Interactive Distributed Ray Tracing of Highly Complex Models

Ingo Wald, Philipp Slusallek, Carsten Benthin
Computer Graphics Group,
Saarland University, Saarbruecken, Germany
{wald,slusallek,benthin}@graphics.cs.uni-sb.de

Abstract. Many disciplines must handle the creation, visualization, and manipulation of huge and complex 3D environments. Examples include large structural and mechanical engineering projects dealing with entire cars, ships, buildings, and processing plants. The complexity of such models is usually far beyond the interactive rendering capabilities of todays 3D graphics hardware. Previous approaches relied on costly preprocessing for reducing the number of polygons that need to be rendered per frame but suffered from excessive precomputation times — often several days or even weeks.
In this paper we show that using a highly optimized software ray tracer we are able to achieve interactive rendering performance for models up to 50 million triangles including reflection and shadow computations. The necessary preprocessing has been greatly simplified and accelerated by more than two orders of magnitude. Interactivity is achieved with a novel approach to distributed rendering based on *coherent ray tracing*. A single copy of the scene database is used together with caching of BSP voxels in the ray tracing clients.

1 Introduction

The performance of todays graphics hardware has been increased dramatically over the last few years. Today many graphics applications can achieve interactive rendering performance even on standard PCs. However, there are also many applications that must handle the creation, visualization, and manipulation of huge and complex 3D environments often containing several tens of millions of polygons [1, 9]. Typical examples of such requirements are large structural engineering projects that deal with entire buildings and processing plants. Without special optimization techniques these environments will stay well below interactive rendering performance.

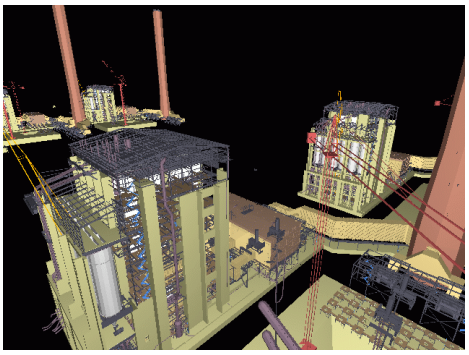

Fig. 1. Four copies of the UNC power-plant reference model with a total of 50 million triangles. In this view a large fraction of the geometry is visible. At 640x480 pixels the frame rate is 3.4 fps using seven networked dual Pentium-III PCs.

In the past these models were usually handled on a component by component basis, as the sheer data volume prohibited any interactive visualization or manipulation of the models as a whole. However,

there are several scenarios that require the interactive visualization and manipulation of entire models. For instance, design reviews and simulation-based design must often deal with the complex interrelations between many components of the model, such as a large processing plant involving many industrial robots and transport devices. Many interactive tasks benefit greatly from the ability to instantly inspect any aspect of the entire model, such as walking or flying around the model as a whole and then zooming in on relevant details.

A minimum requirement for achieving interactivity is the spatial indexing of the geometry. This allows to limit rendering to visible parts of the model by using view frustum and occlusion culling (e.g. [4, 22]). Unless the model has been organized in such a way already, spatial indexing requires sorting the geometry spatially in a relatively simple and efficient preprocessing phase.

Beyond spatial indexing rendering can be improved by performing additional precomputation: computing and rendering only the potentially visible set (PVS) of geometry, creating and selecting levels-of-detail (LOD), simplifying the geometry, and replacing distant geometry using image-based methods. Aliga et al. [1] created a framework using all these methods to achieve interactive performance for our reference power-plant model (see Figure 1).

This advanced preprocessing, however, is very costly and for most cases cannot be fully automated yet. Preprocessing time was estimated to be 3 weeks for the complete model of only a single power-plant from Figure 1.

Because of this tremendous overhead, alternatives need to be found that do not require such costly preprocessing in order to achieve interactivity. Ray tracing is an obvious candidate as it only relies on spatial indexing for efficient rendering and features built-in view frustum and occlusion culling. It is known to have logarithmic scalability in terms of models size and also scales well with available computational resources due to being "embarrassingly parallel".

However, ray tracing is known for its high computational cost and is not usually associated with interactive rendering. Fortunately, this situation is changing rapidly as highly optimized ray tracing implementations become available [20, 14].

1.1 Interactive Ray Tracing on Standard PCs

The *coherent ray tracing* approach by Wald et al. [20] achieves high rendering speed both through a new ray tracing algorithm and low-level optimizations. The latter include simplifying and optimizing code paths, optimizing data structures for caching, and using SIMD extensions (Single Instruction – Multiple Data, such as Intels SSE [6]) for data-parallel implementations of basic ray tracing algorithms. This implementation is limited to triangles only but offers arbitrary shading computations through dynamically loadable modules.

