EUROGRAPHICS 2001 / A. Chalmers and T.-M. Rhyne
(Guest Editors)

Volume 20 (2001), Number 3

Interactive Rendering with Coherent Ray Tracing

Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner

Computer Graphics Group, Saarland University

Abstract

For almost two decades researchers have argued that ray tracing will eventually become faster than the rasterization technique that completely dominates todays graphics hardware. However, this has not happened yet. Ray tracing is still exclusively being used for off-line rendering of photorealistic images and it is commonly believed that ray tracing is simply too costly to ever challenge rasterization-based algorithms for interactive use. However, there is hardly any scientific analysis that supports either point of view. In particular there is no evidence of where the crossover point might be, at which ray tracing would eventually become faster, or if such a point does exist at all.

This paper provides several contributions to this discussion: We first present a highly optimized implementation of a ray tracer that improves performance by more than an order of magnitude compared to currently available ray tracers. The new algorithm makes better use of computational resources such as caches and SIMD instructions and better exploits image and object space coherence. Secondly, we show that this software implementation can challenge and even outperform high-end graphics hardware in interactive rendering performance for complex environments. We also provide an brief overview of the benefits of ray tracing over rasterization algorithms and point out the potential of interactive ray tracing both in hardware and software.

1. Introduction

Ray tracing is famous for its ability to generate high-quality images but is also well-known for long rendering times due to its high computational cost. This cost is due to the need to traverse a scene with many rays, intersecting each with the geometric objects, shading the visible surface samples, and finally sending the resulting pixels to the screen. Due to the cost associated with ray tracing the technique is viewed almost exclusively as an off-line technique for cases where image quality matters more than rendering speed.

On the other hand ray tracing offers a considerable number of advantages over other rendering techniques. Basic ray casting, i.e. sampling the scene with individual rays, is a fundamental task that is the core of a large number of algorithms not only in computer graphics. Other disciplines use the same approach for example to simulate propagation of radio waves [11], neutron transport, and diffusion [33]. Another example is DARPA's large "Data Intensive Systems" (DIS) project [10], which is mainly motivated by the need to speed up ray tracing for computing radar cross sections.

But even if we only concentrate on rendering applications, ray tracing offers a number of benefits over rasterization-based algorithms that dominate todays algorithms targeted at interactive 3D graphics:

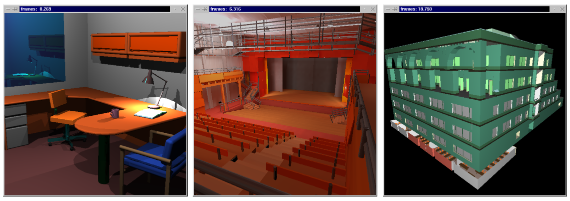


Figure 1: *Interactive ray tracing: The office, conference and Soda Hall models contain roughly 40k, 680k, and 8 million triangles, respectively. Using our software ray tracing implementation on a single PC (Dual Pentium-III, 800 MHz, 256MB) at a resolution of* 512^2 *pixels, these scenes render at roughly 3.6, 3.2, and 1.6 frames per second using both processors.*

Occlusion Culling and Logarithmic Complexity Ray tracing enables efficient rendering of complex scenes through its built in occlusion culling as well as its logarithmic complexity in the number of scene primitives.

Flexibility Ray tracing allows us to trace individual or unstructured groups of rays. This provides for efficient computation of just the required information, e.g. for sampling narrow glossy highlights, for filling holes in image-based rendering, and for importance sampling of illumination [42].

Efficient Shading With ray tracing, samples are only shaded after visibility has been determined. Given the trend toward more and more realistic and complex shading, this avoids redundant computations for invisible geometry.

Simpler Shader Programming Programming shaders that create special lighting and appearance effects has been at the core of realistic rendering. While writing shaders (e.g. for the RenderMan standard [3]) is fairly straightforward, adopting these shaders to be used in the pipeline model of rasterization has been very difficult [31]. Since ray tracing is not limited to this pipeline model it can make direct use of shaders [16, 38].

Correctness By default ray tracing computes mostly physically correct reflections, refractions, and shading. In case the correct results are not required or are too costly to compute, ray tracing can easily make use of the same approximations used to generate these effects for rasterization-based approaches, such as reflection or environment maps. This is contrary to rasterization, where approximations are the only option and it is difficult to even come close to realistic effects.

Parallel Scalability Ray tracing is known for being "trivially parallel" as long as a high enough bandwidth to the scene data base is provided. Given the exponential growth of available hardware resources, ray tracing should be better able to utilize it than rasterization, which has been difficult to scale efficiently [12]. However, the initial resources required for a ray tracing engine are higher than those for a rasterization engine.

Coherence is the key to efficient rendering. Due to the low coherence between rays in traditional recursive ray tracing implementations, performance has been rather low. However, as we show in this paper, ray tracing still offers considerable coherence that can be exploited to speed up rendering to interactive levels even on a standard PCs.

It is due to this long list of advantages that we believe ray tracing is an interesting alternative even in the field of interactive 3D graphics. The challenge is to improve the speed of ray tracing to the extent that it can compete with rasterization-based algorithms. It seems that some hardware support will eventually be needed to reach this goal, however in this paper we concentrate on a pure software implementation.

While it is certainly true that ray tracing has a high computational cost, its low performance on todays computers is also strongly affected by the structure of the basic algorithm. It is well-known that the recursive sampling of ray trees neither fits with the pipeline execution model of modern CPUs nor with the use of caching to hide low bandwidth and high latency when accessing main memory [28].

Many research projects have addressed the topic of speeding up ray tracing [14, 15] by various methods such as better acceleration structures, faster intersection algorithms, parallel computation [34, 8], approximate computations [6], etc. This research has resulted in a large number of improvements to the basic algorithm and is documented in the ray tracing literature of the past decades.

