Hybrid Scheduling for Parallel Ray Tracing

Hybrid Scheduling for Parallel Ray Tracing

rapport

ter verkrijging van het diploma
voor technologisch ontwerper aan
de Technische Universiteit Delft

door

Erik Reinhard

Examencommissie:
Prof. dr ir F. W. Jansen
Prof. dr F. J. Peters
Prof. dr ir J. van Katwijk

Preface

Ray tracing is an easy algorithm to explain, but difficult to implement efficiently. And even then performance is a difficult issue, which calls for every sequential and parallel optimisation that one can think of. This report relates a number of design choices which allowed a sequential ray tracer to evolve into a system for parallel ray tracing and is the result of a two-year (TWAIO) course on parallel rendering, undertaken by the author at the Delft University of Technology, Department of Mathematics and Informatics.

As ray tracing is one of those applications that do not have a regular structure, such as for example solving large matrices and sorting algorithms etc., good parallel solutions of ray tracing are likely to exhibit rather complicated control and data flow. It is far from easy to predict which objects will be referenced most and which processor is going to need what data in what time step. Regular problems are easy in this sense, so that, dependent on the type of application, either data parallel solutions or demand driven approaches usually result in good speed-ups. Because ray tracing and radiosity algorithms lack sufficient structure, for each task generated it should be decided whether it is cheaper to transfer the task to the processor that stores the relevant data, or whether the appropriate data should be fetched from a remote processor. This basic idea leads rather naturally to a combination of data parallel and demand driven scheduling, called hybrid scheduling.

Coherence can be used to distinguish between demand driven tasks and data parallel tasks. If there is a number of tasks that will reference the same small amount of data, these tasks can easily be sent to a processor along with the data in demand driven manner. If for some set of tasks the amount of data referenced becomes too large, then these tasks are better executed data parallel by sending the tasks to the processors that already store the data.

This scheme was suggested by Jansen et. al. [11] to be applied to parallel ray tracing. To prove the validity of this assumption was the main topic of this project.

The TWAIO design engineering course consists of two parts, each of which takes roughly one year to complete. The first year was filled with courses on parallel rendering and computer graphics, which created a basic understanding of the area of research.

The second year of the course was devoted to the TWAIO engineering project. The goal of this project was to develop and implement a parallel programme for realistic visualisation based on the hybrid scheduling approach. The requirements were stated

as follows:

- The system should be able to process (very) large model data bases.
- The system should be implemented on basis of the PVM communication library to allow the programme to run on both distributed systems (clusters of workstations) and dedicated parallel multiprocessors.
- An existing sequential programme (Rayshade) should serve as a starting point for parallelisation.

The project started september 1994 and finished september 1995. At that point the implementation was not yet fully functional. Only models of limited size could be rendered. In the remainder of 1995, the memory bottlenecks were removed, allowing the system to render larger models. The results so far show that hybrid scheduling indeed helps to balance the load, although the speed up obtained with hybrid scheduling, is still a somewhat disappointing. For models with large numbers of light sources or many reflective surfaces, the computation is dominated by the data parallel network. The computational complexity of the data parallel network is not fully matched by the demand driven component. This obviates the need to shift yet more work into the demand driven component of the algorithm. This report will discuss the causes and (possible) sollutions to these problems in detail.

In the course of the project, a number of papers were published on this subject and these are added to this report as appendices ([16] [17] [18] and [4]).

It would have been impossible to complete this work without the invaluable help of my tutor Erik Jansen. Many of the discussions and brainstorm sessions we had, led to new ideas and insights which improved our scheduling algorithm. Also, our graduate students Lucas Tijssen, Maurice van der Zwaan and Maurice Bierhuizen provided new incentives to the project with their enthusiasm and skills. Finally, the people from IC^3A, in particular Dick van Albada and Benno Overeinder from the University of Amsterdam, provided computing power and gave valuable support.

Contents

1 Introduction **1**

2 Previous scheduling techniques **5**

3 Hybrid scheduling **7**

4 Optimisations **11**

5 Implementation issues **15**

6 Results **17**
6.1 Test environment 18
6.2 Optimal parameter settings 19
6.3 Performance of data parallel network 23
6.4 Model dependence 25
6.5 Comparison between scheduling techniques 26
6.6 Large models 28

7 Conclusions **31**

8 Future work **33**

Appendices **37**

A Pyramid Clipping for Efficient Ray Traversal **39**
A.1 Introduction 39
A.2 Pyramid - bintree intersection 40
A.3 Ray traversal 44
A.4 Implementation and Evaluation 44
A.5 Conclusions and future work 47

B Hybrid Scheduling for Efficient Ray Tracing of Complex Images **51**
B.1 Introduction 51
B.2 Data parallel ray tracing 53
B.3 Demand driven pyramid tracing 55

B.4 Architecture and implementation 57
B.5 Conclusions 58

C Environment Mapping for Efficient Sampling of the Diffuse Interreflection 61
C.1 Introduction 61
C.2 Method 63
C.3 Experiments 66
C.4 Discussion 70

D Scheduling Issues in Parallel Rendering 75
D.1 Introduction 75
D.2 Ray tracing 77
D.3 Data parallel ray tracing 78
D.4 Demand driven pyramid tracing 81
D.5 Shadow rays 82
D.6 Architecture and implementation 82
D.7 Conclusions 84

Summary 87

Samenvatting 87

1 Introduction

Rendering algorithms such as ray tracing (explained in figure 1.1) and radiosity provide a high level of realism, which is what is required for many applications in both science and industry. Rendering could play a valuable role in design, provided that high image quality is paired with both interactive rendering times and the capability to render very large models. For example in greenhouse design, it is very important that light is obstructed by the framework as little as possible (see figure 1.2).

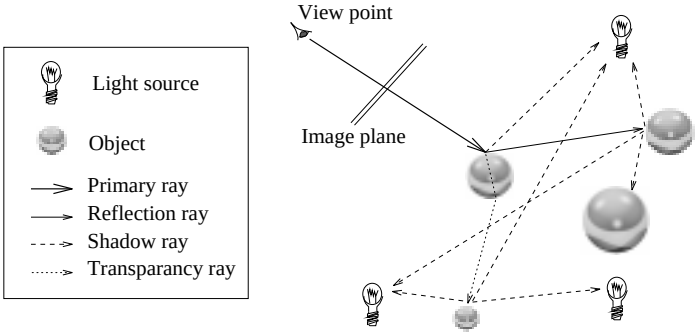


Figure 1.1: Ray tracing consists of shooting primary rays from the eye point through the screen and tracing reflected and refracted rays recursively if they intersect with objects. At each intersection rays are also shot towards light sources for shadow testing.

For other applications, such as scientific visualisation, ray tracing may provide visual cues which are otherwise difficult to attain, see for example figure 1.3. Again, the models used in this field are usually extremely large, while at the same time short rendering times are also preferred. Meeting such demands requires an efficient, possibly massively parallel implementation of a rendering algorithm. In this report the focus is on ray tracing, but whenever techniques applicable to radiosity are presented, this will be mentioned as well.

Previous attempts to parallelise a ray tracing algorithm mainly fall into two categories: demand driven scheduling and data parallel or data driven scheduling. The former method is associated with 'bringing the data to the computation' and the latter can be viewed as 'bringing the computation to the data'. Both have advantages and disadvantages and generally speaking, the advantages of the one are the disadvantages of the other and vice-versa. It is therefore rather obvious to try and mix both scheduling mechanisms into a hybrid scheduling technique which retains the advantages and solves the disadvantages of both. A quick review of demand driven and data parallel

Figure 1.2: Greenhouse lighting. (Model supplied by Bom B.V., Naaldwijk. Rendering done by Maurice Bierhuizen, using Radiance.)

processing, as well as previous attempts at hybrid scheduling, is given in chapter 2.

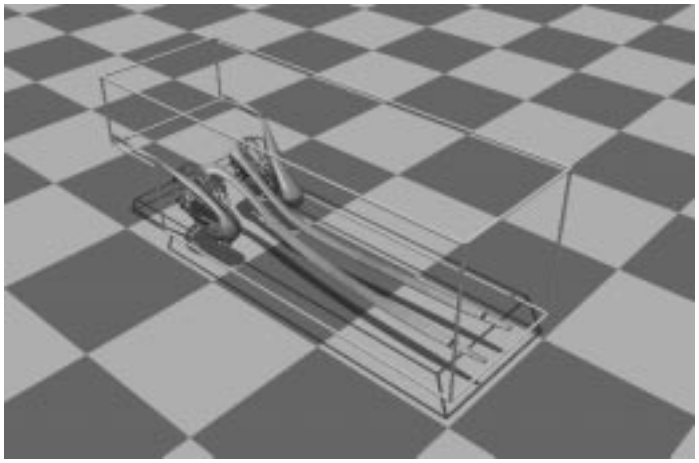


Figure 1.3: Flow visualisation using ray tracing. (Flow data supplied by G. Segal, Faculty of Technical Mathematics and Informatics, Delft University of Technology)

A key notion in this report is ray coherence [21] [9], defined as the probability that rays with the same origin and similar directions, will hit the same objects. An example of ray coherence between primary rays is given in figure 1.4. By pre-selecting the dark objects in this figure, and discarding the light balls, all the primary rays depicted can be traced accurately using the pre-selected objects only. This saves dramatically on the number of intersection tests and can also help reducing the communication overhead in parallel ray tracing. In ray tracing, coherency is highest for primary rays and shadow rays that sample light sources. In radiosity applications, coherence can be observed between rays used for hemisphere sampling.

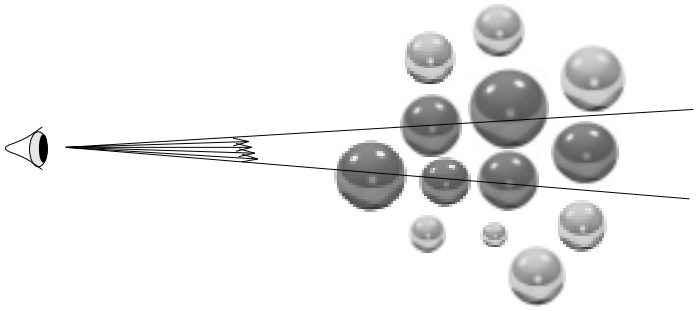


Figure 1.4: Ray coherence between primary rays.

In the hybrid scheduling algorithm described in this report, coherence is applied to distinguish between demand driven and data parallel tasks. Rays that are coherent, i.e.

primary rays in our case, can be scheduled demand driven, and the remainder of the rays is scheduled data parallel. The machinery we use to employ coherence is called Pyramid Clipping, or PyraClipping for short [4]. In the remainder of this report a basic understanding of PyraClipping is assumed, as well as knowledge about general computer graphics issues such as spatial subdivision techniques.

After discussing previous scheduling techniques in the following chapter, our hybrid approach is presented in chapter 3. A number of optimisations have been applied to this basic algorithm, mainly to further reduce the amount of communication overhead and to remove the memory and computational bottlenecks that were present in the host process. These optimisations are presented in chapter 4. The implementation and test results are subsequently presented in chapters 5 and 6 respectively and this report is concluded with chapters discussing results and future work.

2 Previous scheduling techniques

In the past, quite a few attempts have been made to parallelise ray tracing. Perhaps the most obvious of these involves replicating the object data over the processors and splitting the screen into a number of regions [15] [3] [14]. The pixels in each region are then processed by a different processor, which has no need for object communication, as it already stores the entire scene description in its local memory. This demand driven approach is referred to as a screen space subdivision and yields an optimal load balance, as well as minimal communication. Load balancing is achieved by only assigning new regions to processors that have just finished their tasks; a property of demand driven scheduling. However, as the model data is copied to each processor's memory, the above algorithm is not scalable, as larger models will occupy all the memory available.

In order to relax the memory constraint, this approach has been adapted to distribute the objects over the processor's memories. For tracing a ray, a given processor may then need object data that is not available from its own local memory, so that object communication becomes necessary. Because fetching objects from other processors introduces an enormous communication overhead, caching techniques can be effectively used to remedy this problem [7]. However, caching can never completely eradicate object communication, which is global by nature. To minimise the number of cache misses, coherence may be used as a guide to form tasks [20].

As too much global object communication prevents scalability, data parallel algorithms may be used instead. These algorithms also distribute objects over processors, but usually do this according to a (regular) spatial subdivision, such as a grid of voxels [5] [2] [12]. In data parallel solutions objects are not communicated at all, at the expense of task communication. A ray that starts in a particular voxel is assigned to the corresponding processor. As long as the ray stays in that voxel, it is traced as in a sequential ray tracer. However, if the ray leaves this voxel without intersecting any objects, the ray is communicated to the processor that stores the neighbouring voxel. This indicates that communication is only local, which, together with the fact that objects are distributed over the processors, makes data parallel solutions at least theoretically scalable. A disadvantage is that it is very difficult to balance the load in such schemes. This is mainly due to the unequal distribution of objects over the processors and the occurrence of hot spots due to both light sources (figure 2.1) and the camera point.

The descriptions of both demand driven and data parallel methods show that their advantages and disadvantages with respect to load balancing and communication, are largely complementary. If done properly, combining data parallel and demand driven algorithms into one hybrid scheduling algorithm may therefore overcome the problems associated with either of the two, while retaining the advantages of both. Ray tracing algorithms may be split into a demand driven and a data parallel part in various ways.

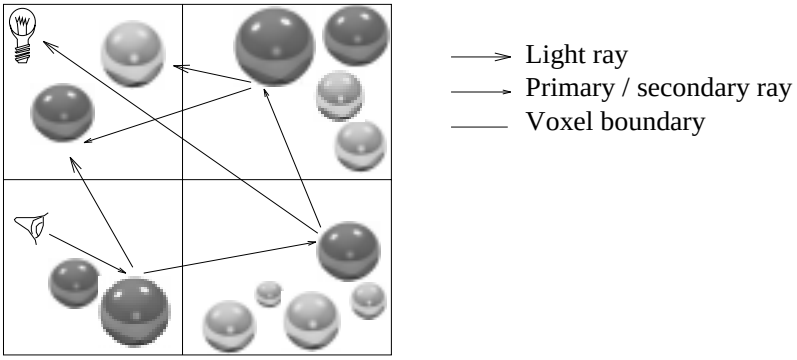


Figure 2.1: Light sources attract a lot of work, because at each intersection, a light ray is sent towards the light source.

The data parallel part should then provide a basic, yet possibly unequal load, which may then be balanced by assigning demand driven tasks to the processors with the lowest loads. In order to be successful, such a functional decomposition should distinguish demand driven tasks which do not require much data. This allows demand driven tasks to be scheduled on a processor without excessive data communication. Also, the amount of work involved with each demand driven task should be sufficient to balance the load adequately.

One such hybrid scheduling algorithm splits the work into ray traversal and ray tracing tasks [19]. The ray traversal tasks intersect rays with a spatial subdivision. This is not data intensive, so these are scheduled demand driven. The ray tracing is performed in data parallel fashion as described above, providing the basic load. Though in itself a good idea, the ray traversal tasks do not provide enough work to balance the unequal load introduced by the data parallel processes.

Recognising the potential of hybrid scheduling, a new criterion for the division of the algorithm into demand driven and data parallel tasks has been developed, a description of which follows in the next chapter.

3 Hybrid scheduling

For the sake of clarity, in this chapter only a very basic hybrid scheduling algorithm is described. A number of features has been added to improve the performance of this algorithm and to remove possible bottlenecks. A description of these is deferred to chapter 4.

Our basic hybrid scheduling algorithm [16] [17] assumes the presence of a number of identical processes (from here on called 'trace processes'), each of which is capable of handling both demand driven and data parallel tasks. There is also a host process, which performs pre-processing and schedules the demand driven tasks. The host will additionally retrieve pixel values from the system and write these to file. The data parallel tasks will provide the processes with a basic load, which may vary from process to process, and the demand driven component will be used to equalise the load.

Although it would be difficult to measure coherence, it is safe to say that secondary rays generally exhibit less coherence than primary rays, or maybe even no coherence at all. For these rays, object communication would not pay off, so these are scheduled data parallel. Primary rays, however, are coherent, and a certain amount of object communication would be justified, as it gives the freedom to schedule these in a demand driven way. Hence, coherence is our criterion to decide which rays to schedule in what way [11].

For tracing secondary rays in a data parallel manner, the scene is subdivided into a number of voxels, where each trace process will store the objects of one such voxel in its local memory. To speed up local tracing, each voxel is further subdivided using a bintree spatial subdivision, effectively resulting in a grid of bintrees data structure, see figure 3.1.

If a ray enters one of the voxels, the associated trace process will trace the ray through its voxel. If an intersection with an object occurs, secondary rays are spawned and these traced recursively. If the ray, however, leaves the voxel, which is detected because no intersection is found, the ray is transferred to the process holding a neighbouring voxel. That trace process will continue tracing the ray.

If a ray terminates, a colour value is computed. In the case of light rays this happens when an intersection with either an object or the light source itself is found and for reflection rays this occurs if the maximum ray depth is reached or a diffuse object is intersected. This colour value is returned to the process which spawned the ray. To this extent, each process keeps track of the intersections it finds and of the secondary rays that are spawned as a result of that. As the order in which colour values for a particular intersection are returned from other processes, is indeterminate, these values are stored until all are returned. The process then performs the shading for that intersection, yielding another colour value. This value is in turn handed to the processor

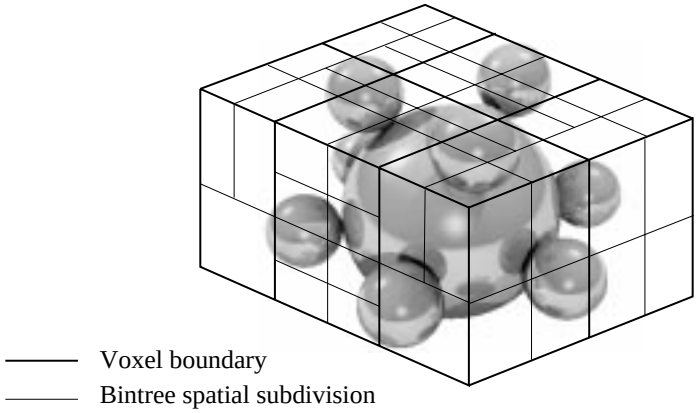


Figure 3.1: Data structure used for tracing secondary rays.

that spawned the corresponding ray.

This idea of delayed shading is introduced to allow processes to do useful work after it transfers a ray to another process. It can then continue tracing the next ray, instead of waiting for a colour value to be returned. The extra bookkeeping involved outweighs the disadvantage of the idle time that would otherwise be introduced.

Each trace process is, next to data parallel ray tracing, capable of handling demand driven primary rays. Before a process can commence with such a task, it has to be prepared by the host process. This preparation is basically the same as the pre-processing step of the Pyra-Clip algorithm. The host selects a bundle of rays and intersects the bounding pyramid of these rays with the grid of bintrees, see figure 3.2. This results in either a list of objects or a list of references to objects, which is communicated to a trace process as a demand driven task. A combination of cells with objects and cells with only references is also possible. For those cells that do not contain objects, a trace process may either fetch those objects from other processes, or may decide to hand over rays to the data parallel cluster.

The host process now holds the entire scene description and consequently all cells that are put in a cliplist, contain objects instead of references. This presents the host with a memory bottleneck, the removal of which is described in chapter 4.

When a trace process receives a demand driven task, i.e. a list of objects and a pyramid, it starts tracing the individual primary rays using this list. As this list is valid for all the rays in the pyramid, the average amount of objects communicated per ray is low, thereby justifying this approach.

If an intersection with an object in this list is found, secondary rays need to be spawned. For this, the voxel in which the intersection occurred, is computed. Subsequently, the primary ray, with the intersection point, is sent to the corresponding process which will actually spawn the secondary rays in a data parallel manner. This process will also perform the shading for the primary ray and send the resulting pixel value back to the host. Figure 3.3 schematically presents the message flow between

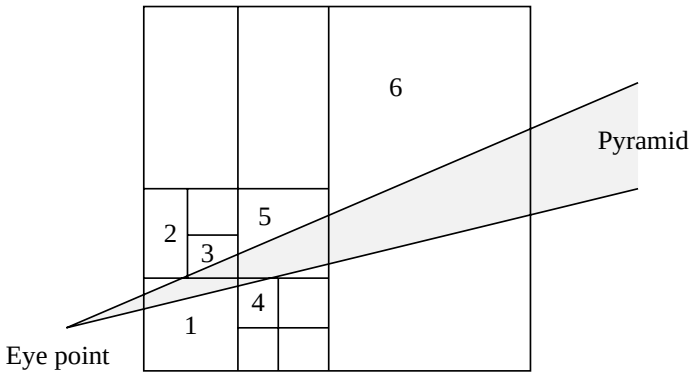


Figure 3.2: Cells 1-6 are included in the cliplist for this pyramid.

processes.

Finally, there are various ways in which primary ray tasks can be scheduled to processes. The object of all of these is to balance the unequal load induced by the data parallel handling of secondary rays. Two main classes of scheduling can be distinguished: static and dynamic scheduling.

Static scheduling means that before the computation starts, the host determines which processes will be selected for receiving demand driven tasks and which will not receive such tasks at all. This selection is based on an estimation of the amount of work each process will receive due to the data parallel tracing of rays. The better this estimation, the more successful any load balancing will be. Which parts of the scene attract more work than others is partially determined by the geometry of the scene, but also by the location of the eye point and the position of the light sources. To a certain extent it should therefore be possible to estimate the distribution of work by counting objects in each voxel or by determining the position of the light sources. These criteria are used to select processes which may receive additional demand driven tasks.

A form of (semi-) dynamic scheduling is also possible. To accomplish this, the host counts the number of rays it has sent to each process and also keeps track of the amount of rays that has returned in the form of pixel values. If the number of primary rays still present with a particular process drops below a certain threshold level, a new primary ray task is sent to that process. Because each process gives data parallel ray tasks a higher priority than demand driven tasks, counting the number of primary rays in the system should give a good estimation of the amount of work still available in the system. However, as some processes may be very busy computing secondary rays, while others are waiting for results to return from these busy processes, the estimation of work still available should be on a per process base, instead of global. Therefore, the expectation is that semi-dynamic scheduling, although better than static scheduling, still performs sub-optimal.

