Ray Tracing of Large Models on a Multi-Projection Display

Vasco Costa João M. Pereira Joaquim A. Jorge
INESC-ID/IST
Lisboa, Portugal
{vasc,jap,jaj}@vimmi.inesc-id.pt

Resumo

O uso de écrans de grande dimensão, no nosso caso uma 'display wall' multi-projector, tem vindo a aumentar. Quando vários utilizadores precisam de visualizar uma cena ou quando um único utilizador necessita de obter uma visão alargada do problema e requer uma resolução mais elevada que a disponível utilizando os outros tipos de écran à sua disposição. É natural que as cenas visualizadas neste tipo de écran de alta definição tenham também uma resolução elevada. Para efeitos de avaliação do desempenho do sistema de visualização utilizamos modelos 3D com uma complexidade na ordem de dezenas de milhões de triângulos. O sistema funciona através da subdivisão do écran de grande dimensão em M grupos de amostras em que M é o número de máquinas no aglomerado de computadores. Cada amostra é sintetizada traçando raios com recurso a um algoritmo de subdivisão espacial por nós concebido. O sistema permite a visualização a ritmos interactivos de cenas deste tipo.

Abstract

The use of large scale displays, in our case a multi-projector display wall, has been increasing in many applications where multiple users need to visualize a scene or when a single user needs to have a comprehensive view of things and requires a larger resolution than otherwise available. Hence it is natural that the scenes to be viewed should have high resolution as well. For benchmarking purposes we used 3D models with a complexity in the order of tens of millions of triangles. The system works by subdividing a large screen into M groups of samples where M is the number of machines in the rendering cluster. Each sample is ray traced using a spatial subdivision algorithm of our own design. The system is able to render such scenes at interactive rates.

Keywords

Raytracing, Parallel rendering, Distributed applications

1. INTRODUCTION

The use of large displays is increasingly more common for several kinds of applications due to the increased amount of information possible to present on such displays, they also make it possible for more than one person to use a display for visualization or other purposes. i.e. such displays are important for applications which are collaborative in nature or require work on minute detail in several areas including architectural, automotive, biomedical, heritage and others.

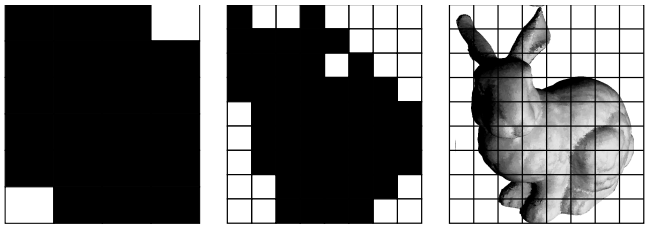
In our specific case we were interested in viewing large architectural or heritage pieces. For benchmarking purposes we selected a large scanned model from a standard dataset, namely the Thai Statue model, with 10 million triangles, from the Stanford archive.

To solve the specific issue of how to display the data we used the hardware resources available, namely a cluster of a dozen PCs, to render the model in a parallel fashion.

There are several standard solutions to visualize models on a cluster. These often use OpenGL [Shreiner09] based libraries such as Chromium [Humphreys08], OpenSceneGraph [Wang10] or OpenSG [Reiners02]. These libraries provide a large amount of flexibility for someone who wishes to write an application. However they have difficulty handling large datasets. We were also interested in evaluating the performance of an alternative rendering algorithm, in this case ray tracing, for this scenario.

Ray tracing backed by an acceleration structure has a performance less sensitive to triangle count and large amounts of overdraw which will be common in future architectural scenes we wish to visualize. The currently benchmarked model has limited depth complexity.

We also decided early on the planning phase that the parallelization algorithm should be as independent of the particular ray tracing acceleration structure or method as possible. Hence we decided, contrary to other parallel ray tracing implementations, to focus on the task of assigning the task of rendering each sample. Since the display has a large resolution of 4096×2304 pixels there should be no issue in finding enough work to assign to computing nodes of



(a) Level 2 macrocells (b) Level 1 cells (c) Grid with per cell triangle lists: *empty cells are compressed*

Figure 1: Multi-level static grid with macrocells for fast empty region traversal

the cluster.