In our implementation we build on this previous work and combine it in a novel and optimized way, paying particular attention to caching, pipelining, and SIMD issues to achieve more than an order of magnitude improvement in ray tracing speed compared to other well-known ray tracers such as Rayshade or POV-Ray. As a result we are able achieve interactive frame rates even on standard PCs (see Figure 1).

We start this paper with a description of our high-performance ray tracing engine, discussing general optimization strategies (Section 2), a vectorized intersection algorithm using Intel's SSE SIMD instructions, which is working on packets of rays (Section 3), a simple and efficient BSP traversal algorithms that also works on ray packets (Section 4), and finally a SIMD shading implementation (Section 5). The performance of our ray tracing engine is then evaluated in Section 6.

We then use our ray tracing engine to show that ray tracing is particularly well suited for efficient rendering of complex models (Section 7). We evaluate the performance and scalability of our ray tracer on a number of models ranging from a few ten-thousand up to 8 million triangles. We also compare our software ray tracer against the performance of OpenGL-based rasterization hardware such as a low-cost Nvidia GPU, a SGI Octane workstation, and a SGI ONYX-3 graphics supercomputer. We show that even today the performance of a software ray tracer on a single PC can challenge dedicated rasterization hardware for complex environments. Additionally, we show early results of distributed ray tracing using a few desktop PCs that outperforms the graphics hardware above for complex scenes.

1.1. Previous Work

Even though ray-tracing is as old as 1968 [4, 20, 43, 9], its use for interactive applications is relatively new. Recently Parker et al. [28, 29, 30] demonstrated that interactive framerates could be achieved with a full-featured ray tracer on a large shared memory supercomputer.

Their implementation offers all the usual ray tracing features, including parametric surfaces and volume objects, but is carefully optimized for cache performance and parallel execution in a non-uniform memory access environment. They have proven that ray tracing scales well in the number of processors in a shared memory environment, and that even complex scenes of several hundred thousand primitives could be rendered at almost real-time framerates.

Pharr et al. [32] have shown that coherence can be exploited by completely reordering the ray tracing computation. They were able to render scenes with up to 46 million triangles. Their approach actively manages the scene geometry and rays through priority queues instead of relying on simple caching as we do. However, their system was far from real time.

Hardware implementations of ray tracing are available [40], but are currently limited to accelerating off-line-rendering applications, and do not target interactive frame rates.

2. An Optimized Ray Tracing Implementation

The following four sections describe our implementation of a highly optimized ray tracing engine that outperforms currently available ray tracers by more than an order of magnitude (see Section 6).

We start with an overview of general optimization techniques and how they have been applied to our ray tracing engine, such as reducing code complexity, optimizing cache usage, reducing memory bandwidth, and prefetching data. A similar discussion of optimization issues – although on a higher-level – can also be found in [39]. In the following sections we then discuss the use of SIMD instructions, commonly available on microprocessors today, to efficiently implement the main three components of a ray tracer: ray intersection computations, scene traversal, and shading.

2.1. Code Complexity

A modern processor has several hardware features such as branch prediction, instruction reordering, speculative execution, and other techniques [17, 18] in order to avoid expensive pipeline stalls. However, the success of these hardware approaches is fairly limited and depends to a large degree on the complexity of the input program code. Therefore, we prefer simple code that contains few conditionals, and organize it such that it can execute in tight inner loops. Such code is easier to maintain and can be well optimized by the programmer as well as by the compiler. These optimizations become more and more important as processor pipelines get longer and the gap between processor speed and memory bandwidth and latency opens further.

For traversing the scene, we use an axis-aligned BSP-tree [23]: Its ray traversal algorithm is shorter and simpler compared to octrees, bounding volume hierarchies (BVH), and grids. Even though most easily formulated recursively, it can be transformed to a compact iterative algorithm [21].

We have also chosen to only support triangles as geometric primitives. As a result, the inner loop that performs intersection computations on lists of objects does not have to branch to a different function for each type of primitive. By limiting the code to triangles we lose little flexibility as all surface geometry can be converted. The same approach is being used by most commercial ray tracers (according to information from their developers). While the number of primitives increases, this is more than compensated by the better performance of the ray tracing engine.

For shading we need the flexibility to support arbitrary shaders and thus allow for dynamic loading of shaders. Advanced shading features like multi-texturing, bump-mapping or reflections could be added without changing the core of the ray tracing engine. Flexibility in the shading stage is much less problematic than for intersection computations, as it is only called once for each shading ray, while we perform an average of 40-50 traversal steps and 5-10 intersection tests per ray.

2.2. Caching

Contrary to general opinion, our careful profiling reveals that a ray tracer is not bound by CPU speed, but is in fact bandwidth-bound by access to main memory. Especially shooting rays incoherently (as done in many global illumination algorithms) results in almost random memory accesses and bad cache performance. On current PC systems, bandwidth to main memory is typically up to 8-10 times less than to primary caches. Even more importantly, memory latency increases by similar factors as we go down the memory hierarchy. For example, our triangle test is more than 60% slower if the data has to be fetched from main memory instead of being in the cache. Memory issues become even more important for BSP traversal, where the ratio of computation to memory bandwidth is lower, thus making it more difficult to hide latencies.

Since data transfer between memory and cache is always performed in entire cache lines of 32 bytes, the *effective* cost when accessing memory is not directly related to the number of bytes read, but the number of cache line transfers. As a general result we need to carefully lay out data such that it makes best use of the available caches and design our algorithms so that we can efficiently hide latency by prefetching data, such that it is already available in a cache when it is needed for computations.