Finally, true dynamic load balancing may be attempted by having trace processes notify the host upon completion of a demand driven task. The host may then decide

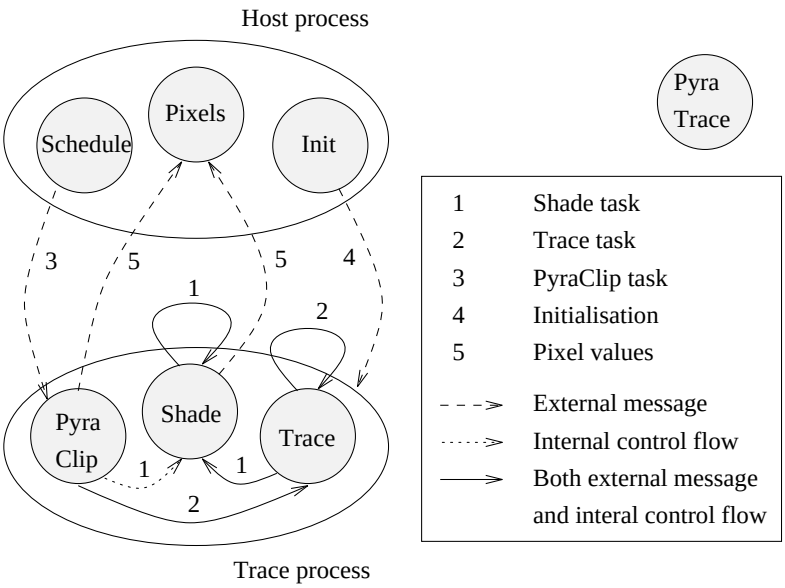


Figure 3.3: Control and message flow between processes.

to sent new demand driven tasks to that processor. Dynamic scheduling is clearly the most accurate, but this accuracy is paid for in the form of extra control messages.

4 Optimisations

The algorithm described in the previous chapter still suffers from a few deficiencies and also allows a number of optimisations. These are discussed in this chapter.

The main problems not addressed so far are due to the host process. This process presents a memory bottleneck as well as a computational bottleneck. The memory bottleneck is due to the fact that the host parses the input file and creates the spatial subdivision. In order to determine which process receives which objects, the scene has to be subdivided into equal sized voxels. To do this, the bounding box of the entire scene needs to be computed, after which the objects can be distributed over the processes. This memory bottleneck is addressed by incrementally computing the bounding box, while deleting the objects after they are used. In a second pass, the objects can be reread and distributed over the processes, using bounding box information obtained in the first pass.

The Pyra-Clip pre-processing performed by the host for each demand driven task also causes a memory bottleneck. This step creates a list of objects by intersecting a pyramid with the spatial subdivision structure. However, the objects contained within the selected bintree cells, are currently put in the list as well. For this reason, the host not only stores the spatial subdivision structure, but also the entire model.

This memory bottleneck is resolved by using the coherence criterion again. As primary rays that are further away from the camera point, tend to diverge, the coherence between them diminishes. This means that bintree cells that are far away from the camera point, will usually contain objects that are not very often referenced for primary rays. We can therefore reduce the length of the lists that are generated for demand driven processing and still find most of the intersections of primary rays using the lists. The pre-processing then involves only the objects that are closest to the view point, so that other objects need not be present at the host (figure 4.1). Alternatively, the host may generate cliplists that partially contain objects and only includes references to objects for cells that are far away from the eye point.

Upon detection of an empty cell, a trace process may either redirect the primary rays to the data parallel cluster, or the trace process may decide to fetch the objects it needs from other processes. Objects should be fetched whenever object communication is cheaper than ray task communication. Suitable criteria to detect this should be developed. For example, the number of remaining rays should still be large enough to justify object communication. Also, the communication distance for objects and rays could be compared to see which communication may be cheaper.

The host may also prove to be a computational bottleneck due to the Pyra-Clip pre-processing. Although it is argued that preparing demand driven tasks is a cheap process, by adding more processors to the system, it will become relatively expensive.

In order to overcome the computational bottleneck, the host may be split up into

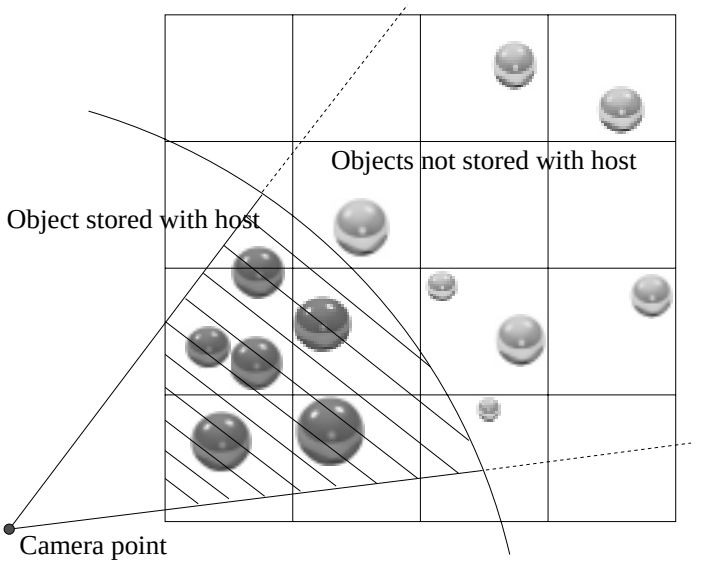


Figure 4.1: Objects close to the camera point will be stored with the host.

several new processes. In that situation there would still be a master process that parses the input, but in addition to spawning a number of trace processes, it also spawns a number of sub-hosts. Each sub-host is capable of generating demand driven tasks for the trace processes it governs. Collecting pixel values may either be done by the sub-hosts or by the master process.

By distributing the PyraClip tasks over multiple processes, the workload of the former host is also distributed, thereby solving the computational bottleneck of the host.

Currently, the host functionality is performed by the subhosts, which store the subset of the scene that is closest to the eye point and within the viewing pyramid. After pyraclipping, a possibly incomplete cliplist is sent to a trace process as a demand driven task. This trace process subsequently traces the individual rays and should these rays survive the objects in the cliplist, they are transferred to the data parallel cluster.

The light sources present in the model may provide a computational hotspot, as every intersection between a ray and an object generates a shadow ray (figure 2.1). Because any data parallel network is very sensitive to the load imbalances due to such distribution of light sources, a separate solution for light rays is presented in the form of light caches. The work associated with tracing shadow rays is preferably performed as another demand driven task, but may alternatively be executed locally by the process that generated the shadow ray. This also minimises task communication for shadow rays and moves work from the processes that store light sources to processes that find ray object intersections. Because there are usually more processes with objects than processes with light sources, the hotspots due to light sources are relocated and spread out over a larger number of processes.

To prevent communication for shadow rays, currently a process caches all objects between the light sources and its voxel, see figure 4.2. This eliminates the need for shadow ray communication completely, but for large models with many light sources, the caches may become quite large. The cache size should therefore be pre determined and light rays that do not intersect with local objects and subsequently generate a cache miss, should be handled as ordinary data parallel rays. This is not yet implemented.

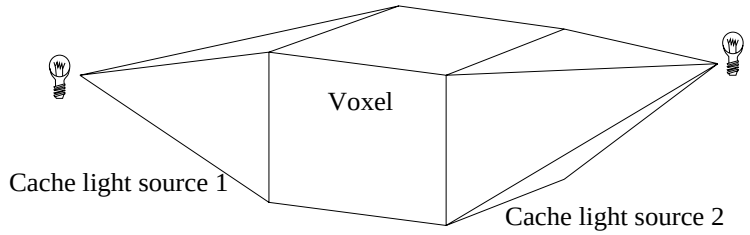


Figure 4.2: Each process has the objects of its voxel and for each light source a light cache.

Finally, in parallel implementations it is always very important to minimise the number of messages sent. Due to startup latencies, it is advantageous to have a few large messages than a large number of small messages. There are a number of possibilities to bundle different messages that originate from a particular process which are destined for another process. For each type of message, a number of buffers equal to the number of trace processes is created. Messages are collected in these buffers until either the process is temporarily out of work, or a maximum buffer size is reached for a particular buffer. In the first case all non-empty buffers are sent, while in the second case just one buffer is sent. The maximum buffer size is a user defined parameter influencing message densities. Buffers are created for pixel values, trace tasks and shading tasks which need to be transferred to another process.

The effect of all these optimisations on processing time will be discussed in chapter 6.

5 Implementation issues

In parallel computing, not only a good algorithm should be devised, but this algorithm should also be mapped onto a suitable architecture. Because of the irregular data accesses of ray tracing, the hardware should be as flexible as possible. In addition, as tracing single rays are usually the smallest tasks in ray tracing, the hardware should allow a rather course grain mode of computation. For this reason we opt for MIMD machines. Within this class, the choice between shared memory multiprocessors and message passing multicomputers is a little more difficult. Shared memory machines have the advantage of good programmability and good speed-ups can be attained with them. However, up until now, these machines were not scalable, because memory access becomes a bottleneck for larger numbers of processors. Should hardware scalability be achieved in the future, shared memory systems will be a very attractive alternative.

As this is currently not the case, message passing or distributed memory multicomputers are used for our parallel ray tracing algorithm. In message passing systems, processors can be added so that the computation power increases, as well as the total amount of memory and communication bandwidth. From the hardware point of view, the only problem that most message passing computers have, is that communication bandwidth and processing speed are poorly matched. As a consequence, in addition to the greater programming effort that these computers require, extra attention should be paid to keep communication overhead to a minimum.

When using distributed memory machines, the network topology is the next issue to be addressed. It can be shown that if an algorithm completes its computations within a certain amount of time on a given network, the algorithm can be modified to run on a different network in the same order of time. For example, if an algorithm is O(n^2) on a hypercube, it can be modified to run in O(n^2) on a mesh topology as well. The choice of network is therefore not all that important. For simplicity we chose a 2D mesh, as this conveniently maps onto a 2D grid of voxels. Another reason for choosing this architecture obviously has to do with hardware availability.

In this case, the hardware used for taking measurements is a Parsytec PowerXplorer with 32 Motorola MPC 601 processors running asynchronously at 80 MHz. Each processor is connected to a T800 transputer, which is used for communication purposes. Each processor also has a 32 Mb local memory, adding up to a total of 1 Gbyte RAM. Peak performance is rated at 2 Gflop/s.

The PVM3 library was chosen to handle communication [6]. This software is widely used and runs on a great variety of platforms, which allows the implementation and debugging to take place on a small cluster of workstations, while testing and measuring is performed on the Power Xplorer. Portability is one of the great assets of PVM, in addition to the relative simplicity of programming.

Because PVM was primarily developed for clusters of workstations, the network topology is completely shielded from the user. This is an advantage when using clusters of workstations, but a disadvantage with respect to multiprocessors, where network topology information is present. In our implementation we assume a 2D mesh topology onto which we map a 2D grid of voxels, but due to this property of PVM, we cannot be sure if the actual mapping is accurate, i.e. if neighbouring voxels are mapped to neighbouring processors. For optimal performance this would be required, so shielding network information is a big disadvantage in PVM.

Another more general problem with PVM is performance related. As PVM is a library implemented using native communication primitives, which may vary with each computer, startup latencies and throughput are lower than if an application uses the communication primitives directly. The differences between the operating system Parix on the Parsytec Power Xplorer and PVM running on top of Parix, are not too alarming, however [10].

Finally, the basis for our hybrid scheduling implementation is the public domain ray tracer Rayshade [13], which has a large user base and is well written. This makes it a good basis for parallelisation. There are also a few difficulties when parallelising an existing programme like Rayshade. The most important of these is the recursive nature of ray tracing, which is reflected in the programme structure of Rayshade. Recursive algorithms are notoriously difficult to parallelise. For this reason, the core of Rayshade needed some thorough rewriting in order to allow tasks to be interrupted halfway and then to be resumed later. This way, a trace process does not have to wait for a shading result after it sends a ray to a neighbouring trace process, but can handle all sorts of other tasks while the next trace process computes a shading result.

6 Results

The goal of parallelising an algorithm is to reduce the overall computation time, preferably linear in the number of processors used. To achieve this, the overhead introduced by the parallelisation process, should be minimised. Overhead generally consists of communication and idle time. Load imbalances may occur due to the unequal distribution of work or to excessive communication which clutters up the network. Therefore, a good speed-up can be achieved if the amount of communication is kept to a minimum and the load is well balanced. Both issues are addressed in this chapter.

Speed-up may be defined as:

$$S_{N,P} = \frac{T_1}{T_{N,P}} \tag{6.1}$$

where N is the number of tasks and P is the number of processors. In the hybrid scheduling algorithm, each ray can be viewed as one task as long as it is not transferred to another process. In the data parallel section, whenever a ray is communicated to the next process, a new task is generated. Also, at an intersection point, new ray tasks are generated, where the direction of the rays depends on geometry. This means that the scene to be rendered (geometry, amount of reflectivity) plays an important role in determining the number of tasks generated and also on which processes these tasks will be executed. For this reason, it is very difficult, if not impossible, to estimate the general performance of the algorithm a priori. Obviously, speed ups can be computed with the above formula after images are generated using different numbers of processors.

Efficiency is a related parameter, which is normally defined as speed-up over the number of processors used to attain this speed-up. This definition also necessitates a comparison between runs of the programme with different numbers of processors, i.e. at least a run with only one processor should be done. It is more convenient to compute an image using a certain number of processors and immediately have an indication of its performance. In order to achieve this, for each processor the idle time is measured as well as the total running time. From these numbers, an efficiency parameter can be computed with:

$$E_{N,P} = \frac{1}{P} \sum_{i=1}^{P} \frac{T_i - T_{i,idle}}{T_i} 100\% \tag{6.2}$$

In this report, the term efficiency will be used in this slightly unorthodox way. Note that this implies that the overhead caused by the PVM packing routines and the extra book keeping that is done to avoid recursion, is discarded completely. As this overhead is unavoidable, this is not a real disadvantage. Finally, synchronisation time need not be measured, because there are no synchronisation points during the computation. Therefore, it is claimed that efficiency measured this way still correlates with speed-up and

that it is a good parameter to evaluate the performance of an implementation, although it will give optimistic figures compared with the traditional efficiency parameter. In the tables and figures in this chapter, the efficiencies are presented in the columns marked E and timing statistics (all of which are in seconds) are indicated by T.

6.1 Test environment

The models used for evaluation of the data parallel and hybrid scheduling algorithms stem from the SPD model database [8] and are slightly modified to better reflect the type of model these algorithms are designed for. From the balls model the plane is removed and the ten largest balls are made non-reflective, see figure 6.1. Although still on the stiff side, the amount of reflectivity is more conform the reflectivity found in industry models. This model consists of 7381 spheres.

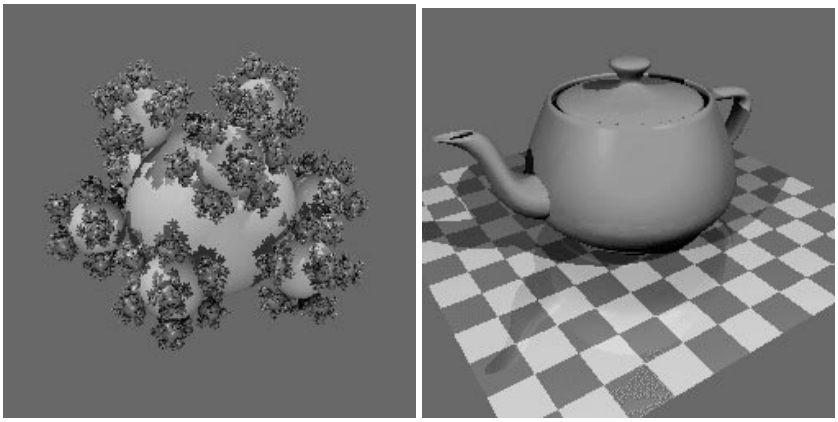


Figure 6.1: Balls model containing 7381 spheres (left) and teapot model with 9264 triangles (right).

The teapot model, also from the SPD database, has 9264 triangles which approximate the surface of the teapot. This model has a diffuse set of ground polygons and a specular surface. Again, the amount of reflectivity is larger than found in realistic models.

When more than four processors are used, both the teapot and the balls model have a rather unequal distribution of objects over space, which mimics the distribution of objects that is expected to be found in real applications.

In order to test the behaviour of the system for large models, some tests have been performed with the balls 6 model (66.430 spheres) and the conference room model, created by Anat Grynberg and Greg Ward (23.966 polygons), see figure 6.2. The balls 6 model has one light source, while the conference room model was tested with 8 and 30 (point) light sources. Testing performed with these models is discussed in section 6.6, while the remaining sections in this chapter discuss the behaviour of the system for the two small models.

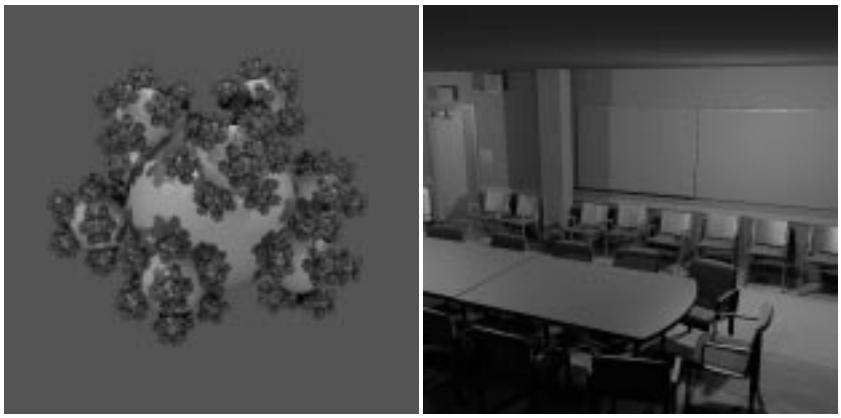


Figure 6.2: Balls model containing 66.430 spheres (left) and conference room model with 23.966 polygons (right). The balls model is shown with three light sources, but measurements were taken with just one light source to lower memory requirements.

Table 6.1: Performance parameters.

Parameter	Optimum
Granularity	100 tasks/message max.
Scheduling	Dynamic
Number of pixels in system	9000 pixels
Pixels per pyramid	16 x 16 pixels
Pyramids per proc	0.8
Number of subhosts	Equal to #slaves

Finally, in all tests performed, the image size is 256 x 256.

6.2 Optimal parameter settings

The hybrid scheduling algorithm has a fair amount of parameters to be set, each of which affects the performance. It is therefore necessary that these parameters are set to optimal values. Table 6.1 lists the parameters that affect performance, together with their optimal settings. Each of these parameters is discussed below.

First of all, the effect of buffering outgoing messages is evaluated. By measuring how many tasks are bundled to a single message on average, the reduction in message densities can be computed. The granularity parameter determines the maximum number of tasks that are allowed to form a single message. If this number is reached, the message is sent right away. The optimum value for this parameter is around 100, allowing a rather spectacular average reduction of 97%. This shows that bundling of messages is a highly effective means to reduce communication requirements, which is

Table 6.2: Dynamic and semi-dynamic scheduling (Balls model; 16 processors).

Criterion	Demand driven trace procs	Pixel count T	Pixel count E	Dynamic T	Dynamic E
500 Objects max	4	34.43	29%	21.26	51%
750 Objects max	9	27.92	34%	24.52	47%
1000 Objects max	11	25.99	37%	28.31	36%
1250 Objects max	16	26.16	39%	26.28	42%
Location	12	29.84	45%	30.25	38%
All but host	15	23.03	41%	20.81	51%
All processes	16	26.75	37%	19.99	58%

always important in parallel processing, and even more so when using PVM.

As far as scheduling is concerned, the static version is not tested on its own, but in combination with semi-dynamic scheduling, i.e. in combination with counting pixel values that are returned to the host and scheduling new tasks based on that count. Static scheduling then consists of determining a number of trace processes that may receive demand driven tasks before the computation starts. This is a subset of the total number of trace processes. The same occurs in dynamic scheduling, where first a subset of trace processes may be selected after which tasks are scheduled over this subset in a dynamic fashion.

Criteria to admit a trace process to the pool of processes that can receive demand driven tasks, are its location in the grid, the number of objects that lie in its voxel, or simply all trace processes.

The different forms of scheduling and the efficiencies obtained using them, are given in Table 6.2. The model used for these computations is the balls model with 7381 objects, no ground plane and the largest ten spheres are diffuse, while the remaining spheres are specular. The number of processors is 16 and the size of the image is 256 x 256.

From table 6.2 we conclude that the more processes are statically determined to be capable of receiving demand driven tasks, the higher the efficiencies obtained. By limiting the number of processes, both semi-dynamic and true dynamic scheduling perform sub optimally. This indicates that none of the criteria mentioned gives a good a priori estimation of the workload. Therefore, some form of dynamic scheduling is necessary.

Table 6.2 also shows that dynamic scheduling performs better than scheduling based on a count of returned pixels. The increased accuracy in dynamic scheduling more than offsets the cost of extra control messages which are needed to inform the host about the end of each task.

At the end of a computation, when most processes have finished, but a few remain busy, performance may drop dramatically. To minimise this effect, the number of pixels

Table 6.3: Rendering time and efficiency as function of the number of subhosts by 16 processors.

	8 x 8 pyramid size		16 x 16 pyramid size	
Number of subhosts	T	E	T	E
Balls model				
0	23.91	38%	19.33	49%
1	37.03	29%	36.13	36%
2	18.73	54%	20.72	54%
4	16.11	61%	21.58	55%
8	16.15	61%	21.28	58%
16	15.18	61%	18.03	62%
Teapot model				
0	62.78	19%	58.70	21%
1	64.34	21%	64.88	22%
2	35.10	40%	36.55	41%
4	26.40	59%	28.34	51%
8	29.05	56%	34.77	51%
16	22.58	45%	26.59	42%

that are being processed at any one time should be as low as possible, while still feeding each processor with sufficient work. This can be regulated by limiting the number of pixels that are sent into the system, i.e. by limiting the number of primary rays. The optimum value for this parameter is determined to be 9000.

A related parameter is the size of the demand driven tasks, i.e. the number of rays within a pyramid. In the sequential algorithm, coherence is optimally exploited if there are 16 x 16 rays in each task [4]. This proves to be the optimal parameter setting for the parallel version as well. Larger tasks mean less opportunities to control the load, while smaller tasks provides the host with a computational bottleneck. However, smaller tasks (8 x 8) can be used if the host task is distributed over a number of processes.

The Pyra clipping task may be performed by the host, but may also be distributed over a subset of the trace processes. The host then merely initialises trace processes and collects pixel values afterwards. The efficiency versus number of subhosts is given in table 6.3 [1]. The demand driven task sizes tested, are 8 x 8 and 16 x 16.

