

Rendering Large Scenes Using Parallel Ray Tracing

Erik Reinhard and Frederik W. Jansen

Faculty of Technical Mathematics and Informatics,
Delft University of Technology
Julianalaan 132, 2628BL Delft, The Netherlands
Email: (erik |fwj)@duticg.twi.tudelft.nl

Abstract

Ray tracing is a powerful technique to generate realistic images of 3D scenes. However, the rendering of complex scenes may easily exceed the processing and memory capabilities of a single workstation. Distributed processing offers a solution if the algorithm can be parallelised in an efficient way. In this paper a hybrid scheduling approach is presented that combines demand driven and data parallel techniques. 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. By combining demand driven and data driven tasks, a good load balance is 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 with fairly no restriction on the size of the model data base to be rendered.

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.

Computer graphics techniques such as ray tracing and radiosity methods can provide the realism that is required for these applications, but the processing and memory capabilities needed for rendering (realistically) large scenes often exceed the capacity of single workstations. Using parallel or distributed systems, either with distributed or shared memory architectures, seems a logical solution to these problems but, unfortunately, it is not trivial to match the processing with the distributed data in an optimal way.

The most common way to parallelise ray tracing is the demand driven approach where each processor is assigned a part of the image (Plunkett & Bailey 1985) (Crow, Demos, Hardy, McLaugglin & Sims 1988) (Lin & Slater 1991). Alternatively, coherent subtasks may be assigned to processors with a low load. This has the advantage of spreading the load evenly over the processors (Green & Paddon 1989) (?), but it requires either the data to be duplicated with each processor, thereby limiting the

model size, or objects have to be communicated on request, introducing a significant extra overhead. Caching mechanisms may be implemented to reduce the amount of communication, but its efficiency is highly dependent on the amount of coherence between subsequent data requests. This may be low in ray tracing, in particular for reflection rays. Also, with realistic rendering often too much data is involved (textures, radiosity meshes, etc.) to be communicated frequently.

Alternatively, scheduling could be performed by distributing the data over the processors according to a (regular) spatial subdivision. Rays are then traced through a voxel (cell) of the spatial subdivision and when a ray enters the next voxel, it is transferred as a task to the processor holding that voxel's objects (Dippé & Swensen 1984) (Cleary, Wyvill, Birtwistle & Vatti 1986) (Kobayashi, Nishimura, Kubota, Nakamura & Shigei 1988). 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 to the size of the model to be rendered. However, there are also some rather severe disadvantages, which include a load balancing problem and an increasing communication overhead with larger numbers of processors.

The problems with either pure demand driven or data driven implementations may be overcome by combining the two, yielding a hybrid algorithm (Scherson & Caspary 1988) (Jansen & Chalmers 1993). Scherson and Caspary 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 parallel 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 of Scherson and Caspary is that the demand driven component and the data parallel component are not well matched, i.e. the ray traversal tasks are computationally not expensive enough to fully compensate for the load balancing problem of the data parallel part. Therefore, our data parallel algorithm (presented in section 2) will be modified to incorporate two extra demand driven components (section 3).

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 the 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.

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 2. Next, the demand driven component and the handling of light rays are introduced

in section 3. The complete algorithm is described in section 4. Experiments with two reasonably complex models are described in section 5. The data parallel and hybrid methods are compared in section 6. Finally, conclusions are drawn in section 7.

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.

A suitable data distribution is to split the object space into (equal sized) voxels exploiting local object coherence. Each processor is assigned a process and a voxel with its objects. Ray tracing is now performed in a master-slave setup. After initialisation, for each primary ray the host determines which voxel it originates in (or enters) and sends this ray as a task to the associated trace process. This tracing process reads the 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 rays are traced and if necessary communicated to neighbouring voxels. The results of these rays are returned to the process that spawned the rays. When all results for an intersection are completed, shading is performed and a colour value is returned. The host collects all the shading information for each pixel and writes the result to an image file (see figure 1).