Chapter 2 reviews some related work in the area focusing on grid acceleration structures for parallel ray tracing. In chapter 3 we present our multi-level static acceleration structure algorithm. Chapter 4 describes the system architecture: developed software and the hardware it runs on. We discuss the system test results in chapter 5. Finally we present our conclusions and pinpoint directions for future work.

2. PREVIOUS WORK

Ray tracing over a cluster of PCs [Wald02, Benthin06] has been an active area of research since the past decade. Ize et al worked on sort-middle cluster ray tracing using grids [Ize06] as well as how to optimize such a task using bounding volume hierarchies (BVH) on a cluster [Ize11] with an Infiniband interconnect. Their work divides acceleration structure (grid, BVH) cells among the cluster nodes so it is possible to view large models which would otherwise not fit in available memory. The issue with parallelizing an acceleration structure in such a fashion is the large performance penalty, up to $10\times$ slower, in their case compared to the case where it is simply replicated in full.

Grid acceleration structures [Fujimoto86] subdivide the space of the scene into cubically shaped cells. The 3D DDA cell traversal method for ray tracing introduced by Fujimoto et al. was later improved [Amanatides87]. This more recent traversal method is still commonly used today. Macrocells [Wald06] can be used to speed up empty space traversal during ray tracing.

Further improvements include efficient compression of the grid acceleration structure via the row displacement algorithm [Lagae08]. It reduces grid memory footprint by a factor of $20:1$ by compressing the empty grid cells which would otherwise use large amounts of memory (see Figure 2).

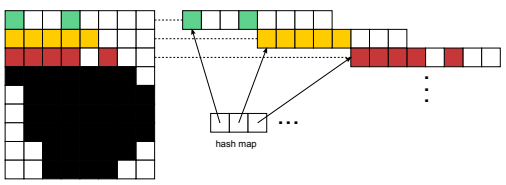


Figure 2: Row displacement compression.

3. MULTI-LEVEL STATIC GRIDS

In order to render complex scenes, with tens of millions of triangles, on current hardware it is necessary to use a spatial subdivision acceleration structure of some sort. In this way geometry which is either obscured or otherwise out of view is not processed resulting in interactive frame rates.

For this work we decided to use a multi-level static grid ray tracing acceleration structure of our own design. Since present hardware architectures have multiple levels of cache with varying bandwidth and latency it is advisable to reduce the working set to a minimum amount of memory. To further enhance rendering performance we speeded up cell traversal with a multi-level hierarchy of macrocells. The macrocells enable faster ray traversal ($2\times$ faster in scanned scenes) by skipping large empty regions of space (see Figure 1). In short the macrocell structure is a small 3D bit array which allows the trivial rejection of empty regions of space with minimum cache memory usage.

Improvements to the basic algorithm (macrocells, row displacement compression) enable the use of a finer grid resolution than would otherwise be possible with conventional algorithms which have severe memory bandwidth and cache thrashing issues [Costa10].

To pick the resolution of the rectangular axis aligned cells for a given scene in a grid we employ an heuristic. The heuristic usually involves the number of triangles in a scene, the scene's bounding box extents or volume, as well as a user defined density parameter. This parameter aims to make the number of cells M linear in regards to the number of triangles N in a scene, where:

$$M = \rho N \tag{1}$$

ρ is the grid density. The number of cells M is equal to the grid resolution $Mx \times My \times Mz$. Cubically shaped cells work best, and most heuristics take this into account. In this work a ρ value of 32 is used.

We use the following heuristic to pick the grid resolution:

$$M_i = S_i \sqrt[3]{\frac{\rho N}{V}} \qquad (i \in \{x, y, z\}) \tag{2}$$

S_i is the scene bounding box size in dimension i, V is the bounding box volume.

