RENDERING COMPLEX SCENES ON CLUSTERS WITH LIMITED PRECOMPUTATION

Gilles Cadet, Sebastian Zambal, Bernard Lécussan
Supaero, Computer Science Department
10, av. Edouard Belin
31055 Toulouse Cedex
France
{gilles.cadet, sebastian.zambal, bernard.lecussan}@supaero.fr

Abstract This paper presents a parallel ray-tracing algorithm in order to compute very large models (more than 100 million triangles) with distributed computer architecture. On a single computer, the size of the used dataset generates an out of core computation. Cluster architectures designed with off-the-shelf components offer extended capacities which allow to keep the large dataset inside the aggregated main memories. Then, to achieve scalability of operational applications, the real challenge is to exploit efficiently the amount of available memory and computing power. Ray-tracing, for high quality image rendering, spawns non-coherent rays which generate irregular tasks difficult to distribute on such architectures. In this paper we present a cache mechanism for main memory management distributed on each parallel computer and we implement a load balancing solution based on an auto adaptive algorithm to distribute the computation efficiently.

Keywords: distributed ray tracing, cluster of commodity computers, load balancing, latency hiding

1. Introduction

Clusters are simple and widespread systems. Although scalability of operational applications remains the main goal, effective use of the distributed memory and the theoretical computing power becomes a difficult task when the number of computers increases. Scalability can be reached by hiding resource latencies and by using original mechanisms to tolerate residual latencies [Amdhal, 1967]. Furthermore, the system must also distribute computations and data while ensuring effective load balancing.

The distribution of the ray tracing algorithm requires a strong optimization of these various parameters. Moreover, dealing with large models which can-

not be loaded completely into the main memory currently raises a technical challenge.

In the first part of this article, we present two methods to handle large datasets: a lazy ray tracing algorithm which limits the size of memory needed for the computations and a geometry cache management which gives the ability to access large dataset. In the second part, we expose a distributed solution of the ray tracing algorithm. We show an original method to approach the optimal load balancing and present ways of distributing the geometry database by duplication or by sharing. In the third part, we show the results obtained with the above methods. Lastly we discuss improvements of these methods that we are currently working on.

2. Background

2.1 Problems of Distributed Ray Tracing

The ray tracing algorithm is mostly used in 3D rendering to produce photorealistic images by accurate simulation of light propagation. To obtain efficient computation times, the simulation is done by following the inverse way of the light. Rays are spawned from the observer's eye through each pixel of the image to render and then are propagated inside the geometrical model by reflection and refraction laws until they reach light sources [Appel, 1968].

Using ray tracing to manage large dataset larger than main memory and to distribute the computations on commodity computers raises two types of problems. The first one is related to the use of distributed memory and the second one is due to the irregular data accesses caused by the traced rays.

Although the cluster architecture offers high cumulated computing power at low price, it suffers from low bandwidth and high latency of communication. Distributed memory implies the reorganization of standard algorithms in order to maximize the data access locality. To handle very large models, solutions exploit either the network and/or the local hard disk as a memory extension. These two points are strong constraints because the differences in bandwidth and latency between main memory and network or hard disk limit the performance of the global system.

From the algorithmic point of view, the rays hit the triangles of the model in a very irregular way. With a significant depth of recursion and a great number of secondary rays, the same triangles are requested several times in an unforeseeable order.

To summarize, the various problems we have to consider are: efficient use of distributed memory, low bandwidth and high latency of remote memories and irregularity of data accesses. Below we give an overview of recent works that treat these problems.

2.2 Previous Work

To reduce rendering time current studies develop three approaches. One of the older solutions is parallel computing either with parallel machines or clusters. The second approach relates to the capacity of new graphic cards to load custom code [Purcell and al, 2002, Purcell and al, 2003] the use of these graphic cards is quite limited because they are only optimized for Z-buffer rendering. However, more recent proposals like AR350 [Hall, 2001] and SaarCor [Schmittler and al, 2002] use graphic cards with a built in ray engine. In fact fields of research related to graphic card solutions are very promising with regard to performances, but are still limited to models which fit in the graphic card memory or in main memory.

Work on large models with clusters is justified overall by the need to visualize the models realistically as a whole and not by parts. Compared to other algorithms like Z-buffer, ray tracing has a complexity which grows logarithmically according to the number of triangles contained in the model. This logarithmic property is guaranteed only if the model can be loaded completely into memory.

Ray tracing can be distributed naturally by computing independently groups of rays on each computer. The master splits the image into parts distributed to each slave in a demand driven computation. In this case, the geometry needed by each slave is transmitted via the network or can be locally loaded from the hard disk. Another way of distributing is to share the model between all computers; the distribution is then controlled by the geometry owned by each slave (data driven) and the rays are propagated among the machines. A third and last way of distributing is a hybrid solution which is a combination of the two preceding methods. More recent works use demand driven distribution because it minimizes exchange of information over the network improving the load balancing. So we expose below some recent results.