We carefully align data to cache lines: This minimizes the additional bandwidth required to load two cache lines simply because some data happen to straddle a cache line boundary. However there are often trade-offs. For instance our triangle data structure requires about 37 bytes. By padding it to 48 bytes we trade-off memory efficiency and cache line alignment.

We keep data together if and only if it is used together: E.g. only data necessary for a triangle intersection test (plane equation, etc.) are stored in our geometry structures, while data that is only necessary for shading (such as vertex colors and normals, shader parameters, etc.) is stored separately. Because we intersect on average several triangles before we find an intersection, we avoid loading data that will not be used.

Given the huge latency of accessing main memory it becomes necessary to load data into the cache before it will be used in computations and not fetch it on demand. This way the memory latency can be completely hidden. Most of todays microprocessors offer instructions to explicitly prefetch data into certain caches. However, in order to use prefetching effectively, algorithms must be simple enough such that it can easily be predicted which data will be needed in the near future.

2.3. Caches and Mailboxes

We also separate read-only, e.g. preprocessed triangle data, from read-write data such as mailboxes [15]. If a mailbox would be stored with the triangle data as in the original proposal, an entire cache line would be marked changed even though only a single integer has actually been modified. This becomes a huge problem in a multi-threaded environment, where by constantly changing mailboxes each processor keeps invalidating cache lines in all other processors.

These problems can easily be resolved by employing a simple hashing mechanism: each thread computing intersections has a small hash table of entries of the form (*triangleId*, *rayId*). A mailbox lookup then simply consists of checking the corresponding hash table entry. Since a good scene traversal algorithm results in only a few triangle intersections per ray, the hash-table can be kept small.

Similarly we do not need an elaborate hash function but simple masking of the triangle id will do. Due to the small amount of memory used and the frequent accesses to its entries, the hash table will stay in the first level caches most of the time. Occasional redundant intersections of objects due to hash collisions are far outweighed by the vastly improved memory-performance. This mechanism is simple enough to be implemented by only a few lines of code.

2.4. Coherence through Packets of Rays

The most important aspect of accelerating ray tracing is to exploit coherence as far as possible. Our main approach is to exploit coherence of primary and shadow rays by traversing, intersecting, and shading *packets of rays* in parallel. Using this approach we can reduce the compute time of the algorithm by using SIMD instructions on multiple rays in parallel, reduce memory bandwidth by requesting data only once per packet, and increase cache utilization at the same time.

2.5. Parallelism through SIMD Extensions

Several modern microprocessor architectures offer SIMD extensions, which allow to execute the same floating point instructions in parallel on several (typically two to four) data values, thereby yielding a significant speedup for floating point intensive applications including 3D graphics. Such extensions also contain instructions for explicit cache management like prefetching. Examples of such extensions are Intel's SSE [19], AMD's 3dNow! [1], and IBM/Motorola's AltiVec [26].

In the following three sections we discuss in more detail how coherent computations with packets of rays and SIMD operations can be used together to speed up the core of a ray tracer, namely triangle intersection, ray traversal and shading.

3. Ray-Triangle Intersection Computation

Optimal ray triangle intersection code has long been an active field of research in computer graphics and has lead to a large variety of algorithms, e.g. Moeller-Trumbore [25], Glassner [15], Badouel [5], Pluecker [13], and many others [24]. Before discussing SIMD implementations, we first describe the triangle test used in our C-code without using assembler or SSE optimizations. This forms the base for our later discussions.

3.1. Optimized Barycentric Coordinate Test

The triangle test used in our implementation is a modification of Badouel's algorithm [5]. It first computes the distance to the point where the ray pierces the plane defined by the triangle, and checks that distance for validity. Only if the distance falls within the interval where the ray is searching for intersections, the actual hit point H is computed and projected into a 2D-plane perpendicular to a coordinate axis.

In order to prevent numerical instabilities, the plane with the largest angle to the triangle normal is chosen for the projection. This results in three cases for the intersection computation. The barycentric coordinates of the hit point H can then be calculated efficiently in 2D. Based on these barycentric coordinates, it can be decided whether the ray pierces the triangle or not.

For the implementation, we need only the properly scaled 2D edge equations for two of the triangle edges, together with the plane equation for the distance calculation, and a tag to mark the projection axis. By preprocessing and proper scaling of these equations, this information can be expressed by 9 floats plus the projection flag. For cache alignment purposes, we pad that data to a total of 48 bytes. An in-depth description of the implementation can be found in [41].

3.2. Evaluating Instruction Level Parallelism

The implementation of the barycentric triangle test requires only few instructions and offers almost no potential for exploiting instruction-level parallelism. Optimizing the algorithms using the Intel SSE extensions results in a speedup of about 20%. It is clear that this speedup is not sufficient for interactive ray tracing.

	Bary. C code	Pluecker SSE	Bary. SSE 4-1	speedup
min	78	77	22	3.5
max	148	123	41	3.7

Table 1: *Cost (in CPU cycles) for the different intersection algorithms. 41 cycles correspond to roughly 20 million intersections per second on a 800 MHz Pentium-III. Measured by using the internal Pentium-III CPU counters.*

As another alternative, we also evaluated a SIMD implementation of the Pluecker triangle test (see [36, 13]). Due to a linear control flow and a somewhat higher computational cost, this triangle test offers much more potential for instruction-level parallelism. The SSE implementation is straightforward and showed good speedups compared to a C implementation of the Pluecker test. However, due to its higher computational cost and particularly its higher bandwidth requirements, the instruction-parallel Pluecker code is effectively not significantly faster than the original barycentric C code (see Table 1).