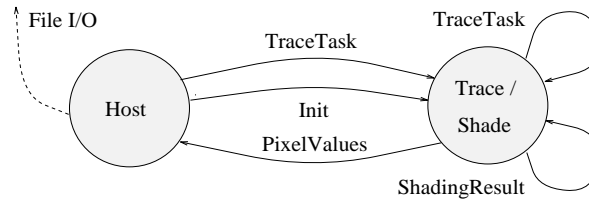


Figure 1: Host and slave processes.

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 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 in principle scalable, although the communication overhead grows with the number of processors.

Disadvantages are that load imbalances may occur, due to 'hot spots' in the scene (near the view-point and near light sources). These may be solved by either static load balancing (Priol & Bouatouch 1989), which most probably yields a suboptimal load balance, or by dynamic load balancing (Dippé & Swensen 1984), which may induce extra overhead in the form of data communication.

3 Demand driven pyramid tracing

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.

The demand-driven tasks are in our approach selected on the amount of coherence between rays. Bundles of rays may show different amounts of coherence. 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.

To exploit this coherence, a bundle of rays can be enclosed in a pyramid or cone. This pyramid is then with the spatial subdivision to retrieve all objects within the pyramid, prior to tracing the rays (Greene 1994). A similar algorithm, called PyraClip (der Zwaan, Reinhard & Jansen 1995), was implemented by us to retrieve a list of cells in depth order from a bintree structure. Figure 2 shows the retrieved cells (1-6) for this example.

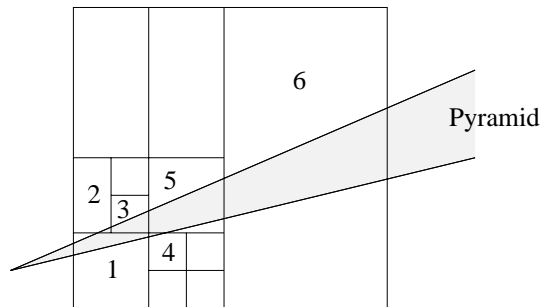


Figure 2: Pyramid traversal generates clip list (cells 1-6).

After the PyraClip preprocessing is completed, the tracing of the individual rays within the pyramid only requires 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 limited.

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, 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. Light pyramids are then preferably processed by the process that initiated them to avoid contention at the processes that contain the light sources (Priol & Bouatouch 1989). It may then be necessary to fetch objects from remote processes, as figure 3 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.

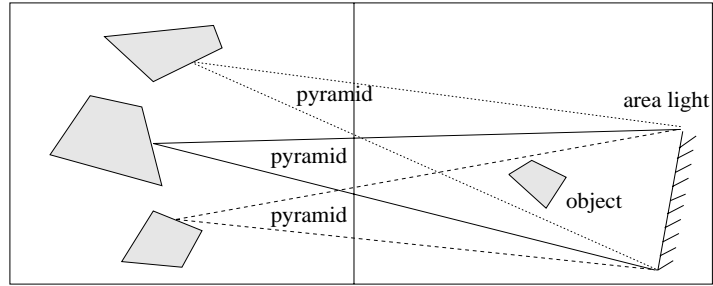


Figure 3: The object in the right voxel is needed for light pyramid tracing by the process managing the left voxel.

4 Hybrid system

Adding the demand-driven components to the data parallel system can be done by giving the host the task to pre-process the PyraClip tasks and schedule these to other processors. However, this would require a complete copy of the data base to be stored at the host, which puts a significant constraint on the size of the data base. Also, the processing involved may lead to a computational bottle-neck at the host. We therefore choose to distribute the PyraClip preprocessing as well.

In the new set-up there is a host process that issues bundles of rays on request to the (distributed) trace processes. The trace processes are, next to the data parallel ray tracing, capable of performing the demand-driven tasks of preprocessing the PyraClip tasks and executing the PyraClip tasks (tracing the rays). To avoid that all trace processes should have access to the whole data base for the PyraClip preprocessing, only the spatial subdivision structure is duplicated with each process. With the PyraClip preprocessing then first an ordered list of intersected cells (the 'clip-list') is created. Second, the actual contents of each cell is requested. A cache can be maintained for objects most often called for. In fact, in our current implementation, the cache is pre-filled with a resident set consisting of objects closest to the viewpoint (see figure 4). Rays that survive these objects are further processed in a data parallel manner.

