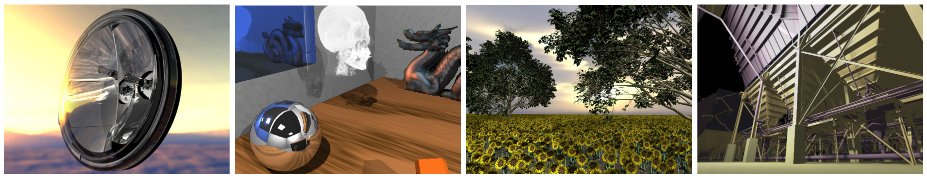
Interactive Ray Tracing on Commodity PC clusters
State of the Art and Practical Applications

Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek

Saarland University, Germany



Abstract. Due to its practical significance and its high degree of parallelism, ray tracing has always been an attractive target for research in parallel processing. With recent advances in both hardware and software, it is now possible to create high quality images at interactive rates on commodity PC clusters.
In this paper, we will outline the "state of the art" of interactive distributed ray tracing based on a description of the distributed aspects of the OpenRT interactive ray tracing engine. We will then demonstrate its scalability and practical applicability based on several example applications.

1 Introduction

The ray tracing algorithm is well-known for its ability to generate high quality images but has also been infamous for its long rendering times. Speeding up ray tracing for interactive use has been a long standing goal for computer graphics research. Significant efforts have been invested, mainly during the 1980ies and early 90ies, as documented for example in [11].

For users of graphical applications the availability of real-time ray tracing offers a number of interesting benefits: The ray tracing algorithm closely models the physical process of light propagation by shooting imaginary rays into the scene. Thus, it is able to accurately compute global and advanced lighting and shading effects. It exactly simulates shadows, reflection, and refraction on arbitrary surfaces even in complex environments (see Figures 1 and 2). Furthermore, ray tracing automatically combines shading effects from multiple objects in the correct order. This allows for building the individual objects and their shaders independently and have the ray tracer automatically take care of correctly rendering the resulting combination of shading effects (cf. Section 3.1). This feature is essential for robust industrial applications, but is not offered by current graphics hardware. Finally, ray tracing efficiently supports huge models with billions of polygons showing a logarithmic time complexity with respect to scene size, i.e. in the number of triangles in a scene (cf. Section 3.2). This efficiency is due to inherent pixel-accurate *occlusion culling* and *demand driven* and *output-sensitive* processing that computes only visible results.

1.1 Fast and Parallel Ray Tracing

However, ray tracing is a very costly process, since it requires to trace millions of rays into a virtual scene and to intersect those with the geometric primitives. In order to improve performance to interactive rates requires to combine highly optimized ray tracing implementations with massive amounts of computational power.

Due to its high degree of parallelism, together with its practical significance for industrial applications (e.g. for lighting simulation, visualization, and for the motion picture industry), especially parallel ray tracing has attracted significant research, not only from the graphics community (e.g. [16, 5]), but also from the parallel computing community (e.g. [15]). Muuss et al. [13] and Parker et al. [14] were the first to show that interactive ray tracing is possible by massive parallelization on large shared memory supercomputers. More recently, Wald et al. [19] demonstrated that interactive frame rates can also be achieved on commodity PC clusters. In their research, Wald et. al have accelerated software ray tracing by more than a factor of 15 compared to other commonly used ray tracers. This has been accomplished by algorithmic improvements together with an efficient implementation designed to fit the capabilities of modern processors. For instance, changing the standard ray tracing algorithm to tracing packets of rays and to perform computations within each packet in breadth first order improves coherence, and enables efficient use of data parallel SIMD extensions (e.g. SSE) of modern processors. Paying careful attention to coherence in data access also directly translates to better use of processor caches, which increasingly determines the runtime of today's programs. The combination of better usage of caches and SIMD extensions is crucial to fully unfold the potential of today's CPUs, and will probably be even more important for future CPU designs.

Even though these optimizations allow some limited amount of interactive ray tracing even on a single modern CPU, one PC alone still cannot deliver the performance required for practical applications, which use complex shading, shadows, reflections, etc. In order to achieve sufficient performance on todays hardware requires combining the computational resources of multiple CPUs.

As has already been shown by Parker et al. [14], ray tracing scales nicely with the number of processors, but only if *fast access to scene data* is provided, e.g. by using shared memory systems. Today, however, the most cost-effective approach to compute power is a distributed memory PC cluster. Unfortunately they provide only low bandwidth with high latencies for data access across the network. In a related publication, Wald et al. [21] have shown how interactive ray tracing can be realized on such a hardware platform. In the following we briefly discuss the main issues of high-performance implementations in a distributed cluster environment, by taking a closer look at the distribution framework of the OpenRT interactive ray tracing engine [18].