From table 6.3 it can be deduced that adding one subhost, which should give a performance comparable to the zero subhosts case, in fact introduces quite some extra overhead. As the host functionality in that case is almost completely transferred to one of the trace processes, it was expected that the impact on performance would be negligible. The subhosts provide slightly better performance than employing no subhosts if their amount equals the number of trace processes. This effect is also manifest with other numbers of processors, as tables 6.4 and 6.5 indicate.

[1] For these figures, the teapot model has a specular surface and a non-specular set of polygons forming the ground plane.

Table 6.4: Performance subhosts vs. no subhosts (balls model).

		No subhosts		Subhosts = #processes	
Processors	Grid	T	E	T	E
1	1x1	63.10	98%	69.04	100%
2	1x2	47.79	97%	40.16	97%
4	2x2	33.88	92%	29.33	86%
8	2x4	24.67	67%	26.41	63%
16	4x4	15.06	58%	17.37	53%
32	4x8	14.56	40%	11.91	51%

Table 6.5: Performance subhosts vs. no subhosts (teapot model).

		No subhosts		Subhosts = #processes	
Processors	Grid	T	E	T	E
1	1x1		%		%
2	1x2	70.33	96%	62.12	97%
4	2x2	51.84	82%	39.27	75%
8	2x4	41.21	53%	31.19	56%
16	4x4	44.81	27%	23.93	43%
32	4x8	48.27	13%	18.16	37%

Table 6.6: Effect of light cache on performance (balls and teapot models).

		With light cache		Without light cache	
Processors	Grid	T	E	T	E
Balls model					
1	1x1	69.04	100%	59.78	100%
2	1x2	40.16	97%	49.05	96%
4	2x2	29.33	86%	98.94	52%
8	2x4	26.41	63%	88.30	34%
16	4x4	17.37	53%	38.82	42%
32	4x8	11.91	51%	47.03	28%
Teapot model					
1	1x1				
2	1x2	62.12	97%	68.11	97%
4	2x2	39.27	75%		
8	2x4	31.19	56%	372.83	20%
16	4x4	23.93	43%	109.39	26%
32	4x8	18.16	37%	62.42	26%

It can be observed that given multiple subhosts, a demand driven task size of 8 x 8 pixels is preferable over larger task sizes. However, because the host is a bottleneck by an 8 x 8 task size, in tables 6.4 and 6.5 figures are given by a task size of 16 x 16 pixels.

The effectivity of caching all objects between each process' voxel and the light sources, is demonstrated in table 6.6. In this comparison, the number of subhosts equals the number of trace processes. The numbers that are missing in this table are due to memory problems, which should be resolved in the future.

Caching objects between a voxel and the light sources proves to be a useful addition which enhances the performance of the data parallel section of the algorithm. It reduces the amount of communication for shadow rays to zero and moreover relieves the processes that store the light sources from a computational bottleneck.

For larger models and larger numbers of processors, caching all objects between a light source and a voxel may not be feasible. Avoiding this memory bottleneck by caching only a subset of these objects, remains future work.

6.3 Performance of data parallel network

One of the arguments of introducing hybrid scheduling are the bad load balancing properties of data parallel rendering. Before comparing data parallel and hybrid scheduling techniques, first the behaviour of data parallel rendering is evaluated.

In data parallel rendering, it is expected that the voxels with the most objects attract the most work. By counting the number of rays that are processed by a processor, and

comparing these with the objects that are in each voxel, a high correlation should be found. Note, that shadow rays are computed by using cached objects. Light sources, therefore, will not be hotspots. The load that light sources provide, are distributed over the processors that find the intersections. Therefore, load balancing will generally be slightly better, although voxels with high object densities may somewhat easier become hotspots. A bigger advantage should be the considerable reduction in communication requirements, as no communication is necessary for shadow rays.

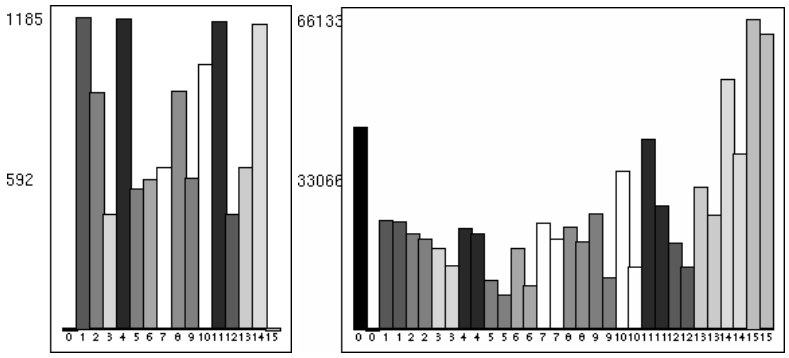


Figure 6.3: Typical object distribution (left) and workload, counted in rays (left bars in right figure). The right bars in this figure present the rays that are communicated to a neighbouring processor.

The object distribution over voxels and the associated workload is depicted in figure 6.3. The workload is very unequal, as expected, but only correlates lightly with the object distribution. We assume that this has to do with the choice of viewpoint, as most processing will be in the voxels near the eye point. The voxels far away from the view point may contain a large number of objects that are not visible. There will not be many intersections in these voxels, despite the amount of objects available. Another possible cause for the absence of a strong correlation may be the fact that communication latencies are different for different communication distances. In data parallel rendering different communication distances may occur for shading messages; ray task messages are communicated only to local neighbours.

In figure 6.3 (right), a large number of rays processed by each processor does not produce an intersection and are subsequently transferred to a neighbouring processor (compare rays processed (left) with rays transferred (right)). Therefore, on average there will be a lot of communication per ray, resulting in a low efficiency. Note that many of these rays are packed into larger messages, so that message densities are not as high as suggested by figure 6.3.

A second assumption regarding data parallel rendering is that the load a processor receives, is on average constant over time. This turns out to be a false assumption, as figure 6.4 shows. The load is clearly not constant over time. A second reason why data parallel rendering behaves rather poorly. This does not have implications for the validity of hybrid scheduling in general, but a hybrid scheduling with a static demand

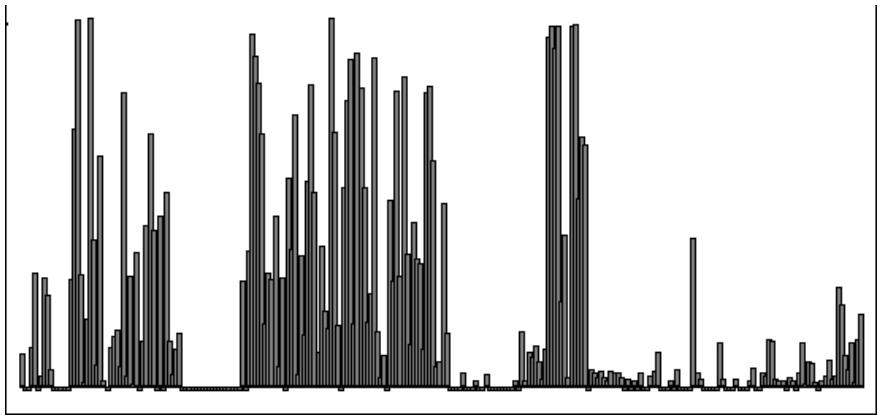


Figure 6.4: Efficiency of one processor over time. Each bar represents a time interval of 0.1 s. The height of a bar represents the fraction of time that work is performed in each interval.

Table 6.7: Rendering time and efficiency of the data parallel algorithm as function of the number of processors.

Processors	Grid	T	E
1	1x1	81.31	98%
2	1x2	85.43	79%
4	2x2	85.80	53%
8	2x4	78.96	47%
16	4x4	87.50	31%
32	4x8	76.60	17%

driven component should prove very difficult.

Again using the balls model, the speed-up characteristics of data parallel scheduling are evaluated. A number of identical images were computed using different numbers of processors. The rendering time of each of them is given in table 6.7. Pre-processing time is not included in these numbers.

Table 6.7 shows rendering times that are even poorer than expected. The almost constant rendering times suggest that the host is the bottleneck, but this is not the case, as figure 6.5 shows.

6.4 Model dependence

Normally, the rendering time is linearly dependent on the size of the image to be computed, but there is also a weaker dependency on the size of the scene. Although the rendering time may vary with different models, in the ideal case the efficiency should

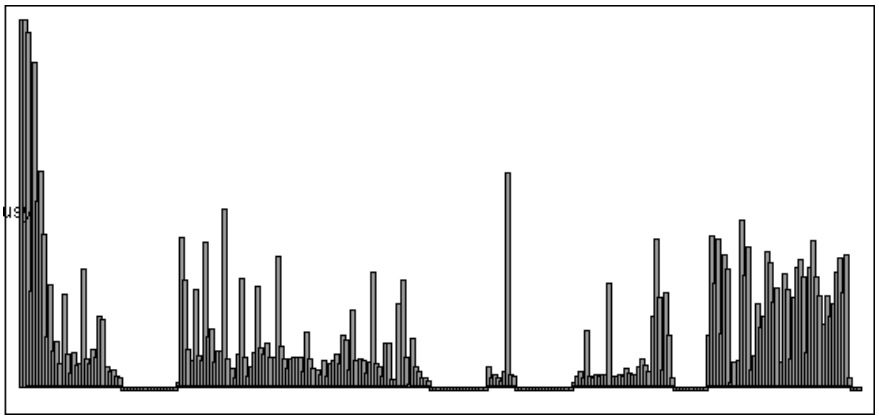


Figure 6.5: Workload of the host as function of time. This workload is obtained when the host controls 16 processors.

be more or less constant.

In practice this will not be the case, as a possible awkward distribution of objects may give rise to such a bad load balance in the data parallel part of the algorithm, that demand driven tasks are not sufficient to balance the load. This effect may be aggravated if there is a substantial amount of reflectivity in the model. If this is the case, then the data parallel cluster will dominate the computation. The balls model allows the amount of reflectivity to be easily adjusted, as in each level of recursion the surfaces in that level may be set to either reflection or diffuse. Results for both hybrid and data parallel scheduling are shown in table 6.8. In this figure, level 1 means that the entire model is reflective, while in level 6 the model is completely diffuse. Although the overall performance of the hybrid algorithm is better than the data parallel algorithm, both react similarly to the amount of reflectivity in the model. This indicates that reflection has an adverse effect on the performance of the data parallel part of the hybrid algorithm, while the demand driven section is, as expected, invariant under different amounts of reflectivity. Hence, the hybrid scheduling algorithm performs best if specular surfaces are not used abundantly.

Because the host has a memory bottleneck, models larger than approximately 10000 objects can not yet be rendered. For this reason tests with different model sizes have only limited value and consequently these were not performed.

6.5 Comparison between scheduling techniques

In section 6.3 it is shown that the data parallel network is not scalable and performs rather poorly. Although data parallel rendering is part of the hybrid scheduling algorithm, the latter scales far better than the former. This can be accounted for by the load balancing capabilities of the demand driven tasks, as is clearly demonstrated in figure 6.6. In this figure the processors that have only a few data parallel tasks, receive

Table 6.8: Effect of reflectivity on performance (balls model, 16 processors)

	Hybrid		Data parallel	
Level	T	E	T	E
1	41.82	41%	102.60	29%
2	28.13	57%	67.87	28%
3	16.57	61%	67.30	29%
4	13.39	64%	55.06	32%
5	12.84	68%	50.90	33%
6	11.34	62%	39.30	36%

a large number of demand driven rays and vice versa. Hence a better load distribution than in data parallel rendering.

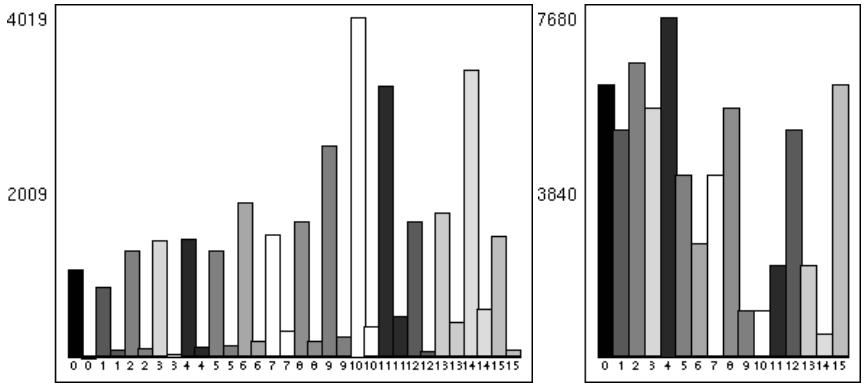


Figure 6.6: Demand driven rays complement the workload of the data parallel cluster (balls model; 16 processors). The left bars in the left figure represent the data parallel rays per processor. The right bars in the left figure represent the fraction of data parallel rays that was transferred to another processor. The right figure depicts the demand driven rays per processor.

If the load is well distributed and there is no excessive communication overhead, the question rises why hybrid scheduling does not yield ideal speed ups for a large number of processors. This question is answered in figure 6.7, where the efficiency over time is given for the balls model with 16 x 16 pyramids and 16 processors. It is clear that the first half of the computation is very efficient, after which there is a sudden drop in efficiency. At this point the demand driven tasks are all finished and the data parallel network computes the remaining secondary rays. The load balancing capabilities are exhausted by then. There is a possibility to hand out demand driven tasks more sparingly, but then the performance in the early stages of the computation will be lower. A more realistic solution would be to have more work executed as demand driven tasks and less work in the data parallel cluster.

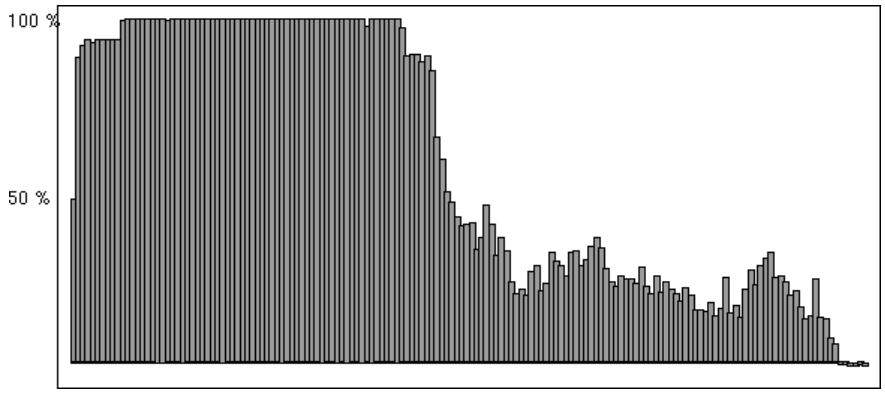


Figure 6.7: Efficiency over time (balls model; 16 processors). Each bar represents a 0.1s time interval. Preprocessing is not included in this graph.

For reasons mentioned above, the current hybrid algorithm still shows a rather disappointing speed-up graph (figure 6.8 left), although it is a whole lot better than the speed up graph for the data parallel algorithm (figure 6.8 right).

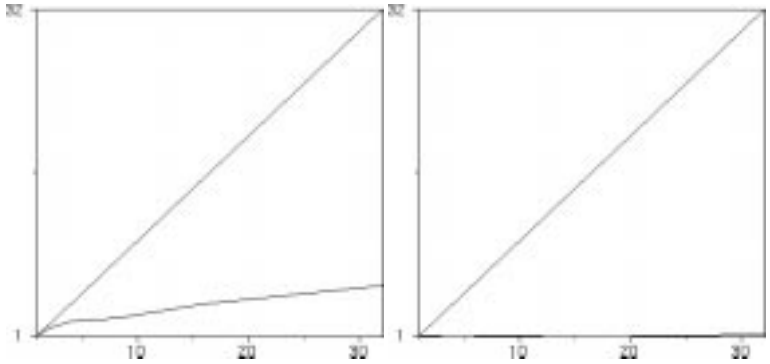


Figure 6.8: Speed up graphs for the hybrid algorithm (left) and data parallel only (right). The balls model was used to obtain these figures.

6.6 Large models

Images have been generated for the two large models described in section 6.1, which resulted in the timing figures given in table 6.9. These figures do not include preprocessing and loading of the data in the distributed memories and caches. Again, the poor results are mainly due to the drop in performance in the second half of the computations, as is explained in section 6.5. This severely reduces scalability.

The one-processor runs did not complete because the models were simply to large

Table 6.9: Timings in seconds for balls5 (1 light source) and conference room (8 resp. 30 light sources). o.m = out of memory. Optimal timings for (sequential) standard Rayshade: 34.5 s, 100.0 s. and 264.2 s. respectively on an SGI Onyx.

	Balls 5		Conference room 8		Conference room 30	
Procs	Data parallel	Hybrid	Data parallel	Hybrid	Data parallel	Hybrid
1	o.m	o.m	o.m	o.m	o.m	o.m
2	o.m	o.m	226.7	368.9	675.0	1266.0
4	o.m	o.m	137.0	167.7	430.7	562.0
8	o.m	38.6	91.5	81.2	262.8	237.0
16	42.9	26.4	88.7	66.3	272.2	236.1
24	47.1	22.8	82.4	49.8	274.3	242.5
32	51.3	26.8	89.3	62.9	193.0	173.5

to fit into memory. For the balls model, the two, four and eight processor runs did not complete, because PVM buffers exceeded the memory available.

Table 6.9 shows that adding light sources to the conference room model, has a drastic effect on the overall performance. While a smaller number of light sources favours the hybrid scheduling approach, the many light sources case shows only a minor performance difference between the two scheduling techniques. This result can be explained by the fact that light sources are traced locally in the data parallel cluster. If many intersections for primary rays are found in the same voxel, the associated process will trace the corresponding light rays. Adding light sources therefore has the effect of increasing the amount of data parallel work, while the demand driven component remains constant. The performance of hybrid scheduling and data parallel scheduling will thus converge.

7 Conclusions

Regarding the assumptions on data parallel rendering, an important conclusion of this work would be that most of them proved unfounded. The load per process is not constant over time and the workload does not follow the distribution of objects over processes. This makes load balancing in a hybrid scheduling algorithm more difficult, as an a priori estimate of which processes will receive what load becomes almost impossible. This explains why the static load balancing criteria that were thought to be useful, did not work. Dynamic load balancing is in such cases a much better idea and the improved performance more than offsets the extra communication overhead associated with dynamic load balancing.

The other assumption on data parallel rendering was right: the load is highly unbalanced. For this reason alone, data parallel rendering in itself is not scalable if measured according to Amdahl's law. This problem may be alleviated somewhat by dynamic redistribution of data [5] or by static load balancing according to a low resolution rendering pass [1]. Our approach involves adding a demand driven component to balance the load, which works fairly well given the abysmal results of data parallel rendering alone.

For large models with few light sources, the demand driven component seems to balance the load, although improvements are still required. Large numbers of light sources tend to reduce the effect of demand driven scheduling, so that hybrid scheduling and data parallel scheduling perform equally poor for such models. Demand driven scheduling for light rays may strongly improve the performance of hybrid scheduling, as tracing light rays constitutes the bulk of the work that needs to be performed.

Currently, light rays are traced locally by the process that spawned them. There is no communication involved, which is a vast improvement for both the data parallel and the hybrid algorithm. In very large environments with a large number of light sources, caching all objects may not be possible. In that case only the objects closest to the light sources should be cached with each processor. In some cases this may re-introduce ray communication for shadow rays, although high cache hit rates are to be expected due to the way the cache is filled.

The load balancing properties of hybrid scheduling are clearly demonstrated. As long as there are demand driven tasks available, each processor can be kept busy, yielding for that part of the computation very high efficiencies. However, half way the computation there are no demand driven tasks left, leaving the data parallel cluster with unacceptable load imbalances. This is the major cause for the efficiency loss that the hybrid algorithm exhibits. Simple parameter tuning is not enough to resolve this problem, but more work should be performed demand driven and less in the data parallel

cluster.

One way to create more demand driven tasks is to perform the tracing of shadow rays demand driven instead of local with the process that generated the shadow rays.

Finally, it should not be forgotten that as far as efficiency is concerned, these results are obtained *despite* PVM, which was designed to be portable but not necessarily efficient. A major gain in efficiency may be obtained if the implementation were rewritten in terms of the native communication primitives of the Parsytec PowerXplorer. However, this would sacrifice much of the portability and thus this implementation would be confined to run on machines with the PARIX operating system, which are not nearly as widespread as simple PVM networks.

8 Future work

If research into parallel rendering were to be continued, a number of different directions may be taken. This chapter discusses some of them, beginning with simple optimisations to the programme which should bring functionality closer to Rayshade as it originally was. Also, hints are given on possible extensions to improve parallel performance. After that, this chapter is concluded with speculation on future hardware developments and its implications to parallel rendering in general.

Although the current implementation works reasonably well for selected models, the functionality offered is a subset of standard Rayshade [13]. One of the most important of omissions is texture mapping. Currently, texture mapping is not supported at all, as there are no functions yet to communicate textures between processes. Because Rayshade allows a large number of arbitrary transformations on texture maps, implementing such communication primitives is not at all trivial. If texture mapping were to be incorporated, the textures are best stored in the data parallel cluster, as this saves memory and the shading, which requires the texture maps, is performed by data parallel processes anyway.

A second optimisation may include biased scheduling. Currently, all demand driven tasks are scheduled with processes that are thought to be out of work. However, by scheduling tasks with processes that already have some of the data needed, some savings on object communication may be acquired. If a low number of processors is available, this optimisation may pay off, but with increasing numbers of processors, the revenue of biased scheduling will decrease, because each process has a smaller portion of the data.

In order to achieve a better load balance, each trace process may govern a number of voxels, instead of just one. In this case, communication between data parallel processes will become more global. A compromise may be reached by subdividing each voxel into a number of smaller voxels and distributing these voxels over a few processes, instead of over all the processes. In that case, there are clusters of processes with local communication between the clusters, while communication between the processes in a single cluster will have a global nature.

Alternatively, the data parallel component may be improved by allowing voxels to overlap. If each voxel extends into the domain of its neighbours by half a voxel's size, tracing a ray may be performed in a more flexible way. Each voxel will have an inner boundary (conforms to the current non-overlapping voxel boundaries) and an outer boundary. If a process is has a large input buffer, new incoming rays may be redirected to a neighbouring data parallel voxel, so that the data parallel workload is spread out over more processes. This approach is expected to work best for the transition from

demand driven primary rays to data parallel secondary rays, as each intersection point is now stored with multiple processes. Overlapping voxels may be employed to spread out uneven load, memory permitting.

A more general and more involved optimisation may be the switch from a regular grid of voxels to a data structure which is more adaptable to the model. This would imply voxels of varying size and also their position would not be fixed in a grid. Voxels would then be generated according to the clustering of objects in the scene, so that they may overlap. This would sacrifice the local communication property of a data parallel network, but may well improve the load balance. Clusters of objects that will attract a large amount of work may then be replicated with more than one process, while objects that will be less referenced occur only once in the system. In addition to the improved load balancing properties, this may also reduce communication in a very natural way.