3.3. SIMD Barycentric Coordinate Test

The speedup achieved with the instruction-parallel implementations is too small to be of significant impact on rendering time. We therefore went back to the already fast barycentric code, and used data parallelism by performing four ray-triangle tests in parallel. However, this means to either intersect one ray with four triangles, or to intersect a packet of four rays with a single triangle. The latter case requires a change to the overall architecture of the ray tracing engine.

Intersecting one ray with four triangles would require us to always have four triangles available for intersection to achieve optimal performance. However, voxels of acceleration data structures should contain only few triangles on average (typically 2-3). More importantly, triangles fall into three different projection cases with separate code each, which lowers the options to use data parallelism even more and precludes the use of this approach.

In contrast it is much simpler to bundle four rays together and intersect them with a single triangle. However, this approach requires us to always have a bundle of four rays available together, which requires a completely new scene traversal algorithm, which we discuss in the next section.

The data-parallel implementation corresponds almost exactly to the original algorithm and is straightforward to implement in SSE. A potential source of overhead is that even though some rays may have terminated early, all four rays have to be intersected with a triangle. Information on which of the four rays is still active is kept in a bit-field, which can be used to mask out invalid rays in a *conditional move* instruction when the hit point information is stored. In practice we obtain almost perfect parallelism for primary rays and to a somewhat lesser degree for shadow rays.

In our implementation, the SSE code for intersecting four rays with a single triangle requires 86-163 CPU cycles. Amortizing this cost over the four rays results in only 22 to 41 cycles per intersection, which corresponds to roughly 20 to 36 million ray-triangle intersection test per second on a 800 MHz Pentium-III CPU. The observed speedup is 3.5-3.7 (see Table 1), and is close to the maximum expected value. Note that this algorithm could also be used to accelerate other ray tracing-based rendering algorithms such as memory coherent ray tracing [32].

4. BSP Traversal

Even before accelerating the triangle test, traversal of the acceleration structure was typically 2-3 times as costly as ray-triangle intersection. As the SSE triangle intersection code reduces the intersection cost by more than a factor of three, traversal is the limiting factor in our ray tracing engine. Since our SSE intersection procedure requires us to always have four rays available this suggests a data parallel traversal of a bundle of at least four rays.

A wide variety of ray tracing acceleration schemes have been developed, such as octrees, general BSP-trees, axis-aligned BSP-trees, regular and hierarchical grids, ray classification, bounding volume hierarchies, and even hybrids of several of these methods. See [37, 15] for an overview and further references. Our main reason for using a BSP tree in our implementation is the simplicity of the traversal code: Traversing a node is based on only two binary decisions, one for each child, which can efficiently be done for several rays in parallel using SSE. If any ray traverses a child, all rays will traverse it in parallel.

This is in contrast to algorithms like octrees or hierarchical grids, where each of the rays might take a different decision of which voxel to traverse next. Keeping track of these states is non-trivial and was judged to be too complicated to be implemented efficiently. Bounding Volume Hierarchies have a traversal algorithm that comes close in simplicity to BSP trees. However, BVHs do not implicitly order their child nodes, which is another reason for our choice of axis-aligned BSP trees.

4.1. Traversal Algorithm

Before describing our algorithm for traversal of four rays in parallel, we first take a look at the traversal of a single ray, as presented in [23]: In each traversal step, we maintain the *current ray segment* $[near, far]$, which is the part of the ray that actually intersects the current voxel. This ray segment is first initialized to $[0, \infty)$, then clipped to the bounding box of the scene, and is updated incrementally during traversal. For

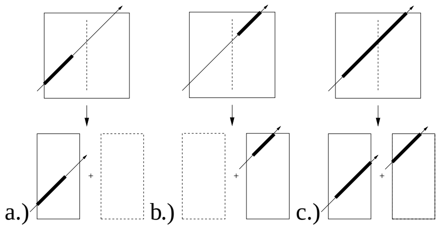


Figure 2: *The three traversal cases in a BSP tree: A ray segment is completely in front of the splitting plane (a), completely behind it (b), or intersects both sides (c).*

each traversed node, we calculate the distance d to the splitting plane defined by that node, and compare that distance to the current ray segment.

If the ray segment lies completely on one side of the splitting plane ($far < d$ or $d < near$), we immediately proceed to the corresponding child voxel. Otherwise, we traverse both children in turn, with the ray segment clipped to the respective child voxel. This avoids problems for rays that are forced by another ray from the same packet to traverse a voxel they would not otherwise traverse. For those rays the ray segment will be the empty interval.

The algorithm for tracing four different rays is essentially the same: For each node, we use SSE operations to compute the four distances to the splitting plane and to compare these to the four respective ray segments, all in parallel. If all rays require traversal of the same child, we immediately proceed to that child without having to change the ray segments. Otherwise, we traverse both children, with each ray segments updated to $[near, min(far, d)]$ for the closer, respectively $[max(near, d), far]$ for the distant child.

When traversing several rays at the same time, the order of traversal can be ambiguous, since different rays might require a different traversal order. Since the order is based only on the sign of the respective direction, this can happen only if the signs of the four direction vectors do not match, which is a rare case if we assume the rays to be coherent. Additionally, it can be shown that no two rays with the same origin can require different traversal orders. This completely resolves this problem for pinhole cameras and point light sources. If rays are allowed to start in different locations, a straightforward solution is to only allow rays with matching direction signs in the same packet, and tracing the few special cases separately.

4.2. Memory Layout for Better Caching

As mentioned above, the ratio of computation to the amount of accessed memory is very low for scene traversal. This requires us to carefully design the data structure for efficient caching and prefetching. Memory bandwidths and cache utilization have been improved with a compact, unified node layout. For inner nodes in a BSP node we have to store

- Pointers to the two child nodes. By implicitly storing the right child immediately after the left child, this can be represented with a single pointer.
- A flag on whether it is a leaf node or the type of inner node (splitting axis). This requires two bits.
- The split coordinate, which is the coordinate where the plane intersects its perpendicular axis.

