Scheduling Issues in Parallel Rendering

Erik Reinhard, Frederik W. Jansen*

Abstract

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.

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 (Plunkett and Bailey 1985) (Crow, Demos, Hardy, McLaugglin and Sims 1988) (Lin and 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 and Paddon 1989) (Shen 1993), 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.

*Delft University of Technology
Faculty of Technical Mathematics and Computer Science
Julianalaan 132
2628BL Delft
Netherlands

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 (Dippé and Swensen 1984) (Cleary, Wyvill, Birtwistle and Vatti 1986) (Kobayashi, Nishimura, Kubota, Nakamura and 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 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 (Scherson and Caspary 1988) (Jansen and Chalmers 1993). Scherson and Caspary (1988) 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 and Caspary (1988) 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 3 and 4) will be modified to incorporate some extra demand driven components differently from Scherson and Caspary (1988).

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 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 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.

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 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 4 and 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 6. Finally, conclusions are drawn in section 7.

2 Ray tracing

Ray tracing is a technique for creating a two-dimensional image of a three-dimensional virtual environment (Whitted 1980). 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 1. Three classes are summed up here (Glassner 1989):

- 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) (Ward, Rubinstein and Clear 1988).
- Transparency rays. These rays carry light through an object.

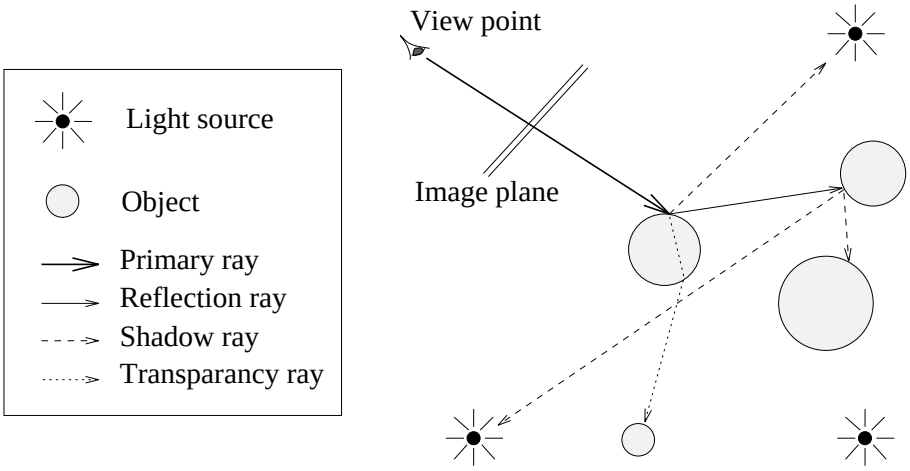

Figure 1: Overview of ray tracing

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.

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 (Green and Paddon 1989). 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 2.

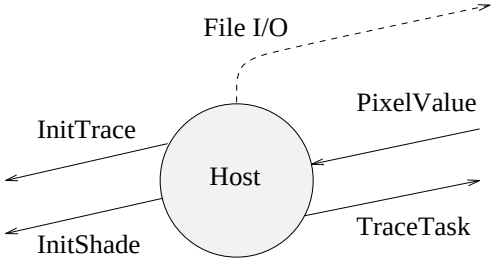

Figure 2: Host process

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 3.

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

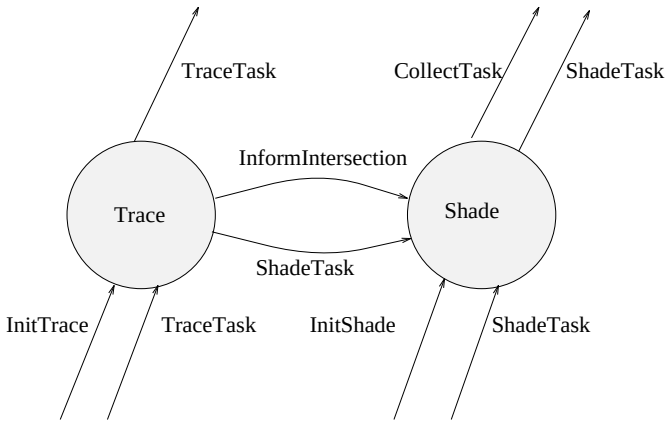

Figure 3: Trace and shade processes

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 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.

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 (Glassner 1989). Examples are cone tracing (Amanatides 1984), beam tracing (Heckbert and Hanrahan 1984) (Greene 1994) and pencil tracing (Shinya, Takahashi and Naito 1987). 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 4.