The RTRT team [Wald and al, 2001a] proposes the first animation of large models by distributing an optimized ray tracing engine on commodity computers [Wald and al, 2001b]. The model is precomputed by the construction of a binary space partition (BSP) tree. Each voxel of this tree is an independent element recorded in a file which contains geometry, textures, precomputed data and a BSP subtree. One voxel has an average size of 256 KB and is directly loadable into main memory.

To take advantage of the bi-processor architecture, each slave runs two threads for processing rays. Moreover there is another thread for loading the data asynchronously. The geometry is centralized on a data server which distributes the voxels when the slaves request them. Their downloadable animation shows a powerplant of 12.5 million of triangles with outdoor and indoor views. The file has a size of 2.5 GB after a precomputation of 2 hours and 30

minutes. Results perform about 5 frames per second on a cluster of 7 computers. The animation requires an average data bandwidth of 2 MB/s with some peaks about 13 MB/s which saturate the network. The analyse of these results seems to show that the working set stays rather small which indicates that impacted triangles are localized and that the slaves do not often have to deal with low bandwidth and high latency of the network.

Another study [Demarle and al, 2003], which is very close to the RTRT's one, but less optimized in the ray core engine, shows another possible distribution on a cluster to handle large dataset. The main difference is the way of recovering data by each slave. Each one launches two processing threads and a voxel loader thread, called "DataServer", which emulates a shared memory. A processing thread sends its geometry data requests to the local DataServer which either loads the data from the main memory or transmits the received request to a remote DataServer. To work efficiently, the model is partially loaded by each slave and thus is wholly loaded into the distributed memories. The performance reaches more than 2 frames per second for a model size of 7.5 GB with a cluster of 31 processors. This model must be precomputed before the rendering using hierarchical voxels in 40 minutes.

2.3 Goals

As we have mentioned above, current studies render very large scenes on commodity computers efficiently. All these studies render images in "real time" after a precomputation step which gives the possibility to load parts of the model independently and with spatial coherency. Each slave manages a voxel cache with a least recently used (LRU) policy. However, the suggested methods are only effective for a quite small working set and moreover after a time consuming precomputation step.

The study presented in this paper wants to differ in this last point by strongly reducing the precomputation step. Using a traditional model file we seek to optimize the total rendering time, taking care of the complete computation from the model recording to its visualization. This approach makes sense with the modelling process which unceasingly changes the geometry integrity. By reducing the precomputation time, we expect that the time of the first visualization can be considerably reduced.

Moreover, we want to handle very large models while keeping a photorealistic quality of the rendered image without any restrictions (e.g. without limiting the depth of recursion of the propagated rays). The objective is to render models which include about 100 million triangles.

3. Methods to Handle Large Dataset

To handle large dataset with ray tracing, the strategy of finding and loading data must be modified. We first describe a method to quickly find data without using extra memory and then we present a cache manager which offers the possibility of working with a model that is larger than the main memory.

3.1 Lazy Construction of the Octree

A geometrical model is built with objects which are discretized into triangles. Each triangle is linked to physical properties giving its color and the way to reflect and refract rays when they hit its surface [Whitted, 1979]. The hot spot when developing a ray tracing engine is to have an efficient function to compute the intersection between triangles and rays.

To reach this goal, the model is generally divided into voxels by a three-dimensional regular grid. However, with voxels of the same size, the distribution of the triangles is unbalanced; this introduces useless computation time to traverse empty voxels. Moreover, this structure takes a lot of memory because the dimension of the grid must be large enough to keep the number of triangles in denser voxels as small as possible. A well known solution is to build a hierarchy of voxels like an octree [Glassner, 1984]; in this case only not empty voxels are divided into subvoxels which saves memory. However, the exploration of the octree is a little more complex because of the hierarchy and the irregularity of the data structure.

The dynamic construction of the octree is an efficient solution to keep the memory occupancy low [Bermes and al, 1999]. Instead of building the whole octree, the algorithm starts with only one voxel which represents the bounding box of the model. During the rendering computation, each time a ray traverses a voxel not yet divided, the subvoxels are dynamically built and the traversal of the children is done recursively; with this principle, only impacted triangles are loaded into memory and are used directly. This construction, called lazy octree construction, is the basic technique to quickly find triangles in large models.

This lazy algorithm shows the following properties: first, a child node is evaluated only if it contains necessary data for the computation; then, the node evaluation result is definitively stored in the octree and will be reused for neighbor ray computation. With distributed memory architecture, the data structure is built dynamically and locally for the subset of rays computed by each processor. Thereby, the algorithm exploits spatial ray coherence.

Efficiency is obtained for models which fit in the main memory of one computer (see results in Table 2). The main objective of this new study is to implement mechanisms to overcome this limitation.

