division of frustum into sub-frustums

plane that best divides the objects in the frustum into two sub-frustums such a way that:

- The difference in loads of the sub-frustums is low that results in almost equal work; and
- The deviation of the load of each sub-frustum from half of the total load is low that reduces object duplication.

This process is in turn applied on each of the sub-frustums until we have a sub-frustum for each client. This splitting operation is done along the horizontal and vertical planes alternatively.

The final list of objects in each sub-frustum and the sub-frustum parameters are communicated to the specific client using MPI.

3.5 Visibility Determination Module

The rendering phase runs independently on each client in parallel. Each Client works on two GPUs using two parallel graphic pipelines (Figure 6). The first GPU uses a pipeline for identifying the visibility of the objects in its sub-frustum.

This pipeline loads the skeleton and the bounding boxes if available or the bounding spheres for each of the objects in the list received from the server. It uses the visible objects from the previous frame's sub-frustum as occluders. The lower LOD geometry of the visible objects from previous frame's sub-frustum that are present in the current sub-frustum are loaded and rendered to the depth buffer using the current sub-frustum parameters. The geometry of any potential occluder in the current sub-frustum is also loaded and rendered to the depth buffer. The rest of the objects in the current sub-frustum are tested for visibility using hardware occlusion queries on their bounding box/sphere.

Algorithm 1 Load Balanced Frustum Division algorithm

P = array of leftmost and rightmost bounds of nodes
Lw_i = total weight of nodes with P_i as leftmost bound
Rw_i = total weight of nodes with P_i as rightmost bound
L_i = array of nodes with P_i as leftmost bound
R_i = array of nodes with P_i as rightmost bound
nd = Number of sub-frustums required
$Min_i(f(i)) = i$ for which $f(i)$ is minimum

procedure DIVIDEFRUSTUM($nodes, direction, nd$)
  if $nd = 1$ **then return**
  end if
  Generate P by projecting the bounding spheres on horizontal or vertical axis based on $direction$
  n = total number of points
  $sort\ (P, Lw, Rw, L, R)$ with P as key
  $totalweight = \sum_{i=0}^{n} Lw_i$
  for $i = 0$ to n **do**
    // $exclusive\_scan$ on Lw of points from 0 to i
    $Lw_i = \sum_{j=0}^{i-1} Lw_j$

    // $exclusive\_scan$ on Rw of points from n to i
    $Rw_i = \sum_{j=n-1}^{i+1} Rw_j$
  end for
  $lower\_bound = i$ such that $Lw_{i+1} > totalweight/2$
  $upper\_bound = i$ such that $Rw_{i-1} > totalweight/2$
  $divide = Min_i(OptimalWeight(Lw_i, Rw_i, totalweight/2))$
    where $lower\_bound <= i <= upper\_bound$
  $leftsubfrustum = \uplus\{Ln_i\}\ \forall i \in [0, divide)$
  $rightsubfrustum = \uplus\{Rn_i\}\ \forall i \in (divide, n]$
  $divideFrustum(leftsubfrustum, !direction, nd/2)$
  $divideFrustum(rightsubfrustum, !direction, nd/2)$
end procedure

procedure OPTIMALWEIGHT(l, r, h)
  return $|l - h| + |r - h| + |l - r|$
end procedure