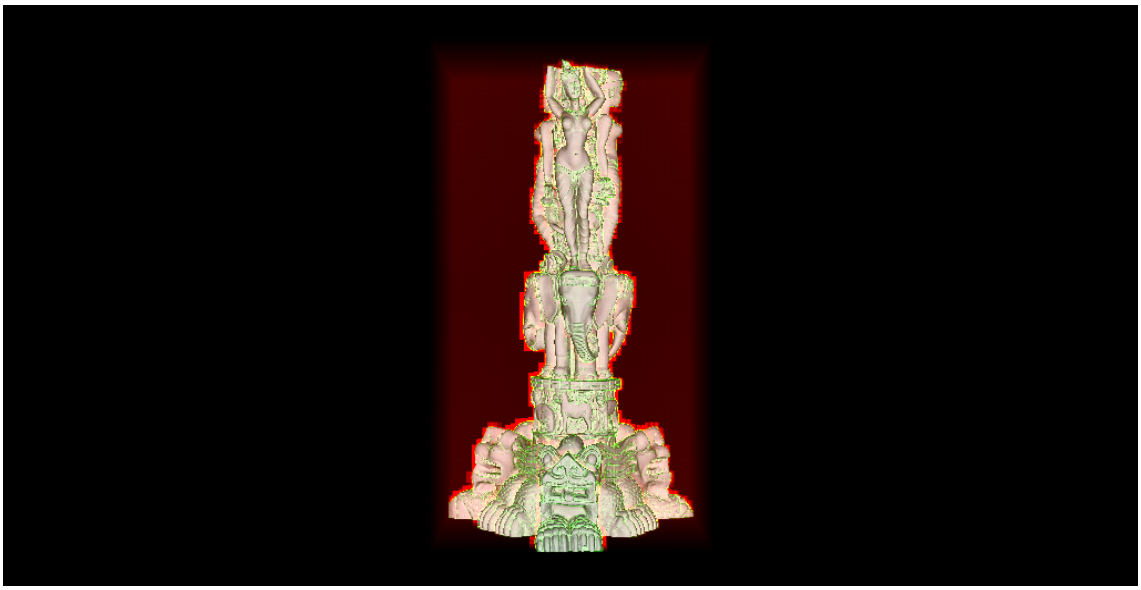


Figure 3: Visualization of the Thai Statue model (10 million triangles) which shows the ray tracing complexity for the given scene. Ray tracing was done using a multi-level static grid acceleration structure. Cell traversal costs are displayed in red, while triangle intersection costs are displayed in green.

The rendering time when using an acceleration structure of this type is approximately:

$$t_{rendering} = t_{cell\,traversals} + t_{triangle\,intersections} \quad (3)$$

Figure 3 shows an example of the ray tracing complexity of the scene.

4. SYSTEM ARCHITECTURE

We developed parallel ray tracing rendering system software which is run on a cluster of PCs.

We replicate the entire dataset on all cluster nodes. We are working on a Gigabit Ethernet cluster with much worse bandwidth and latency characteristics than the Infiniband used in [Ize11] so we chose not to use their kind of solution. This is done prior to visualization. Since the dataset is large but each node we have contains a limited amount of memory (1 GB DRAM) we needed to employ several forms of compression to enable the model to fit into the available memory space. We hope to upgrade the cluster powering the display soon which will enable us to further improve performance in the near future.

Below we describe the heuristics used and the software and hardware architectures of the implemented system in further detail.

4.1. WORKLOAD DIVISION ALGORITHMS

We divide the screen samples among the cluster nodes to split the workload. The cluster node assigned to work on that sample ray traces the full ray tree for the ray intersecting that point in the view area. We use a couple of heuristics to decide how to split the workload, namely the interleaved and tiled heuristics.

All of these heuristics are static in nature. They are easy to compute quickly but do not take the displayed scene characteristics into account. If we had feedback on the rendering cost of each work unit of interest (sample, tile) we could attempt to do dynamic load balancing during the rendering process. This could be done with a parallel scheduler of some sort.

4.1.1. INTERLEAVED HEURISTIC

In the interleaved heuristic we farm out samples in the following fashion: sample *i* is assigned to machine:

$$i \bmod M \quad (4)$$

where M is the number of machines in the cluster.