3.2 Geometry Cache Management

To get along with the problem of huge dataset, in a first step the large model is subdivided into small blocks. These blocks are loaded into a cache when they are requested by the ray tracing engine (Figure 1). It must be taken care that this geometrical cache does not interfere with the virtual memory of the operating system. Recent works have demonstrated that classical virtual memory is not very effective for applications using large quantities of data [Cozette, 2003]. Overall there are two ways to deal with this: either the virtual memory of the operating system is adapted or the flow of data is controlled directly. In order to have free control over the caching process we have decided to implement the later solution.

Before considering parallelization, a sequential version of the cache was developed. One part of the complete caching system is a compiler which initially converts the model into a structured binary file. This binary file then serves as a database for the actual cache supplying the ray tracing engine with blocks.

The geometrical data of a model is initially given as a text file containing triangles. The compiler transforms this text file into a binary file which consists of a sequence of blocks. Each block contains information about several triangles and can be loaded into memory independently. To accelerate the system, a non destructive LZO compression [Oberhumer, 2002] is applied to the blocks which reduces the size of the blocks by about a half and leads to a speed-up of 25% for the time spent for loading the model. A well chosen size for the blocks minimizes the redundancy and maximizes the compression rate which leads to better performance. Experimentally we found an optimal block size of 64 KB.

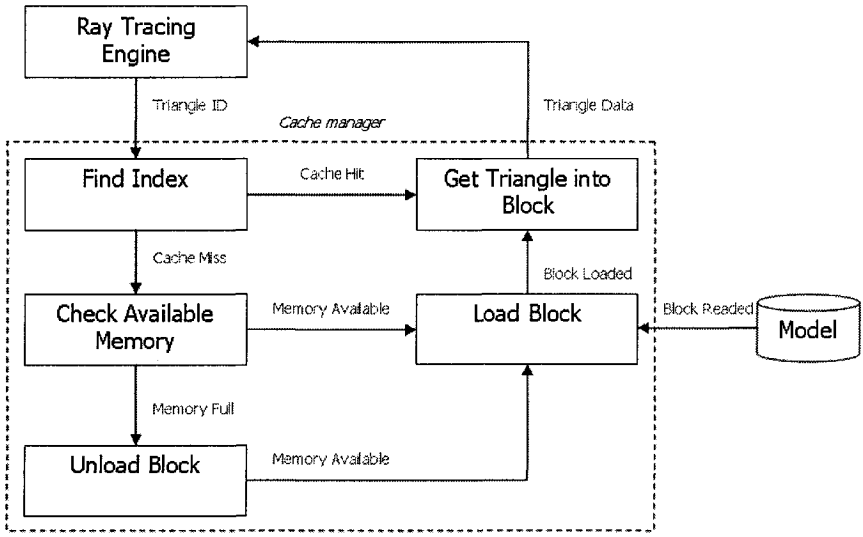
The compilation of the model is the only needed step to initialize the cache memory and to start the image rendering; this takes about three minutes for a model size of 3.4 GB.

During the rendering process the cache delivers requested triangles to the ray tracing engine. Each time a triangle is demanded, the cache first has to determine the block containing the triangle; if the block is not in memory at this time the cache has to load it. That might include unloading another block. Finally the demanded triangle can be returned.

The cache management is purely associative. A table indicates for each block if it is loaded or not. Thus the time to find a block is negligible and the most important item to improve the cache efficiency is to increase the hit rate.

The cache policy of unloading blocks is LRU. The number of allocations (and deallocations) of memory is minimized at each cache miss by considering memory areas that have been allocated before. This approach does not fragment the memory and limits the number of system calls.

Figure 1. Data flow of the geometry cache.



We have tested several strategies to reduce the time of loading blocks and also to proceed the computations while loading blocks. The first idea was to use two different levels of subdivision: large blocks for loading data and small blocks for internal use in the cache. Another idea was to use a small secondary cache which is able to load a sequence of blocks simultaneously. Furthermore we tried to apply an own thread dedicated only to anticipate block loading. However all these approaches did not lead to notable improvements of the rendering time. In fact, all the overhead caused by the additional complexity (useless loading of blocks, management of additional functionality, overhead when synchronizing threads) lowered the performance of the global system.

4. Distribution of Large Dataset

What it has not been answered so far is the question how to handle large geometrical data in a distributed environment. The following topics discuss the task distribution as well the data distribution.

4.1 Task Distribution

In this section we will first talk about how to distribute computations for the ray tracing algorithm in general. Then we will present an efficient method to dynamically reassigning the work load of the slaves.

The rendering process is subdivided according to a partition of the image into tiles. This typically results in a classical master/slave architecture. The master subdivides the image and assigns its tiles to the remote slaves. After each slave has returned its locally rendered region, the master reassembles and displays the final image. When using this strategy, no geometrical data is exchanged between the master and the slaves. Only information about tiles to be rendered has to be transmitted. To exploit the locality of each slave's cache, the master has to assign to each slave the tiles which are close together.