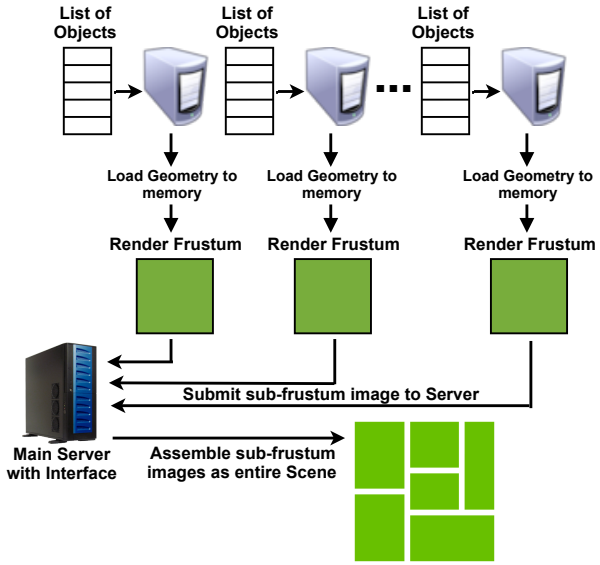

Figure 5: Overview of Rendering Cluster

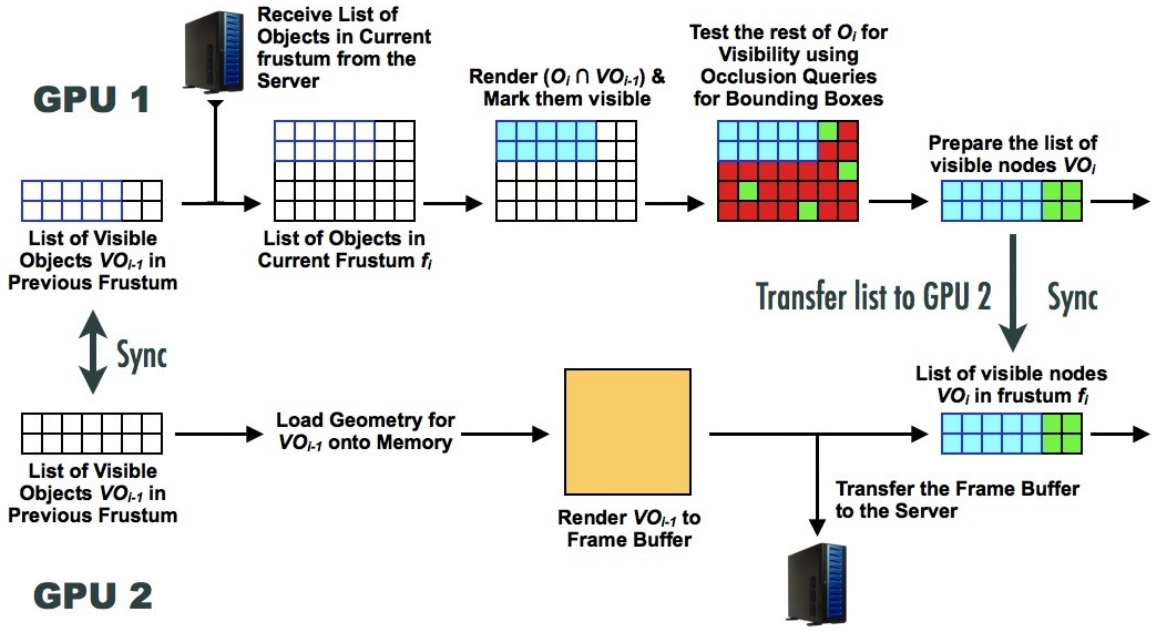

Figure 6: Occlusion culling and Sub-frame rendering modules on the GPUs of a client

The objects that passed the visibility test are marked visible for the next n frames assuming consistency between the consecutive frames. This helps in reducing the load of occlusion queries with a safe assumption that a geometry which passed the visibility threshold once will atleast be partially visible for the next n frames, where n is a configurable parameter. These marked objects will again be tested for visibility in their n^{th} frame if they still fall within the sub-frustum.

The list of visible objects is prepared comprising of the occluders in sub-frustum, nodes that passed the occlusion queries and already marked nodes that fall within the sub-frustum. This list is passed to the rendering module on the second GPU.

3.6 Sub-frame Rendering Module

The rendering module uses the second GPU on the client. This module receives the list of visible objects from the occlusion culling module. It loads the skeleton with all the occlusion query nodes disabled. The VBOs of the common visible objects from previous sub-frame are reused. The geometry of the rest of the visible objects are loaded as VBOs onto the GPU for efficient buffering and memory management. Textures, lighting effects and other required settings are applied at this level and the final sub-frame is rendered offscreen with the specified sub-frustum and viewport configuration to a frame-buffer. This frame-buffer is transferred to the server as compressed data stream.

3.7 Assemble Frame Module

The server acquires the compressed sub-frame-buffers from all the clients. These sub-frame-buffers are decompressed and assembled according to their offsets to form the complete rendered image of the scene.

3.8 Cache Management

We implemented a multi level cache management system. Each client maintains a buffer on the GPU in the form of VBOs. The next level of cache is maintained by the CPU on the ramdisk. The third level of caching remains on the HDD. Whenever a geometry not present on the GPU is to be loaded, the process tries to retrieve it from the ramdisk failing which it looks up for the geometry on the HDD. The geometry found on the HDD is loaded on the ramdisk and then sent to the GPU. The caching follows a LRU algorithm where the least recently used geometry is unloaded from the cache when it is full.