For a cluster with a dozen machines: sample 0 is worked by machine 0, sample 1 is worked by machine 1, ..., sample 12 is worked by machine 0, sample 13 is worked by machine 1, ...

4.1.2. TILED HEURISTIC

In the tiled heuristic we subdivide the screen into tiles of 8×8 samples each. This should allow for increased memory coherency during ray tracing compared to the previous interleaved heuristic which has worse memory coherency in particular for clusters with large numbers of machines where samples are further apart.

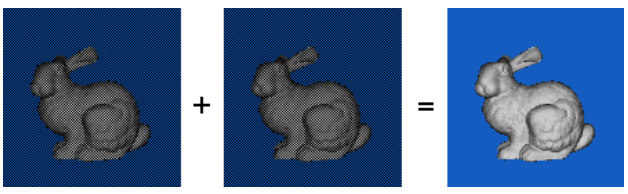


Figure 4: Interleaved heuristic sample distribution for a cluster with two nodes.

In the tiled heuristic we farm out tiles in the same fashion as we farm out samples in the interleaved heuristic: tile *t* is assigned to machine:

$$t \bmod M \tag{5}$$

where *M* is the number of machines in the cluster.

For example, for a cluster with a dozen machines: tile 0 is worked by machine 0, tile 1 is worked by machine 1, ..., tile 12 is worked by machine 0, tile 13 is worked by machine 1, ...

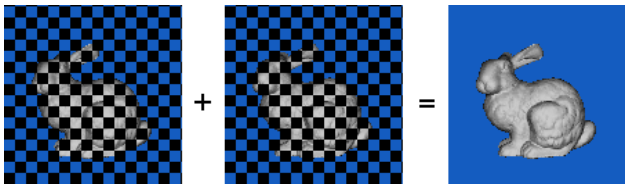


Figure 5: Tiled heuristic sample distribution for a cluster with two nodes.

4.2. SOFTWARE ARCHITECTURE

All task assignment work is performed by the Compositor Server (CS). The CS runs on a separate machine. Rendering machines connect to the CS as they are brought online.The CS is resilient to rendering machine breakdowns and allows dynamic joining and leaving of rendering processes.

Each rendering machine contains a process which does the rendering of the samples it is assigned to. Upon the beginning of each frame the CS sends a packet to each rendering machine with the instructions on which samples it needs to process. The results are then sent back to the CS as soon as work on the frame is finished. This ensures there is no visible tearing or other rendering glitches.

Once a frame is finished rendering the CS outputs the results to the screen, which may be the display wall, or any other OpenGL rendering surface.

The software was programmed in ANSI C++ without use of machine specific assembly instructions or intrinsics.

The scene memory usage can be computed as requiring 12 bytes per triangle for storing vertex index information (three machine words for each vertex index), plus 12 bytes for each vertex (three floating point numbers per coordinate). In this way we can reduce the storage requirements for those mesh triangles which have shared vertexes. Ray/triangle intersections are computed using the Möller-Trumbore [Möller05] intersection algorithm because it does not require any additional memory.

4.3. COMPOSITOR PACKETS

The compositor server interacts with the rendering servers using two kinds of packets as shown in Figures 6, 7.

The output samples are sent as uncompressed RGB data using one byte per color component.

machine-id	: int
machines	: int
eye	: float[3]
ll	: float[3]
u	: float[3]
v	: float[3]
xres	: int
yres	: int

Figure 6: Packets written to each rendering process.

packet-size	: int
samples-rgb	: byte[packet-size]

Figure 7: Packets read from each rendering process.

4.4. HARDWARE ARCHITECTURE

The rendering cluster contains twelve PCs with a Pentium 4 CPU at 3.0 GHz each with 1 GB of RAM. The compositor server runs on a similarly specced machine. The cluster machines run the Linux operating system.

The cluster is connected via Gigabit Ethernet. The available network bandwidth from the rendering machines to the compositor server was benchmarked using *iperf* at 680 Mbits/sec.