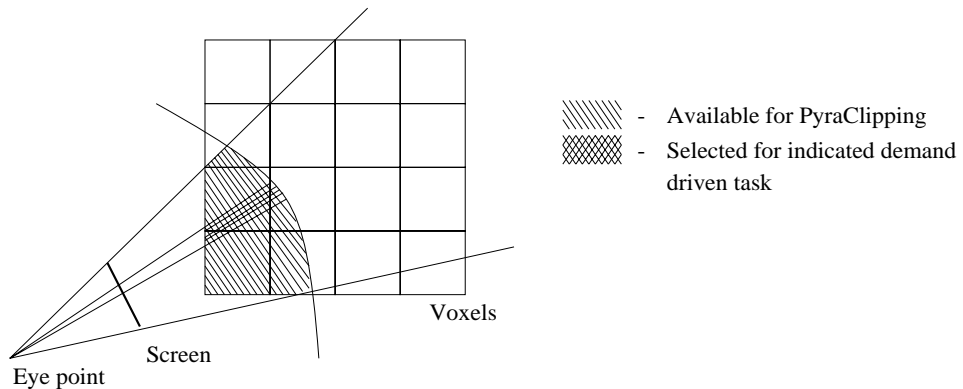


Figure 4: Data distribution for demand driven tasks. Each process holds the objects closest to the eye point and pre selects from these objects to form demand driven tasks.

When during demand-driven tracing of primary rays an intersection is found then further processing is transferred to the process that keeps the intersected object (in the data parallel scheme). This process then spawns the secondary rays that are traced data parallel. This process also spawns the shadow rays. As mentioned earlier, the shadow tracing is performed locally, supported by the shadow cache. When the results of the secondary rays are gathered, the shading is performed and the result is returned to the host. When no intersection is found during demand-driven tracing of primary rays, this result is returned directly to the host process.

In figure 5, the resulting setup of host, data parallel trace and demand driven PyraClip tasks is depicted. PyraClip tasks can be generated either for primary rays and for tracing shadow rays.

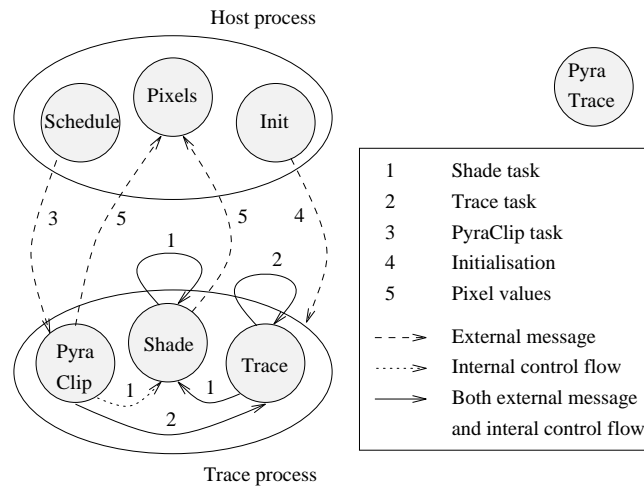


Figure 5: Control and message flow between processes.

There are further a large number of difficult control issues involved which deal with memory usage (buffer and cache sizes, etc.), communication (buffer overflow, message packing, saturation and deadlock), task scheduling (where to do PyraClip tracing, shadow tracing and shading) and with trade-offs between demand-driven and data-parallel processing. See for further details Reinhard & Jansen (1996).

5 Implementation and Experiments

The described scheduling and data management techniques were implemented in the (sequential) public domain ray tracer Rayshade (Kolb 1992). For communication support we choose the PVM library (Geist, Beguelin, Dongarra, Jiang, Manchek & Sunderam 1993). Although standard libraries may be less efficient than hard coding for a specific architecture, it has great advantages in terms of portability. Our current system runs on a cluster of workstations as well as on dedicated multi-processor systems.