4. EXPERIMENTAL RESULTS

We performed experiments on two different clusters for various models. The clusters were setup using 5 different systems. System 1 has a Core 2 Duo processor, 2GB RAM with a GeForce GTX 480 GPU as GPU-1 and a Quadro FX 3700M as a GPU-2. System 2 has an i7 Quad Core processor, 6GB RAM and two Tesla C2070 GPUs. System 3 has an i7 Quad Core processor, 4GB RAM with a GeForce GTX 580 as GPU-1 and Tesla C2050 as GPU-2. System 4 has an Core 2 Duo processor, 2GB RAM with a GeForce GTX 280 as GPU-1 and a GeForce 8600 GTS as GPU-2. System 5 has a Core 2 Duo processor, 4GB RAM with a GeForce 8600GTS as GPU-1 and a Tesla S1070 as GPU-2. First cluster is a 2-client setup with System 1 as server using GTX 480 GPU and Systems 2 and 3 as clients. Second cluster is a 4-client setup with System 5 as server using Tesla S1070 GPU and Systems 1, 2, 3 and 4 as clients. The CPU cache on the clients is set dynamically to half the total size of objects in the model being rendered for the 2-client setup and to a quarter of the total size of objects for the 4-client setup. The clusters are setup on a Gigabit network with the

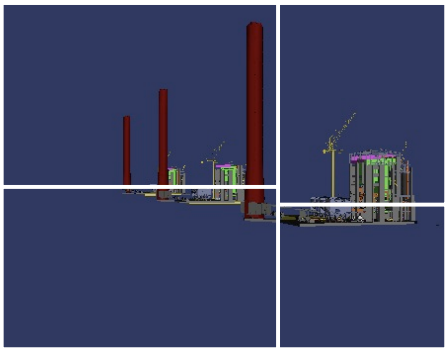

Figure 7: Tiling of a frame on 4-client cluster with power plants

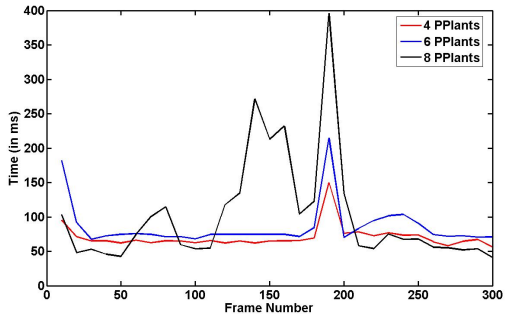

Figure 8: Frame Time in ms (averaged per 10 frames) for 4, 6 and 8 power plants models

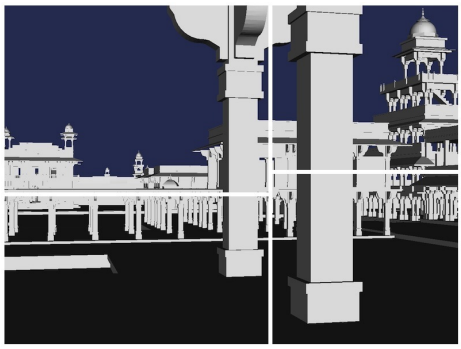

Figure 9: Tiling of a frame on 4-client cluster with Fatehpur Sikri models

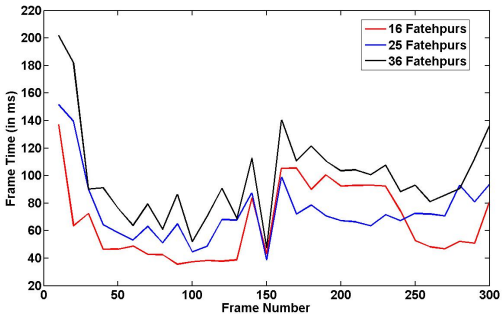

Figure 10: Frame Time in ms (averaged per 10 frames) for 10, 20 and 30 Fatehpur Sikri models

communications between the systems through MPICH2.