The display wall is a canvas illuminated by twelve projectors with a 4×3 geometry. Each projector runs at 1024×768 resolution. The canvas surface has dimensions of $4.00m \times 2.25m$.

5. RESULTS AND DISCUSSION

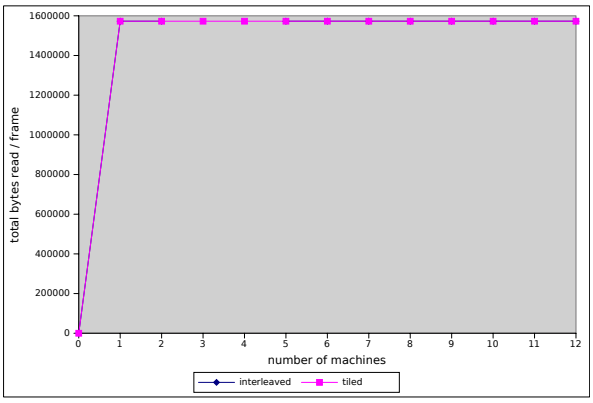
For each test statistics were gathered over an interval of five seconds of rendering in order to collect enough samples to reduce measuring error.

The test scene consisted of the ten million triangle Thai Statue from the Stanford 3D Scanning Repository. This scene is large enough to be representative of the kinds of scenes we wish to be able to visualize in the future.

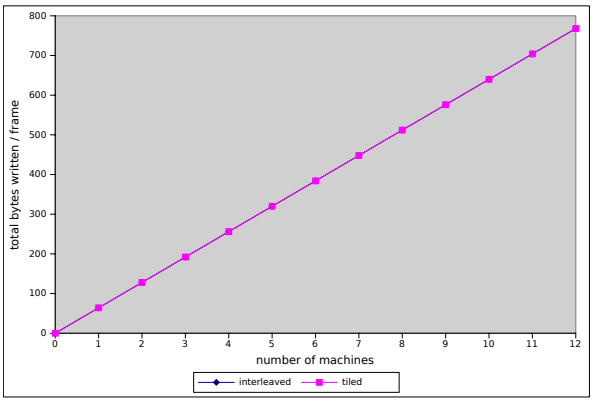
The tests were run using zero to twelve rendering machines in action to measure the speedup achieved from adding more machines to the system. Running the tests with zero ray tracing machines connected to the compositor server also allowed us to measure the speed at which it is able to send frames to the screen without any time being spent in ray tracing whatsoever.

Each frame has a resolution of 1024×512 with one sample per pixel. A frame is then scaled up to 4096×2304 screen resolution using the bilinear filtering facilities provided by OpenGL. This frame resolution was selected because our version of OpenGL does not support non power of two textures and this provided a similar aspect ratio to the actual screen resolution. This resolution also allows us to achieve interactive ray tracing frame rates for the scene in question.

As expected the bandwidth incoming to the compositor server remains constant since incoming data is composed of sampled pixels resulting from the ray tracing algorithm