The first model used for evaluation of the data parallel and hybrid scheduling algorithms stems from the SPD model database (Haines 1992) and is slightly modified to better reflect the type of model

these algorithms are designed for. From the balls5 model the plane is removed and the ten largest balls are made non-reflective, see figure 6 (left). Although still on the stiff side, the amount of reflectivity is more conform the reflectivity found in architectural models. Moreover only one light source is used instead of the standard three (otherwise the shadow caches would be too strongly reduced in size to be effective). This model consists of 66.430 spheres. When more than four processors are used, the balls model has a rather unequal distribution of objects over space. A more realistic situation is presented with the Conference room, a model created by Ward¹ (figure 6 right). The conference room has 23177 polygons. The area light sources were replaced by 8 resp. 30 point light sources.

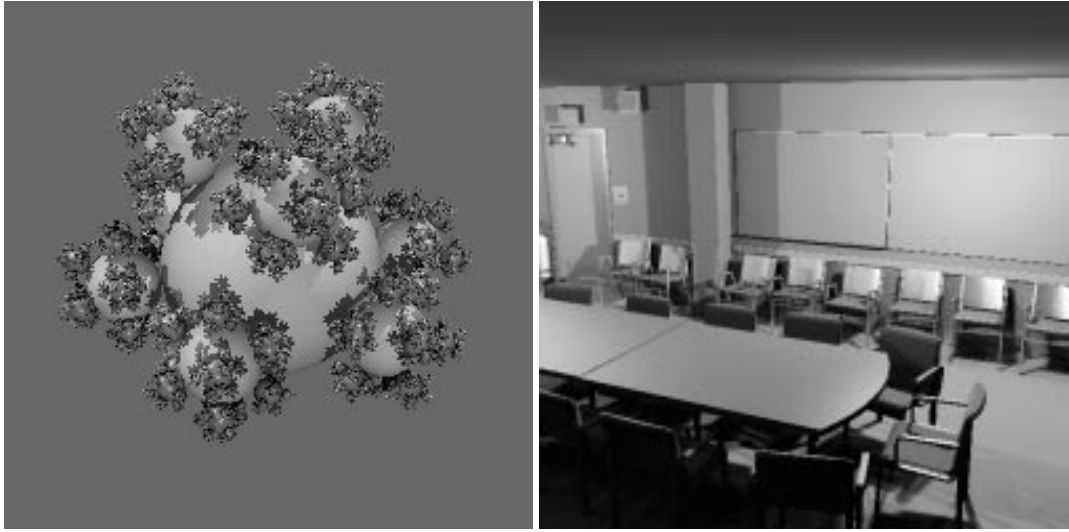


Figure 6: Balls model containing 66.430 spheres (left) and conference room with 23.966 polygons (right).

The experiments were carried out on 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. In all tests performed, the image size is 256 x 256. For comparison we also give the optimal performance for the sequential version of Rayshade for these models on a SGI Onyx with one R8000 processor and 256 MB of local memory.

6 Comparison between scheduling techniques

The timing results are listed in table 1. Preprocessing and (pre)loading of the data in memories and caches is not included.

The figures show that data parallel rendering alone is not scalable and performs rather poorly. The hybrid scheduling algorithm performs (slightly) better. This can be accounted for by the load balancing capabilities of the demand driven tasks, as is clearly demonstrated in figure 7. In this figure

¹Obtained from <http://radsite.lbl.gov/mgf/scenes.html>

Table 1: 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.

Procs	Balls 5		Conference room 8		Conference room 30	
	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

the processors that have only a few data parallel tasks, receive a large number of demand driven rays and vice versa. Hence a better load distribution than in data parallel rendering.