The principal difficulty of this algorithm is to achieve an efficient load balancing to optimize the total rendering time. The most important aspects in this context are to reduce the number of communications, to improve the locality of computations on each slave and to keep the sizes of the tiles as large as possible. For a tile, it has been observed that the computation time per pixel is inversely proportional to its area. This rule is a direct result of anti aliasing which exceeds the actual border of the tile and causes redundant computations.

The partition of the image can either be done statically (before rendering) or dynamically (during rendering). The static assignment is simple but possibly leads to unbalancing load on the slaves because the complexities of the individual parts of the image can be very different. One possible solution for this problem is to estimate the complexities in a preprocessing step. However this additional preprocessing costs a lot of time and moreover the estimation of complexities might not be sufficient to achieve a good load balancing [Farizon and al, 1996]. The conclusion is that a dynamic subdivision of the tiles has to be applied.

To subdivide the image, the size of the tiles has to be chosen carefully. This size is guided by the following antagonism: large tiles reduce the number of communications and increase the time per pixel; small tiles allow a fine regulation of the load balancing which is particularly important at the end of the rendering process. It is thus interesting to subdivide the image into large tiles at the beginning of the rendering and then to subdivide them before the rendering ends. The difficulty is to know when exactly the subdivision should start.

To find a good solution for the subdivision, three main items are considered:

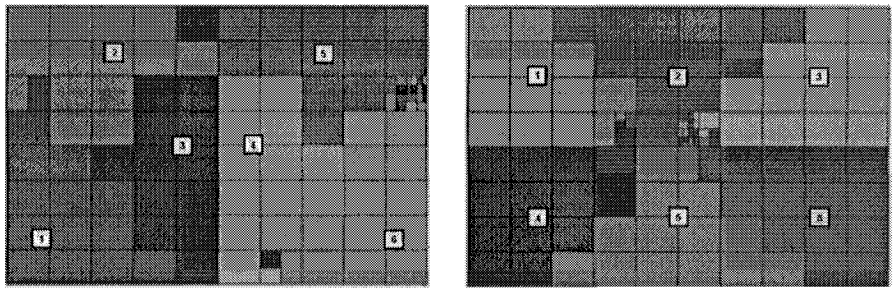
- the definition of central points which guide the assignment process,
- the assignment process itself,
- an adaptive subdivision method for the assigned tiles.

Definition of the Rendering Zones. The initial granularity of the assigned tiles is chosen in a way to achieve at least five tiles per slave. This value has empirically been found out to satisfy the models used.

To preserve the locality, we assign to each slave a position in the image plane called central point. The tiles assigned to each slave are as near as possible

to the slave's central point. Two different ways to choose the central points were examined. The first approach was to place the points into the image in form of an X (Figure 2. left); this strategy assumes that most images have their highest complexity in the center and allows treatment of the largest tiles there. The second approach places the central points homogeneously taking into account the image resolution (Figure 2. right); this generally leads to better performances.

Figure 2. Tile assignments with 6 slaves. Each slave has a central point and is represented by a specific colour. Left: "X" placement of central points and nearest tile policy to assign the next tile. Right: homogeneous placement of central points and trade off policy to assign the next tile.



Assignment and Subdivision. The assignment of the tiles must reuse already built parts of the octree by optimizing the data locality. One strategy we tried out was to assign the free tiles nearest to each slave's central point. Although this method was efficient at the beginning of the rendering there may be no free tile near a central point at the end. A slightly modified strategy keeps the tiles near the central points unassigned as long as possible. The tiles are now assigned by minimizing the distance to the own central point and maximizing the distances to the other central points.

Large tiles are defined at the beginning of the rendering and are subdivided during the rendering process. An algorithm using the estimation of the remaining computation time allows us to subdivide the tiles as late as possible and leads to an efficient load balancing. An estimated remaining computation time is associated with each free tile as a function of previously rendered tiles. Each estimation is weighted according to the inverse of the distance between the currently and the previously rendered tiles. When a new tile has to be assigned, the master checks if the total time of the current and future computations is greater than the time estimated for this tile. If this is not the case, the tile is subdivided and the tile assignment function is recalled. This continues until an acceptable computation time or a minimum size of the tile is reached.

The principle of image subdivision with the assignment and the subdivision of tiles offers an efficient load balancing (Table 4). The method of estimating the remaining time in fact suffers from underestimation but keeps the scale between estimated intervals and real intervals. A little amelioration of load balancing could be realized by assigning an amount of time to each tile. When this critical time is exceeded, the slave must stop and retransmit the partially computed tile.

5. Data Distribution

5.1 Duplication

The simplest approach to deal with the data locality is duplication. Each slave has a complete copy of the model on its hard disk. Consequently, each slave is able to render a part of the image independently of the others, accessing its local data using a local cache.