The new algorithm improves on basic recursive ray tracing by making significantly better use of coherence through reordering rays and tracing, intersecting, and shading them in packets of four or more rays in SIMD fashion. This is similar in spirit to [16]. It reduces the memory bandwidth proportionally to the number of rays in a packet because data needs to be fetched only once for the whole packets instead of once for each ray. It also improves caching behavior through better data locality.

By tracing rays in packets the usual depth-first ray tracing algorithm is essentially reordered to be breadth-first within each packet. This reordering can exploit the coherence among adjacent eye rays as well as among shadow and other secondary rays.

With all these optimization the coherent ray tracing algorithm runs almost com-

pletely within the data caches of the CPU, thus achieving speedup factors between 11 to 15 (!) compared to conventional ray tracers. It challenges the performance of high-end graphics hardware already for scene complexities of more than half a million triangles and moderate screen resolutions of 512^2 using only a single Intel Pentium-III CPU [20].

1.2 Interactive Distributed Ray Tracing

Due to the "embarrassingly parallel" nature of ray tracing these results scale well with the use of multiple processors as long as they all get the necessary bandwidth to the scene database. However, contrary to other approaches [10, 14, 1] our target platform is not an expensive shared-memory supercomputer but the inexpensive cluster of workstations (CoW) that is commonly available everywhere. Unless the complete data base is replicated on each machine the bandwidth of the network limits the performance for distributed ray tracing in this scenario.

We use the classic setup for distributed ray tracing using a single master machine responsible for display and scheduling together with many working clients that trace, intersect, and shade rays. The main challenges of this approach are efficient access to a shared scene data base that can contain several GB of data, load balancing, and efficient preprocessing.

We solve these issues with a novel approach that is again based on exploiting coherence using the same basic ideas as in [20] but on a coarser level. Our main contributions are:

Scene subdivision In a preprocessing step a high-level BSP-tree is built while adaptively subdividing the scene into small, self-contained voxels. Since preprocessing is only based on the spatial location of primitives it is simple and fast. Each voxel contains the complete intersection and shading data for all of its triangles as well as a low-level BSP for this voxel. These voxels form the basis for our explicit cache management.

Scene cache management The complete preprocessed scene is stored on a server only once and all clients request voxels on demand. The clients explicitly manage a cache of voxels, thus exploiting coherence between rays (within the rays of a single packet and between multiple adjacent packets) and in time (between similar rays in subsequent frames).

Latency hiding By reordering the computations we hide some of the latencies involved in demand loading of scene data across the network by continuing computations on other rays while waiting for missing data to arrive. This approach can easily be extended by prefetching data for future frames based on rays coarsely sampling a predicted new view.

Load balancing We use the usual task queue approach based on image tiles for load balancing. Instead of randomly assigning image tiles to clients we try to assign tiles to clients that have traced similar rays in previous frames. This approach maximizes cache reuse across all clients. Ideally this leads to the working set (visible geometry) being evenly distributed among the caches of all clients.

The paper is organized as follows: we start with a review of related work in the next section before presenting and discussing the main issue of distributed data management in Section 3. Section 4 describes our preprocessing algorithm, which is followed by a discussion of our load balancing algorithm in Section 5. Results are presented in Section 6 before we conclude and offer suggestions for future work in Section 7.

2 Related Work

Regarding the visualization of large models, the most closely related previous work is the UNC "Framework for Realtime Walkthrough of Massive Models" [1]. In particular we have chosen to directly compare our results with the performance published in this paper. The UNC framework focuses on walkthroughs using high end graphics hardware and a shared-memory multiprocessor machine. It consistently achieves interactive frame rates between 5 and 15 frames per second on an early SGI Onyx with four R4400 and an InfiniteReality graphics subsystem.

The UNC framework uses a combination of different speedup techniques. The largest effect is due to the replacement of distant geometry by textured depth meshes [1, 19, 3], which results in an average reduction of rendered polygons by 96%. This reduction is due to occlusion as well as sparse resampling of the environment with image-based methods. Both effects are implicit in ray tracing, even though the resampling in ray tracing is dynamic and uses the original geometry instead of rendering a smaller simplified scene. The resulting scintillations or temporal noise can at least partially be resolved by a temporal coherent sampling strategy [8].

Another reduction by 50% each resulted from view frustum and level-of-detail (LOD) selection. While the first is again an implicit feature of ray tracing, we do not implement LODs even though they could easily be used once they are generated [5]. This is a typical time/image quality trade-off because not using LOD increases aliasing but avoids the long preprocessing times for LOD creation.