The rendering system is tested with two different sets of models. First set comprises of models made of different number of replicated Coal Power Plant models. Each power plant model is made up of approximately 12M triangles across 20,000 geometry nodes with 8 levels of hierarchy. A fixed path walkthrough (demonstrated in the video) of the model to make all instances of the power plant visible was used to measure the results. The second set comprises of models made of different number of replicated Fatehpur Sikri models. Each Fatehpur Sikri model is made up of approximately 4.8M triangles across 1113 geometry nodes. A fixed path walkthrough of the model (demonstrated in the video) is used for the tests.

The average runtimes are measured for different modules and the network transfer for both the models. The tables show average times over 300 frames and the graphs show average fps of 10 consecutive frames.

The three blocks in the tables, viz., Server, GPU 1 and GPU 2 represent the respective modules running on them. These 3 modules operate in parallel while synchronizing at the network communications. On the server, *'Receive and assemble sub-frames'* module runs on separate thread in parallel to the *'Frustum division'* module. It represents the effective frame rate of the whole system which includes the time spent waiting for the framebuffers from clients, transfer over network, decompression and assembling of the framebuffer. For both GPU 1 and GPU 2, the runtimes on all clients are provided which operate in parallel asynchronously. On GPU1 *'Enable occlusion queries'* module includes the time to load the occluder geometry and the bounding boxes of the objects in the sub-frustum, unload any geometry/bounding boxes of the objects from the previous sub-frustum that went out-of-bound and the time to issue occlusion queries. *'Retrieve occlusion queries'* represents the time to retrieve the occlusion query results and prepare the list of visible objects. On GPU 2, *'Load geometry to render'* is the time taken to load new geometry onto the memory before rendering to the frame buffer. *'Send to server'* includes the sub-image compression time and network transfer time to the server.

The effective frame time (Figures 8,10) averaged for every 10 consecutive frames, the runtime of the load balanced frustum division algorithm (Figures 12,13) and the time for preparing the occlusion queries (Figure 11) are shown for different models. The frustum division algorithm resizes the sub-frustum of each client every frame to balance the load of the objects in the frustum. When a large number of objects move in to the frustum in a new frame, the sub-frustum holding the newly entered objects loads more geometry than the previous frames and hence the corresponding frame takes more time to process. This behavior can be noticed in the graphs(Figure 8) at around 200^{th} frame where the time taken increases momentarily. The clients which are assigned the sub-frustums with new objects take more time to prepare the occlusion queries for the new objects which can be noticed in Figure 11. We could notice from Figures 8 and 10 that difference in the frame time is very little in general even as the complexity of the model scales up.

Table 1: Timings for multiple power plant(12M triangles) models on a cluster of 2 clients

	Tests	Four Plants (80k nodes)	Six Plants (120k nodes)	Eight Plants (160k nodes)
FPS and Frame time (average over 200 frames)		7fps / 136 ms	7fps / 147 ms	4-12fps / 150 ms
Server	Frustum Division	15.890 ms	11.705 ms	11.437 ms
	Send to GPU 1 over network	36.828 ms	35.287 ms	35.642 ms
	Receive & Assemble sub-frames	136.122 ms	147.131 ms	150.386 ms
GPU 1 (client1/client2)	Enable Occlusion Queries	8.87 / 9.99 ms	8.32 / 10.27 ms	7.96 / 9.70 ms
	Retrieve Query results	1.77 / 1.61 ms	1.68 / 1.74 ms	1.64 / 1.70 ms
	Send to GPU 2	16.06 / 26.26 ms	16.65 / 16.80 ms	16.02 / 17.35 ms
GPU 2 (client1/client2)	Load Geometry to render	0.44 / 0.29 ms	0.37 / 0.36 ms	0.34 / 0.32 ms
	Send to Server over network	16.48 / 11.96 ms	14.14 / 15.09 ms	16.12 / 12.75 ms

Table 2: Timings (in ms) for multiple power plant (12M triangles) models on a cluster of 4 clients