2 Distribution Aspects of the OpenRT Interactive Ray Tracing Engine

Before discussing some of the parallel and distribution details, we first have to take a closer look at the overall system design.

2.1 General System Design

Client-Server Approach: Even though our system is designed to run distributed on a cluster of PCs, we assume this ray tracer to be used by a single, non-parallel application running on a single PC. As a consequence, we have chosen to follow the usual client/server approach, where a single master centrally manages a number of slave machines by assigning a number of tasks to each client. The clients then perform the actual ray tracing computations, and send their results back to the server in the form of readily computed, quantized color values for each pixel.

Screen Space Task Subdivision: Effective parallel processing requires to break the task of ray tracing into a set of preferably independent subtasks. For predefined animations (e.g. in the movie industry), the usual way of parallelization is to assign different frames to different clients in huge render farms. Though this approach successfully improves throughput, it is not applicable to a real-time setting, where only a single frame is to be computed at any given time.

For real-time ray tracing, there are basically two approaches: *object space* and *screen space* subdivision [16, 5]. Object space approaches require the entire scene database to be distributed across a number of machines, usually based on an initial spatial partitioning scheme. Rays are then forwarded between clients depending on the next spatial partition pierced by the ray. However, the resulting network bandwidth would be too large for our commodity environment. At today's ray tracing performance individual rays can be traced much faster than they can be transferred across a network. Finally, this approach often tends to create *hot-spots* (e.g. at light sources that concentrate many shadow rays), which would require dynamic redistribution of scene data.

Instead, we will follow the screen-based approach by having the clients compute disjunct regions of the same image. The main disadvantage of screen-based parallelization is that it usually requires a local copy of the whole scene to reside on each client, whereas splitting the model over several machines allows to render models that are larger than the individual clients' memories. In this paper, we do not consider this special problem, and rather assume that all clients can store the whole scene. In a related publication however, it has been shown how this problem can be solved efficiently by caching parts of the model on the clients (see [21]).

Load Balancing: In screen space parallelization, one common approach is to have each client compute every n-th pixel (so-called pixel-interleaving), or every n-th row or scanline. This usually results in good load balancing, as all clients get roughly the same amount of work. However, it also leads to a severe loss of *ray coherence*, which is a key factor for fast ray tracing. Similarly, it translates to bad cache performance resulting from equally reduced *memory coherence*.

An alternative approach is to subdivide the image into quadrangular regions (called *tiles*) and assign those to the clients. Thus, clients work on neighboring pixels that expose a high degree of coherence. The drawback is that the cost for computing different tiles can significantly vary if a highly complex object (such as a complete power plant as shown in Figure 2) projects onto only a few tiles, while other tiles are empty. For *static task assignments* – where all tiles are distributed among the clients before any actual computations – this variation in task cost would lead to extremely bad client utilization and therefore result in bad scalability.

Thus, we have chosen to use a tile-based approach with a dynamic load balancing scheme in our system: Instead of assigning all tiles in advance, corresponding to a *data driven* approach, we pursue a *demand driven* strategy by letting the clients themselves ask for work. As soon as a client has finished a tile, it sends its results back to the server, and requests the next unassigned tile from the master.

Hardware Setup: To achieve the best cost-effective setting, we have chosen to use a cluster of dual-processor PCs interconnected by commodity networking equipment. Currently, we are using up to 24 dual processor AMD AthlonMP 1800+ PCs with 512 MB RAM each. The nodes are interconnected by a fully switched 100 Mbit Ethernet using a single Gigabit uplink to the master display and application server for handling the large amounts of pixel data generated in each image. Note, that this hardware setup is not even state of the art, as much faster processors and networks are available today.

2.2 Optimization Details

While most of the above design issues are well-known and are applied in similar form in almost all parallel ray tracing systems, many low-level details have to be considered in order to achieve good client utilization even under interactivity constraints. Though we can not cover all of them here, we want to discuss the most important optimizations used in our system.

Communication Method: For handling communication, most parallel processing systems today use standardized libraries such as MPI [8] or PVM [10]. Although these libraries provide very powerful tools for development of distributed software, they do not meet the efficiency requirements that we face in an interactive environment.