For best performance we use one float for the split coordinate and squeeze the two flag bits into the 2 low order bits of the pointer, which results in 8 bytes per node or 4 nodes per cache line. By aligning the two children of a node on half a cache line we make sure that both children are fetched together since they are likely to be traversed together. The additional computations to extract these two bits from the pointer are negligible as the traversal code performs very few computations anyway compared to the amount of memory it accesses.

For leaf nodes the pointer addresses the list of objects, and the other fields can be used to store the number of objects in the list. Using the same pointer for both node types allows us to reduce memory latencies and pipeline stalls by prefetching, as the next data (either a node or the list of triangles) can be prefetched before even processing the current node. Even though prefetching can only be used with SSE cache control operations, the reduced bandwidth and improved cache utilization also affect the pure C implementation.

4.3. Traversal Overhead

Traversing packets of rays through the acceleration structure generates some overhead: Even if only a single ray requires traversal of a subtree or intersection with a triangle, the operation is always performed on all four rays. Our experiments have shown that this overhead is relatively small as long as the rays are coherent. Table 2 shows the overhead in additional BSP node traversals for different packet sizes.

As can be seen from this experiment, overhead is in the order of a few percent for 2×2 packets of rays, but goes up for larger packets. On the other hand, increasing screen resolution also increases coherence between primary rays.

Most important is the fact that the effective memory bandwidth has been reduced essentially by a factor of four through the new SIMD traversal and intersection algorithms as triangles and BSP nodes need not be loaded separately for each ray. This effect is particularly important for ray traversal as the computation to bandwidth ratio in relatively low.

Of course one could operate on even larger packets of rays to enhance the effect. However, our results show that we are running almost completely within the processor caches even with only four rays. We have therefore chosen not to use more rays per ray packet, as it would additionally increase the

	2×2	4×4	8×8	256^2	1024^2
Shirley 6	1.4%	4.4%	11.8%	5.8%	1.4%
MGF office	2.6%	8.2%	21.6%	10.4%	2.6%
MGF conf.	3.2%	10.6%	28.2%	12.2%	3.2%

Table 2: *Overhead (measured in number of additional node traversals) of tracing entire packets of rays at an image resolution of* 1024^2 *in the first three columns: As expected, overhead increases with scene complexity (800, 34k, and 280k triangles, respectively) and packet size, but is tolerable for small packet sizes. The two columns on the right show the overhead for* 2×2 *packets at different screen resolutions.*

overhead due to redundant traversal and intersection computations.

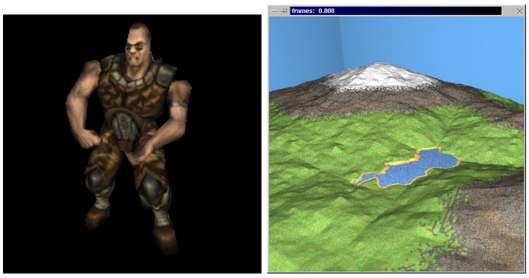
5. SIMD Phong Shading

As data-parallel intersection and traversal has shown to be very effective, the same benefits should also apply for shading computations. Similar to the traversal and intersection code, we can shade four rays in parallel. Since the four hit points may have different materials, data has to be rearranged. Although this setup results in some overhead, the following shading operations can be very efficiently implemented in SSE, yielding almost perfect utilization of the SSE units.

Light sources are processed in turn: For each light source, we first determine its visibility by shooting shadow rays using the traversal and intersection algorithms described above. If a light source is visible from at least one pixel, its contribution to all four hit points is computed in parallel. This contribution is then added to the visible hit points only, by masking out shadowed points. This procedure computes information that may get discarded later and thus has some overhead. However, this happens only in the case that the visibility of a light source is different between the set of hit points. For a coherent set of rays this happens but rarely.

Special care has to be taken when shooting the shadow rays. Since shadow rays typically make up the largest fraction of all rays in a ray tracer, shooting them with the fast SSE traversal code is desirable. This, however, is only efficient as long as the rays are coherent, which is not automatically true for shadow rays, since all shadow rays from a single hit point typically go in very different directions. However, coherent primary rays are also likely to hit similar locations in the scene, yielding coherent shadow rays if connected to one of the light sources. In the worst case (i.e. if the rays are incoherent), performance degrades to the performance achieved when tracing each ray on its own.

Implementing the shading in SSE operations gives a speedup of 2 to 2.5 as compared to the C implementation on top of the speedup obtained by the general optimizations discussed above. Texturing has shown to be relatively cheap. Even an unoptimized version has reduced frame rates by less than 10 percent, as can be seen in Figure 3. This cost could probably be reduced even more due to a large potential for prefetching and parallel computations that we currently do not take advantage of. As shading typically makes up for less than 10 percent of total rendering time, more complex shading operations could easily be added without a major performance hit.



scene	flat	textured
Quake	4.22fps	3.85fps
Terrain	1.05fps	0.96fps

Figure 3: *Texturing comes rather cheaply: Even for a complex scene with incoherent texture access, texturing only slightly reduces the framerate, rendering even a scene of one million triangles interactively. Image resolution is* 512^2.