The main advantage of this approach is that almost no communication over the network is necessary during the computation. The master only communicates with the slaves to send all assignments and receive all rendered tiles. The main drawback is the time needed to transfer the file to each local disk. This transfer takes about three minutes for the "Engine x729" model (Figure 5) and can be partially masked by the time needed to parameterize the computation by the user.

5.2 Sharing

When a slave needs a block not yet loaded locally, it can avoid the loading of the block from the hard disk if this block was already loaded into the main memory of another slave. We will now discuss the conditions under which it is better to access remote caches for a requested block than to load it from the local disk.

To find the more effective strategy (either to load the block or to transmit it over the network) we have compared the communication bandwidth between two slaves using MPI (Message Passing Interface) to the hard disk bandwidth. With a Gigabit Ethernet network the bandwidth of MPI is better for block sizes greater or equal to 16 KB. The bare latency of MPI communication was measured with about 136 μs compared to 5000 μs for the hard disk (for random access).

According to the above characteristics the previous cache mechanism has been modified. The function that loads the data from the disk is replaced by a function that downloads the requested block from another remote cache.

Two different kinds of slaves are used. The first plays the classical role of a "worker" and uses a cache with limited memory. If the worker has to load

a new block, it transmits its request via MPI to another slave. The other slave receiving the demand is in the role of the "reader". This type of slave only waits for the block requests and does not communicate with the master. If the requested block is available the reader simply returns it to the worker.

The tests, carried out with a small model but with a high miss rate of the cache, show that the data accesses over the network allows to double the rendering performance. Because the network offers better performance than the hard disk with models that cause a high miss rate of the cache, it is interesting to study an architecture that uses remote accesses.

We will consider two possible configurations. The first is to use as many readers and as many writers as there are machines available. With this configuration, there is a reader for each worker but the memory is used commonly by all slaves. The second configuration is to use a single reader. This means that a single machine is busy with loading the blocks and distributing them over the network to the other slaves.

During its initialization phase, each slave receives a role assignment from the master. If it is a worker, it waits for the tile assignments. If the slave is a reader, it links itself to the local files of blocks and waits for workers' requests.

Each worker is related to all the readers and each reader manages a part of the geometry. The geometry is shared uniformly so that each reader has to manage the same amount of data. A worker sends its requests to a specific reader depending on the index block to load.

This algorithm is applicable for models that have a file size smaller than the total readers' memory. In fact access over the network is not efficient when blocks have to be loaded from the remote hard disk. Each used block should reside in one of all memories to avoid remote hard disk accesses.

Table 1. Rendering times depending on the number of readers.

Medium for loading	Hard Disk	Network	
Nb. of Readers	-	1	4
Nb. of Workers	4	4	4
Rendering Time (sec.)	70	37	35

The above table (Table 1) shows the execution times depending of the roles given to the slaves. The "Engine" model (Figure 5) used for this measurement has a size of 5 MB with blocks of 4 KB. Furthermore the memory of the workers is limited to 10% of the total model size. The readers can load the whole model into the cache. These results show that it is much more efficient to load the blocks over the network than to load them from the local hard disk. The best result is achieved by using 4 readers because this avoids a network bottleneck.

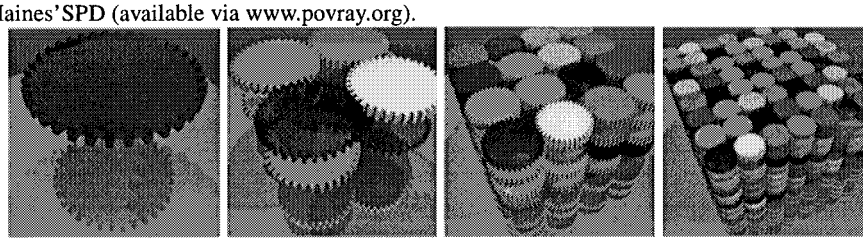
6. Results

The results are obtained with a cluster of six machines linked together by an Ethernet network with a bandwidth of 1 GB/s. Each machine is equipped with an AMD Athlon MP 2000+ biprocessor (1.7 Ghz), has got an IBM Deskstar 60 GXP hard disk (60 GB capacity, 7200 rpm, 2 MB of cache) and 1 GB of DDRAM. The operating system used is Windows 2000 Professional.

6.1 Lazy Octree Construction

The following performance comparison (Table 2) shows the efficiency of the lazy octree construction in terms of computation time and memory size needed. Previous comparisons can be found in [Bermes and al, 1999].

Figure 3. From left to right: Gears1, Gears2, Gears4, Gears8. All test scenes from Eric Haines'SPD (available via www.povray.org).



All rendered images (Figure 3) have a size of 512x512 pixels and are computed by only one machine. PovRay for Windows V3.5 is used and configured with its default options except for the following parameters: deactivation of the anti-aliasing, first pass with ray tracing and shadowing with ray tracing.