	Tests	Four Plants (80,000 nodes)	Six Plants (120,000 nodes)	Eight Plants (160,000 nodes)
FPS and Frame time		6-16fps / 78 ms	5-14fps / 86 ms	3-23fps / 101 ms
Server	Frustum Division	18.89 ms	20.08 ms	21.05 ms
	Send to GPU 1 over network	67.866 ms	79.1 ms	90.413 ms
	Receive & Assemble sub-frames	78.365 ms	85.774 ms	101.782 ms
GPU 1 (4 clients)	Prepare Occlusion Queries	4.7/33.8/4.9/29.6	6.4/48.1/10.8/45.3	97.8/60.7/80.7/69.2
	Retrieve Query results	1.3/2/1.3/2.1	1.4/2.1/1.4/2	1.7/2.8/2/2.7
	Send to GPU 2	5.8/7.9/5.4/7.7	6.3/10.7/6.9/8.7	14.3/15.2/16.1/14.4
GPU 2 (4 clients)	Load Geometry to render	3.1/4.4/0.8/0.7	4.6/4.7/14.2/10.9	30.2/8.3/28.8/12
	Send to Server over network	18.1/4.4/15.2/3.6	17/4.8/17.6/6.5	29.2/9.8/25.6/30.1

Table 3: Timings (in ms) for multiple Fatehpur Sikri models on a cluster of 4 clients

	Tests	16 Fatehpur (77M triangles)	25 Fatehpur (120M triangles)	36 Fatehpur (172M triangles)
FPS and Frame time		9-26fps / 64 ms	9-25fps / 79 ms	7-20fps / 93 ms
Server	Frustum Division	9.07 ms	16.18 ms	16.48 ms
	Send to GPU 1 over network	45.89 ms	44.52 ms	52.28 ms
	Receive & Assemble sub-frames	64.603 ms	79.321 ms	93.4 ms
GPU 1 (4 clients)	Prepare Occlusion Queries	1.7/4.1/7.5/3.7	7/8.2/4.8/9.1	3.7/12.8/10.1/13.5
	Retrieve Query results	0.2/0.4/0.2/0.4	0.4/0.8/0.4/0.7	0.5/1/0.6/0.9
	Send to GPU 2	8,2/17.8/9.7/19.8	7.8/6.7/11.5/13.7	9.9/10.7/9.7/14.9
GPU 2 (4 clients)	Load Geometry to render	1.5/0.7/10.7/0.5	11.4/0.8/5.1/0.5	6.6/0.9/11.8/0.6
	Send to Server over network	35.4/9.1/37.4/11.4	34.6/11.3/32.7/10.4	35.4/8.3/37.5/14

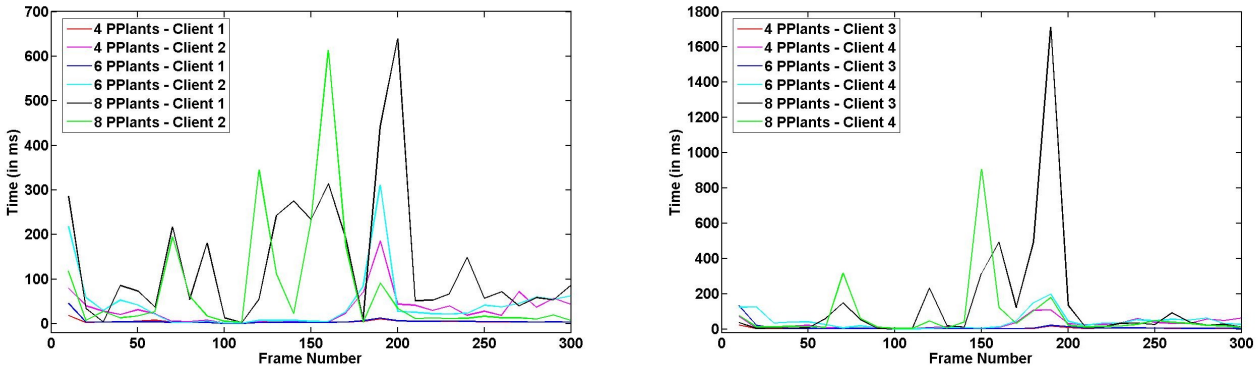

Figure 11: Time to prepare Occlusion Queries on Clients 1,2 (left) and Clients 3,4 (right) (averaged per 10 frames) for 4, 6 and 8 power plants models