In order to assign voxels to clusters of objects, techniques should be available to identify clusters. Although this is a little off topic - initially the user may indicate which objects belong to which clusters during modelling - some automatic criterion to find clusters may be helpful when rendering scenes generated with other tools.

Incorporating environment maps may be a further improvement on the overlapping voxels idea [18]. This truncates (secondary) rays that would otherwise travel a long distance through the scene, thereby further reducing the amount of communication at the cost of a small error in the resulting image.

Finally, adding importance driven techniques may be beneficial to the efficiency of many rendering algorithms. By giving each object some importance, indicating whether its contribution to the image is small or large, a better data distribution may be achieved. Important objects for example, may be replicated a number of times, whereas objects with low importance may occur only once. Also, there may be opportunities to distribute work according to importance. Importance driven techniques may be thought of as exploiting object coherence. This line of research therefore shifts from ray coherence to object coherence.

Bibliography

[1] K Bouatouch and T Priol. Parallel space tracing: An experience on an iPSC hypercube. In N Magnenat-Thalmann and D Thalmann, editors, *New Trends in Computer Graphics (Proceedings of CG International '88)*, pages 170–187, New York, 1988. Springer-Verlag.

[2] J G Cleary, B M Wyvill, G M Birtwistle, and R Vatti. Multiprocessor ray tracing. *Computer Graphics Forum*, 5(1):3–12, mar 1986.

[3] F C Crow, G Demos, J Hardy, J McLaugglin, and K Sims. 3d image synthesis on the connection machine. In *Proceedings Parallel Processing for Computer Vision and Display*, Leeds, 1988.

[4] Maurice van der Zwaan, Erik Reinhard, and Frederik W Jansen. Pyramid clipping for efficient ray traversal. In Pat Hanrahan and Werner Purgathofer, editors, *Rendering Techniques '95*, pages 1–10. Trinity College, Dublin, Springer - Vienna, June 1995. proceedings of the 6th Eurographics Workshop on Rendering.

[5] Mark A. Z. Dippé and John Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 149–158, jul 1984.

[6] A Geist, A Beguelin, J Dongarra, W Jiang, R Manchek, and V Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee, may 1993. Included with the PVM 3 distribution.

[7] S A Green and D J Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, 9(6):12–26, nov 1989.

[8] E A Haines. Standard procedural database, v3.1. 3D/Eye, feb 1992.

[9] P Hanrahan. Using caching and breadth first search to speed up ray tracing. In *Proceedings Graphics Interface '86*, pages 55–61, Toronto, 1986. Canadian Information Processing Society.

[10] A G Hoekstra, F van der Linden, P M A Sloot, and L O Hertzberger. Comparing the parix and PVM parallel programming environments. In J van Katwijk, J J Gerbrands, M R van der Steen, and J F M Tonino, editors, *Proceedings of the first*

annual conference of the Advanced School for Computing and Imaging, pages 288–292, Heijen, May 1995. ASCI.

[11] F W Jansen and A Chalmers. Realism in real time? In Michael F. Cohen, Claude Puech, and Francois Sillion, editors, *4th EG Workshop on Rendering*, pages 27–46. Eurographics, jun 1993. held in Paris, France, 14–16 June 1993.

[12] H Kobayashi, S Nishimura, H Kubota, T Nakamura, and Y Shigei. Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The Visual Computer*, 4(4):197–209, 1988.

[13] C E Kolb. *Rayshade User's Guide and Reference Manual*, jan 1992. Included in Rayshade distribution, which is available by ftp from princeton.edu:pub/Graphics/rayshade.4.0.

[14] Tony T Y Lin and Mel Slater. Stochastic ray tracing using SIMD processor arrays. *The Visual Computer*, 7(4):187–199, 1991.

[15] D J Plunkett and M J Bailey. The vectorization of a ray-tracing algorithm for improved execution speed. *IEEE Computer Graphics and Applications*, 5(8):52–60, aug 1985.

[16] Erik Reinhard and Frederik W Jansen. Hybrid scheduling for efficient ray tracing of complex images. In M Chen, P Townsend, and J A Vince, editors, *High Performance Computing for Computer Graphics and Visualisation*, pages 78–87, London, july 1995. University of Swansea, Springer.

[17] Erik Reinhard and Frederik W Jansen. Scheduling issues in parallel rendering. In J van Katwijk, J J Gerbrands, and J F M Tonino, editors, *Proceedings of the First Annual Conference of the Advanced School for Computing and Imaging*, pages 268–277, Heijen, May 1995. ASCI.

[18] Erik Reinhard, Lucas U Tijssen, and Frederik W Jansen. Environment mapping for efficient sampling of the diffuse interreflection. In G Sakas, P Shirley, and S Müller, editors, *Photorealistic Rendering Techniques*, pages 410–422, Darmstadt, jun 1994. Eurographics, Springer Verlag. proceedings of the 5th Eurographics Workshop on Rendering.

[19] I D Scherson and C Caspary. A self-balanced parallel ray-tracing algorithm. In P M Dew, R A Earnshaw, and T R Heywood, editors, *Parallel Processing for Computer Vision and Display*, volume 4, pages 188–196, Wokingham, 1988. Addison-Wesley Publishing Company.

[20] L S Shen. *A Parallel Image Rendering Algorithm and Architecture Based on Ray Tracing and Radiosity Shading*. PhD thesis, Delft University of Technology, Delft, sep 1993.

[21] L R Speer, T D DeRose, and B A Barsky. A theoretical and empirical analysis of coherent ray-tracing. In *Computer-Generated Images*, pages 11–25, Tokyo, 1985. Springer-Verlag. Proceedings Graphics Interface '85.

Appendices

The following four appendices contain papers that were written in the course of the project on hybrid scheduling. References for these papers are:

Appendix A Maurice van der Zwaan, Erik Reinhard and Frederik W. Jansen: 'Pyramid Clipping for Efficient Ray Traversal', in Hanrahan, P. and Purgathofer, W. eds. 'Rendering Techniques '95', proceedings of the 6th Eurographics Workshop on Rendering, Trinity College Dublin, June 1995, Springer Verlag - Wien, pp. 1-10.

Appendix B Erik Reinhard and Frederik W. Jansen: 'Hybrid Scheduling for Efficient Ray Tracing of Complex Images', in Chen, M. ed. 'proceedings of the Workshop on HPC for Computer Graphics and Visualisation', University of Swansea, July 1995, to be published by Springer Verlag - London.

Appendix C Erik Reinhard, Lucas U. Tijssen and Frederik W. Jansen: 'Environment Mapping for Efficient Sampling of the Diffuse Interreflection', in Sakas, G. Shirley, P. and Müller, S. eds. 'Photorealistic Rendering Techniques', proceedings of the Fifth Eurographics Workshop on Rendering, Darmstadt, June 1994, Springer Verlag - Heidelberg, pp 410-422

Appendix D Erik Reinhard and Frederik W. Jansen: 'Scheduling Issues in Parallel Rendering', in Katwijk, J. van, Gerbrands, J. J. and Tonino, J. F. M. eds. 'Proceedings of the First Annual Conference of the Advanced School for Computing and Imaging', Heijen, May 1995, pp. 268-277

A Pyramid Clipping for Efficient Ray Traversal

Rays having the same origin and similar directions frequently appear in the form of viewing rays and shadow rays to area light sources in ray tracing, and in hemisphere shooting or gathering in radiosity algorithms. The coherence between these rays can be exploited by enclosing a bundle of these rays with a pyramid and by classifying objects with respect to this pyramid prior to tracing the rays. We present an implementation of this algorithm for a bintree spatial subdivision structure and compare the performance with a recursive bintree traversal and the standard grid traversal algorithm. In parallel implementations the technique can be used to create coherent intersection tasks, allowing demand-driven scheduling with low communication overheads.

A.1 Introduction

With high quality rendering becoming more and more widespread, the demand for shorter rendering times increases. Both ray tracing and ray tracing-based radiosity algorithms may satisfy the needs of users in terms of quality, but still lack behind when it comes to speed of execution. Spatial subdivision techniques have greatly improved the efficiency of ray tracing but further optimizations in that direction will be difficult to achieve.

Parallel processing offers an interesting alternative to further speed up the ray tracing process. A problem, however, occurs with object models that are too large to be replicated at each processor memory. Then object data will have to be communicated to the intersection tasks or the tasks will have to be brought to the data. In both cases severe communication overheads and/or load unbalances may be introduced. Efficient parallel rendering can only be achieved when enough data coherence is present in the handling of subsequent ray intersection tasks [1].

A way to create coherent ray intersection tasks is to bundle neighboring rays and to pre-select those objects that are likely to be intersected by this bundle of rays (see figure A.1). In Shen et al. [2] such a technique is used to schedule ray tasks on a number of demand-driven intersection processors. Shooting a coherent bundle of rays is a powerful computing primitive that can be applied to tracing primary rays, tracing shadow rays for area light sources and in ray tracing based radiosity algorithms for shooting or gathering energy from one patch to another. Also with brdf-reflection models, incoming rays will spawn a large number of coherent reflection rays.

Earlier ray coherence methods exploiting the notion that rays with similar directions and origins, are likely to intersect the same objects, can be found in [3, 4, 5]. Also shaft culling [6] is a method that classifies objects as in or outside of a shaft con-

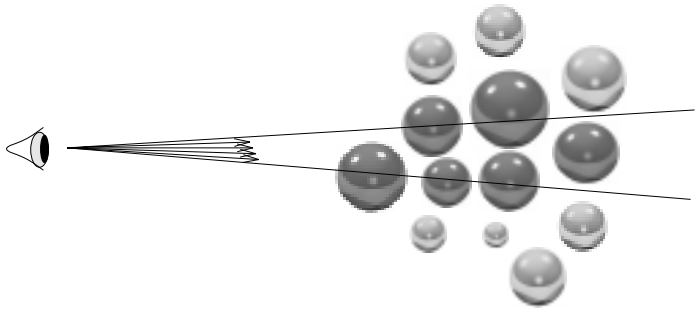


Figure A.1: Coherence between rays.

structed, for instance, between a surface and a light source, or between a shooting patch and a receiving patch.

Recently, Greene published an algorithm for intersecting arbitrary convex polyhedrons with rectangular solids that can be used to efficiently cull polygons that are inside a viewing pyramid from an octree spatial subdivision structure [7]. The viewing pyramid is intersected with the octree and each (axis-aligned) cell of the octree is recursively tested for overlap with the pyramid. A maximum of three tests may be needed to conclude if the pyramid intersects with a cell. In the first test, the bounding box of the pyramid is tested against the cell. If some overlap is found, the second test is invoked, which tests on which side of each of the planes making up the pyramid the cell lies. If the cell is inside any of these planes, the third test is necessary. In this test, the cell and the pyramid are projected onto the three orthographic planes. If in all three views the projection of the cell is outside the projection of the pyramid, then the two do not intersect.

We have been experimenting with a similar algorithm that we originally gave the name of cone clipping [8] and recently renamed in pyra-clip, an abbreviation of pyramid clipping. Our method only differs from Greene's method in that a bintree is used instead of an octree, and for the second and third test a Cohen-Sutherland clipping test is used to determine whether the cell intersects the pyramid. Our version is explained in more detail in sections A.2 and A.3. In section A.4 we compare the method with two regular single-ray traversal methods, one for a bintree and one for a grid. The results are discussed in the final section.

A.2 Pyramid - bintree intersection

The pyra-clip algorithm assumes the presence of a bintree structure. This structure is created in a preprocessing step. The pyra-clip algorithm itself consists of two parts. In the first part, a pyramid is created (this can, for instance, be the viewing pyramid or a subpart of it) that encloses a number of rays having the same origin and similar directions. This pyramid is intersected with the bintree structure. This results in an ordered list of bintree cells that are (partly) inside the pyramid. Then, in the second step, the individual rays that formed the pyramid are traced through the cells. The

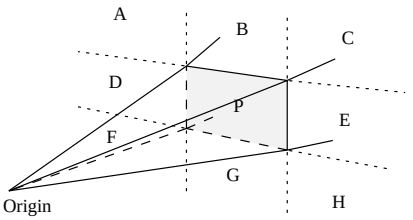


Figure A.2: The planes defining the pyramid generate nine subspaces.

algorithm is done when for each ray an intersection is found. The first step is discussed in this section, while the ray traversal is discussed in section A.3.

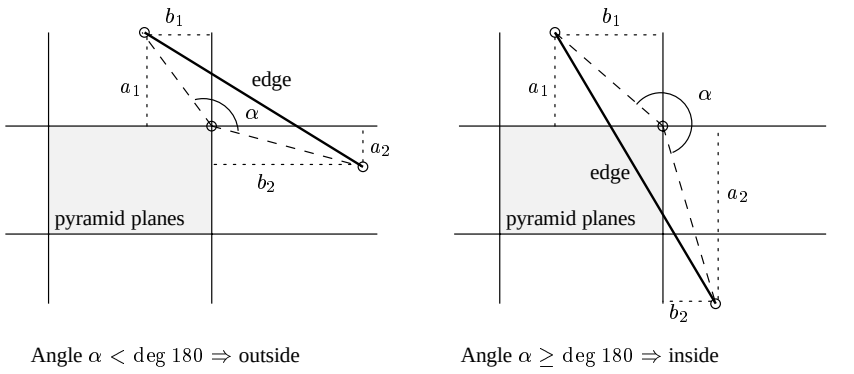
Central to the first step is the classification of each bintree cell to the pyramid. This classification uses a variant of the Cohen-Sutherland line clip algorithm. To this extent, the bintree is recursively tested for intersection with the pyramid. Each level requires at most two tests to determine whether an intersection occurs between a cell and the pyramid.

The first test examines the position of the vertices of a cell with respect to the planes of the pyramid. The four planes of the pyramid define nine subspaces which may contain one or more of the vertices of the cell, see figure A.2. The position of the vertices is derived from the distances of the vertices to the planes of the pyramid. The following cases now may occur:

- All distances are positive indicating that the vertices are inside sector P (see figure A.2). This means that the cell is completely contained within the pyramid.
- Not all vertices are inside sector P. The cell will be partially inside the pyramid.
- The vertices are in three consecutive subspaces, excluding P (for example A-B-C or C-E-H). Now the cell will be completely outside the pyramid.
- The vertices are in both subspace B and G or in both D and E. Because of the convexity of the pyramid, the cell is partially inside the pyramid.
- The vertices are located in the subspace A, C, F and H. This means that the pyramid is contained within the cell.

These tests can be efficiently implemented with bitwise operations. For each vertex of the cell, a bitmask is set, indicating which of the subspaces it is in. The bitmasks for each vertex are OR-ed together, resulting in a bitmask indicating the position of the entire cell with respect to the pyramid.

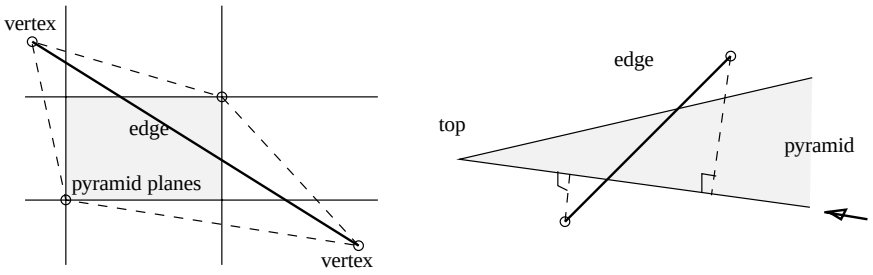
All these tests are relatively simple and therefore fast. For bintree cells that are completely inside or completely outside, no further testing is needed. The cells that are partially inside require their descendants to be recursively tested. This test is expected to be conclusive for most situations, but there are exceptions. An example is if vertices are both in partly opposite subspaces (e.g. D and C). For these situations, a more conclusive second edge test is needed.



Angle $\alpha < \deg 180 \Rightarrow$ outside

Angle $\alpha \geq \deg 180 \Rightarrow$ inside

Figure A.3: Simplified clipping test.



Edge vertices in opposite sectors require at most two tests

Side view of clipping test

Figure A.4: Left: opposite sectors. Right: side view.

The second test explicitly classifies the edges of the bintree cell with respect to the pyramid. Because it is not necessary to know what exactly the shape of the intersecting volume is, no clipping in the true sense of the word is performed. The following test is performed for each pair of vertices defining an edge. For each vertex of a bintree cell, the distance to each of the pyramid planes is known. Using these distances, we can easily compute whether an edge intersects the pyramid or not, see figure A.3. In this example, the distances between the vertices and the pyramid planes are given by a_1, b_1 and a_2 and b_2 respectively. An edge intersects the plane if and only if

$$\frac{a_1}{b_1} \leq \frac{a_2}{b_2} \tag{A.1}$$

A special situation occurs if the vertices of an edge are in diagonally opposite sectors, as is depicted in figure A.4 (left). In this case, two evaluations of equation A.1 are necessary. If the first of these tests indicates that the edge does not intersect the pyramid, the second test can be omitted, otherwise the second one is conclusive.

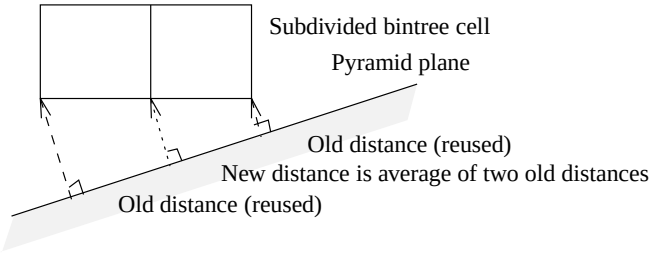


Figure A.5: Distances can be computed incrementally.

All edges are tested this way until an edge is found to intersect with the pyramid or until all twelve edges are tested. Then we can still have the case that no edge intersects the pyramid, but the cell completely surrounds the pyramid. In order to handle this case, for each edge a mask similar to the one in the first test, is updated. After each edge is tested, the orientation of the pyramid with respect to the cell is known and the intersection routine returns with the result.

In 3D, the distances that are used to classify an edge as inside or outside, are all with respect to the planes of the pyramid. Because for both vertices defining an edge, the distances to the same planes are used, the method is valid in 3D. Looking along the line indicated by the arrow in figure A.4 (right), the image of figure A.4 (left) would be seen.

The distances are not computed for each vertex of each cell anew. Instead the regular structure of the bintree is used to efficiently calculate at each level of recursion the distances of the children nodes out of the distances of their parent node, see figure A.5. Per subdivision only four new distances have to be computed, each by averaging two old distances. Eight old distances can be retained. Then, the pyramid - cell intersection routine can be called for the two new bintree cells.

The cell farthest from the view point is put on a stack and the cell nearest the view point is processed first. In this way, the first leaf cell reached is the one closed to the view point. All following leaf cells are automatically derived in the correct order.

The ordered cells are stored in a list, which is called the cliplist. Each pyramid thus generates its own cliplist during the traversal of the bintree that effectively contains all the cells that are traversed by at least one ray of the bundle and most likely by a majority of these rays (see figure A.6). The six planes of each cell (x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}) are also copied to the cliplist. These are needed for the final ray traversal.

In addition, the objects in the cells may also be tested with respect to the pyramid planes and marked as in or out (excluding objects that will never be intersected by any of the rays).

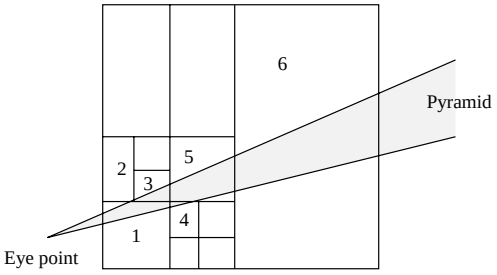


Figure A.6: Ray traversal generates cliplist (cells 1-6).

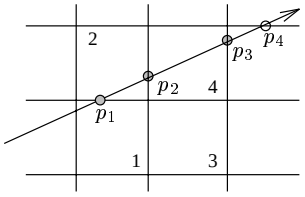


Figure A.7: Ray traversal.

A.3 Ray traversal

After the cliplist has been generated, the first step of the pyraclip algorithm is finished. The next step is to determine which objects intersect with the rays within the pyramid. The ray traversal traverses the cells in list order until an intersection is found. While examining a new cell from the cliplist, the ray parameter value of the exit point for that cell is calculated. The ray exit point is the closest intersection with one of the (three) backfacing cell planes. Which of the six planes are the backfacing planes can be determined from the ray direction. The ray parameter of the exit point is then calculated with three multiplications, three adds and two compare. With an additional compare, it is determined whether the ray actually intersects the cell. If the new ray parameter is smaller than the parameter of the previous exit point, then the cell is not intersected by the ray and can be skipped. For instance in figure A.7, the exit point for cell 1 is p_1 (minimum of p_1 and p_2). The exit point for cell 2 is p_2 (minimum of p_2 and p_4). For cell 3 exit point p_1 would be selected again, which is closer to the origin of the ray than the exit point of the previous cell. Therefore, it is concluded that cell 3 can be skipped, after which p_3 is found to be the exit point of cell 4.

What we have obtained with this algorithm, is a traversal for an adaptive spatial subdivision but with a ray traversal cost comparable with the standard grid method.

A.4 Implementation and Evaluation

In order to have an idea how efficient the algorithm is, we have implemented it in Rayshade [9], a widely used public domain ray tracer. Earlier tests have shown that the

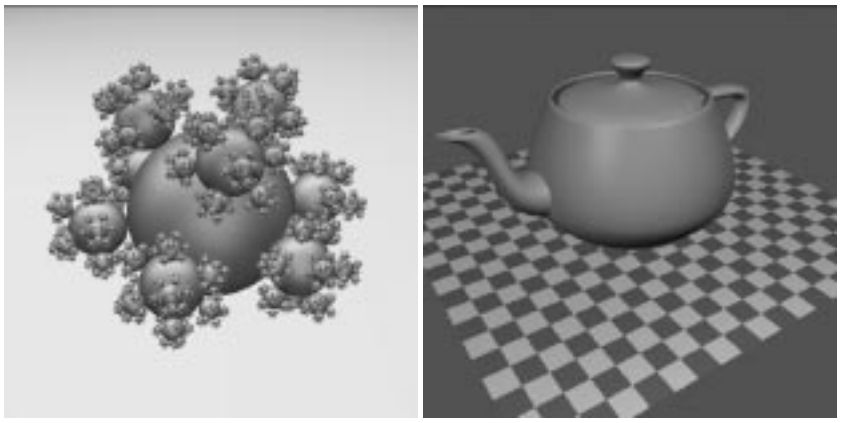


Figure A.8: Standard Procedural Models (balls and teapot)

grid traversal method in Rayshade is one of the fastest ray traversal methods [10, 11]. The grid traverseral in Rayshade is fast, because the standard DDA algorithm is enhanced with raybox testing [12] and ray hit caching. In our tests we have also included a comparison with the recursive bintree traversal algorithm [13, 14], implemented in Rayshade by de Leeuw [15].