Table 2. Performance comparison between PovRay and ray tracing with lazy octree construction.

Model	Gears1		Gears2		Gears4		Gears8	
Number of triangles	27 552		219 744		1 757 280		14 057 568	
Ray Engine	Pov	Lazy	Pov	Lazy	Pov	Lazy	Pov	Lazy
Parse Time (sec.)	0.0	0.1	1.0	0.3	3.0	2.3	153.0	15.8
Trace Time (sec.)	17.0	7.8	16.0	6.9	17.0	6.7	21.0	7.6
Full Time (sec.)	17.0	7.9	17.0	7.2	20.0	9.0	174.0	23.4
Peak Memory (MB)	0.6	0.7	4.1	1.9	32.3	9.3	258.6	58.9

Theses results confirm that the lazy octree construction can efficiently save memory, especially for large models, and reduce the total computation time.

6.2 Cache Management Performance

The cache management is tested with a large model called "Engine x729". This model is built using 83 million triangles contained in a file of 3.4 GB (not compressed) with approximately 40 bytes per triangle. The following table (Table 3) summarizes the various tests. After compression, the blocks' file takes 1.8 GB on the hard disk with block size of 64 KB. Measurements are made on only one machine. Evaluation of the chosen model faces up to difficulties because it includes a lot of transparencies which implies that the major part of the triangles is hit. Results show that a model of more than 83 million triangles can be rendered with only one machine in less than one hour.

Table 3. Geometry cache performance with one machine.

Ray Depth	1	2	12
Rendering Time (RT)	59.4 min.	82.7 min.	421 min.
Loading Time (LT)	31.3 min.	47.5 min.	-
Nb. of Rays	1.20 e+06	1.56 e+06	-
Nb. of Cache Accesses	1.86 e+09	1.93 e+09	-
Nb. of Block Loads	1.26 e+06	1.50 e+06	-
$RT/(RT - LT)$	2.11	2.40	-

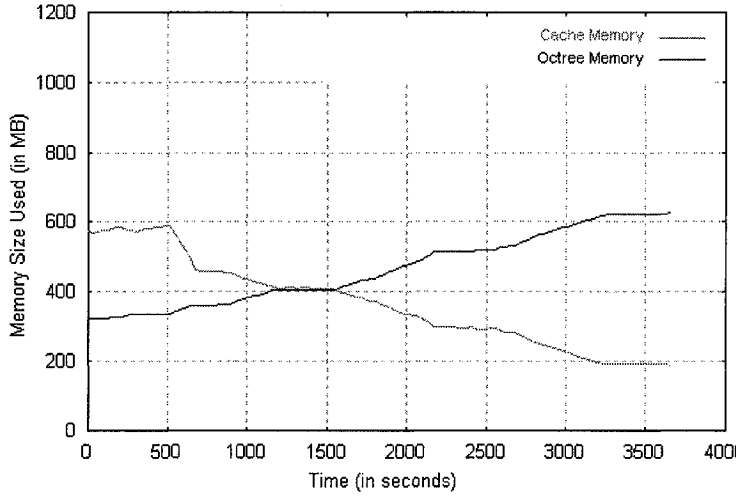
The next figure (Figure 4.) shows that the size of the octree increases constantly during the computation from 320 MB to 625 MB. This means that the cache manager runs with only 200 MB because it adapts dynamically to the free available memory. Thus the cache management works, at the end of computations, with less than 6% of memory compared to the model size. This small part of memory does not disturb the progression of computations because the increasing accuracy of the octree during the rendering process permits the localization of the ray engine data requests.

6.3 Ray Tracing Parallelization

Table 4. Load balancing performance with temporal refinement of the tiles (Times in seconds). The parallel efficiency is: $\frac{SequentialTime}{ParallelTime \times NumberOfSlaves}$.

		Sequential	2 slaves		4 slaves		6 slaves	
Model	Nb. Tri.	Time	Time	Eff.	Time	Eff.	Time	Eff.
Engine	114 7977	31.7	17	93%	9.6	83%	7.1	75%
Stairs	74 182	76.2	39.3	97%	21.3	89%	15.3	83%
ChevyInd	29 630	170.2	90.3	94%	49.1	87%	35	81%

Figure 4. Memory evolution while rendering.



The above table (Table 4) summarizes parallelization benches done with one, two, four and six slaves for three different models. All measured times are obtained with the temporal refinement method with a single parameter setting:

- the initial size of the tiles is set to 64x64 pixels,
- the minimal size of the tiles is limited to 8x8 pixels,
- an option allowing the slave to stop an assigned tile after a critical time is used.

Thanks to this adaptive core, which does not need specific parameters for each model, we obtain an efficient load balancing. The parallel efficiency with 6 slaves is close to 80% and thus offers a good acceleration of the rendering time.