Finally, occlusion culling based on hierarchical occluder maps (HOM) [22] for the near geometry reduces the number of rendered polygons by another 10%. Again occlusion culling is implicit in a ray tracer and does not require additional preprocessing.

The main drawback of the UNC approach is the tremendous preprocessing time — estimated to be three weeks for a single copy of the power-plant model. The technique is also less scalable both in terms of model size (the preprocessing is apparently super-linear) and graphics performance, where performance of ray tracing can easily be scaled by adding more client PCs.

Recently Parker et al. [14, 12, 13] demonstrated that interactive frame rates could also be achieved with a full-featured ray tracer on a large shared-memory supercomputer. Their implementation offers all the usual ray tracing features, including parametric surfaces and volume objects, but is carefully optimized for cache performance and parallel execution in a non-uniform memory-access environment. They have proven that ray tracing scales well in the number of processors in a shared memory environment, and that even complex scenes of several hundred thousand primitives could be rendered at almost real-time frame rates.

A similar system has been realized by Muuss [9, 10]. It used CSG objects as its primitives, which makes direct comparisons difficult. It also uses an optimized general ray tracer and was able to render highly complex models at a few frames per second on high-end shared-memory supercomputers with up to 96 CPUs. However, both this and the Utah system require an expensive shared-memory machine, while we concentrate our effort on low cost PCs in standard network environments.

Memory coherent ray tracing by Pharr et al [15] has also been able to efficiently render highly complex objects by exploiting coherence between rays. In addition to basic ray tracing their system also implements global illumination effects through path tracing [7].

We share the basic idea of splitting the scene into smaller voxels and using these for manual caching. However, our usage of the voxel structure is quite different, as

Pharr et al. performs significantly more reordering and scheduling of computations. In their system intersection computations are scheduled based on voxels: all rays that have entered a voxel so far are intersected in batches with the geometry in this voxel and rays are forwarded to adjacent voxels if there is no hit. The intersection computations for a voxels are scheduled based on the number of rays waiting, their weight, the amount of geometry, the state of the geometry cache, and other factors.

The system made it possible to ray trace scenes of up to 50 million triangles, but was far from realtime with rendering times still in the order of several hours for images of moderate size. On the other hand this system was mainly designed for generality and not for highest performance.

Another difference to our approach is the regular structure of the scheduling grid and the local forwarding of rays. The first issue results in large numbers of voxels even in empty regions of the scene (particularly relevant in the power-plant model) and the second has a significant overhead in tracing individual rays through empty space. By tracing packets of rays in parallel until they all terminate we eliminate this overhead while our BSP hierarchy better adapts to the local structure of the scene.

Of course, there has been a tremendous amount of previous work on parallel and distributed ray tracing in general. Detailed surveys can be found in [2, 18, 17]. Most of the techniques used in our ray tracing engine have been proposed in one or another way in previous publications, but never in the combination and with the optimizations as presented here. To our knowledge this is the first time anything close to interactive performance has been reported for distributed ray tracing with models of this size.

3 Distributed Data Management

The main problem we had to deal with for highly complex scenes are related to file size, limited address space, network distribution of the model data, and stalls due to demand loading. File size is not really a problem any more as most platforms (Linux in particular) are now supporting file sizes beyond 2 GB. More problematic is the limited virtual address space on 32 bit architectures such as Intel's Pentium CPUs.

In the original implementation of coherent ray tracing [20] we created a single binary file containing the model. It used the main memory layout so that we could directly map the entire file into our address space using the Unix mmap-facilities. However this is no longer possible with files larger than the supported address space. One possible solution would be to map only parts of a larger file and change the mappings on demand (essentially using a cache of mappings).

On the other hand we did not want to replicate the entire model of several GB on each of our client machines. This means that demand loading of mapped data would be performed across the network with its low bandwidth and large latency. While this approach is technically simple by using mmap across an NFS-mounted file system, it drastically reduces performance for large models. For each access to missing data the whole ray tracing process on the client is stalled while the operating system reads a single memory page across the network.

Even stalling for a few milliseconds due to network latency is very costly for an interactive ray tracer: Because tracing a single ray costs roughly one thousand cycles [20], we would lose several thousand rays for each network access. Instead we would like to suspend work on only those rays that require access to missing data. The client can continue working on other rays while the missing data is being fetched asynchronously. However, this approach is not possible with the automatic demand loading facilities of the operating system unless large numbers of threads would be used, which would be

infeasible.

3.1 Explicit Data Management

Instead of relying on the operating system we had to explicitly manage the data ourselves. For this purpose we decompose the models into small voxels. Each voxel is self-contained and has its own local BSP tree. In addition, all voxels are organized in a high-level BSP tree starting from the root node of the entire model (see Figure 2). The leaf nodes of the high-level BSP contain additional flags indicating whether the particular voxel is in the cache or not.