For the test we have only traced primary rays (no shadow and reflection rays). The screen is subdivided into a number of non-overlapping rectangular regions and for each region a pyramid is constructed with its top at the view point. The size (width) of the pyramids (number of rays) is an important parameter that will directly affect the performance. If there are too few rays in a pyramid, then the overhead induced by the clipping algorithm will be more than the gain in the reduction of ray object intersections. On the other hand, if the pyramids are too large, the number of cells tested as outside will reduce, resulting in more ray - object intersection tests per ray. Other parameters that will affect performance are the number of objects (polygons/balls), the maximum number of objects allowed in a cell and the maximum depth of the bintree

For the test we have used two models, balls and teapot, from the standard procedural database [16]. The models are depicted in figure A.8. We chose these models because they do not distribute all objects uniformily over object space, which would give an unrealistic test result compared to models normally used in ray tracing and radiosity algorithms. We used the models in different resolutions.

First we verified the optimum setting of the subdivision parameters. For the grid method (uniform spatial subdivision) the rule of thumb given by de Leeuw [15] was confirmed. The number of voxels should equal the number of objects, or stated otherwise the number of voxels in each direction should be approximately equal to $\sqrt[3]{N}$, where N is the number of objects in the model. In fact, we found $2 + \sqrt[3]{N}$ to be the optimal setting for the two models.

The bintree algorithms were less sensitive to different parameter settings. For both model ranges, a maximum of 12 (or 16) objects per cell proved to be optimal. The maximum number of subdivisions varied from 18 for the small models to 24 for the

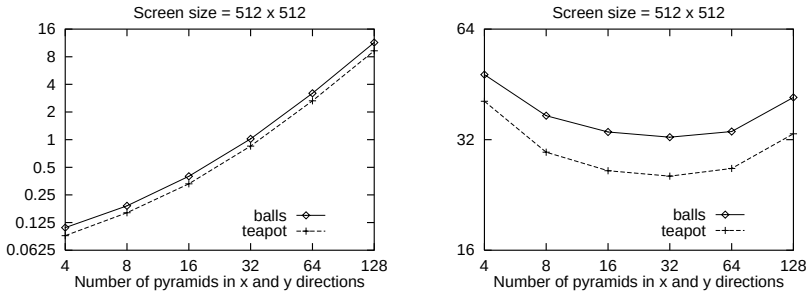


Figure A.9: Preprocessing time (left) and rendering time (right) as function of pyramid size (timing in seconds).

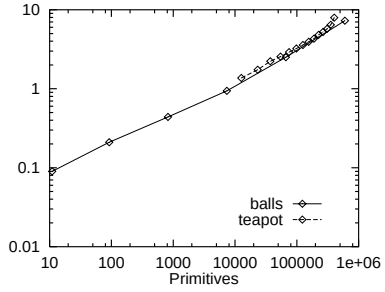


Figure A.10: Preprocessing time (in seconds) as function of the number of primitives.

large models (a maximum subdivision level of 24 corresponds with a maximum resolution of $2^8 = 256$ in both x, y and z directions). For other parameter settings, the times are only a few seconds off.

Then the optimal size for the pyramids was tested. The image size is 512 by 512 pixels. For each of the models, images were computed with $2^2, 4^2, 8^2, \ldots, 128^2$ pyramids per image. The optimum size lies around 32x32 pyramids (16^2 rays per pyramid; see figure A.9 (right)). The pyra-clip time to generate the cell list is about 1 second for 32x32 pyramids and 2 seconds for 64x64 pyramids (see figure A.9 (left)). The preprocessing time is linear in the number of objects, see figure A.10. We choose therefore the 32x32 pyramid subdivision.

Rendering times for the 16^2 rays per pyramid pyraclipping, the bintree recursive and the uniform grid algorithm, are shown in figure A.11. The tests were done on a SGI Power Onyx with 256 Mb internal memory. The memory available prohibited testing larger models. The pyraclip numbers give the pure traversal time, without the overhead of the clipping and without tracing secondary rays.

The figures show that the grid performs linear with respect to the number of objects, while the recursive bintree is logarithmic and wins for larger N. A multi grid method would probably have shown a similar performance. In fact this is the first time that we are able to show that the performance of the recursive bintree traversal is $\log N$ for such large models. Unfortunately, the pyraclip traversal adheres more to the grid

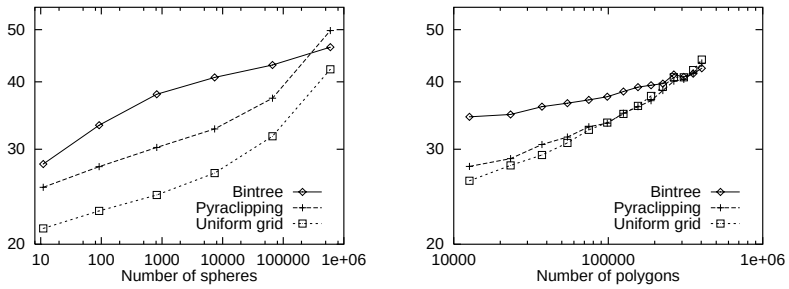


Figure A.11: Pyraclipping compared with uniform grid and bintree methods. Models used: balls (left) and teapot (right); timing in seconds.

method than to the recursive bintree traversal and thus looses in the end compared to the recursive ray traversal. The figures show that there is no use for the pyraclip method for efficiency reasons only. All methods perform equally well (within the tested range). We found this also to be true for the other SPD models.

A.5 Conclusions and future work

The pyra-clip algorithm intersects a bundle of rays with a hierarchical spatial subdivision structure. First an ordered list of bintree cells is generated using an efficient pyramid - cell intersection method. Then the separate rays are traced through the list of cells.

The tests showed that the pyraclip method is not faster than the standard single ray traversal methods and thus for speed purposes only, the method as such does not give any additional benefit. However, the method is ideal in that it can generate with minor overhead coherent ray intersection tasks (a bundle of rays and matching objects) that can be executed as fast as the fastest ray traversal methods. We will therefore use this method to generate demand-driven tasks in a parallel ray tracing and radiosity algorithm [17].

Each processor in such a parallel ray tracer would be capable of both performing data parallel tasks and demand driven tasks. The data parallel tasks would provide a basic, though uneven load by tracing non-coherent secondary rays, such as reflection and refraction rays. Primary rays are then handled in demand driven manner and scheduled to processors with a low basic load. A processor receiving a bundle of rays and a cliplist will do the intersection calculations using this data. Shadow rays that sample area light sources are handled in a similar way. Such hybrid scheduling should both minimise data communication and balance the workload, yielding an efficient and scalable algorithm [18].

References

[1] Green, S. A., Paddon, D. J.: 'Exploiting coherence for multiprocessor ray tracing', *IEEE Computer Graphics and Applications* pp. 12–27. (1989)

[2] Shen, L. S., Deprettere, E., Dewilde, P.: A new space partition technique to support a highly pipelined parallel architecture for the radiosity method, *in* Advances in Graphics Hardware V, proceedings Fifth Eurographics Workshop on Hardware, Springer-Verlag. (1990)

[3] Speer, L. R., DeRose, T. D., Barsky, B. A.: A theoretical and empirical analysis of coherent ray-tracing, *in* Computer-Generated Images, Springer-Verlag, Tokyo, pp. 11–25. Proceedings Graphics Interface '85. (1985)

[4] Hanrahan, P.: Using caching and breadth first search to speed up ray tracing, *in* Proceedings Graphics Interface '86, Canadian Information Processing Society, Toronto, pp. 55–61. (1986)

[5] Glassner, A. S., ed.: *An Introduction to Ray Tracing*, Academic Press, San Diego. (1989)

[6] Haines, E. A., Wallace, J. R.: Shaft culling for efficient ray-traced radiosity, *in* Photorealistic Rendering in Computer Graphics, proceedings Second Eurographics Workshop on Rendering, Springer-Verlag 1994, pp. 122–138. (1991)

[7] Greene, N.: Detecting intersection of a rectangular solid and a convex polyhedron, *in* P. Heckbert, ed., Graphics Gems IV, Academic Press, Boston, pp. 74–82. (1994)

[8] van der Wal, P. O.: De coneclip versnellingsmethode voor raytracing, Master's thesis, Delft University of Technology. (1994)

[9] Kolb, C. E.: Rayshade User's Guide and Reference Manual, included in the Rayshade distribution, which is available by ftp from princeton.edu:pub/Graphics/rayshade.4.0. (1992)

[10] Jansen, F. W., de Leeuw, W. C.: Recursive ray traversal. In Ray Tracing News, vol 5, no 1. (1992)

[11] Jansen, F. W.: Comparison of ray traversal methods. In Ray Tracing News, vol 7, no 2. (1994)

[12] Snyder, J., Barr, A.: Ray tracing complex models containing surface tessellations, *Computer Graphics* **21**(4), 119–128. (1987)

[13] Jansen, F. W.: Data structures for ray tracing, *in* L. R. A. Kessener, F. J. Peters, M. L. P. Lierop, eds, Data Structures for Raster Graphics, Springer-Verlag, Berlin, pp. 57–73. (1985)

[14] Sung, K., Shirley, P.: *Ray tracing with the BSP Tree*, Graphics Gems III, Academic Press, Boston, chapter 6, pp. 271–274. (1992)

[15] de Leeuw, W. C.: Recursieve ray traversal, Master's thesis, Delft University of Technology. (1992)

[16] Haines, E. A.: Standard procedural database, v3.1, 3D/Eye. (1992)

[17] Jansen, F. W., Chalmers, A.: Realism in real time?, *in* 4th EG Workshop on Rendering, pp. 1–20. (1993)

[18] Reinhard, E., Jansen, F. W.: Hybrid scheduling for efficient ray tracing of complex images. (1995) International Workshop on High Performance Computing for Computer Graphics and Visualisation, Swansea, United Kingdom, july 3-4, To be published by Springer London, 1995.

B Hybrid Scheduling for Efficient Ray Tracing of Complex Images

Ray tracing is a powerful technique to generate realistic images of 3D scenes. A drawback is its high demand for processing power. Multiprocessing is one way to meet this demand. However, when the models are very large, special attention must be paid to the way the algorithm is parallelised. Combining demand driven and data parallel techniques provides good opportunities to arrive at an efficient scalable algorithm. Which tasks to process demand driven and which data driven, is decided by the data intensity of the task and the amount of data locality (coherence) that will be present in the task. Rays with the same origin and similar directions, such as primary rays and light rays, exhibit much coherence. These rays are therefore traced in demand driven fashion, a bundle at a time. Non-coherent rays are traced data parallel. By combining demand driven and data driven tasks, a good load balance may be achieved, while at the same time spreading the communication evenly across the network. This leads to a scalable and efficient parallel implementation of the ray tracing algorithm.

B.1 Introduction

From many fields in science and industry, there is an increasing demand for realistic rendering. Architects for example need to have a clear idea of how their designs are going to look in reality. In theatres the lighting aspects of the interior are important too, so these should be modelled as accurately as possible. It is evident that making such designs is an iterative and preferably interactive process. Therefore next to realism, short rendering times are called for in these applications. Another characteristic of such applications is that the models to be rendered are typically very large.

It is very difficult to develop algorithms which meet all these demands. Z-buffering algorithms are known for their speed (they are even suitable for animation purposes), but lack sufficient realism for architectural imagery. On the other hand, those algorithms that do offer an acceptable level of realism, such as ray tracing (see figure B.1) and ray tracing based radiosity, are computationally too expensive. Past attempts to overcome these problems generally follow two different approaches. One is to increase the quality of z-buffer algorithms, while retaining their speed. The other method is to speed up radiosity and ray tracing algorithms by using parallel or distributed systems. By having multiple processors in either distributed or shared memory architectures to compute parts of the same problem, considerable speed-ups are to be expected.

The most common way to parallelise ray tracing is the demand driven approach where each processor is assigned a part of the image [1] [2] [3]. Alternatively, coherent

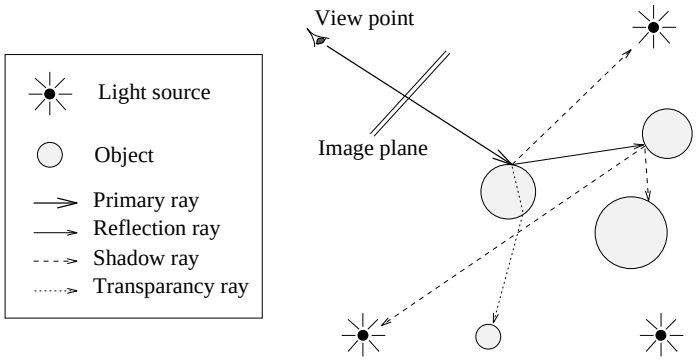


Figure B.1: Ray tracing consists of shooting primary rays from the eye point through the screen and tracing reflected and refracted rays recursively if they intersect with objects. At each intersection rays are also shot towards light sources for shadow testing.

subtasks may be assigned to processors with a low load. This has the advantage of spreading the load evenly over the processors [4] [5], but it requires either the data to be duplicated with each processor, thereby limiting the model size, or an efficient caching mechanism to be implemented. Caching may reduce the amount of communication, but its efficiency is highly dependent on the amount of coherence between subsequent data requests.

Alternatively, scheduling could be performed by distributing the data over the processors according to a spatial subdivision consisting of voxels. Rays are then traced through a voxel and when a ray enters the next voxel, it is transferred as a task to the processor holding that voxel's objects [7] [8] [9]. This is called the data parallel or data driven approach and it is the basis of our parallel implementation. The advantage of such an approach is that there is virtually no restriction on the size of the model to be rendered. However, there are also some rather severe disadvantages, which include a load balancing problem and a large overhead on communication when there is a large number of processors.

The problems with either pure demand driven or data driven implementations may be overcome by combining the two, yielding a hybrid algorithm [10] [6]. Scherson and Caspary [10] propose to take the ray traversal task, i.e. the intersection of rays with the spatial subdivision structure, such as an octree, to be the demand driven component. As an octree does not occupy much space, it may be replicated with each processor. Provided the load on a processor is sufficiently low, a processor can then perform demand driven ray traversal tasks, in addition to ray tracing through its own voxel in a data driven manner. The demand driven task then compensates for the load balancing problem induced by the data parallel component, while at the same time the amount of data communication is kept low.

Computationally intense tasks that require little data are thus preferably handled in a demand driven manner, while data intensive tasks are better suited to the data parallel approach. A problem with the hybrid algorithm by Scherson and Caspary [10] is that the demand driven component and the data driven component are not well matched,

i.e. the ray traversal tasks are computationally not expensive enough to optimally compensate for the load balancing problem of the data parallel part. Therefore, our data parallel algorithm (presented in section B.2 and B.3) will be modified to incorporate two extra demand driven components differently from Scherson and Caspary [10].

A key notion for our implementation is coherence, which means that rays that have the same origin and almost the same direction, are likely to hit the same objects. Both primary rays and shadow rays directed to area light sources exhibit this coherence. To benefit from coherence, primary rays can be traced in bundles, which are called pyramids in this paper. A pyramid of primary rays has the viewpoint as top of the pyramid and its cross section will be square. First the pyramids are intersected with a spatial subdivision structure, which yields a list of cells containing objects that possibly intersect with the pyramids. Then the individual rays making up the pyramid are intersected with the objects in the cells. By localising the data necessary, these tasks can very well be executed in demand driven mode.

Shadow tracing for area light sources provides another opportunity to exploit coherence, as per intersection, a bundle of rays would be sent towards them. These rays originate from the intersection point and all travel towards the area light. The coherence between these rays is considerable, and they can also be traced in a pyramid, much in the same way primary rays are handled. Further, instead of communicating the shadow ray tasks to neighbouring voxels, shadow tracing can also be performed as a demand driven task at the local processor that found the intersection. This may necessitate fetching object data from other processes, so that caching becomes a relevant technique. However, because the same objects may prove to block a light sources for many possible intersections within a voxel, caching is expected to be highly efficient.

Adding demand driven tasks (processing of primary rays) to the basic data driven algorithm, will improve the load balance and increase the scalability of the architecture.

In this paper, first a description of the data parallel component of our parallel implementation is given in section B.2. Next, the demand driven component and the handling of light rays are introduced in section B.3. The complete algorithm is described in terms of processes in these sections. A suitable mapping to a processor topology is described in section B.4. Finally, conclusions are drawn in section B.5.

B.2 Data parallel ray tracing

To derive a data parallel ray tracer, the algorithm itself must be split up into a number of processes and the object data must be distributed over these processes. Both issues are addressed in this section, beginning with the data distribution.

Objects in a model generally exhibit coherence, which means that objects consist of separate connected pieces bounded in space, and distinct objects are disjoint in this space [4]. Other forms of coherence, such as coherence between rays, where rays with similar origins and directions are likely to intersect the same objects, are derived from this definition. To exploit object coherence in a data parallel algorithm, it is best to store objects that are close together, with the same processor. Such a distribution may

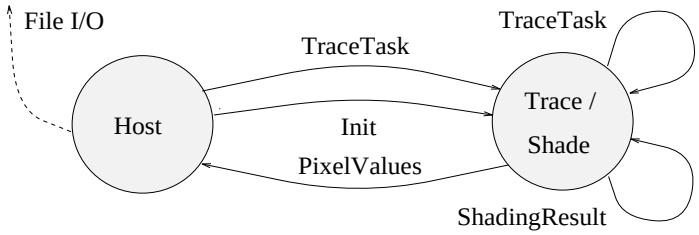


Figure B.2: Host and slave processes.

be achieved by splitting the object space into (equal sized) voxels and assigning each voxel with its objects to a process.

Ray tracing is now performed in a master-slave setup. A host process initialises a number of slave processes, and then determines which slave gets which data. After initialisation, for each primary ray the host determines which voxel it originates in and sends this ray as a task to the associated trace process. When all primary rays are dispatched, the host will wait for incoming pixel values, which it will write to an image file. The host and the trace processes and their incoming and outgoing messages are depicted in figure B.2.

Each tracing process has the objects pertaining to its voxel. This process reads a ray task from its buffer and traces the ray. Two situations may occur. First, the ray may leave the voxel without intersecting any object. If this happens, the ray is transferred to a neighbouring voxel. Else, an intersection with some locally stored object is found and secondary rays are spawned. These are subsequently traced until one of these two criteria is met.

Shading is performed by the process that found the intersection. To that extend it performs some extra book keeping by storing intersection information. Then the secondary rays are spawned. Somewhere in the future colour values for these secondary rays will be returned, which are stored until all results for an intersection are completed. Then shading is performed and a colour value is returned, see figure B.2.

The advantages of this way of parallel ray tracing are that very large models may be rendered as the object database does not need to be duplicated with each process, but can be divided over the distributed memories instead. Ray tasks will have to be transferred to other processes, but as each voxel borders on at most a few other voxels, communication for ray tasks is only local. This means that data parallel algorithms are scalable without too much loss of efficiency.

Disadvantages are that load imbalances may occur. These may be solved by either static load balancing, which most probably yields a suboptimal load balance, or dynamic load balancing, which may induce too much overhead in the form of data communication.

In order to have more control over the average load of each process, and thereby overcome most of the problems associated with data parallel computing, some demand driven components may be added to this algorithm, which is the topic of the following section.

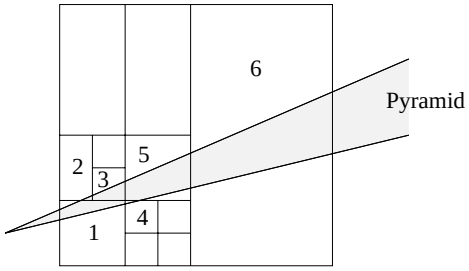


Figure B.3: Pyramid traversal generates clip list (cells 1-6).

B.3 Demand driven pyramid tracing

As tracing rays is by far the most expensive operation in ray tracing, this should be performed as efficiently as possible, i.e. ray coherence should be exploited where present. In ray tracing, different types of rays are generated and each of them has unique properties which call for different handling. We classify rays according to the amount of coherence between them. For example, primary rays all originate from the eye point and travel in similar directions. These rays exhibit much coherence, as they are likely to intersect the same objects. This is also true for shadow rays that are sent towards area light sources.

Restricting ourselves for the moment to primary rays, a common technique to trace a number of rays together, is by replacing them by a generalised ray [12]. Examples are cone tracing [16], beam tracing [17] [18] and pencil tracing [19]). A slightly different method is presented here, consisting of two steps. First, the primary rays are bundled together to form pyramids. These are then intersected with a spatial subdivision. The result of this pyramid traversal is a clip-list, which is an ordered list of cells that intersect (partially) with a pyramid. No object data is necessary to perform this step; only the spatial subdivision structure and the pyramids. Assuming the subdivision structure is a bintree, the process of pyramid traversal is depicted in figure B.3.

The second step consists of tracing the individual rays within each pyramid, using the information obtained from the pyramid traversal. Tracing the rays within a pyramid will therefore require a relatively small number of objects, namely the objects that lie within the cells traversed. For this reason, pyramid tracing may be executed in a demand driven manner, as the data communication involved is restricted.

As the pyramids diverge the further they are away from the origin, coherence decreases. There are two other causes for the decrease in coherence. Some of the rays within a pyramid may intersect with an object, leaving gaps between the primary rays that continue. Also, intersections cause secondary rays that may travel in completely different directions. It may therefore be necessary after a certain distance is travelled, to switch from demand driven execution to data driven execution, as described in the preceding section.

Another opportunity to exploit ray coherence, is provided by shadow rays. Whereas in data parallel tracing, the voxels that contain light sources, may become bottlenecks,

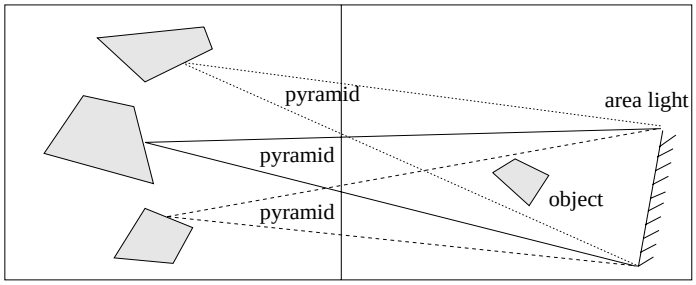


Figure B.4: The object in the right voxel is needed for light pyramid tracing by the process managing the left voxel.