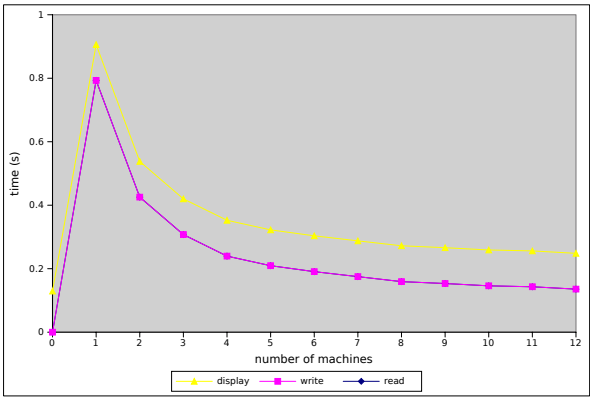


(a) Bandwidth incoming to compositor server

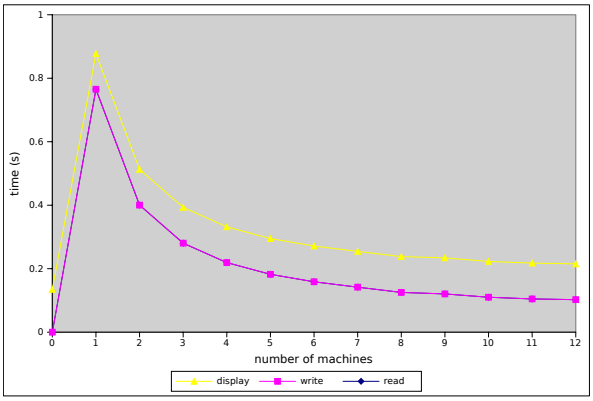


(b) Bandwidth outgoing from compositor server

Figure 8: The above charts show the bandwidth used by the compositor server while rendering a single frame.



(a) Interleaved rendering algorithm



(b) Tiled rendering algorithm

Figure 9: The above charts show the time required to render a single frame using each of the two heuristics.

which are $xres \times yres \times 3$ in size no matter what the number of machines in the cluster is. Outgoing bandwidth scales linearly with the number of machines in the cluster since the compositor server sends one packet to each machine telling it which rays it must render.

Running the tests showed we have a bottleneck sending each frame from the compositor server to the large screen display. This can clearly be seen in Figure 9. Just to display a blank screen takes a whole $114ms$ which results in a frame rate of 8.77 FPS.

This means the rendering time for one frame will always be inferior to this value. This requires more in depth investigation of our setup. We are using Chromium for doing the 2D frame scaling and screen blitting operations and this may be the bottleneck given the large size of the texture and/or the way we are displaying it. For the tiled rendering ray tracing algorithm it gets so ridiculous it takes more time to display the frame to the screen than to ray trace the whole scene.

The speedup is clearly sublinear. This may be due to a bottleneck in the compositor server which is not multi-threaded: the machine running it does not support the OpenMP parallelization primitives that we have in our implementation. Yet there is a clear performance improvement even at the larger cluster sizes.

The goal of achieving speedup versus a single machine is clearly achieved since the ray tracing rendering operation is around $4\times$ faster with the whole cluster rather than a single machine. However the performance speedup peters out as the number of machines in the cluster grows. Linear ray tracing speedup is observed with two machines but speedup starts decreasing as more machines are added to the configuration.

6. CONCLUSIONS AND FUTURE WORK

Clearly the cause of the bottleneck in displaying a frame to the multi-projector screen must be figured out and solved before any further improvements in ray tracing performance are to be attempted. It presently constitutes the majority of the time spent during rendering in our implementation.

Another thing which needs to be attempted is multi-threading the compositor server. The compositor server must handle multiple ray tracing machines at the same time and the current implementation may be the cause of the sublinear ray tracing performance speedup. The tiled rendering heuristic has proved superior but globally the results are not as impressive as expected in the current implementation.