If a missing voxel is accessed by a ray during traversal of the high-level BSP, we suspend this ray and notify an asynchronous loader thread about the missing voxel. Once the data of the voxel has been loaded into memory by the loader thread, the ray tracing thread is notified, which resumes tracing of rays waiting on this voxel. During asynchronous loading, ray tracing can continue on all non-suspended rays currently being processed by the client. More latency could still be hidden by deferring shading operations until all rays are stalled or a complete tile has been traced. We use a simple least-recently-used (LRU) strategy to manage a fixed size geometry cache.

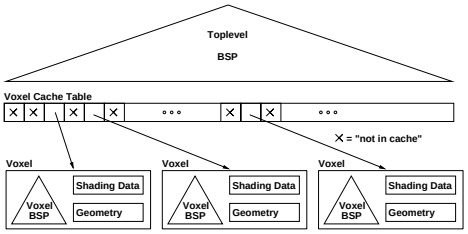

Fig. 2. The data structure used to organize the model data. Voxels are the smallest entity for caching purposes. Their average compressed size is roughly 75 KB.

3.2 Compressed Data

We had to come up with a reasonable compromise between the file size of voxels and the overhead through replication of triangle data. That compromise resulted in an average file size of voxels of 250 KB. With files of this size the voxel loading time is strongly dominated by the amount of data transfered over the network. This means that reducing the file size would also reduce the loading time. We pack our voxels using a method that allows fast and space/cache efficient decompression using the LZO compression library [11].

Though this compression is more optimized towards speed, its compression ratio is approximately 3:1 for our voxel data. Decompression performance is significantly higher than the network bandwidth, taking at most a few hundred microseconds, thus making the decompression cost negligible compared to the transmission time even for compressed voxels.

3.3 Shared Voxel Cache

All the PCs in our system are dual-processor PCs and run two ray tracing threads in parallel. In addition to a good price/performance ratio, it offers the additional advantage that network bandwidth can be reduced by up to a factor of two: whenever data is loaded, it is made available to both threads. Of course, this requires that both threads share the same voxel cache. In order to keep overhead as low as possible all cache management functionality is bundled in a third cache management and voxel fetcher thread, which shares the address space with the rendering threads.

4 Preprocessing

The total size of a single power-plant model is roughly 2.5 GB after preprocessing including BSP-trees, replicated triangles, and shading data. Due to this large data size we need an out-of-core algorithm to spatially sort and decompose the initial model.

The algorithm reads the entire data set once in order to determine its bounding box. It then recursively determines the best splitting plane for the current BSP node and sorts all triangles into the two child nodes. Triangles that span both nodes are replicated. Note that the adaptive decomposition is able to subdivide the model finely in highly populated areas and generates large voxels for empty space. At this stage each node is a separate file on disk in a special format that is suitable for streaming the data through the preprocessing programs.

Once the size of a BSP node is below a given threshold we create a voxel and store it in a file that contains its data (triangles, BSP, shading data, etc.). This is a binary file format that is suitable for directly reading it into memory. In order to avoid large directories with the associated lookup cost on some file systems, the files are automatically sorted into a directory hierarchy.

This preprocessing algorithms is easy to set up as it has only two parameters that need to be set: the number of triangles in a voxel and the maximum depth of the BSP trees. The rendering speed is fairly insensitive to the exact setting of these parameters as long as the minimum size of voxels is reasonably small. However, the size of the generated data set increases steadily with smaller voxel size and larger BSP depth. It is still unclear if there is an automatic way to determine good values for these two parameters.

The cost of preprocessing algorithms has a complexity of $O(n \log n)$ in the model size. Preprocessing is mainly I/O bound as the computation per triangle is minimal. We are currently using a serial implementation, where each step in the recursive decomposition is a separate invocation of a single program. Currently, the resulting files are all located on a single machine acting as the model server.

With a little more programming effort we could significantly speed up preprocessing by distributing the triangle data to multiple machines. A master process would control the decomposition, distribute the load across the machines, and build the high-level BSP tree. Storing the data base on several machines would have the additional benefit that access is distributed across all data base servers, thus making better use of the available bandwidth in a fully switched network. Of course, once the data set of a node is small enough an in-core algorithm should be used for better preprocessing performance.

5 Load Balancing

The efficiency of distributed/parallel rendering depends to a large degree on the amount of parallelism that can be extracted. We are using demand driven load balancing by subdividing the image into tiles of a fixed size (32 by 32 pixels). As the rendering time for different tiles can vary significantly (e.g. see the large variations in model complexity in Figures 1 and 7), we must distribute the load evenly across all client CPUs. This has to be done dynamically, as the frequent camera changes during an interactive walkthrough make static load-balancing impossible.