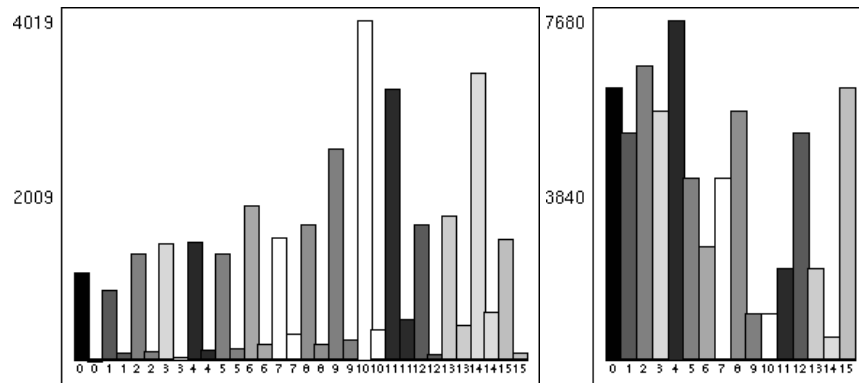


Figure 7: Demand driven rays complement the workload of the data parallel cluster (balls model; 16 processors). The left bars in the left figure represent the data parallel rays per processor. The right bars in the left figure represent the fraction of data parallel rays that was transferred to another 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 the expected speed ups for a large number of processors. This question is answered in figure 8, where the efficiency over time is given for the balls model with 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 and to improve the performance of the remaining rays in the data parallel cluster.

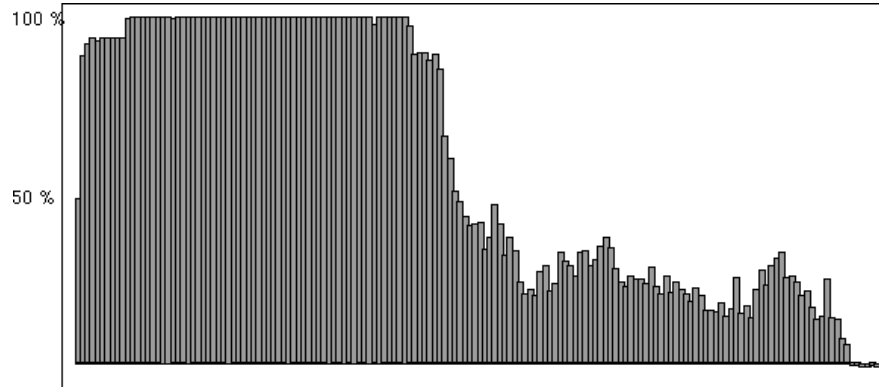


Figure 8: Efficiency over time (balls model; 16 processors). Each bar represents a 0.1s time interval.

7 Conclusions

The two basic approaches to parallel rendering, demand driven and data parallel, both have their advantages and shortcomings. Demand driven approaches suffer from a communication bottleneck due to insufficient cache performance for large models and non-coherent data requests. Data driven scheduling allows the algorithm to be scaled, but almost invariably leads to load imbalances. The experiments also show that the unbalance does not follow the distribution of objects over processes and -even worse- is not constant over time. The unbalance is thus hard to correct with traditional load adapting strategies as dynamic redistribution of data (Dippé & Swensen 1984) or static load balancing according to a low resolution rendering pass (Bouatouch & Priol 1988).

The hybrid approach of adding demand driven components to balance the load works fairly well given the abysmal results of the data parallel rendering alone. 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.

So far, we have performed the shadow tracing locally (data parallel). Adding more light sources therefore adds to the data parallel load (unbalance) and diminishes the effect of the demand driven component. Shadow tracing is therefore the next candidate to be scheduled demand driven.

Our aim is to render much larger models than the examples given here. With an increasing amount of data, data parallel processing will gain in importance. Texture mapping and radiosity processing were not involved in our experiments yet, and these will add to the data parallel component as well. Therefore, data parallel processing should be improved, both by better load balancing and by reducing the latency of the ray flow through the system.

A way to improve the data parallel processing would be to switch from a regular grid of voxels to a data structure that is more adapted to the model. We will also continue to experiment with hierarchical sampling techniques to reduce the number and length of secondary and shadow rays (Reinhard, Tijssen & Jansen 1994) (Kok 1994).

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 parallel machine. However, this would sacrifice much of the portability and thus the implementation would be confined to run on machines with a specific operating system.