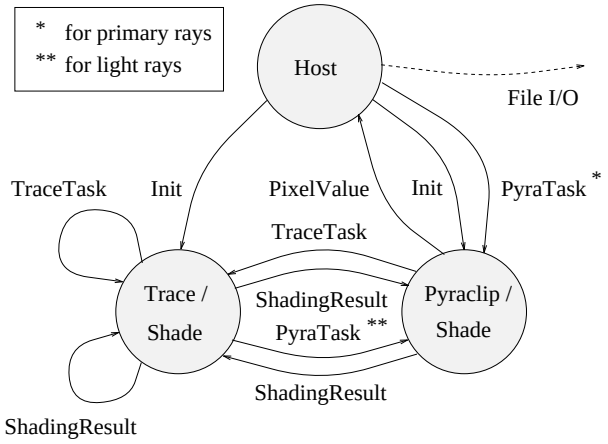


Figure B.5: Host, trace and pyraclip tasks and the communication between them.

a demand driven approach may successfully circumvent this problem. Especially area light sources, which generate a bundle of rays per intersection, are problematic in data parallel tracing. It is therefore advantageous to apply some form of pyramid tracing to these rays as well. To avoid contention at the processes containing light sources, light pyramids may be processed locally by the processes that initiated them [14]. If we choose to trace shadow pyramids with the process that spawned them, it may be necessary to fetch objects from remote processes, as figure B.4 illustrates. In this figure, the object in the right voxel, which is in front of the area light source, is needed by all light pyramids depicted.

Each process that may spawn light pyramids should be equipped with a cache, which reduces data communication. The effectiveness of such a cache is estimated to be high, because many pyramids generated from within a voxel, will intersect with the same objects.

In figure B.5, the resulting setup of host, data parallel trace and demand driven pyraclip tasks is depicted. Pyraclip tasks generated by the host are for primary rays and pyraclip tasks originating from a trace process are for tracing shadow rays.

B.4 Architecture and implementation

Up until this point, the algorithm is described in terms of processes, which is architecture independent. If this algorithm is to be implemented, these processes must be mapped onto a certain hardware architecture.

There is a choice between message passing and shared memory architectures. Message passing architectures are naturally scalable, but communication between processes must be explicitly specified by the application. To reduce the amount of communication, caching may be used. The performance of the cache depends strongly on the amount of data locality. Similar arguments apply for advanced shared memory architectures that also use local caching to reduce the amount of communication with main memory [15]. Therefore, our algorithm is suitable for both types of architecture, as it is designed to minimise the amount of data needed per intersection. Our current system is implemented on a message passing architecture, the mesh connected Parsytec GCel.

The algorithm distinguishes a number of different processes. First of all, there is a host process, which initiates pyramid traversal tasks and receives pixel values. Second, there are pyramid tracing processes which operate in demand driven mode. Ray tracing processes for tracing individual secondary rays form a third category. These processes also perform light pyramid tracing, using a cache for objects. Shading is performed by the process that found the intersection.

One possible mapping of processes onto processors is depicted in figure B.6. There is one host processor and a cluster of slave processors. Each of these slave processors runs data parallel tracing processes. Some of them, typically the ones with a low load, may also run a pyramid tracing process. Pyramid tracing tasks should preferably be scheduled to processors that are located close to the host processor, in order to minimise long distance communication.

In a data parallel tracing cluster, the optimal number of processors will be bounded by the amount of task communication which increases when new processors are added. If this optimal number of processors is lower than the number of processors available for data parallel tracing, it may be possible to add other identical data parallel clusters. In that case the object database is distributed within a cluster, but duplicated over the clusters. Ray tasks may then be executed in any of the available clusters, without ever having to be transferred between clusters. This scheme of cluster replication works in ray tracing by virtue of the static nature of the object database, but not in radiosity and other algorithms, such as Radiance [20], which update the object database in the course of execution.

For the current implementation we have used an adapted version of Rayshade [21], which is a sequential public domain ray tracer. The language used is C, extended with the PVM parallel library [22]. Although using standard libraries may be less efficient than hard coding for a specific architecture, it has great advantages in terms of portability.

Currently, the demand driven tasks consist of primary rays only. Scheduling is performed statically and there are three scheduling criteria. These are processors with a voxel on the model boundary, processors with a low number of objects and processors with a low number of light sources.

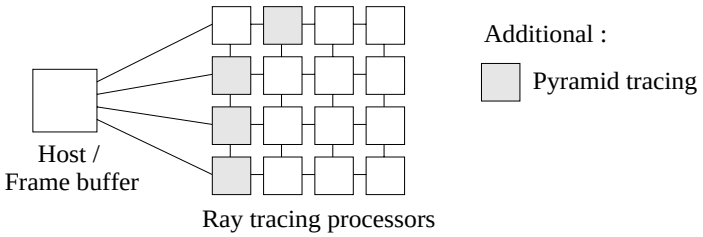


Figure B.6: Mapping of processes onto processors

B.5 Conclusions

The two basic approaches to parallel rendering, demand driven and data parallel, both have their advantages and shortcomings. Data driven scheduling allows the algorithm to be scaled, but almost invariably leads to load imbalances. Demand driven approaches achieve better load balancing, but may suffer from a communication bottleneck due to insufficient cache performance, which makes them less scalable.

In order to arrive at an efficient scalable algorithm for ray tracing, the algorithm is best split into a demand driven and a data parallel component. Rays that show much coherence are then most efficiently traced by a demand driven algorithm which exploits ray coherence as much as possible. The pyramid tracing algorithm described in this paper does precisely that.

For rays that are less coherent, such as reflection rays and transparency rays, the data parallel approach pays off, as fetching object data, which is associated with demand driven solutions, is too expensive for single rays.

As both message passing and shared memory systems rely on caching mechanisms for efficiency, an application running on such machines should optimise data locality. By exploiting ray and object coherence present in ray tracing, data locality can be preserved. This holds for simple ray tracing applications, but also for more involved ray tracing based radiosity and Monte Carlo algorithms, where for each patch or intersection, a hemisphere is sampled.

Features to be implemented next include the demand driven scheduling of light rays and the implementation of dynamic scheduling for pyramid tasks.

References

[1] D. J. Plunkett, M. J. Bailey, The vectorization of a ray-tracing algorithm for improved execution speed, IEEE Computer Graphics and Applications **5**(8), 52–60, (1985).

[2] F. C. Crow, G. Demos, J. Hardy, J. McLaugglin, K. Sims, 3d image synthesis on the connection machine, *in* Proceedings Parallel Processing for Computer Vision and Display, Leeds, (1988).

[3] T. T. Y. Lin, M. Slater, Stochastic ray tracing using SIMD processor arrays, The Visual Computer **7**(4), 187–199, (1991).

[4] S. A. Green, D. J. Paddon, Exploiting coherence for multiprocessor ray tracing, IEEE Computer Graphics and Applications **9**(6), 12–27, (1989).

[5] L. S. Shen, A Parallel Image Rendering Algorithm and Architecture Based on Ray Tracing and Radiosity Shading, PhD thesis, TUDelft, Delft, (1993).

[6] F. W. Jansen, A. Chalmers, Realism in real time?, *in* 4th EG Workshop on Rendering, pp. 1–20, (1993).

[7] M. A. Z. Dippé, J. Swensen, An adaptive subdivision algorithm and parallel architecture for realistic image synthesis, ACM Computer Graphics **18**(3), 149–158, (1984).

[8] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle, R. Vatti, Multiprocessor ray tracing, Computer Graphics Forum, **5**(4), 3–12, (1986).

[9] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, Y. Shigei, Load balancing strategies for a parallel ray-tracing system based on constant subdivision, The Visual Computer **4**(4), 197–209, (1988).

[10] I. D. Scherson, C. Caspary, A self-balanced parallel ray-tracing algorithm, *in* P. M. Dew, R. A. Earnshaw, T. R. Heywood, eds, Parallel Processing for Computer Vision and Display, Vol. **4**, Addison-Wesley Publishing Company, Wokingham, pp. 188–196, (1988).

[11] T. Whitted, An improved illumination model for shaded display, Communications of the ACM **23**(6), 343–349, (1980).

[12] A. S. Glassner, ed., An Introduction to Ray Tracing, Academic Press, San Diego, (1989).

[13] G. J. Ward, F. M. Rubinstein, R. D. Clear, A ray tracing solution for diffuse interreflection, ACM Computer Graphics **22**(4), 85–92, (1994).

[14] T. Priol, K. Bouatouch, Static Load Balancing for a Parallel Ray Tracing on a MIMD Hypercube, The Visual Computer, **5**(12), 109–119, (1989).

[15] J. S. Singh, A. Gupta, M. Levoy, Parallel Visualization Algorithms: Performance and Architectural Implications, IEEE Computer, **27**(7), 45-55, (1994).

[16] J. Amanatides, Ray tracing with cones, ACM Computer Graphics **18**(3), 129–135, (1984).

[17] P. S. Heckbert, P. Hanrahan, Beam tracing polygonal objects, ACM Computer Graphics **18**(3), 119–127, (1984).

[18] N. Greene, Detecting intersection of a rectengualr solid and a convex polyhedron, *in* P. S. Heckbert, ed., Graphics Gems IV, AP Professional, Cambridge, MA, chapter 1.7, pp. 74–82, (1994).

[19] M. Shinya, T. Takahashi, S. Naito, Principles and applications of pencil tracing, ACM Computer Graphics **21**(4), 45-54 (1987).

[20] G. J. Ward, The radiance lighting simulation and rendering system, ACM Computer Graphics pp. 459–472. SIGGRAPH ’94 Proceedings, (1994).

[21] C. E. Kolb, Rayshade User’s Guide and Reference Manual. Included in Rayshade distribution, which is available by ftp from princeton.edu:pub/Graphics/rayshade.4.0, (1992).

[22] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM 3 Users Guide and Reference Manual, Oak Ridge National Laboratory, Oak Ridge, Tennessee. Included with the PVM 3 distribution, (1993).

C Environment Mapping for Efficient Sampling of the Diffuse Interreflection

Environment mapping is a technique to compute specular reflections for a glossy object. Originally proposed as a cheap alternative for ray tracing, the method is well suited to be incorporated in a hybrid rendering algorithm. In this paper environment mapping is introduced to reduce the amount of computations involved in tracing secondary rays. During rendering, instead of tracing the secondary rays all through the scene, values are taken from the maps for the rays that would otherwise hit distant objects. This way the quality of the image is retained while providing a cheap alternative to stochastic brute force sampling methods. An additional advantage is that due to the local representation of the entire 3D scene in a map, parallelising this algorithm should result in a good speed-up and high efficiency.

C.1 Introduction

In the past, a number of approaches to rendering have been proposed. Image quality and speed are the two main goals set in the computer graphics community. If speed is the dominant factor, e.g. in flight simulators and architectural walk-throughs where images must be produced in real-time, z-buffer-based algorithms and hardware can be used. By using a radiosity pre-processing, a more realistic shading can be obtained. Disadvantages are that specular reflection can not be modelled and that the quality of the solution strongly depends upon the resolution of the radiosity mesh.

The other approach to rendering is the use of sampling algorithms, such as ray tracing or ray tracing based radiosity. There are several versions of two-pass algorithms which consist of a radiosity and a ray tracing pass. These hybrid algorithms differ in the amount of sampling done during the rendering pass. The standard two-pass algorithm only samples specular reflection during rendering, while the radiosity values are used for diffuse reflection and direct lighting (Sillion and Puech [14]; Wallace et al. [15]). An extended version also samples direct light during rendering (Shirley [12], [13]; Chen et al. [2]; Kok and Jansen [6]) and finally, diffuse light may be sampled in the rendering stage in addition to specular reflection and direct light (Rushmeier [10]; Chen et al. [2]).

Sampling of indirect light allows the simulation of intricate reflection details. However, in complex environments this method will be very expensive and moreover it may bring about aliasing problems. To make aliasing less visible and to obtain a reasonable estimate of the indirect diffuse reflection, either a huge number of samples has to be taken, or stochastic techniques may be applied. Stochastic sampling effectively turns aliasing into noise, which is less perceptible to the human eye.

However, a large number of samples is still needed. Therefore, using today's hardware, it is not possible to compute an image interactively with these methods. There are two approaches to reduce computation times of high quality rendering algorithms. First the algorithm can be optimised by reducing the number of redundant computations. Examples of such improvements are spatial subdivision techniques (Glassner [3]) and grouping (Rushmeier [11]; Kok [7]). Second, rendering algorithms may provide good opportunities to be efficiently implemented on multicomputers.

An implementation of a hybrid algorithm on a distributed memory MIMD computer will likely present problems when the scenes to be rendered are large. Then the scene database can not be replicated with every processor due to memory restrictions. Some distribution scheme will have to be applied instead. Objects can be assigned to the processors randomly, completely ignoring scene coherence. A better idea is to base the object distribution upon a spatial subdivision structure. Then the objects that are contained within a region of the environment, are assigned to the same processor. A task is executed by the processor that holds the relevant data in its local memory. This data parallel solution may, however, give rise to excessive load imbalances. Another approach is to schedule the computation tasks in a demand driven way, but then processors may need to request data from other processors, resulting in large communication overheads. This is in particular the case for secondary rays where data coherence is low.

In order to solve both sampling and parallelisation problems, we propose to use environment mapping in an adapted form. Where in Blinn and Newell ([1]) environment mapping is presented as a cheap alternative for ray tracing, in our approach, which is similar to (Greene [4]), environment mapping is part of a hybrid rendering algorithm to handle secondary rays efficiently. The sampling related problems are solved by prefiltering the maps. The communication problem is releaved by providing a means to efficiently store information of remote objects in a local environment map. Thus data retrieval becomes a local process, which removes the need to link data management and task scheduling. Therefore, a more flexible parallel implementation may be achieved.

The complete rendering algorithm consists of three separate stages. First, a standard radiosity preprocessing is performed. A coarse mesh is used in this stage, as in subsequent stages only a rough estimate of patch radiances is needed. In the second stage, objects are grouped and for each group an environment map is generated. This is a view independent operation. The map generation may be viewed as a pre-processing of secondary rays. Normally, the origin of secondary rays is determined by the intersection point of a primary ray and a surface. During map building, all rays are traced from the centre point. However, as in the rendering stage secondary rays are generally not traced from the centre point, an error will occur. How to keep this error small is discussed in the following section. In the last stage the scene is rendered using environment maps and radiances.

The map generation and rendering stages are presented in greater detail in the next section. Experiments with an implementation of the environment mapping algorithm are discussed subsequently, while in the last section conclusions are drawn.

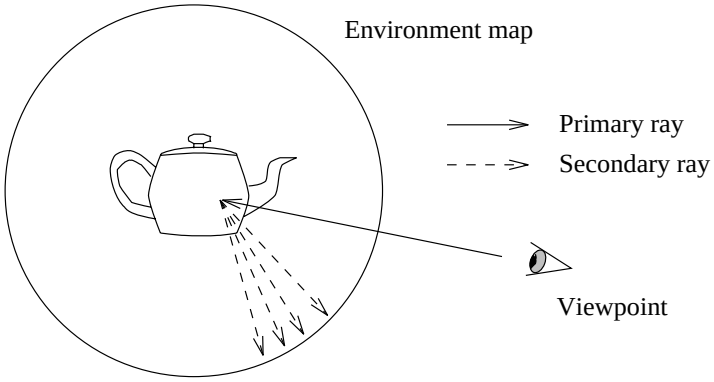


Figure C.1: For every secondary ray, a look-up in the environment map is done, instead of casting secondary rays

C.2 Method

The environment mapping technique was first introduced to computer graphics by Blinn and Newell ([1]). It is a method to enhance an object with reflections without explicitly tracing secondary rays. This is accomplished by projecting the 3D environment onto a 2D environment map that surrounds the glossy object. Instead of intersecting secondary rays with objects, an index in the environment map is computed from the surface normal of the object and the angle of the incoming ray (see Figure C.1). Environment mapping can also be used to reflect digitised photographs in objects.

Following (Miller and Hoffman [9]), in (Greene [4]) environment mapping was augmented with filtering capabilities to support a more general reflection model with both diffuse and specular reflection. The single environment map was replaced by a cube with six maps that were placed around the object. For each of the six maps, a mip-map was generated. They are indexed according to the specularity of the object that the maps surround. Mip-mapping is a pre-filtering method (Williams [16]) that filters colour arrays, such as texture maps or environment maps.

In the environment mapping algorithm as proposed in this paper, the emphasis is on cost reduction of tracing secondary rays, as suggested in (Hall [5]). To achieve this, the scope of these rays is limited to the boundaries of the cube on which the maps are projected. Within these limits, rays are actually traced, as other objects within the cube may be hit first. Because most of the scene is outside the cube, many intersection computations do not have to be performed. This suggests that the smaller the cube is, the greater the savings, as more objects will be on the outside.

However, this also increases the error that environment mapping introduces, because the maps are generated with respect to the centre of the cube, but due to the size of the object(s) within the cube, secondary rays generally do not originate from the centre of the cube. Therefore, the size of the cube is preferably large with respect to the objects within. Also, the objects for which environment mapping is a suitable technique, are relatively small and cubicly shaped. Another error source is the distance of

outside objects to the environment map. The closer these objects are to the environment map, the smaller the error.

As long as the condition of small and cubic objects is satisfied, there is no objection to clustering multiple objects together and to assigning them a single environment map. Good candidates for clustering are for example plants, keyboards and generally small objects with much detail. Not very suitable are long stretched objects such as floors and walls etc.

Environment mapping consists of two distinct stages. First the maps must be generated and then the maps can be used in the rendering phase. During the generation of the environment maps, first the centre of the clustered object is determined. Then around this centre point a large cube is placed on which six environment maps will be projected, one for each side. The resolution of the maps should be high enough to capture sufficient detail for the subsequent rendering stage. Especially specular surfaces need detailed environment maps. From the centre point a number of rays are shot through each map element. Only objects outside the maps are intersected with these rays. A map entry is generated by performing standard ray tracing. After the map entries have been computed, the mip-maps are built by recursively down-filtering the environment maps. This completes the view-independent map generation.

In the rendering stage, for each pixel primary rays are shot, which gives rise to a large number of secondary rays. In our algorithm, with the exception of shadow testing point light sources and selected area light sources, these functions are largely performed by sampling the environment maps. According to the angle of incidence of the primary ray and the bi-directional reflection distribution function (brdf) of a surface, a number of secondary rays are spawned and traced within the scope of the surrounding cube.

An example of possible trace paths is given in Figure C.2. Both teapot and table possess an environment map. A primary ray hits the teapot and spawns secondary rays. Some of these rays will hit the table. For these rays according to the brdf of the table, new rays are spawned which are bounded by the environment map belonging to the table.

The distribution of secondary rays is determined by the brdf of the surface, i.e. more rays are sent in the reflected direction than in other directions. Therefore, the angle between neighbouring rays is smaller in the reflected area. Figure C.3 shows two brdf's, where the one on the left belongs to a surface that is partly diffuse and partly specular. The brdf on the right belongs to a completely diffuse surface. Because the environment maps are mip-mapped, the filter level must be computed before an entry can be calculated. The brdf determines the angle between successive rays (Figure C.3) and this angle provides a means to determine the mip-map level. The closer the rays are together, the less filtering is needed. For more diffuse areas, stronger filtered versions of the environment map should be used. The link between brdf and index level is depicted in Figure C.4.

Because no directed shooting is used for both building the environment maps and sampling the maps, there is no guarantee that point light sources are sampled accurately. Even if they are traced separately during the generation of the maps, point light sources are not well represented in the map. Therefore, these light sources are traced in the rendering stage, partly bypassing the environment maps. Also, sources that are

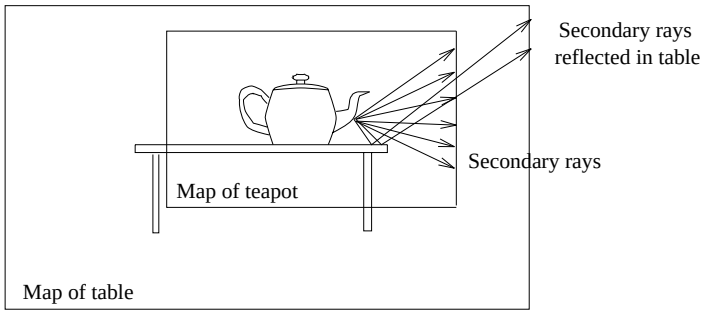


Figure C.2: Example of possible paths of secondary rays

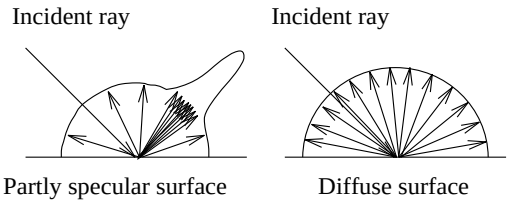


Figure C.3: Secondary rays spawned according to angle of incidence and brdf

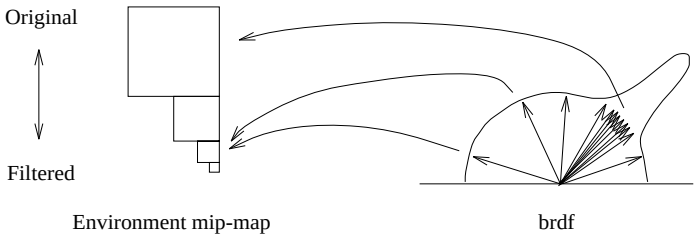


Figure C.4: Brdf determines mip-map index level

selected for source sampling are kept outside the map in order to keep the amount of noise low.

Pure specular surfaces may be handled separately as well, given the fact that the error introduced by using environment maps is largest for perfect mirrors. Finally, the advantage of having a limited scope for secondary rays becomes less when the objects outside the environment map are located close to the environment map. These examples show that environment mapping is most suitable in those cases where sampling does not significantly influence the quality of the image.

C.3 Experiments

Tests have been performed to establish the image quality that can be achieved using environment mapping. Also, the method is compared with respect to quality and rendering times with two two-pass radiosity algorithms. The first shoots primary rays and uses the pre-computed radiosity values to avoid any diffuse secondary rays. Also source selection is performed to limit the number of secondary shadow rays. In the remainder, this algorithm will be called 'source selection' (Kok and Jansen [6]). The second algorithm taking part in the comparison, shoots primary rays and for each object/primary ray intersection, both diffuse and specular secondary rays are shot. However, instead of shooting tertiary diffuse rays, the pre-computed radiosity values are used. The number of shadow rays is limited by using source selection. This algorithm is also known as 'one-level path tracing' (Rushmeier [10]); in this paper to be abbreviated as 'path tracing'.