We employ the usual dynamic load balancing approach where the display server distributes tiles on demand to clients. The tiles are taken from a pool of yet unassigned tiles, but care is taken to maintain good cache locality in the clients. Currently, the scheduler tries to give clients tiles they have rendered before, in order to efficiently

reuse the data in their geometry caches. This approach is effective for small camera movements but fails to make good use of caches for larger movements.

This simple assignment can be improved using an idea from image-based rendering — essentially a simplified RenderCache [21, 8]. For each traced ray the 3D intersection points would be stored together with its rendering cost and the client that computed it. This information can then reprojected into the next frame. For each new tile, its cost is estimated by averaging the cost over all the intersection points reprojected to this tile.

Additionally, for each tile a *client affinity value* is estimated based on the fraction of reprojected samples computed by a particular client. Tiles are then assigned to clients primarily based on affinity in order to maximize cache reuse. Additionally we hand out costly tiles first in order to minimize load imbalance towards the end of a frame.

In order to avoid idle times while the clients are waiting for the next task from the master each client buffers one additional task. This way, when a client has completed its current tile, it can immediately proceed working on the next tile without having to wait for the servers reply. In a similar way, we double-buffer workload in the server: If all tiles from the current frame have been assigned to clients, the server starts assigning tiles from the next frame while waiting for the last frame to complete.

6 Implementation and Results

Our current setup uses two servers — one for display and one for storing and distributing the preprocessed models. Both machines are connected via Gigabit Ethernet to a Gigabit Ethernet switch. These fast links help in avoiding network bottlenecks. In particular we require a high bandwidth connection for the display server in order to deal with the pixel data at higher resolutions and frame rates. The bottleneck for the model data could be avoided by distributing it among a set of machines as mentioned above.

For our experiments we have used seven dual P-III 800-866 MHz machines as ray tracing clients. These clients are normal desktop machines in our lab but were mostly unused while the tests were performed. The client machines are connected to a FastEthernet switch that has a Gigabit uplink to the server switch.

The model server has two very fast striped disks for storing the preprocessed model data. The disks can sustain a bandwidth of roughly 55 MB/s, which almost exactly matches the maximum measured Gigabit network bandwidth. We use NFS with large read and write buffers to access the model data from the clients. Each client is able to almost saturate the full bandwidth of its network connection, such that our potential bottleneck is now the connection to the model server. The display and scheduling server runs at a very light computational load but must handle a large and constant stream of pixels from all clients.

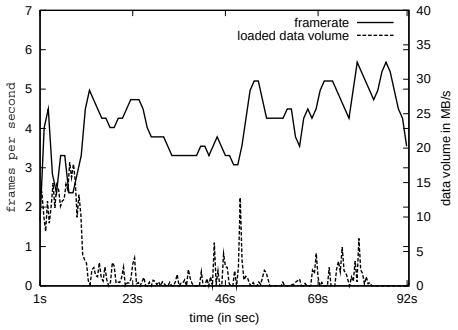

Fig. 3. Frame rate and transfered data rate after decompression during a walkthrough heading from the outside to the inside of the powerplant building. The frame rate is pretty constant around 4-5 fps unless large amounts of data are transfered (at the beginning where the whole building is visible). The frame rate is achieved without the SIMD optimization, which should the frame rate by at least a factor a two to 6–12 fps.

We have tested our setup with the power-plant model from UNC [1] to allow for a direct comparison with previous work. This also provides for a comparison of algorithms based on rasterization versus ray tracing.

The power-plant test model consists of roughly 12.5 million triangles mostly distributed over the main building that is 80 meter high and 40 by 50 meters on either side (see Figure 1). Each triangle of the model also contains vertex normals that allow smooth shading and reflection (see below).

Preprocessing including conversion from the original PLY-files into our format as well as voxel decomposition took roughly 2.5 hours, with our unoptimized, sequential implementation. This already is significantly faster than the preprocessing required in [1]. That approach required 17 hours for a partial preprocessing that did only allow for interactive movements in a small fraction of the overall model. Their preprocessing time for the whole model was estimated to take three weeks [1]. We estimate that once parallel preprocessing is fully implemented, our preprocessing time could be reduced to less than half an hour.

Figure 3 gives a compact summary of our overall results. It shows the frame rate achieved by our system as well as the amount of geometry fetched over the course of a walk through the model. All images in this paper are computed at 640 by 480 resolution. The total time of the walkthrough is 92 seconds using all seven clients. Note that we only trace primary rays for this test in order to allow direct comparison with the results from [1]. We only show the results of a single walkthrough, as they closely match those from other tests.