References

- Bouatouch, K. & Priol, T. (1988), Parallel space tracing: An experience on an iPSC hypercube, *in* N. Magnenat-Thalmann & D. Thalmann, eds, 'New Trends in Computer Graphics (Proceedings of CG International '88)', Springer-Verlag, New York, pp. 170–187.
- Cleary, J. G., Wyvill, B. M., Birtwistle, G. M. & Vatti, R. (1986), 'Multiprocessor ray tracing', *Computer Graphics Forum* **5**(1), 3–12.
- Crow, F. C., Demos, G., Hardy, J., McLauglin, J. & Sims, K. (1988), 3d image synthesis on the connection machine, *in* 'Proceedings Parallel Processing for Computer Vision and Display', Leeds.
- der Zwaan, M. ., Reinhard, E. & Jansen, F. W. (1995), Pyramid clipping for efficient ray traversal, *in* P. Hanrahan & W. Purgathofer, eds, 'Rendering Techniques '95', Trinity College, Dublin, Springer - Vienna, pp. 1–10. proceedings of the 6th Eurographics Workshop on Rendering.
- Dippé, M. A. Z. & Swensen, J. (1984), An adaptive subdivision algorithm and parallel architecture for realistic image synthesis, *in* H. Christiansen, ed., 'Computer Graphics (SIGGRAPH '84 Proceedings)', Vol. 18, pp. 149–158.
- Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. & Sunderam, V. (1993), *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, Oak Ridge, Tennessee. Included with the PVM 3 distribution.
- Green, S. A. & Paddon, D. J. (1989), 'Exploiting coherence for multiprocessor ray tracing', *IEEE Computer Graphics and Applications* **9**(6), 12–26.
- Greene, N. (1994), Detecting intersection of a rectangular solid and a convex polyhedron, *in* P. Heckbert, ed., 'Graphics Gems IV', Academic Press, Boston, pp. 74–82.
- Haines, E. A. (1992), Standard procedural database, v3.1, 3D/Eye.
- Jansen, F. W. & Chalmers, A. (1993), Realism in real time?, *in* M. F. Cohen, C. Puech & F. Sillion, eds, '4th EG Workshop on Rendering', Eurographics, pp. 27–46. held in Paris, France, 14–16 June 1993.
- Kobayashi, H., Nishimura, S., Kubota, H., Nakamura, T. & Shigei, Y. (1988), 'Load balancing strategies for a parallel ray-tracing system based on constant subdivision', *The Visual Computer* **4**(4), 197–209.

- Kok, A. J. F. (1994), *Ray Tracing and Radiosity Algorithms for Photorealistic Image Synthesis*, PhD thesis, Delft University of Technology, The Netherlands. Delft University Press, ISBN 90-6275-981-5.
- Kolb, C. E. (1992), *Rayshade User's Guide and Reference Manual*. Included in Rayshade distribution, which is available by ftp from [princeton.edu:pub/Graphics/rayshade.4.0](ftp://princeton.edu/pub/Graphics/rayshade.4.0).
- Lin, T. T. Y. & Slater, M. (1991), 'Stochastic ray tracing using SIMD processor arrays', *The Visual Computer* **7**(4), 187–199.
- Plunkett, D. J. & Bailey, M. J. (1985), 'The vectorization of a ray-tracing algorithm for improved execution speed', *IEEE Computer Graphics and Applications* **5**(8), 52–60.
- Priol, T. & Bouatouch, K. (1989), 'Static load balancing for a parallel ray tracing on a MIMD hypercube', *The Visual Computer* **5**(1/2), 109–119.
- Reinhard, E. & Jansen, F. W. (1996), A parallel ray tracing system for rendering complex scenes, Technical Report TBA, Faculty of Technical Mathematics and Informatics, Delft University of Technology.
- Reinhard, E., Tijssen, L. U. & Jansen, F. W. (1994), Environment mapping for efficient sampling of the diffuse interreflection, in G. Sakas, P. Shirley & S. Müller, eds, 'Photorealistic Rendering Techniques', Eurographics, Springer Verlag, Darmstadt, pp. 410–422. proceedings of the 5th Eurographics Workshop on Rendering.
- Scherson, I. D. & Caspary, C. (1988), 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.