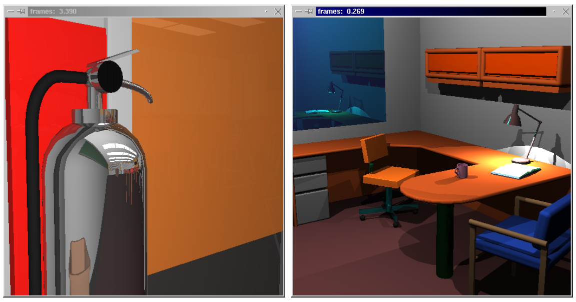


Figure 4: *Frames from the accompanying video showing the entire conference room being reflected in the fire extinguisher (left). Performance only drops slightly even when a large fraction of the scene is reflective. The office has been rendered with many reflective materials (window, lamp, mug, and others) and three point light sources.*

6. Performance of the Ray Tracing Engine

After all the parts of a full ray tracer are now together we can evaluate the overall performance of our system (RTRT). We start by evaluating the performance for primary rays as this will allow us to compare the ray tracing algorithm directly to rasterization-based algorithms that do not directly support shadows, reflection, and refraction effects.

	Tris	Rayshade	POV-Ray	RTRT
MGF office	40k	29	22.9	2.1
MGF conf.	256k	36.1	29.6	2.3
MGF theater	680k	56.0	57.2	3.6
Library	907k	72.1	50.5	3.4
Soda Floor 5	2.5m	OOM	OOM	2.9
Soda Hall	8m	OOM	OOM	4.5

Table 3: *Performance comparison of our ray tracer against Rayshade and POV-Ray. All rendering times are given in microseconds per primary ray including all rendering operations for the same view of each scene at a resolution of* 512^2 *(OOM = out of memory).*

On a single 800 MHz Pentium-III, we achieve a rendering performance from about 200,000 to almost 1.5 million primary rays per second for the SSE version of our algorithm. If we compare the performance of the this version to our optimized C code we see an overall speedup between 1.8 and 2.5. This is a bit less than that for ray-triangle intersection but is due to the worse ratio of memory accesses to computations in the traversal stage and the strict sequential traversal order that does not allow for better prefetching.

6.1. Comparison to Other Ray Tracers

In order to evaluate the performance of our optimized ray tracing engine we tested it against a number of freely available ray tracers, including POV-Ray [27] and Rayshade [22]. We have chosen the set of test scenes so that they span a wide range regarding the number of triangle and the overall occlusion within the scene. Unfortunately, both other systems failed to render some of the more complex test scenes due to memory limitations even with 1GB of main memory.

The numbers of the performance comparison for the case of primary rays are given in Table 3. It demonstrates clearly that our new ray tracing implementation improves performance consistently by a factor between 11 and 15 (!) compared to both POV-Ray and Rayshade. The numbers show that paying careful attention to caching and coherence issues can have a tremendous effect on the overall performance, even for such well-analyzed algorithms as ray tracing.

The numbers also seem to indicate that the performance gap widens slightly for more complex scenes, which indicate that the caching effect get even more pronounced in these cases. Our implementation was tested on a machine with only 256 MB of main memory, while we had to use a machine with 1GB of memory for the other ray tracers.

Some comments on these results are necessary: Rayshade is using a uniform grid as an acceleration structure and we had to determine the best grid size for each scene by trial and error. No such manual optimizations was necessary for POV-Ray and our implementation. Also both other ray tracers could not deal well with large scenes and reported out of memory errors for scenes beyond 1 million triangles.

Of course POV-Ray and Rayshade offer considerably more features than our ray tracer engine. However, most of these features are shading related and could easily be added to our engine using dynamically loadable shaders. This would have little effect on the performance of the core engine unless those features are used. The other ray tracers are also not limited to only use triangles to represent objects. However, we believe this is actually an advantage for us and is partly the reason for the good performance. Finally, these other ray tracers are not written with highest optimization in mind but are more targeted towards a large feature set. We believe that we will be able to show in the future that these two goals do not contradict each other.

6.2. Reflection and Shadow Rays

Of course a ray tracing engine would not be complete if it could not handle shadows, reflection, and refraction. These effects also challenge our overall approach as ray coherence can be considerably less for shadow or even reflection rays. Although the handling of secondary rays is not yet fully optimized in our implementation we were surprised by the good performance we observed even for extreme cases of reflectivity.

The accompanying video shows a walkthrough of the MGF conference scene, where most of the material has at least a slight contribution by reflection rays. Even the doors, wall panels with fixtures, and metal frames of the seats generate reflection rays, often resulting in multiple reflections as clearly visible when zooming towards the fire extinguisher, which reflects the entire scene (see Figure 4).

Sphere-like objects such as the fire extinguisher are potential hot spots in scenes like these as they can trigger large numbers of reflection rays that sample the entire visible environment and are likely to have adverse effect on caching. It is interesting to see that the effect is hardly noticeable as long as these objects cover only moderate parts of the image. In this case only a few rays are reflected almost randomly into the environment. Those rays potentially sample the entire scene but our acceleration structure successfully limits the data being accessed to only a few BSP-cells and triangles along the paths of those few rays. As a result the impact on performance remains low.

Performance degrades significantly only if zooming in on a reflective object such that it fills the field of view. In this case almost all visible geometry will actually be sampled and caching will no longer be effective for large scenes. However, this is an unavoidable consequence of dealing with a working set much larger than the cache (our largest scenes occupy close to 2GB of memory but render fine with 256 MB of main memory). In those cases it seems unavoidable to use approximations such as a reflection map.

An example image rendered with reflections and shadows can also be seen on the right in Figure 4. Many object are reflective and generate reflections rays. Also three shadow rays are sent for each intersection. The performance is mainly influenced by the number of shadow rays.

7. Ray Tracing of Complex Scenes

There is a strong trend towards more and more complex scenes that need to be rendered. Many disciplines need to visualize large assemblies such as whole cars, ships, airplanes, power plants, and similar structures. It is often very time consuming to preprocess the data in order to reduce the complexity to a level manageable by current rendering technology. These preprocessing steps are non-trivial and often require considerable user interaction [2]. We expect this trend to more complex models to increase as computing and memory resources make large assemblies easier to store and handle.