The walkthrough starts outside the main building while sweeping the view across most of the power-plant. This is very demanding as the building is mostly open and allows to see much of the complex inside structure. A lot of geometry needs to be loaded, which saturates the network connection to the model server at up to 17 MB/s of geometry (uncompressed) for a single client only.

We then move inside the main building. The working set is much lower here, but can change very quickly as we move past a large occluder. During this time we still fetch an average of 1–2 MB/s of geometry data (with spikes up to 13 MB/s). However, the latency of those transfers can mostly be hidden by our asynchronous fetching approach. We do not yet perform any form of prefetching, even though this would help even more in avoiding network effects.

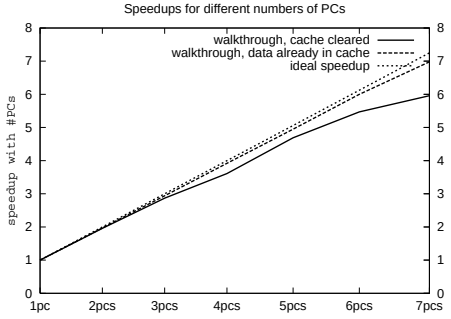

Fig. 4. The system shows almost perfect scalability from 1 to 7 dual CPU PCs if the caches are already filled. With empty caches we see some network contention effects with 4 clients but scalability is still very good. Beyond 6 or 7 clients we start saturating the network link to the model server.

With seven dual CPU machines we achieve a pretty constant frame rate of 3–5 fps. However, all numbers are computed with plain C++ code, as the SIMD code was still being adapted to the new distributed algorithm. Early tests of the SIMD optimized ray tracing code has consistently achieved speedups by a factor greater than 2. This brings our frame rate up to 6–12 fps, which is about the same as achieved in [1]. Note that we still render the original model with all details and not a simplified version of it.

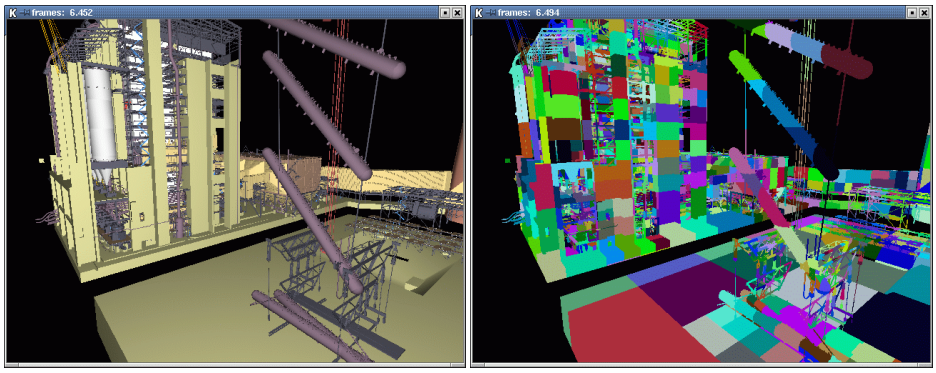

Fig. 5. Two images showing show the structure of the high-level BSP tree by color coding geometry in each voxel (bottom image). Voxels are relatively large for the walls but become really small in regions with lots of details.

Figure 5 visualizes the BSP structure that is built by our preprocessing algorithm. The voxel size decreases significantly for areas that have more geometric detail.

The original model does not provide material information other than some meaningless surface colors. In order to test some of the advanced features of ray tracing, we added a distant light source to the model and made some of the geometry reflective (see Color Plate 6). Of course, we see a drop in performance due to additional rays being traced for shadows and reflections. However, the drop is mostly proportional to the number of traced rays, and shows little effect due to the reduced coherence of the highly diverging rays that are reflected off the large pipe in the front as well as all the tiny pipes in the background.

We also tested the scalability of our implementation by using one to seven clients for rendering exactly the same frames as in the recorded walkthrough used for the tests above and measured the total runtime. The experiment was performed twice — once with empty caches and once again with the caches filled by the previous run. The difference between the two would show network bottlenecks and any latencies that could not be hidden. As expected we achieved almost perfect scalability with filled caches (see Figure 4), but the graph also shows some network contention effects with 4 clients and we start saturating the network link to the model server beyond 6 or 7 clients. Note, that perfect scalability is larger than seven because of variations in CPU clock rates.

Because we did not have more clients available, scalability could not be tested beyond seven clients. However, our results show that scalability is mainly bound by the network bandwidth to the model server, which suggests that a distributed model data base would allow scalability well beyond our numbers. Of course we could also replicate the data — space permitting.

Color Plate 7 shows some other views of the power-plant showing some of the complexity hidden in this huge test model.