Therefore, we had to implement all communication from scratch with standard UNIX TCP/IP calls. Though this requires significant efforts, it allows to extract the maximum performance out of the network. For example, consider the 'Nagle' optimization implemented in the TCP/IP protocol, which delays small packets for a short time period to possibly combine them with successive packets to generate network-friendly packet sizes. This optimization can result in a better throughput when lots of small packets are sent, but can also lead to considerable latencies, if a packet gets delayed several times. Direct control of the systems communication allows to use such optimizations selectively: For example, we turn the Nagle optimization on for sockets in which updated scene data is streamed to the clients, as throughput is the main issue here. On the other hand, we turn it off for e.g. sockets used to send tiles to the clients, as this has to be done with an absolute minimum of latency. A similar behavior would be hard to achieve with standard communication libraries.

Differential Updates: Obviously, the network bandwidth is not high enough for sending the entire scene to each client for every frame. Thus, we only send differential updates from each frame to the next: Only those settings that have actually changed from the previous frame (e.g. the camera position, or a transformation of an object) will be sent to the clients. Upon starting a new frame, all clients perform an update step in which they incorporate these changes into their scene database.

Asynchronous Rendering: Between two successive frames, the application will usually change the scene settings, and might have to perform considerable computations before the next frame can be started. During this time, all clients would run idle. To avoid this problem, rendering is performed asynchronously to the application: While the application specifies frame N, the clients are still rendering frame $N-1$. Once the application has finished specifying frame N, it waits for the clients to complete frame $N-1$, displays that frame, triggers the clients to start rendering frame N, and starts specifying frame $N+1$. Note, that this is similar to usual *double-buffering* [17], but with one additional frame of latency.

Asynchronous Communication: As just mentioned, the application already specifies the next frame while the clients are still working on an old one. Similarly, all communication between the server and the clients is handled asynchronously: Instead of waiting for the application to specify the complete scene to be rendered, scene updates from the application are *immediately* streamed from the server to all clients in order to minimize communication latencies. Asynchronously to rendering tiles for the old frame, one thread at the client already receives the new scene settings and buffers them for future use. Once the rendering threads have finished, (most of) the data for the next frame has already arrived, further minimizing latencies. These updates are then integrated into the local scene database, and computations can immediately be resumed without losing time receiving scene data.

Multithreading: Due to a better cost/performance ratio, each client in our setup is a dual-processor machine. Using multithreading on each client then allows to share most data between these threads, so the cost of sending scene data to a client can be amortized over two CPUs. Furthermore, both client and server each employ separate tasks for handling network communication, to ensure minimum network delays.

Task Prefetching: If only a single tile is assigned to each client at any time, a client runs idle between sending its results back to the server and receiving the next tile to be computed. As a fast ray tracer can compute tiles in at most a few milliseconds, this delay can easily exceed rendering time, resulting in extremely bad client utilization. To avoid these latencies, we let each client request ("prefetch") several tiles in advance. Thus, several tiles are 'in flight' towards each client at any time. Ideally, a new tile is just arriving every time a previous one is sent on to the server. Currently, each client is usually prefetching about 4 tiles. This, however depends on the actual ratio of compute performance to network latency, and might differ for other hardware configurations.

Spatio-Temporal Coherence: In order to make best use of the processor caches on the client machines load balancing also considers the *spatio-temporal coherence* between rays by assigning the same image tiles to the same clients in subsequent frames whenever possible. As soon as a client has processed all of 'its' old tasks, it starts to 'steal' tasks from a random other machine.

3 Applications and Experiments

In the following, we demonstrate the potential and scalability of our system based on several practical examples. If not mentioned otherwise, all experiments will run at video resolution of 640×480 pixels. As we want to concentrate on the practical results, we will but closely sketch the respective applications. For more details, also see [18, 20, 3].

3.1 Classical Ray Tracing

Plug'n Play Shading: One of the main advantages of ray tracing is its potential for unsurpassed image realism, which results from its ability to support specialized shader programs for different objects (e.g. a glass shader), which can then easily be combined with shaders on other objects in a plug and play manner. This is also the reason why ray tracing is the method of choice of almost all major rendering packages. The potential of this approach can be seen in Figure 1: An office environment can be simulated with several different shaders which fit together seamlessly. For example, a special shader realizing volume rendering with a skull data set seamlessly fits into the rest of the scene: i.e. it is correctly reflected in reflective objects, and casts transparent shadows on all other objects. Using our distribution framework, the ray tracer scales almost linearly to up to 48 CPUs, and achieves frame rates of up to 8 frames per second (see Figure 4).