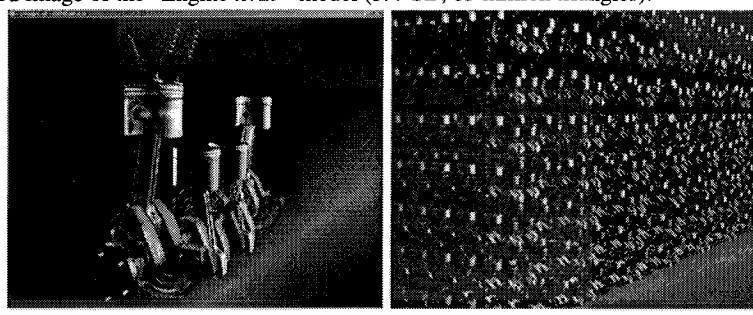
For comparison, we keep the principle of dynamic affectation of the tiles but we inhibit the subdivision procedure. Like in another study [Wald and al, 2001b], a fixed size of the tiles (32x32 pixels) is set. Results show that parallel efficiency decreases about by 3% and thus validate the contribution of the temporal subdivision method of tiles.

Moreover the adaptive method presented easily deals with the problem of heterogeneous computers [Qureshi, 2000] because it balances the computation load according to the power of each computer.

6.4 Large Dataset Distribution

The following tests have been carried out with using the cache management and the parallelization of the ray tracing together. Each slave has a local copy of the model on its hard disk.

Figure 5. Left: rendered image of the "Engine" model (5 MB, 114.000 triangles). Right: rendered image of the "Engine x729" model (3.4 GB, 83 million triangles).



The "Engine x729" model (Figure 5) is rendered in 74 minutes with a parallel efficiency of 95% for a depth of 12 (Table 5) which corresponds to more than 13 million rays traced. This efficiency must be carefully interpreted because during the sequential execution the octree size exceeded 900 MB and the cache manager reached a minimal size of 100 MB which caused a memory system swap. But in parallel, thanks to the distributed memory each slave can efficiently finish its execution.

Table 5. Geometry cache performance with 6 machines. The last column shows the ratios of the distributed computations to the sequential computation.

				Ratios
Ray Depth	1	6	12	-
Rendering Time (RT)	26 min.	51 min.	74 min.	-
Loading Time (LT)	12 min.	27 min.	44 min.	-
Nb. of Rays	1.19 e+06	8.40 e+06	13.8 e+06	1.0
Nb. of Cache Accesses	5.51 e+09	7.08 e+09	7.64 e+09	2.9
Nb. of Block Loads	3.50 e+06	5.20 e+06	6.78 e+06	2.8
$RT/(RT - LT)$	1.9	2.1	2.4	-
Parallel Efficiency	38%	-	95%	-

With a ray depth of 1, the parallel efficiency is only 38%. This lack of scalability is due to an high memory miss rate within each slave. To obtain good performances, the ratio values of the last column should be near to one meaning that each slave can work in a complementary way. This overloading

comes from a trade off between the depth of the octree and the number of triangles inside each voxel. Actually, the leaves of the octree contain too many triangles which implies an increase of data requests on such large models.

7. Conclusion

The computing power and the size of available memories of a cluster allow to handle very large datasets and to compute complex models efficiently. In this paper we have presented solutions and mechanisms to render complex scenes without using a time consuming model precomputation. First, a lazy ray-tracing has been implemented to limit the precomputation time and to save memory. Then, we have used each machine to cache in its own main memory the dataset stored locally on the hard disk and to compute parts of the image distributed by an original master/slave algorithm. Finally, as the whole model can be distributed within all main memories we have developed mechanisms to access required data remotely taking care of the high latency and low bandwidth of the network. Presented results show the efficiencies of the proposed solutions with some drawbacks that we expect to solve within future works.

8. Future Work

Related experiences show that it is important that the finer granularity of the octree does not cause extended use of memory (Figure 4). It seems possible to reduce the size of references from voxels to triangles and from triangles to vertices by limiting the scope of these references to each voxel. As an example, the maximum number of triangles inside a voxel is 65 536 which can limit the reference size to two bytes instead of four actually. This kind of optimizations already allows to increase the exploration depth of the octree.

To increase the locality of computations and to minimize the cache misses, the blocks of data must be made spatially coherent by a preprocessing step of the model. To reduce the execution time of this preprocessing step, we suggest to use two levels of granularity for the octree management. The coarse level, which is precomputed, is sufficient to create blocks of data that can be loaded into memory directly (about 64 KB). The fine level of the octree, which is used for efficiently finding intersections between rays and triangles, is built during computations with a lazy algorithm.

Using voxels as blocks allows an implementation of an asynchronous process for loading blocks. According to Matt Pharr [Pharr and al, 1997] a buffer of rays could be used to obtain the ability to treat rays while loading blocks. Furthermore it will be possible to examine various strategies for selecting rays from this buffer to increase locality and to improve performance.