Ray tracing still needs preprocessing for those data sets, but this preprocessing is limited to pure spatial ordering and does not involve any complex computations on the geometry itself, as would for instance geometric simplification. Additionally ray tracing has occlusion culling built into the algorithms and does not require complex precomputation of data, such as the potentially visible set (PVS) or similar data structures [2]. Consequently, ray tracing is especially well suited for large and complex models.

7.1. Comparison with Rasterization Hardware

We started this paper with the claim made by researchers in the past that ray tracing would eventually become faster than rasterization hardware. However, it was unclear at which point that crossover would happen, if at all. With the ray tracing system described above we are now in a position to answer this questions: We have already reached the crossover point and can now even outperform rasterization hardware with a software ray tracer – at least for complex scenes and moderate screen resolutions.

For this demonstration we compared the performance of our ray tracing implementation with the rendering performance of the OpenGL-based hardware. In order to get the highest possible performance on this hardware we chose to render the scenes with SGI Performer [35], which is well-known for its highly optimized rendering engine that takes advantage of most available hardware resources including multiprocessing on our multiprocessor machines. We have used the default parameters of Performer when importing the scene data via the NFF format and while rendering. The 32-bit version of Performer that we used was unable to handle the largest scene (Soda Hall) because it ran out of memory. We used simple constant shading in all cases.

The rasterization measurements of our experiments were conducted on three different machines in order to get a representative sample of todays hardware performance. On our PCs (dual Pentium-III, 800 MHz, 256 MB) we used a Nvidia GeForce II GTS graphics card running under Linux. Additionally, we used an SGI Octane (300 MHz R12k, 4 GB) with the recently introduced V8 graphics subsystem as well as a brand new SGI Onyx-3 graphics supercomputer (8x 400 MHz R12k, 8 GB) with InfiniteReality3 graphics and four raster managers. The results are are shown in Table 4.

Scene	Tris	Octane	Onyx	PC	RTRT
MGF office	40k	>24	> 36	12.7	1.8
MGF conf.	256k	>5	> 10	5.4	1.6
MGF theater	680k	0.4	6-12	1.5	1.1
Library	907k	1.5	4	1.6	1.1
Soda Floor	2.5m	0.5	1.5	0.6	1.5
Soda Hall	8m	OOM	OOM	OOM	0.8

Table 4: *OpenGL rendering performance in frames per second with SGI Performer on three different graphics hardware platforms compared with our software ray tracer at a resolution of* 512^2 *pixels on a dual processor PC. The ray tracer uses only a single processor, while SGI Performer actually uses all available.*

The results show clearly that the software ray tracer already outperform the best hardware rasterization engines for scenes with a complexity of roughly 1 million triangles and more and is already competitive for scenes of about half the size. The ray tracing numbers can be scaled easily by adding more processors — just enabling the second CPU on our machines doubles our RTRT numbers given in Table 4.

In order to visualize the scaling behavior of rasterization and ray tracing-based renderers, we used the large terrain scene shown in Figure 3 and subsampled the geometry. The results are shown in Figure 5. Even though SGI Performer uses a number of techniques to reduce rendering times, we see the typical linear scaling of rasterization. Even occlusion culling would not help in this kind of scene. Ray tracing benefits from the fact that each ray visits roughly a constant number of triangles but needs to traverse a BSP tree with logarithmically increasing depth. Ray tracing also subsamples the geometry for the higher resolution terrain as the number of pixels is less than the number of triangles.

For scenes with low complexity, rasterization hardware, benefits from the large initial cost per ray for traversal and intersection required by a ray tracer. However, we believe that for these cases there is still room for performance improvements. The large initial cost per ray also favors rasterization for higher image resolutions. However, this effect is linear in the number of pixels and can be compensated by adding more processors, for instance in form of a distributed ray tracer.

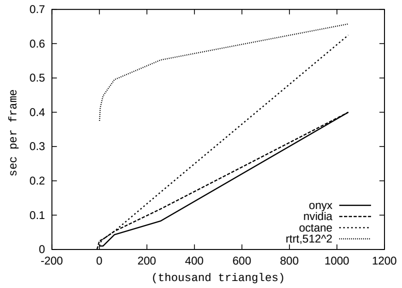


Figure 5: *This figure shows the logarithmic scaling of ray tracing with input complexity. We also show the linear scaling of different rasterization hardware. The scenes were obtained by subsampling the high resolution terrain from Figure 3 with more than one million triangles. Other scenes show even better results due to more occlusion.*

Scene	Tris	framerate
Library	907k	7.7 fps
MGF Theater	680k	7.3 fps
MGF office	40k	2.4 fps(*)

Table 5: *Interactive rendering performance of a distributed ray tracing implementation based on the optimized ray tracing core. All images are rendered at a resolution of* 512^2 *on five PCs (Pentium-III, 800 MHz machines). (*) We render primary rays only, except for the office with contains reflections and shadows from three point light sources.*

7.2. Distributed Ray Tracing

So far we have concentrated on simple ray tracing with primary rays only that can be compared to rasterization-based hardware. As we add special ray tracing effects such as shadows, reflections, or even global illumination, we are confronted with the need to trace an increasing number of rays. However, ray tracing is well-known for its almost perfect scalability in a distributed environment.

Our small distributed ray tracing system uses the typical master/slave approach with socket-based communication. A simple load balancing scheme based on a work queue on the master keeps the client processors busy almost all the time.

Table 5 show the preliminary performance of our engine when connecting to five dual Pentium-III desktop PCs (800 and 866 MHz, 256 to 768 MB of memory). In particular we get a decent rendering speed with the office scene even with reflections and shadows from three point light sources.