For a stress test of our system we have placed four copies of the power-plant model next to each other resulting in a total model complexity of roughly 50 million triangles (see Figure 1). Preprocessing time increased as expected, but the frame rate stayed almost identical compared to the single model. Essentially the depth of the higher-level BSP tree was increased by two, which hardly has any effects on inside views.

However, for outside views we suffer somewhat from the relatively large voxel granularity, which results in an increased working set and accordingly longer loading times that can no longer be completely hidden during movements. When standing still the frame rates quickly approach the numbers measured for a single copy of the model.

7 Conclusions and Future Work

Previously, interactive rendering performance for highly complex models with tens of millions of polygons and more could only be achieved with high-end graphics hardware on supercomputers and required very expensive preprocessing techniques that makes the technique mostly infeasible.

In this paper, we have shown that using a software ray tracing approach, interactive rendering performance can be achieved for more complex models even on inexpensive clusters of workstations. We use a two–level, adaptive scene decomposition with BSP trees that allows explicit data management for caching and reordering purposes.

We have shown that a high degree of parallelism can be extracted from such systems by using efficient load balancing and paying careful attention to network bandwidth and latencies. Stalling due to network latencies can be avoided to some degree by reordering the computations within the clients.

Even though our system already achieves interactive rendering performance by using only seven rendering clients, there are many ideas for further improvements:

Obviously, faster computers and networks are already available. They would allow for almost twice the performance while being only slightly more expensive.

The other obvious extension to the system is to increase speed by fully activating the SIMD extensions and prefetching methods as described in [20]. Also a distributed scene data base would avoid the server bottleneck and allow for even more rendering clients.

More computational resources would allow us to spend more effort on illumination and shading computations. In particular, anti-aliasing and more complex shading computations, like programmable shading, would be interesting. It would also be interesting to implement more complex global illumination algorithms and deal with the reduced coherence for illumination rays.

Bandwidth could be reduced further by separating BSP, geometry, and shading information into separately loadable entities. This would prevent loading shading data even for voxels that never generate an intersection. Similarly, we need to avoid the increased file size due to replicating information for geometry contained in multiple voxels.

Finally, as is the case with all algorithms working on a fixed spatial decomposition, we are limited to static environments. New algorithms and data structure that can deal with complex dynamic environments are desperately needed for interactively ray tracing.

8 Acknowledgements

We thank the the Computer Graphics Group at the University of North Carolina and Anselmo Lastra in particular for providing the power-plant model. Georg Demme and Marcus Wagner provided invaluable help and support with programming and system setup. We also thank the anonymous reviewers for the helpful comments.

References

1. D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Strzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 199–206, Atlanta, USA, April 1999.
2. Alan Chalmers and Erik Reinhard. Parallel and distributed photo-realistic rendering. In *SIGGRAPH 98 Course*, pages 425–432. ACM SIGGRAPH, Orlando, July 1998.
3. Lucia Darsa, Bruno Costa, and Amitabh Varshney. Navigating static environments using image-space simplification and morphing. In *ACM Symposium on Interactive 3D Graphics*, pages 25–34, Providence, RI, 1997.
4. Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-buffer visibility. *Computer Graphics*, 27(Annual Conference Series):231–238, 1993.
5. Homan Igehy. Tracing ray differentials. *Computer Graphics*, 33(Annual Conference Series):179–186, 1999.
6. Intel Corp. *Intel Pentium III Streaming SIMD Extensions*. http://developer.intel.com/vtune/-cbts/simd.htm.
7. James T. Kajiya. The rendering equation. *Computer Graphics*, 20(4):143–150, August 1986.
8. William Martin, Steven Parker, Erik Reinhard, Peter Shirley, and William Thompson. Temporally coherent interactive ray tracing. Technical Report UUCS-01-005, Computer Graphics Group, University of Utah, 2001.
9. Michael J. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium '95*, June 1995.
10. Michael J. Muuss and Maximo Lorenzo. High-resolution interactive multispectral missile sensor simulation for atr and dis. In *Proceedings of BRL-CAD Symposium '95*, June 1995.
11. Markus Oberhume. LZO-compression library. available at http://www.dogma.net/-DataCompression/LZO.shtml.
12. Steven Parker, Michael Parker, Yaren Livnat, Peter Pike Sloan, Chuck Hansen, and Peter Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Computer Graphics and Visualization*, 5(3):238–250, July-September 1999.
13. Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98*, pages 233–238, 1998.
14. Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive ray tracing. In *Interactive 3D Graphics (I3D)*, pages 119–126, april 1999.
15. Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Computer Graphics*, 31(Annual Conference Series):101–108, August 1997.
16. E. Reinhard and F. W. Jansen. Rendering large scenes using parallel ray tracing. In *Eurographics Workshop of Parallel Graphics and Visualization*, pages 67–80, September 1996.
17. Erik Reinhard. *Scheduling and Data Management for Parallel Ray Tracing*. PhD thesis, University of East Anglia, 1995.
18. Erik Reinhard, Alan Chalmers, and F.W. Jansen. Overview of parallel photorealistic graphics. In *Eurographics '98, State of the Art Reports*. Eurographics Association, August 1998.
19. Francois SIllion, George Drettakis, and Benoit Bedelet. Efficient imposter manipulation for real-time visualization of urban scenery. *Computer Graphics Forum, Proceeding Eurographics '97*, 16(3):207–218, September 1997.
20. Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, 20(3), 2001. available at http://graphics.cs.uni-sb.de/ wald/Publications.
21. Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. *Eurographics Rendering Workshop 1999*, 1999. Granada, Spain.
22. Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. *Computer Graphics*, 31(Annual Conference Series):77–88, August 1997.