Visualisation of Car Headlights for Industrial Applications: Being able to efficiently simulate such advanced shading effects also allows to offer solutions to practical industrial problems that have so far been impossible to tackle. For example, the limitations of todays graphics hardware force current Virtual Reality (VR) systems to make heavy use of approximation, which in turn makes reliable quantitative results impossible, or at least hard to achieve. Examples are the visualization of reflections of the car's dash board in the side window where it might interfere with backward visibility through the outside mirror at night. Another example is the high-quality visualization of car headlights that are important design features due to being the "eyes of the car".

Figure 1 shows an 800.000-triangle VRML model of a car headlight rendered with our ray tracing engine [3]. Special reflector and glass shaders are used that carefully model the physical material properties and intelligently prune the ray trees. The latter is essential for achieving real-time frame rates because the ray trees otherwise tend to get very large due to many recursive reflections and refractions. For good visual results we still have to simulate up to 25 levels of recursion for certain Pixels (cf. Figure 1d).

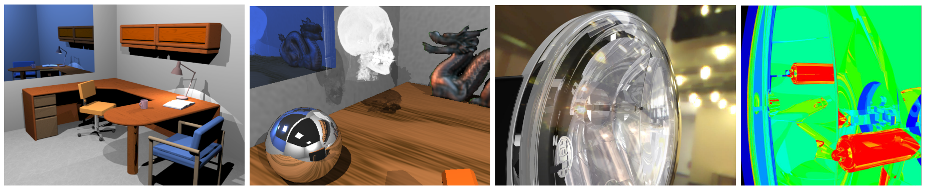


Fig. 1. Classical ray tracing: (a) Typical office scene, with correct shadows and reflections, and with programmable procedural shaders. (b) The same scene with volume and lightfield objects. Note how the volume casts transparent shadows, the lightfield is visible through the bump-mapped reflections on the mirror, etc. (c) The headlight model with up to 25 levels of reflection and refraction. (d) False-color image showing the number of reflection levels per pixel (red: 25+).

Simulating this level of lighting complexity is currently impossible to compute with alternative rendering methods. Thus, for the first time this tool allows automotive designers and headlight manufacturers to interactively evaluate a design proposal. This process previously took several days for preparing and generating predefined animations on video. Now a new headlight model can be visualized in less than 30 minutes and allows designers the long missing option of freely interacting with the model for exploring important optical effects.

3.2 Interactive Visualization of Highly Complex Models

Apart from its ability to simulate complex lighting situations, another advantage of ray tracing is that in practice it's time complexity is *logarithmic* in scene size [12], allowing to easily render even scenes with several million up to billions of triangles. This is of great importance to VR and engineering applications, in which such complex models have to be visualized. Currently, such models have to be simplified before they can be rendered interactively. This usually requires expensive preprocessing [1, 2], significant user intervention, and often negatively affects the visualization quality. With our distributed ray tracing system, we can render such models interactively (see Figures 2 and 4), and – even more importantly – without the need for geometric simplifications.

Figure 2 shows screenshots from two different models: The image on the left shows three complete power plants consisting of 12.5 million triangles each. Using our ray tracer, we can easily render several such power plants at the same time, and can even interactively move parts of the power plant around. Especially the latter is very important for design applications, but is usually impossible with alternative technologies, as simplification and preprocessing ususally work only in static environments.

Yet another advantage of ray tracing is its possibility to use *instantiation*, the process of re-using parts of a model several times in the same scene. For example, the second model in Figure 2 consists of only ten different kinds of sunflowers (of roughly 36,000 triangles each) and one type of tree, which are then instantiated several thousand times to form a complete landscape of roughly one *billion* triangles. Furthermore, the scene is rendered including transparency textures for the leaves, and computes even pixel-accurate shadows cast by the sun onto the leaves (see Figures 2c and 2d). Using our framework, both scenes can be rendered interactively, and achieve almost-linear scalability, as can be seen in Figures 2 and 4.

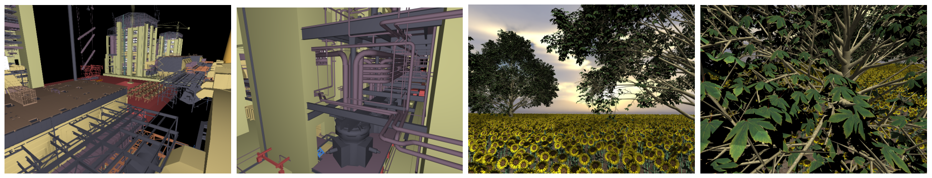


Fig. 2. Complex models: (a) Three powerplants of 12.5 million individual triangles each, rendering interactively at 23 fps. (b) A closeup on the highly detailed geometry. (b) An outdoor scene consisting of roughly 28,000 instances of 10 different kinds of sunflowers with 36,000 triangles each together with several multi-million-triangle trees. The whole scene consists of roughly one billion triangles and is rendered including shadows and transparency. (d) A closeup of the highly detailed shadows cast by the sun onto the leaves.