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 there-

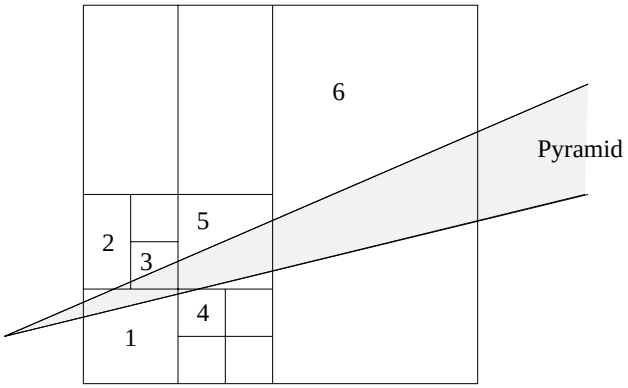

Figure 4: Pyramid traversal generates clip list (cells 1-6).

fore be necessary after a certain distance is travelled, to switch from demand driven execution to data driven execution, as described in the preceding section.

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 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.

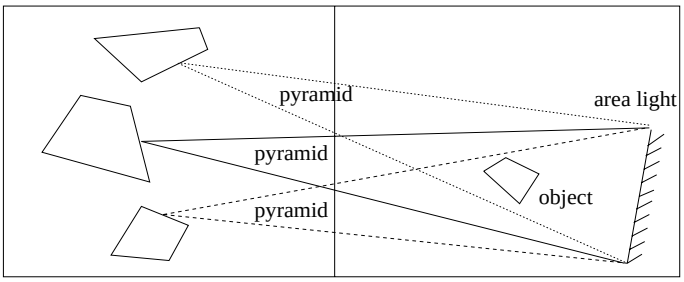

Figure 5: The object in the right voxel is needed for light pyramid tracing by the process managing the left voxel.

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.

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 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

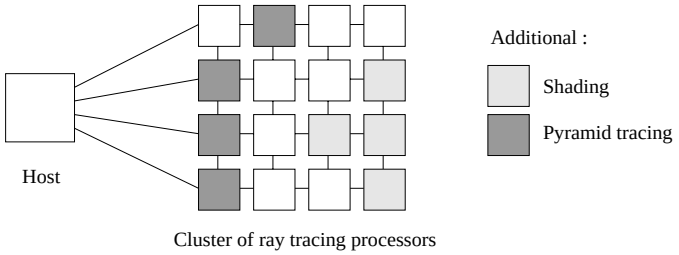

Figure 6: Mapping of processes onto processors

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 (Ward 1994)).

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 7).

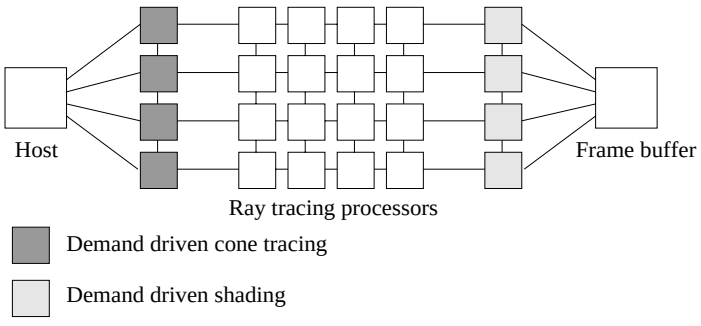

Figure 7: “Pipelined” architecture

For the current implementation we have used an adapted version of Rayshade (Kolb 1992), which is a

sequential public domain ray tracer. The language used is C, extended with the PVM parallel library (Geist, Beguelin, Dongarra, Jiang, Manchek and Sunderam 1993). 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.

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

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

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

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

Dippé, M. A. Z. and 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. and Sunderam, V. (1993), *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, Oak Ridge, Tennessee. Included with the PVM 3 distribution.

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

Green, S. A. and 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.

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

Jansen, F. W. and Chalmers, A. (1993), Realism in real time?, *in* M. F. Cohen, C. Puech and 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. and Shigei, Y. (1988), 'Load balancing strategies for a parallel ray-tracing system based on constant subdivision', *The Visual Computer* **4**(4), 197–209.

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.

Lin, T. T. Y. and Slater, M. (1991), 'Stochastic ray tracing using SIMD processor arrays', *The Visual Computer* **7**(4), 187–199.

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

Scherson, I. D. and Caspary, C. (1988), 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.

Shen, L. S. (1993), A Parallel Image Rendering Algorithm and Architecture Based on Ray Tracing and Radiosity Shading, PhD thesis, Delft University of Technology, Delft.

Shinya, M., Takahashi, T. and Naito, S. (1987), 'Principles and applications of pencil tracing', *Computer Graphics (SIGGRAPH '87 Proceedings)*.

Ward, G. J. (1994), The RADIANCE lighting simulation and rendering system, *in* A. Glassner, ed., 'Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)', Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, ACM Press, pp. 459–472.

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

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