All tests have been performed on a test scene consisting of a table on which a ball is placed. The table is in a room with a textured ceiling and four area light sources. The table has an environment map, so that the walls and the ceiling are outside the map and the table and the ball are inside the map. The objects outside are projected onto the map, which is shown in Figure C.5.

Two potential sources for errors have been defined in the preceding paragraph. These are the finite resolution of the maps and the discrepancy between the origin of secondary rays during map generation and rendering. The test scene (with a specular reflecting table top) has been rendered using different map resolutions, results of which are shown in Figure C.6. The rendering times for these pictures are given in Table C.1[1]. The effect of differences between the origin of rays during map generation and rendering can be varied by enlarging the environment map with respect to the table. In Figure C.7, the size of the environment map is increased from left to right. Corresponding rendering times are shown in Table C.1.

These images show that when the resolution is chosen too low, aliasing occurs. For this test environment, which is a worst case in the sense that the camera-point is zoomed in on a specular reflecting table, a resolution of at least 512 x 512 is needed to generate acceptable images. For more distant view points a lower resolution can be used. The same holds for more diffuse objects.

[1] All rendering has been performed on a Silicon Graphics Indigo.

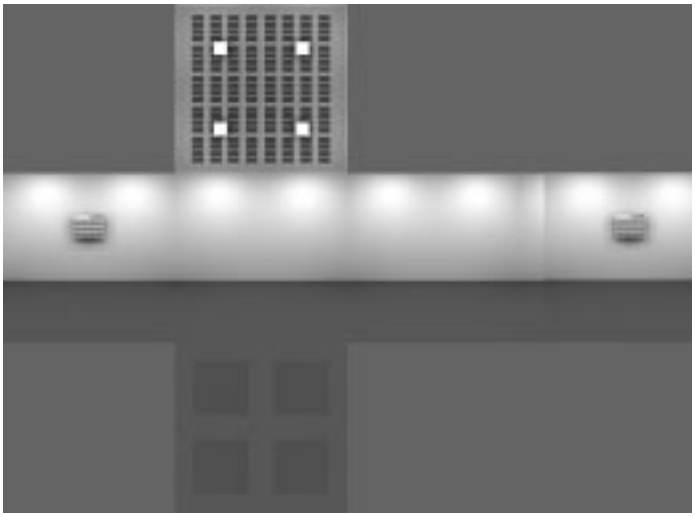


Figure C.5: Environment map generated for table in test environment

Table C.1: Rendering times in min:sec for test scene using different environment map resolutions and different distances between centre point and map

	Resolution[2]				Distance ratio[3]			
	64	128	256	512	1 : 1.25	1 : 5	1 : 10	1 : 14
Radiosity	9 : 45	9 : 45	9 : 45	9 : 45	9 : 45	9 : 45	9 : 45	9 : 45
Env. gen.	0 : 16	1 : 05	4 : 22	17 : 23	31 : 28	23 : 00	17 : 59	16 : 42
Rendering	6 : 51	6 : 54	6 : 53	6 : 55	9 : 35	9 : 33	9 : 44	9 : 49
Total	16 : 52	16 : 44	21 : 00	33 : 53	50 : 38	42 : 12	37 : 28	36 : 16

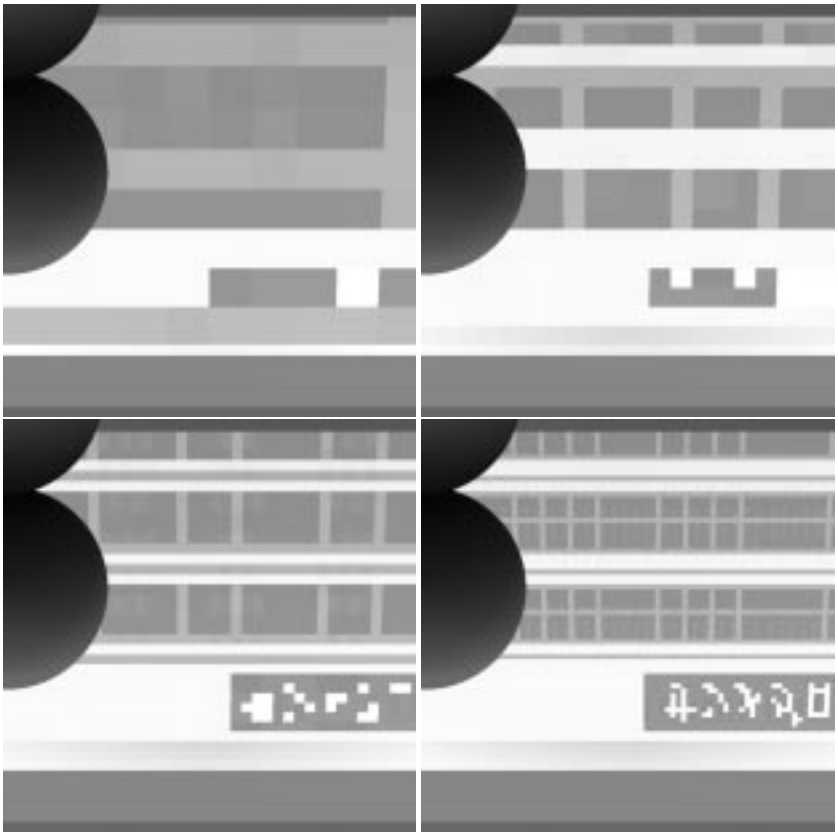


Figure C.6: Test scene rendered with increasing environment map resolution. The table top is a purely specular reflector

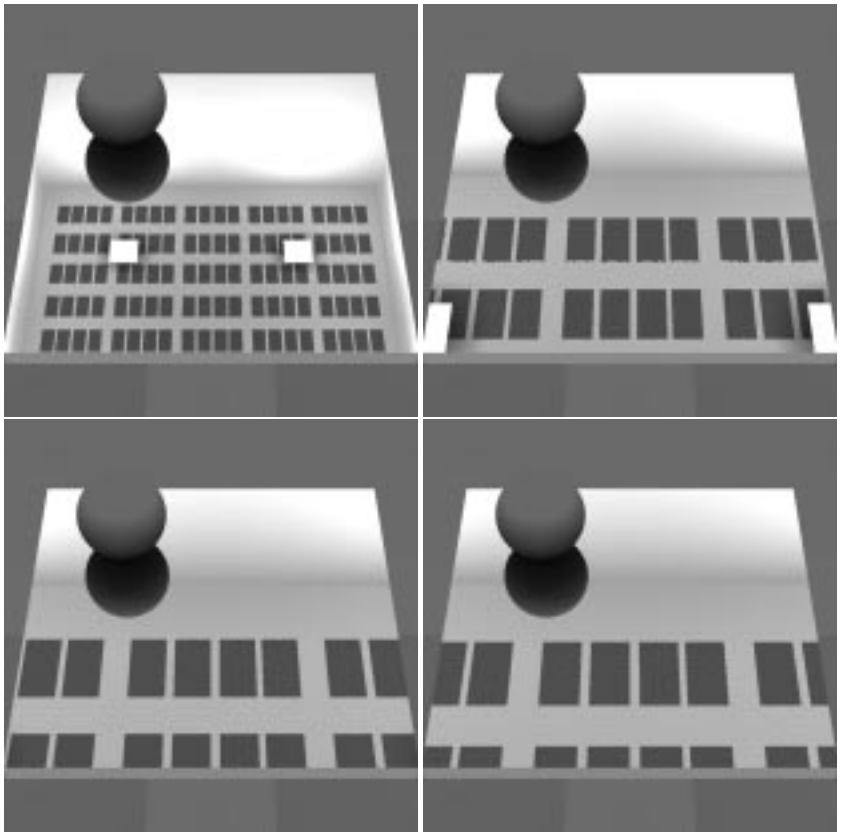


Figure C.7: Test scene rendered with increasing map size. The table top is a purely specular reflecting surface

The amount of time needed to generate the environment maps quadruples when doubling the resolution in both u and v directions. This is in accordance with expectations, as four times as many rays are cast. Actual rendering times do not vary as a function of the map resolution, because the number of secondary rays that are traced remains constant.

Varying the distance between table and environment map, it turns out that the dislocation of objects outside the environment map becomes within bounds when the distance ratio is around $1 : 10$. If the map is placed more closely to the table, the dislocation of the reflection increases.

During rendering the effect of the size of the maps on the rendering times is less significant. Here, smaller environment maps mean slightly shorter rendering times. This result is opposite to the relation between size and map generation time, because now a smaller map allows secondary rays to be traced along a shorter distance.

The test scene as used in the preceding test has been used for the comparison with the afore mentioned algorithms as well. The table top, however, is now 20% specular reflecting and 80% diffuse. In Figure C.8, the results of the algorithms are shown. Both the environment mapping and the path tracing picture exhibit colour bleeding between the ball and the table and the ball casts a more accurate and darker shadow upon the table than is the case with the source selection algorithm. The ball also receives more reflected light from the table. These effects can be attributed to the diffuse sampling performed by the environment mapping and path tracing algorithms. As the shading is completely independent of the local radiosity mesh, artifacts due to insufficient meshing are thus avoided with this algorithms.

The number of rays for each algorithm are given in Table C.2 and the rendering times are given in Table C.3. For this simple test scene, the differences are not very prominent. For more complex environments the rendering times for the environment mapping algorithm will not increase much, while the rendering times for path tracing will grow significantly. The environment map generation times are expected to grow slowly with increasing scene complexity, while the source selection algorithm's time complexity is relatively independent of the scene to be rendered.

C.4 Discussion

Environment mapping is a technique which can be incorporated in a ray tracing based radiosity algorithm. With the exception of elongated objects, environment mapping can be used for single objects or groups of objects. Three important parameters that influence the accuracy of the images are the resolution of the maps, the distance between the centre point and the environment maps and the location of objects outside the environment maps. The resolution needs to be highest for perfect mirrors, while a lower resolution is sufficient for glossy and diffuse surfaces.

[2]The resolution is the number of map entries in both u and v directions. The distance ratio is $1 : 12.5$.

[3]The ratios given in these columns are the size of the table divided by the distance of the centre point of the environment map. The resolution of the map is 512 x 512.

[4]Number of rays per pixel or per map element.

[5]Number of rays generated per intersection.

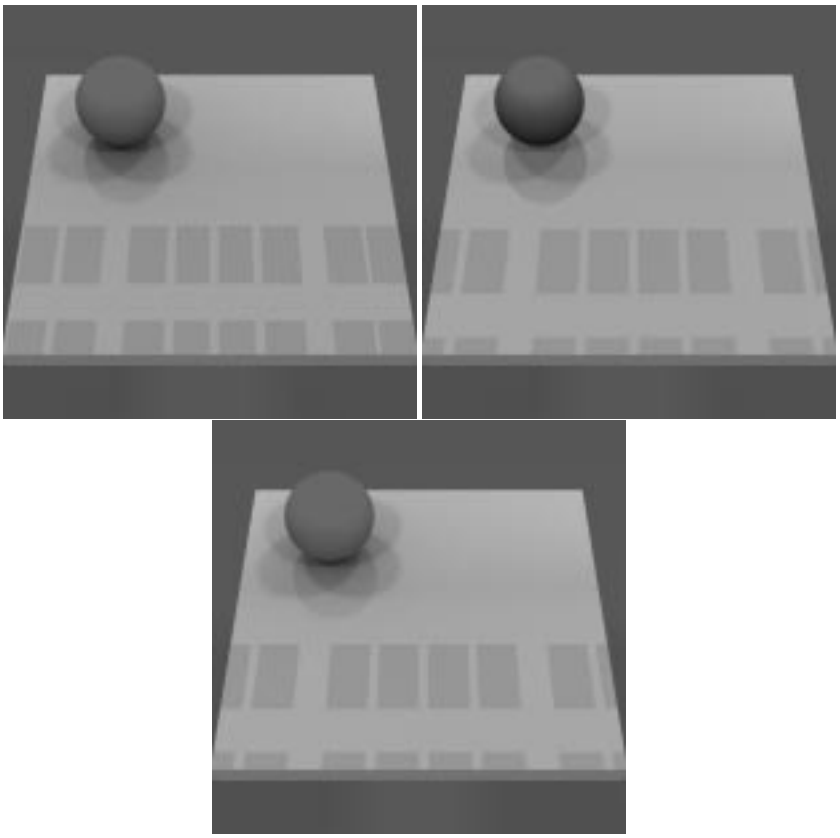


Figure C.8: Qualitative comparison of hybrid rendering algorithms: the three stage environment mapping algorithm (top), the result of the source selection algorithm (middle) and the path tracing algorithm (bottom)

Table C.2: Number of rays shot

Type of ray	Env. mapping		Source sel.	Path Tr.
	Map gen.	Rend.		
Primary[4]	1	1	1	1
Shadow[5]	16	16	16	16
Diffuse[5]	0	100	0	100
Specular[5]	1	1	1	1

Table C.3: Timings of the the rendering algorithms in hour:min:sec

Pass	Env. mapping	Source sel.	Path Tr.
Radiosity	9 : 45	9 : 45	9 : 45
Env. map generation	1 : 50 : 52	n.a.	n.a.
Rendering	7 : 17 : 43	5 : 39 : 08	11 : 17 : 36
Total	9 : 08 : 20	5 : 48 : 53	11 : 27 : 21

The distance between maps and the centre point depends on the size of the objects for which the maps are generated. If the ratio between these two is chosen too small, then the reflected surroundings appear dislocated. With respect to the quality of the images, both environment mapping and path tracing allow more accurate shadows due to the diffuse sampling that is performed during rendering. In addition, these methods are capable of handling more complicated brdf's than source selection, which splits a surface's reflection properties into a specular and a diffuse component of which the latter is taken from the radiosity mesh.

A disadvantage may be that this rendering technique requires more memory, as in addition to the scene description, a number of possibly large environment maps need to be stored. However, if a similar qualitative result were to be obtained without environment mapping, a much finer radiosity mesh would be needed, which largely cancels out the memory disadvantage of environment mapping.

Due to the more local data references made during rendering, environment mapping is better suited for parallel implementation than the source selection and path tracing algorithms. Each processor in a MIMD computer could be assigned a number of objects with their associated environment maps. This results in an object space subdivision where all objects within an environment cube, are physically stored with the same processor. Rendering can then be accomplished with minimal communication requirements, as only primary rays need to be distributed and results must be transferred to the frame buffer. Secondary rays may also induce communication, but all diffuse and specular secondary rays are handled locally. For the generation of the environment maps, communication between processors will still be required. However, the number of rays needed in this stage is relatively small compared with the amount of sampling which would otherwise be needed during rendering.

As only a sequential implementation of the environment mapping algorithm exists, our future plans include implementing this algorithm on a transputer system. Performance and scalability issues will then be examined. The ability to use more accurate brdf's has not been fully exploited yet in the implementation discussed in the preceding paragraph. Therefore, inclusion of more realistic brdf's remains work to be done. Finally, we would like to extend this algorithm so that large flat objects can be handled correctly as well.

References

[1] Blinn, J. F., Newell, M. E.: Texture and reflection in computer generated images, Communications of the ACM **19**(10), 542–547, (1976).

[2] Chen, S. E., Rushmeier, H. E., Miller, G., Turner, D.: A progressive multi-pass method for global illumination, Computer Graphics **25**(4), 165–174, (1991).

[3] Glassner, A. S.: An Introduction to Ray Tracing, Academic Press, San Diego (1989).

[4] Greene, N.: Environment mapping and other applications of world projections, IEEE Computer Graphics and Applications (1986) 21–29.

[5] Hall, H.: Hybrid Techniques for Rapid Image Synthesis, course notes, SIGGRAPH '86: Image Rendering Tricks, (1986).

[6] Kok, A. J. F., Jansen, F. W.: Source selection for the direct lighting computation in global illumination, *in* Proceedings Eurographics Workshop on Rendering, Barcelona, Spain (1991).

[7] Kok, A. J. F.: Grouping of patches in progressive radiosity, *in* M. Cohen, C. Puech , F. Sillion, eds, Fourth Eurographics Workshop on Rendering, Paris, France, 221–231 (1993).

[8] Kok, A. J. F., Jansen, F. W., Woodward, C.: Efficient, Complete Radiosity Ray Tracing Using a Shadow-coherence Method, The Visual Computer, **10**(1993) 19–33.

[9] Miller, G. S., Hoffman, C. R.: Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments, SIGGRAPH '84: Advanced Computer Graphics Animation Seminar Notes, (1984)

[10] Rushmeier, H. E.: Realistic Image Synthesis for Scenes with Radiatively Participating Media, PhD thesis (1988).

[11] Rushmeier, H. E., Patterson, C., Veerasamy, A.: Geometric Simplification for Indirect Illumination Calculations, Graphics Interface '93, 227–236 (1993).

[12] Shirley, P.: A Ray Tracing Method for Illumination Calculation in Diffuse Specular Scenes, Graphics Interface '90, 205–212 (1990).

[13] Shirley, P.: Physically Based Lighting Calculations for Computer Graphics, PhD thesis, Urbana-Champaign (1991).

[14] Sillion, F., Puech, C.: A general two-pass method integrating specular and diffuse reflection, ACM Computer graphics (1989), SIGGRAPH '89.

[15] Wallace, J. R., Cohen, M. F., Greenberg, D. P.: A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods, ACM Computer Graphics **21**(4), 311–320, (1987), SIGGRAPH '87.

[16] Williams, L.: Pyramidal parametrics, ACM Computer Graphics **17**(3), 1–11, (1983).

D Scheduling Issues in Parallel Rendering

Ray tracing is a powerful technique to generate realistic images of 3D scenes. A drawback is its high demand for processing power. Multiprocessing is one way to meet this demand. However, when the models are very large, special attention must be paid to the way the algorithm is parallelised. Combining demand driven and data parallel techniques provides good opportunities to arrive at an efficient scalable algorithm. Which tasks to process demand driven and which data driven, is decided by the data intensity of the task and the amount of data locality (coherence) that will be present in the task. Rays with the same origin and similar directions, such as primary rays and light rays, exhibit much coherence. These rays are therefore traced in demand driven fashion, a bundle at a time. Non-coherent rays are traced data parallel. By combining demand driven and data driven tasks, a good load balance may be achieved, while at the same time spreading the communication evenly across the network. This leads to a scalable and efficient parallel implementation of the ray tracing algorithm.

D.1 Introduction

From many fields in science and industry, there is an increasing demand for realistic rendering. Architects for example need to have a clear idea of how their designs are going to look in reality. In theatres the lighting aspects of the interior are important too, so these should be modelled as accurately as possible. It is evident that making such designs is an iterative and preferably interactive process. Therefore next to realism, short rendering times are called for in these applications. Another characteristic of such applications is that the models to be rendered are typically very large.

It is very difficult to develop algorithms which meet all these demands. Z-buffering algorithms are known for their speed (they are even suitable for animation purposes), but lack sufficient realism for architectural imagery. On the other hand, those algorithms that do offer an acceptable level of realism, ray tracing and radiosity, are computationally too expensive. Past attempts to overcome these problems generally follow two different approaches. One is to increase the quality of z-buffer algorithms, while retaining their speed. The other method is to speed up radiosity and ray tracing algorithms by using parallel or distributed systems. By having multiple processors to compute parts of the same problem, considerable speed-ups are to be expected.

The most common way to parallelise ray tracing is the demand driven approach where each processor is assigned a part of the image [14] [3] [13]. Alternatively, co-

herent subtasks may be assigned to processors with a low load. This has the advantage of spreading the load evenly over the processors [7] [16], but it requires either the data to be duplicated with each processor, thereby limiting the model size, or an efficient caching mechanism to be implemented. Caching may reduce the amount of communication, but its efficiency is highly dependent on the amount of data coherence. Therefore caching can never completely eliminate inter processor communication.

Alternatively, scheduling could be preformed by distributing the data over the processors according to a spatial subdivision consisting of voxels. Rays are then traced through a voxel and when a ray enters the next voxel, it is transferred as a task to the processor holding that voxel's objects [4] [2] [11]. This is called the data parallel or data driven approach and it is the basis of our parallel implementation. The advantage of such an approach is that there is virtually no restriction on the size of the model to be rendered. However, there are also some rather severe disadvantages, which include a load balancing problem and a large overhead on communication when there are a large number of processors.

The problems with either pure demand driven or data driven implementations may be overcome by combining the two, yielding a hybrid algorithm [15] [10]. Scherson et. al. [15] propose to take the ray traversal task, i.e. the intersection of rays with the spatial subdivision structure, such as an octree, to be the demand driven component. As an octree does not occupy much space, it may be replicated with each processor. Permitting the load on a processor is sufficiently low, a processor can then perform demand driven ray traversal tasks, in addition to ray tracing through its own voxel in a data driven manner. The demand driven task then compensates for the load balancing problem induced by the data parallel component, while at the same time the amount of data communication is kept low.

Computationally intense tasks that require little data are thus preferably handled in a demand driven manner, while data intensive tasks are better suited to the data parallel approach. A problem with the hybrid algorithm by Scherson et. al. [15] is that the demand driven component and the data driven component are not well matched, i.e. the ray traversal tasks are computationally not expensive enough to optimally compensate for the load balancing problem of the data parallel part. Therefore, our data parallel algorithm (presented in sections D.3 and D.4) will be modified to incorporate some extra demand driven components differently from Scherson's.

A key notion for our implementation is coherence, which means that rays that have the same origin and almost the same direction, are likely to hit the same objects. Both primary rays and shadow rays directed to area light sources (see section D.2) exhibit this coherence. To benefit from coherence, primary rays can be traced in bundles, which are called pyramids in this paper. A pyramid of primary rays has the viewpoint as top of the pyramid and its cross section will be square. First the pyramids are intersected with a spatial subdivision structure, which yields a list of cells containing objects that possibly intersect with the pyramids. Then the individual rays making up the pyramid are intersected with the objects in the cells. By localising the data necessary, these tasks can very well be executed in demand driven mode.

Shadow tracing for area light sources provides another opportunity to exploit co-

herence, as per intersection, a bundle of rays would be sent towards them. These rays originate from the intersection point and all travel towards the area light. The coherence between these rays is considerable, and they can also be traced in a pyramid, much in the same way primary rays are handled. Further, instead of communicating the shadow ray tasks to neighbouring voxels, shadow tracing can also be performed as a demand driven task at the local processor that found the intersection. This may necessitate fetching object data from other processes, so that caching becomes a relevant technique. However, because the same objects may prove to block a light sources for many possible intersections within a voxel, caching is expected to be highly efficient.