3.3 Interactive Lighting Simulation

Even though classical ray tracing as described above considers only direct lighting effects, it already allows for highly realistic images that made ray tracing the preferred rendering choice for many animation packages. The next step in realism can be achieved by including indirect lighting effects computed by *global illumination* algorithms as a standard feature of 3D graphics (see Figure 3).

Global illumination algorithms account for the often subtle but important effects of indirect lighting effects in a physically-correct way [6, 7] by simulating the global light transport between all mutually visible surfaces in the environment. Due to the need for highly flexible visibility queries, virtually all algorithms today use ray tracing for this task. Because of the amount and complexity of the computations, rendering with global illumination is usually even more complex than classical ray tracing, and thus slow and far from interactive, taking several minutes to hours even for simple diffuse environments. The availability of real-time ray tracing should now enable to compute full global illumination solutions also at interactive rates (see Figure 4). We will not cover the details of the algorithm here, which are described in close detail in [20] and [4].

Using our system, we can interactively simulate light propagation in a virtual scene, including soft shadows, reflections and indirect illumination. As each frame is recomputed from scratch, interactive changes to the environment can be handled well, allowing to modify geometry, lights and materials at interactive rates.

4 Results and Conclusions

In this paper, we have shown how efficient parallelization of a fast ray tracing kernel on a cluster of PCs can be used to achieve interactive performance even for high-quality applications and massively complex scenes. We have sketched the parallelization and distribution aspects of the OpenRT distributed interactive ray tracing engine [18], which uses screen-space task subdivision and demand-driven load balancing, together with low-level optimization techniques to minimize bandwidth and to hide latencies. The combination of these techniques allow the system to be used even in a real-time setting, where only a few milliseconds are available for each frame. We have evaluated the performance of the proposed system (see Figure 4) in a variety of test scenes, for three different applications: Classical ray tracing, visualising massively complex models, and interactive lighting simulation: Due to the limited bandwidth of our display

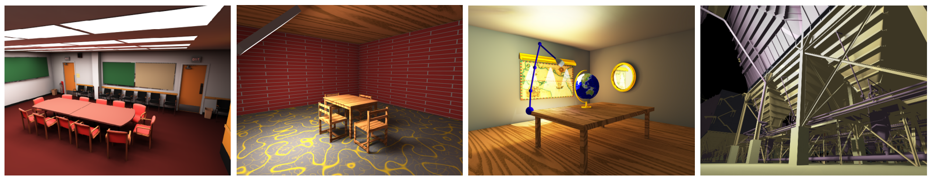


Fig. 3. An interactive global illumination application in different environments. From left to right: (a) A conference room of 280.000 triangles, (b) The "Shirley 6" scene, with global illumination including complex procedural shaders, (c) An animated VRML model, and (d) global illumination in the power plant with 37.5 million triangles. All scenes render at several frames per second.

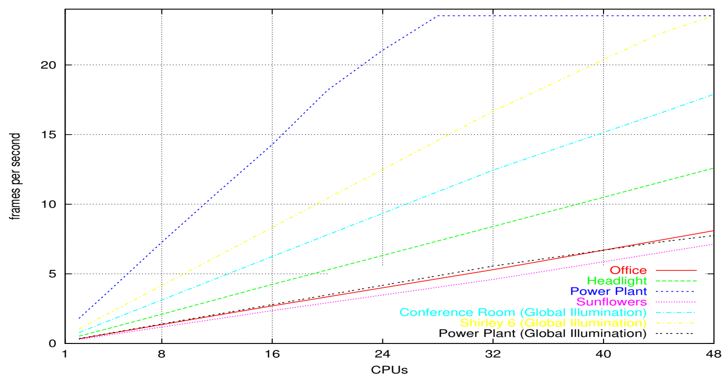


Fig. 4. Scalability of our system on a cluster of up to 48 CPUs. Note that the system is limited to about 22–24 frames per second due to the limited network connection of the display server. Up to this maximum framerate, all scenes show virtually linear scalability.

server (which is connected to the Gigabit uplink of the cluster), our framerate is limited to roughly 22–24 frames per second at 640×480 pixels, as we simply cannot transfer more pixels to the server for display. Up to this maximum framerate however, all scenes show virtually linear scalability for up to 48 CPUs (see Figure 4).