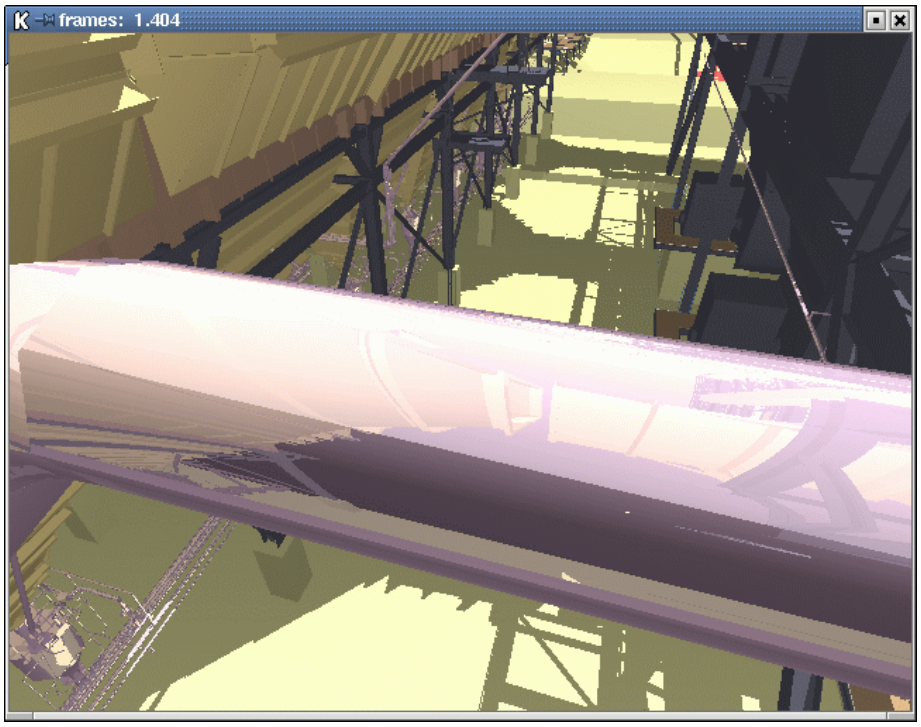

Fig. 6. Shadow and reflection effects created with ray tracing using one light source. The performance drops roughly proportional to the number of total rays traced but the size of the working set increases. Note the reflections off all the small pipes near the ground. Diffuse case: 1 ray per pixel, 4.9 fps, with shadow and reflection (multiple of 2 rays): 1.4 fps.

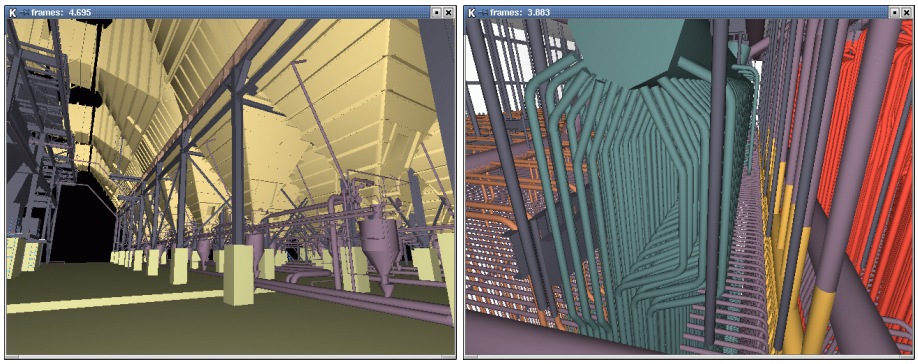

Fig. 7. Two complex view of the power-plant. Both still render at 4.9 and 3.9 fps, respectively.