For each intersection found, shading is subsequently performed by separate shading processes. These shading processes may be scheduled as a demand driven task by assigning them to processors which have a low load. As these tasks also involve communication, they may also be scheduled on processors that have a low communication load.

Adding these three demand driven tasks (processing of primary and shadow rays and shading) to the basic data driven algorithm, will improve the load balance and increase the scalability of the architecture.

In this paper, first some general properties of ray tracing are discussed, followed in section D.3 by a description of the data parallel component of our parallel implementation. Next, the demand driven component and the handling of light rays are introduced in sections D.4 and D.5 respectively. The complete algorithm is described in terms of processes in these sections. A suitable mapping to a processor topology is described in section D.6. Finally, conclusions are drawn in section D.7.

D.2 Ray tracing

Ray tracing is a technique for creating a two-dimensional image of a three-dimensional virtual environment [20]. This environment, or model, typically consists of surfaces and light sources. The viewpoint determines which part of the model is displayed on the screen. In front of the viewpoint an image plane is selected. The part of the model which is visible from the viewpoint through the image plane, will form the image.

The light sources can be thought of as emitting rays of light. These rays can be emitted in all directions. A number of these rays will hit a surface where they are partially absorbed, reflected and refracted. Some light rays will continue along the reflected and refracted directions and may or may not hit the viewpoint.

To calculate an image, we can capture this incoming light by reversing this process, i.e. rays are traced from the eye point back into the environment. These rays are called primary rays. When such a ray intersects a surface, the shading of the surface is determined, given the direction of the incoming light, and it is assigned to the pixel in the image plane through which the ray was shot. This is the most basic form of ray tracing. A number of extensions are possible to account for more lighting effects, to improve the quality of the image or to reduce the number of computations involved.

To account for effects such as shadows and transmission of light through objects, secondary rays starting at the intersection point can be cast, see figure D.1. Three classes are summed up here [6]:

- Shadow rays or illumination rays. These rays carry light from a light source directly to a surface.
- Reflection rays. These rays are used to compute the contribution of light coming from the reflected direction. Additionally, secondary rays can be used to account for indirect reflection (radiosity) [19].
- Transparency rays. These rays carry light through an object.

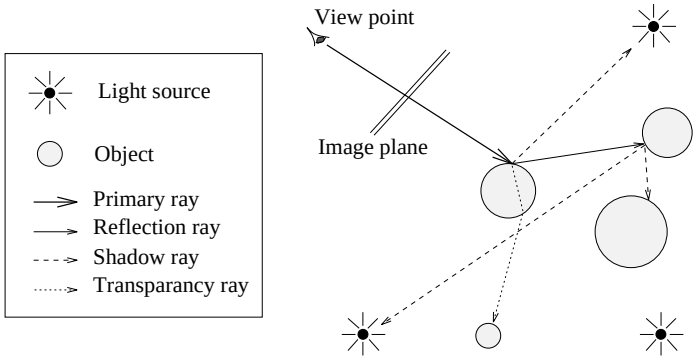


Figure D.1: Overview of ray tracing

Reflection and transparency rays introduce recursion into the algorithm, because they are treated in exactly the same way primary rays are treated. As a stopping criterion, usually a threshold is set to specify the maximum level of recursion.

An easy way to add a lot of detail to the model is by using texture mapping. Instead of assigning a single colour to each surface, a pattern (the texture) may be projected onto the surfaces of objects. Textures are usually defined as bitmaps. Whenever an intersection is found with an object that has a texture assigned to it, in addition to shooting secondary rays, this texture must be sampled as well. This is normally considered part of the shading of a surface.

D.3 Data parallel ray tracing

To derive a data parallel ray tracer, the algorithm itself must be split up into a number of processes and the object data must be distributed over these processes. Both issues are addressed in this section, beginning with the data distribution.

Objects in a model generally exhibit coherence, which means that objects consist of separate connected pieces bounded in space, and distinct objects are disjoint in this space [7]. Other forms of coherence, such as coherence between rays, where rays with

similar origins and directions are likely to intersect the same objects, are derived from this definition. To exploit object coherence in a data parallel algorithm, it is best to store objects that are close together, with the same processor. Such a distribution may be achieved by splitting the object space into (equal sized) voxels and assigning each voxel with its objects to a process.

Ray tracing is now performed in a master-slave setup. A host process initialises a number of slave processes, and then determines which slave gets which data. After initialisation, for each primary ray the host determines which voxel it originates in and sends this ray as a task to the associated trace process. When all primary rays are dispatched, the host will wait for incoming pixel values, which it will write to an image file. The host process and its incoming and outgoing messages is depicted in figure D.2.

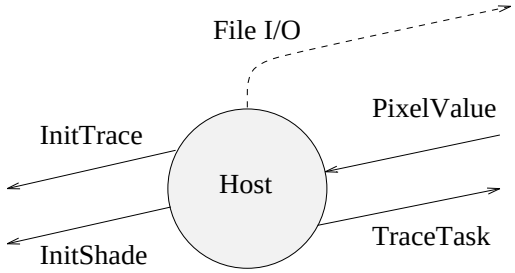


Figure D.2: Host process

There are two different types of slave processes, one which traces rays through the voxel it is assigned, and one which performs shading for each intersection detected by the tracing processes. Each tracing process has the objects pertaining to its voxel. This process reads a ray task from its buffer and traces the ray. Two situations may occur. First, the ray may leave the voxel without intersecting any object. If this happens, the ray is transferred to a neighbouring voxel. Else, an intersection with some locally stored object is found and secondary rays are spawned. These are subsequently traced until one of these two criteria is met. Also, when an intersection is found, one of the shading processes is informed, so that it reserves storage space to hold the results returned by tracing and shading secondary rays.

The shading process performs two main functions, the actual shading and the book keeping necessary as the order in which secondary rays are completed is indeterminate. It may receive intersection information from tracing processes, which tells the shading process that an intersection is found that caused secondary rays to be spawned. Somewhere in the future this shading process will receive colour values upon completion of these secondary rays.

A shading process may receive colour values from both tracing processes and (other) shading processes. For a particular intersection these will arrive later in time than the intersection information itself. Therefore, these colour values may be stored with the data concerning the intersection that generated the secondary rays. When a colour value is received, it is also determined if this colour value is the last one belonging to that particular intersection. If this is the case, all necessary information to shade

a ray has been received, and the shading is performed. After that, the resulting colour value is sent to the process that spawned the ray, which completes this shading task. The shading and tracing processes with their communication are depicted in figure D.3.

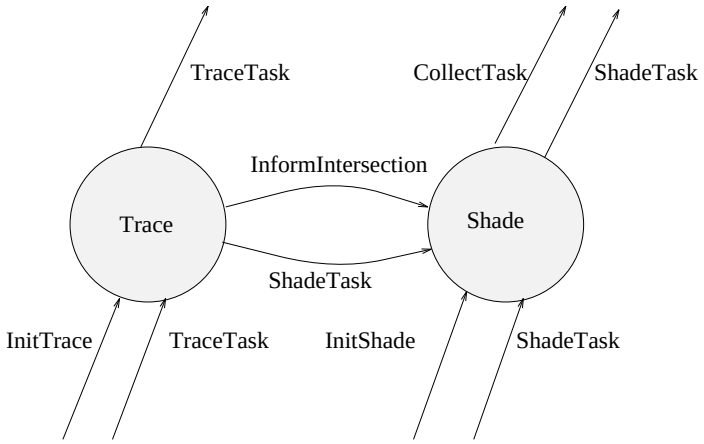


Figure D.3: Trace and shade processes

The advantages of this way of parallel ray tracing are that very large models may be rendered as the object database does not need to be duplicated with each processor, but can be divided over the processor's memories instead. Ray tasks will have to be transferred to other processors, but as each voxel borders on at most a few other voxels, communication for ray tasks is only local. This means that data parallel algorithms are scalable without too much loss of efficiency.

Disadvantages are that load imbalances may occur. These may be solved by either static load balancing, which most probably yields a suboptimal load balance, or dynamic load balancing, which may induce too much overhead in the form of data communication.

Some flexibility may be obtained by performing the shading primarily by those processors that are less occupied. For this, shading must be independent of tracing, and hence be independent of object data. To achieve this, texture mapping is assumed to be part of the tracing task. Because shading is computationally rather cheap, this may not solve the load balancing problem completely, although it is expected to balance communication requirements across the network, see section D.6.

In order to have more control over the average load of each process, and thereby overcome most of the problems associated with data parallel computing, some demand driven components may be added to this algorithm, which is the topic of the folowing sections.

D.4 Demand driven pyramid tracing

As tracing rays is by far the most expensive operation in ray tracing, this should be performed as efficiently as possible, i.e. ray coherence should be exploited where present. In ray tracing, different types of rays are generated and each of them has unique properties which call for different handling. We classify rays according to the amount of coherence between them. For example, primary rays all originate from the eye point and travel in similar directions. These rays exhibit much coherence, as they are likely to intersect the same objects. This is also true for shadow rays that are sent towards area light sources.

Restricting ourselves for the moment to primary rays, a common technique to trace a number of rays together, is by replacing them by a generalised ray [6]. Examples are cone tracing [1], beam tracing [9] [8] and pencil tracing [17]. A slightly different method is presented here, consisting of two steps. First, the primary rays are bundled together to form pyramids. These are then intersected with a spatial subdivision. The result of this pyramid traversal is a clip-list, which is an ordered list of cells that intersect (partially) with a pyramid. No object data is necessary to perform this step; only the spatial subdivision structure and the pyramids. Assuming the subdivision structure is a bin-tree, the process of pyramid traversal is depicted in figure D.4.

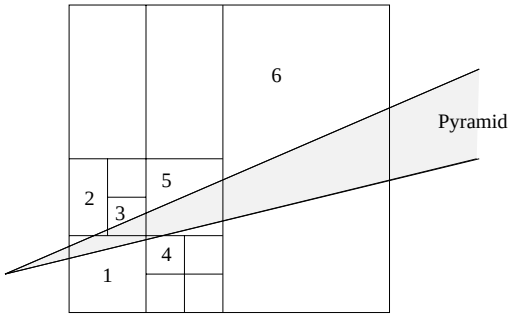


Figure D.4: Pyramid traversal generates clip list (cells 1-6).

The second step consists of tracing the individual rays within each pyramid, using the information obtained from the pyramid traversal. Tracing the rays within a pyramid will therefore require a relatively small number of objects, namely the objects that lie within the cells traversed. For this reason, pyramid tracing may be executed in a demand driven manner, as the data communication involved is restricted.

As the pyramids diverge the further they are away from the origin, coherence decreases. It may therefore be necessary after a certain distance is travelled, to switch from demand driven execution to data driven execution, as described in the preceding section.

D.5 Shadow rays

In ray tracing, for each intersection, rays are shot towards each light source. This means that in data parallel ray tracing the voxels that contain light sources, may become bottlenecks. Especially area light sources, which generate a bundle of rays per intersection, are problematic. It is therefore advantageous to apply some form of pyramid tracing to these rays as well. To avoid contention at the processes containing light sources, light pyramids may be processed locally by the processes that initiated them. If we choose to trace shadow pyramids with the process that spawned then, it may be necessary to fetch objects from remote processes, as figure D.5 illustrates. In this figure, the object in the right voxel, which is in front of the area light source, is needed by all light pyramids depicted.

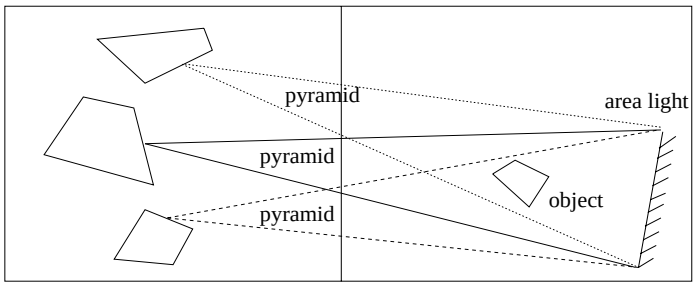


Figure D.5: The object in the right voxel is needed for light pyramid tracing by the process managing the left voxel.

Each process that may spawn light pyramids should be equipped with a cache, which reduces data communication. The effectiveness of such a cache is estimated to be high, because many pyramids generated from within a voxel, will intersect with the same objects.

D.6 Architecture and implementation

Up until this point, the algorithm is described in terms of processes, which is architecture independent. If this algorithm is to be implemented, these processes must be mapped onto a certain network topology. Often used network topologies are hypercubes and meshing architectures. The data parallel part of the algorithm has its object data subdivided according to a voxel structure. Because voxels map very conveniently to a meshing architecture, hypercubes are not considered any further.

The algorithm distinguishes a number of different processes. First of all, there is a host process, which initiates pyramid traversal tasks and receives pixel values. Second, there are pyramid tracing processes which operate in demand driven mode. Ray tracing processes for tracing individual secondary rays form a third category. These processes also perform light pyramid tracing, using a cache for objects. Tracing is performed in data driven manner. Shading processes, finally, perform shading excluding texture

mapping.

One possible mapping of processes onto processors is depicted in figure D.6. There is one host processor and a cluster of slave processors. Each of these slave processors runs data parallel tracing processes. Some of them, typically the ones with a low load, may also run a pyramid tracing or a shading process

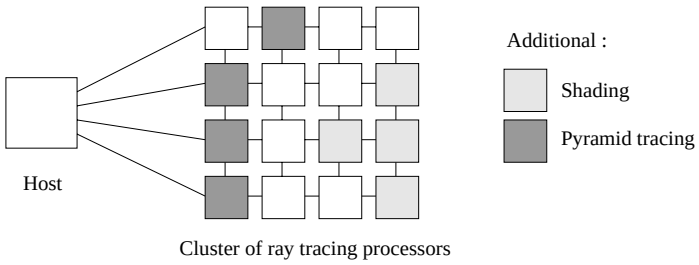


Figure D.6: Mapping of processes onto processors

The pyramid tracing processes should be located on processors that are preferably close to each other, because objects may be transferred between these processes (caching). Pyramid tracing tasks should also be scheduled to processors that are located close to the host processor, in order to minimise long distance communication.

As shading is not computationally expensive, a small number of shading processes may suffice. These may be located on those processors where due to the tracing processes, the load is expected to be low. This may be achieved through either static or dynamic load balancing.

In a data parallel tracing cluster, the optimal number of processors will be bounded by the amount of task communication which increases when new processors are added. If this optimal number of processors is lower than the number of processors available for data parallel tracing, it may be possible to add other identical data parallel clusters. In that case the object database is distributed within a cluster, but duplicated over the clusters. Ray tasks may then be executed in any of the available clusters, without ever having to be transferred between clusters. This scheme of cluster replication works because the objects in the scene do not change during the computation, therefore making it applicable to ray tracing, but not to radiosity or other algorithms which update the object database in the course of execution (e.g. Radiance [18]).

An alternative strategy would be to add more processors that work only in demand driven mode, shifting the point where demand driven tasks are converted to data driven tasks. This may lead to a more pipelined architecture (figure D.7).

For the current implementation we have used an adapted version of Rayshade [12], which is a sequential public domain ray tracer. The language used is C, extended with the PVM parallel library [5]. Although using standard libraries may be less efficient than hard coding for a specific architecture, it has great advantages in terms of portability. In addition to that, parallel ray tracing provides an almost ideal test case for such parallel tools.

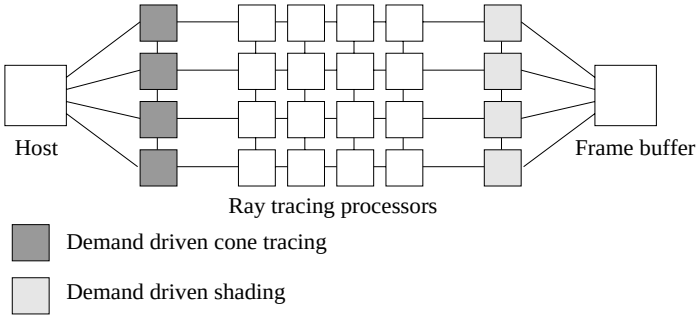


Figure D.7: “Pipelined” architecture

D.7 Conclusions

The two basic approaches to parallel rendering, demand driven and data parallel, both have their shortcomings. Data driven allows the algorithm to be scaled, but almost invariably leads to load imbalances. Demand driven approaches achieve better load balancing, but may suffer from a communication bottleneck, which makes them less scalable.

In order to arrive at an efficient scalable algorithm for ray tracing, the algorithm is best split into a demand driven and a data parallel component. Rays that show much coherence are then most efficiently traced by a demand driven algorithm which exploits ray coherence as much as possible. The pyramid tracing algorithm described in this paper does precisely that.

For rays that are less coherent, such as reflection rays and transparency rays, the data parallel approach pays off, as fetching object data, which is associated with demand driven solutions, is too expensive for single rays.

The current status of the project is that the data driven cluster is finished. All the extra demand driven processes still have to be implemented. For this reason, no speed-up of scalability measures have been taken yet.

References

[1] Amanatides, J.: ‘Ray tracing with cones’, *ACM Computer Graphics* **18**(3), 129–135. (1984)

[2] Cleary, J. G., Wyvill, B. M., Birtwistle, G. M. and Vatti, R.: ‘Multiprocessor ray tracing’, *Computer Graphics Forum* pp. 3–12. (1986)

[3] Crow, F. C., Demos, G., Hardy, J., McLaugglin, J. and Sims, K.: ‘3d image synthesis on the connection machine’, *in* ‘Proceedings Parallel Processing for Computer Vision and Display’, Leeds. (1988)

[4] Dippé, M. A. Z. and Swensen, J.: 'An adaptive subdivision algorithm and parallel architecture for realistic image synthesis', *ACM Computer Graphics* **18**(3), 149–158. (1984)

[5] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V.: *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, Oak Ridge, Tennessee. Included with the PVM 3 distribution. (1993)

[6] Glassner, A. S., ed.: *An Introduction to Ray Tracing*, Academic Press, San Diego. (1989)

[7] Green, S. A. and Paddon, D. J.: 'Exploiting coherence for multiprocessor ray tracing', *IEEE Computer Graphics and Applications* pp. 12–27. (1989)

[8] Greene, N.: Detecting intersection of a rectangular solid and a convex polyhedron, *in* P. Heckbert, ed., 'Graphics Gems IV', Academic Press, Boston, pp. 74–82. (1994)

[9] Heckbert, P. S. and Hanrahan, P.: 'Beam tracing polygonal objects', *ACM Computer Graphics* **18**(3), 119–127. (1984)

[10] Jansen, F. W. and Chalmers, A.: Realism in real time?, *in* '4th EG Workshop on Rendering', pp. 1–20. (1993)

[11] Kobayashi, H., Nishimura, S., Kubota, H., Nakamura, T. and Shigei, Y.: 'Load balancing strategies for a parallel ray-tracing system based on constant subdivision', *The Visual Computer* **4**(4), 197–209. (1988)

[12] Kolb, C. E.: *Rayshade User's Guide and Reference Manual*. Included in Rayshade distribution, which is available by ftp from princeton.edu:pub/Graphics/rayshade.4.0. (1992)

[13] Lin, T. T. Y. and Slater, M.: 'Stochastic ray tracing using SIMD processor arrays', *The Visual Computer* **7**, 187–199. (1991)

[14] Plunkett, D. J. and Bailey, M. J.: 'The vectorization of a ray-tracing algorithm for improved execution speed', *IEEE Computer Graphics and Applications* **5**(8), 52–60. (1985)

[15] Scherson, I. D. and Caspary, C.: 'A self-balanced parallel ray-tracing algorithm', *in* P. M. Dew, R. A. Earnshaw and T. R. Heywood, eds, 'Parallel Processing for Computer Vision and Display', Vol. 4, Addison-Wesley Publishing Company, Wokingham, pp. 188–196. (1988)

[16] Shen, L. S.: 'A Parallel Image Rendering Algorithm and Architecture Based on Ray Tracing and Radiosity Shading', PhD thesis, TUDelft, Delft. (1993)

[17] Shinya, M., Takahashi, T. and Naito, S.: 'Principles and applications of pencil tracing', *ACM Computer Graphics*. (1987)

[18] Ward, G. J.: 'The radiance lighting simulation and rendering system', *ACM Computer Graphics* pp. 459–472. SIGGRAPH '94 Proceedings. (1994)

[19] Ward, G. J., Rubinstein, F. M. and Clear, R. D.: 'A ray tracing solution for diffuse interreflection', *ACM Computer Graphics* **22**(4), 85–92. (1988)

[20] Whitted, T.: 'An improved illumination model for shaded display', *Communications of the ACM* **23**(6), 343–349. (1980)

Summary

Ray tracing is a widely accepted technique for generating realistic images of artificial models. One of its major drawbacks is the time needed to compute an image. For this reason parallel processing is often used to accelerate image generation. This report presents a ray tracer in which a hybrid scheduling technique is employed. Scheduling consists of a data parallel and a demand driven component. The latter uses the Pyra clip method to efficiently compute intersections for a large number of rays and at the same time keeping communication requirements within bounds. Which rays to compute in a demand driven manner and which using the data parallel algorithm, is decided by the amount of coherence between rays. Single rays are traced by the data parallel algorithm and coherent rays are processed by the demand driven algorithm. The result is a scalable ray tracing algorithm capable of rendering images of large models.

Samenvatting

Ray tracing is een algemeen toepasbare techniek voor het genereren van realistische afbeeldingen van kunstmatige modellen. Een van de grootste nadelen van deze techniek is de rekentijd die nodig is om een plaatje te berekenen. Om die reden wordt vaak gebruik gemaakt van parallel rekenen om het process van beeldgeneratie te versnellen.

De belangrijkste manieren waarop ray trace algoritmen geparallelliseerd worden, zijn volgens data parallelle, dan wel demand driven methoden. Beide hebben voor- en nadelen welke nagenoeg complementair zijn. Een data parallelle opdeling geeft load balancing problemen, maar is, althans volgens de theorie, min of meer schaalbaar. Een demand driven algoritme veroorzaakt bij grote modellen object communicatie, maar de load is gemakkelijker te balanceren. Het is daarom een voor de hand liggende gedachte beide methoden te combineren tot een hybride scheduling algoritme, waarbij gepoogd wordt de voordelen van beide te behouden en de nadelen op te heffen. Welke delen van de berekening op demand driven manier gescheduled worden, en welke op data parallelle wijze, wordt beslist aan de hand van de hoeveelheid coherentie die tussen verschillende rays aanwezig is.