The presented applications have also shown that a fast and distributed software implementation of ray tracing is capable of delivering completely new types of interactive applications: While todays graphical hardware solutions (including *both* the newest high-performance consumer graphics cards *and* expensive graphics supercomputers) can render millions of triangles per second, they can not interactively render whole scenes of many million to even billions of triangles as shown in Section 3.2.

Furthermore, such hardware architectures can not achieve the level of image quality and simulation quality that we have shown to be achievable with our system (see Section 3.1). This is especially true for interactive lighting simulation (Section 3.3), which – due to both its computational cost and its algorithmic complexity – is unlikely to be realized on graphics hardware any time soon. With this capability of enabling completely new applications, our system provides real value for practical applications, and is already being used in several industrial projects.

Still, typical VR applications demand even higher resolutions and even higher framerates. Typically, resolutions up to 1600×1200 are required to drive equipment such as PowerWalls etc, and 'real-time' applications usually demand frame rates of 25 frames per second and more. This leaves enough room for even more parallelization, and also requires to eventually think about more powerful network technologies (such as Myrinet [9]) to provide the required network performance.

Acknowledgements

The RTRT/OpenRT project has been supported by Intel Corp. The Sunflowers and Powerplant models have been graciously provided by Oliver Deussen and Anselmo Lastra.

References

1. D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stürzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An Interactive Massive Model Rendering System Using Geometric and Image-Based Acceleration. In *ACM Symposium on Interactive 3D Graphics*, pages 199–206, Atlanta, USA, April 1999.
2. William V. Baxter III, Avneesh Sud, Naga K Govindaraju, and Dinesh Manocha. Gigawalk: Interactive Walkthrough of Complex Environments. In *Rendering Techniques 2002*, pages 203 – 214, June 2002. (Proceedings of the 13th Eurographics Workshop on Rendering 2002).
3. Carsten Benthin, Ingo Wald, Tim Dahmen, and Philipp Slusallek. Interactive Headlight Simulation – A Case Study for Distributed Interactive Ray Tracing. In *Proceedings of Eurographics Workshop on Parallel Graphics and Visualization (PGV)*, pages 81–88, 2002.
4. Carsten Benthin, Ingo Wald, and Philipp Slusallek. A Scalable Approach to Interactive Global Illumination. to be published at Eurographics 2003, 2003.
5. Alan Chalmers, Timothy Davis, and Erik Reinhard, editors. *Practical Parallel Rendering*. AK Peters, 2002. ISBN 1-56881-179-9.
6. Micheal F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Morgan Kaufmann Publishers, 1993. ISBN: 0121782700.
7. Philip Dutre, Kavita Bala, and Philippe Bekaert. Advanced Global Illumination. In *SIGGRAPH 2001 Course Notes, Course 20*. 2001.
8. MPI Forum. MPI – The Message Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi.
9. Myrinet Forum. Myrinet. http://www.myri.com/myrinet/overview/.
10. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, Cambridge, 1994.
11. Andrew Glassner. *An Introduction to Raytracing*. Academic Press, 1989.
12. Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, 2001.
13. Michael J. Muuss. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium '95*, June 1995.
14. Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics (I3D)*, pages 119–126, 1999.
15. Tomas Plachetka. Perfect Load Balancing for Demand-Driven Parallel Ray Tracing. In B. Monien and R. Feldman, editors, *Lecture Notes in Computer Science*, pages 410–419. Springer Verlag, Paderborn, August 2002. (Proceedings of Euro-Par 2002).
16. Erik Reinhard. *Scheduling and Data Management for Parallel Ray Tracing*. PhD thesis, University of East Anglia, 1995.
17. B. Schachter. *Computer Image Generation*. Wiley, New York, 1983.
18. Ingo Wald, Carsten Benthin, and Philipp Slusallek. OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Saarland University, 2002. Available at http://graphics.cs.uni-sb.de/Publications.
19. Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics 2001).
20. Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive Global Illumination using Fast Ray Tracing. *Rendering Techniques 2002*, pages 15–24, 2002. (Proceedings of the 13th Eurographics Workshop on Rendering).
21. Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. *Rendering Techniques 2001*, pages 274–285, 2001. (Proceedings of the 12th Eurographics Workshop on Rendering).