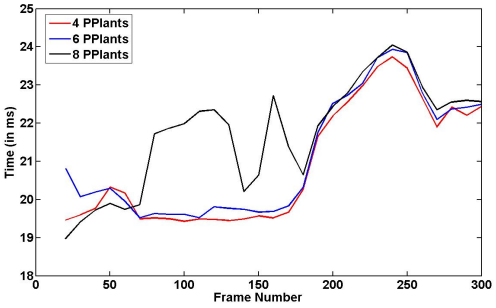

Figure 12: Load balanced frustum division (averaged per 10 frames) for 4, 6 and 8 power plants models

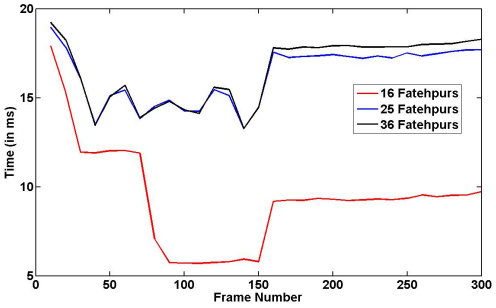

Figure 13: Load balanced frustum division (averaged per 10 frames) for 16, 25 and 36 Fatehpur Sikri models

The Power plant model is made of large number of objects while the Fatehpur Sikri model is made of lesser number of complex objects spread spatially. As described before this system is designed for massive models that complex as well as spatially large with much occlusion like the Fatehpur Sikri monuments. This can be seen by comparing the frame times of the Power Plant and the Fatehpur Sikri replicated models where the Fatehpur Sikri walkthrough performed much faster than the Power Plant walkthrough.

A single system with an i7 Quad core processor, 4GB RAM and a GeForce GTX 580 GPU was able to render only up to the 3 power plants model. When compared against this, our distributed rendering system performed about 6 times faster for the 3 power plants model on an average of 300 frames of the walkthrough.

5. CONCLUSIONS

In this paper, we presented a distributed rendering framework for a cluster of processors to handle truly massive models. Our method uses a client-server framework with the server partitioning the rendering load among the clients. This is a difficult problem when the assets required for rendering are truly bulky. We showed the results from rendering massive models using small clusters of processors. Our experience demonstrates that scalable rendering of massive models can be achieved using our approach when more clusters are available. However, our system can be used for rendering huge monuments at interactive rates for walkthroughs by the users.

6. ACKNOWLEDGMENTS

We thank the IDH project of DST, India for their partial financial support for this research work.

7. REFERENCES

[1] Openscenegraph : http://www.openscenegraph.org.

[2] J. Allard, V. Gouranton, L. Lecointre, E. Melin, and B. Raffin. Net juggler and softgenlock: Running vr juggler with active stereo and multiple displays on a commodity component cluster. In *Proceedings of IEEE Virtual Reality Conference 2002*, pages 273–274, 2002.

[3] K.-U. Doerr and F. Kuester. Cglx: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):320–332, May 2011.

[4] S. Eilemann, A. Bilgili, M. Abdellah, J. Hernando, M. Makhinya, R. Pajarola, and F. Schürmann. Parallel rendering on hybrid multi-gpu clusters. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 109–117, March 2012.

[5] S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):436–452, May 2009.

[6] C. Erikson, D. Manocha, and W. V. Baxter, III. Hlods for faster display of large static and dynamic environments. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, I3D '01, pages 111–120, New York, NY, USA, 2001. ACM.

[7] N. K. Govindaraju, A. Sud, S.-E. Yoon, and D. Manocha. Interactive visibility culling in complex environments using occlusion-switches. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, I3D '03, pages 103–112, New York, NY, USA, 2003. ACM.

[8] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 693–702, New York, NY, USA, 2002. ACM.

[9] Nirnimesh, P. Harish, and P. J. Narayanan. Garuda: A scalable, geometry managed display wall using commodity pc's. *TVCG*, 13(5):864–877, 2007.

[10] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 2005.

[11] R. Samanta, T. Funkhouser, and K. Li. Parallel rendering with k-way replication. In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, PVG '01, pages 75–84, Piscataway, NJ, USA, 2001. IEEE Press.

[12] A. Sud, N. K. Govindaraju, and D. Manocha. Gigawalk: Interactive walkthrough of complex environments. *Eurographics Workshop on Rendering (EGWR)*, 2002.

[13] S.-E. Yoon, B. Salomon, R. Gayle, and D. Manocha. Quick-vdr: Out-of-core view-dependent rendering of gigantic models. *TVCG*, 11(4):369–382, July 2005.