With these mechanisms we expect to improve scalability when using more machines. The performance will allow us to realize animation in "real time"

and to face new topics like dynamic moving of objects [Reinhard and al, 2000, Lext and al, 2001] or the elimination of temporal artifacts [William and al, 2001].

Acknowledgments

The authors want to thank Christophe Coustet and Sébastien Bermes for their invaluable councils on the techniques for wave propagation simulation and especially photorealistic rendering.

References

[Amdhal, 1967] G. Amdhal. *Validity of the single-processor approach to achieving large scale computing capabilities.* In AFIPS Conference Proceedings vol. 30 (Atlantic City, N. J., Apr. 18Ű20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

[Appel, 1968] A. Appel. *Some techniques for shading machine renderings of solids.* In Proc. Spring Joint Computer Conference (Atlantic City, April 30-May 2, 1968), AFIPS Press, Arlington, Va., pp. 37-45.

[Bermes and al, 1999] S. Bermes, B. Lécussan, C. Coustet. *MaRT : Lazy Evaluation for Parallel Ray tracing.* High Performance Cluster Computing, Vol 2 Prentice Hall 1999.

[Cozette, 2003] O. Cozette. *Contributions systèmes pour le traitement de grandes masses de données sur grappes* Thèse de l'Université de Picardie Jules Verne, Amiens, soutenue le 18 décembre 2003.

[Demarle and al, 2003] D. E. Demarle, S. Parker, M. Hartner, C. Gribble, C. Hansen. *Distributed Interactive Ray Tracing for Large Volume Visualization.* IEEE Symposium on Parallel and Large-Data Visualization and Graphics October 20 - 21, 2003 Seattle, Washington.

[Farizon and al, 1996] B. Farizon, A. Ital. *Dynamic Data Management for Parallel Ray Tracing.* Computer Science Department, Technion, Haifa, Israel Ű Mars 1996.

[Glassner, 1984] A. Glassner. *Space subdivision for fast ray tracing.* IEEE Computer Graphics and Applications. 4(10), pages 15-22, 1984.

[Hall, 2001] D. Hall. *The AR350: Today's ray trace rendering processor.* In Proceedings of the Eurographics/SIGGRAPH workshop on Graphics hardware, Hot 3D Session 1, 2001.

[Lext and al, 2001] J. Lext, T. Akenine-Möller. *Towards rapid reconstruction for animated ray tracing.* In Eurographics 2001 Ű Short Presentations, pages pp. 311Ű318, 2001.

[Oberhumer, 2002] M. F. X. J. Oberhumer. *http://www.oberhumer.com/opensource/lzo/.*

[Pharr and al, 1997] M. Pharr, C. Kolb, R. Gershbein, P. Hanrahan. *Rendering Complex Scenes with Memory-Coherent Ray Tracing.* Proceedings of SIGGRAPH 1997.

[Purcell and al, 2002] T. J. Purcell, I. Buck, W. R. Mark, P. Hanrahan. *Ray Tracing on Programmable Graphics Hardware.* In to appear in Proc. SIGGRAPH, 2002.

[Purcell and al, 2003] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, P. Hanrahan. *Photon Mapping on Programmable Graphics Hardware.* Graphics Hardware 2003. pp. 41-50, 2003.

[Qureshi, 2000] K. Qureshi, M. Hatanaka. *An introduction to load balancing for parallel ray-tracing on HDC.* Current Science, Vol. 78, pp. 818-820, No. 7, 10 april 2000.

[Reinhard and al, 2000] E. Reinhard, B. Smits, C. Hansen. *Dynamic acceleration structures for interactive ray tracing*. In Proceedings Eurographics Workshop on Rendering,pages 299–306, Brno, Czech Republic, June 2000.

[Schmittler and al, 2002] J. Schmittler, I. Wald, P. Slusallek. *SaarCOR - A Hardware Architecture for Ray Tracing*. In Proceedings of EUROGRAPHICS Graphics Hardware 2002.

[Wald and al, 2001a] I. Wald, P. Slusallek, C. Benthin, M. Wagner. *Interactive rendering with coherent ray tracing*. In Eurographics 2001.

[Wald and al, 2001b] I. Wald, P. Slusallek, C. Benthin. *Interactive distributed ray tracing of highly complex models*. In Proceedings of the 12th EUROGRPAHICS Workshop on Rendering, June 2001. London.

[Whitted, 1979] J. T. Whitted. *An improved illumination model for shaded display*. ACM Computer Graphics, 13(3):1–14, 1979. (SIGGRAPH Proceedings).

[William and al, 2001] M. William, S. Parker, E. Reinhard, P. Shirley, W. Thompson. *Temporally coherent interactive ray tracing*. Technical Report UUCS-01-005, Computer Graphics Group, University of Utah, 2001.

III

NUMERICAL COMPUTATIONS