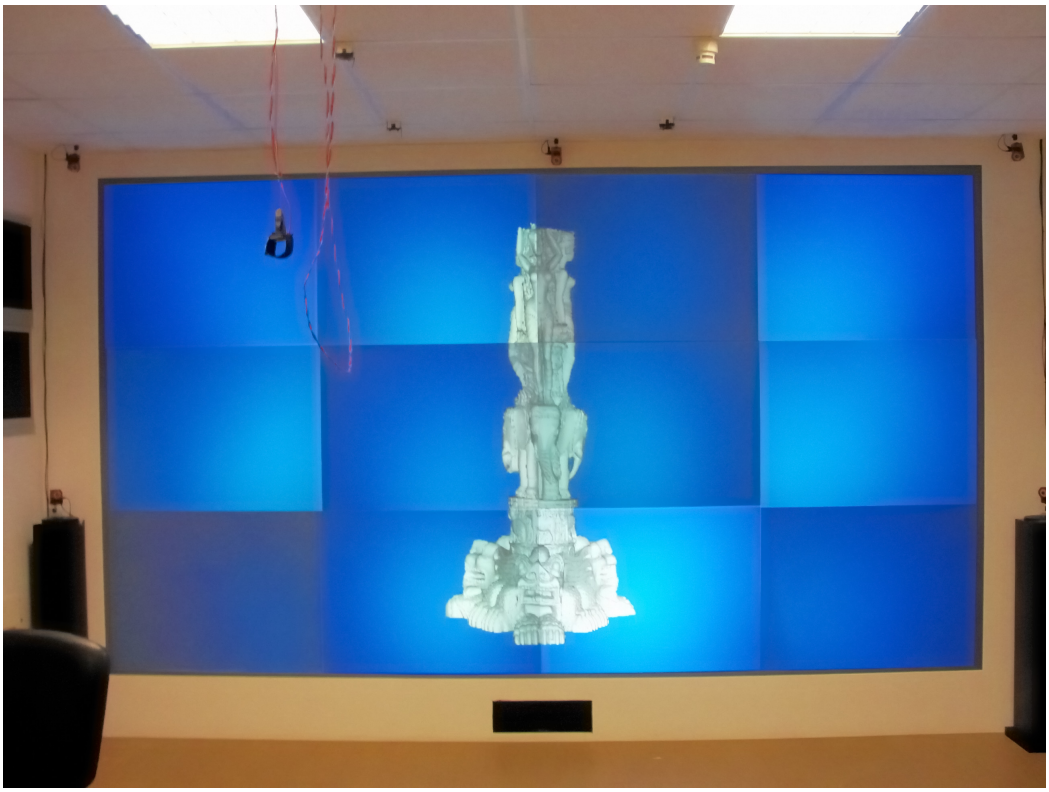


Figure 10: Rendering of the Thai Statue (10 Mtri) on the $4.00m \times 2.25m$ display wall. The hardware powering this application consists of a ray tracing cluster of twelve machines with the screen compositor server running on a separate machine.

7. ACKNOWLEDGEMENTS

We would like to thank the Stanford 3D Scanning Repository for the Thai Statue model.

This work was supported by national funds through FCT - Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2011.

8. REFERENCES

[Amanatides87] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *Proceedings of EUROGRAPHICS*, volume 87, pages 3–10, 1987.

[Benthin06] C. Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, 2006.

[Costa10] V. Costa, J. Pereira, and J. Jorge. Multi-level hashed grid construction methods. In *WSCG*, 2010.

[Fujimoto86] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray-tracing system. *Computer Graphics and Applications, IEEE*, 6(4):16–26, 1986.

[Humphreys08] 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 *ACM SIGGRAPH ASIA 2008 courses*, page 43. ACM, 2008.

[Ize06] T. Ize, I. Wald, C. Robertson, and S.G. Parker. An evaluation of parallel grid construction for ray tracing dynamic scenes. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 47–55, 2006.

[Ize11] T. Ize, C. Brownlee, and C.D. Hansen. Real-time ray tracer for visualizing massive models on a cluster. In *Proceedings of the 2011 Eurographics Symposium on Parallel Graphics and Visualization*, 2011.

[Lagae08] Ares Lagae and Philip Dutré. Compact, fast and robust grids for ray tracing. *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering)*, 27(8), 2008.

[Möller05] T. Möller and B. Trumbore. Fast, minimum storage ray/triangle intersection. In *International Conference on Computer Graphics and Interactive Techniques*. ACM Press New York, NY, USA, 2005.

[Reiners02] D. Reiners. *OpenSG: A scene graph system for flexible and efficient realtime rendering for virtual and augmented reality applications*. PhD thesis, Darmstadt University, 2002.

[Shreiner09] D. Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Addison-Wesley Professional, 2009.

[Wald02] I. Wald, C. Benthin, and P. Slusallek. A flexible and scalable rendering engine for interactive 3d graphics. *Computer Graphics Group, Saarland University*, 2002.

[Wald06] I. Wald, T. Ize, A. Kensler, A. Knoll, and S.G. Parker. Ray tracing animated scenes using coherent grid traversal. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 485–493. ACM, 2006.

[Wang10] R. Wang and X. Qian. *OpenSceneGraph 3.0*. Packt Publishing, 2010.