The machines are connected with a switched 100-Mbit Ethernet and rendering speed is mainly limited by the bandwidth to the computer used for display. The uncompressed pixel stream easily saturates the available links and we are currently experimenting with higher bandwidth network components to eliminate or at least reduce the current bottleneck. However, the bandwidth requirements are only dependent on the size of the output image and are not affected as we proceed to more costly rendering operations, such as more light sources, more expensive shaders, super sampling, and global illumination.

7.3. Description of Accompanying Video

This submission also contains a video. Each video clip is recorded live from the screen of one of our PCs. Ray tracing is computed at a resolution of 640 by 480 pixels. The ray tracing computations are performed on five PCs, which send the computed pixels to the display host across a switched 100 Mbit Ethernet. Higher resolutions could not be rendered due to network bandwidth restrictions. Note that some synchronization artifacts are visible, which are due to the early stage of development of this distributed rendering system. The current framerate is always displayed in the title area of the display window.

The video starts with a simple model of a Quake monster rendered with and without textures to show that texturing has hardly any effect on the rendering performance. In particular texturing performance is independent of scene complexity as each pixel is only shaded once. All scenes are rendered with only a single primary ray for each pixel unless otherwise mentioned.

The next clip shows the rendering of a textured terrain scene containing 1 million triangles. The camera starts zoomed in on a few triangles with the pointer outlining one of them. We then zoom out until almost all triangles are visible. The rendering performance changes only slightly in the process.

We then show a walkthrough of the MGF theatre scene containing 200k triangles. The illumination has been precomputed with stochastic radiosity using the RenderPark system by Bekaert and Suykens [7].

The next clip actually shows two different models of the fully furnished, seven floor Soda Hall building from Berkeley. The initial view is the non-illuminated model consisting of 1.5 million triangles. As we enter the building we switch to a illuminated model subdivided by the radiosity computation into roughly 8 million triangles. Note that the model contains coplanar triangles of which some are black. This creates artifacts that are unrelated to the rendering algorithm.

The next two clips show the MGF office and conference scene with reflection by almost all materials. Please note the reflection in the doors, rails along the walls, and their fixtures. Multiple reflection can for instance be seen in the lower part of the desk lamp in the office clip. Performance stays fairly high except when displaying the reflective lamp in full screen.

The final clip shows the office scene again, this time with reflections and shadows from three point light sources. The performance drops according to the additional numbers of rays that need to be traced.

8. Conclusions

The computational cost of ray tracing is known to be logarithmic in terms of the number of triangles. In contrast, the rendering cost using a rasterization pipeline appears linear in the number of triangles even with optimizations such as view frustum culling. Therefore, a break-even point in model complexity was expected, above which ray tracing would be preferred over rasterization hardware. The main goal of this paper was to investigate where this break-even point is located, comparing an efficient software implementation of the ray tracing algorithm on a commodity PC with state-of-the-art rasterization hardware.

Our ray tracing implementation exploits a number of novel techniques, described above, that make it more than an order of magnitude faster than other ray tracers we could compare with:

- Careful attention is paid to exploiting coherence in the ray tracing algorithms in order to achieve good caching behavior such that the algorithms can essentially run within the first and second level data caches of the processors. Our experiments indicate that this results in a speed-up of roughly half an order of magnitude.
- Several strategies have been investigated for utilizing SIMD instructions found on commodity processors. In our implementation, we used Intel's SSE extensions on a Pentium-III processor. A significant speed-up can only be obtained by re-ordering the ray tracing algorithm so that rays are traced in packets of four coherent rays. This reduces the memory bandwidth by a factor of four and gives an additional speed-up factor of about 2.

We have compared rendering speeds of this ray tracing implementation on a 800MHz Pentium-III based Linux PC with those obtained using a high-end commercial visualization package (SGI Performer) on three different graphics accelerators (NVidia GeForce II GTS, SGI Octane with V8 graphics board, SGI Onyx-3 with InfiniteReality 3 graphics). Our experiments on a variety of models (see Table 4), suggest that the break-even point is reached for models of the order of magnitude of 1 million triangles at a screen resolution of 512 × 512. For larger models, ray tracing wins.

Both ray tracing and the Z-buffer algorithm have a cost component linear in the number of screen pixels as well however. In hardware implementations of the rasterization pipeline, this cost component is almost negligible. The cost of ray tracing is directly proportional to the number of pixels. The break-even point therefore shifts towards more complex models, proportional to screen resolution. Moreover, more sophisticated occlusion culling algorithms currently being developed may reduce the cost of a rasterization pipeline to sub-linear, similar to ray-tracing.

On the other hand, by paying careful attention to caching issues, ray tracing is not limited by memory bandwidth but runs within the processor caches and performance scales linearly with the image resolution, the speed of the processor, and with the number of processors.

We tested our implementation also on 4-CPU system with no performance degradation and estimate a gradual bottleneck due to limited memory bandwidth only at around 6-8 CPUs with current PC technology. The memory bandwidth of current PC systems is rather poor and measures at about 200 MB per second to main memory. A hardware implementation would allow for memory bandwidth in the order of several GB per second, enough to keep a large number of parallel ray tracing units busy. This would allow for real-time visualization for a very wide range of models. We are actively investigating suitable hardware architectures for this approach.

We conclude that, unlike widely believed, the ray tracing algorithm is a viable alternative for a Z-buffer based rasterization pipeline especially when it comes to visualizing large polygonal datasets.

9. Acknowledgements

We would like to thank the Max-Planck-Institut für Informatik in Saarbrücken, Germany, for giving us access to their equipment and rasterization graphics hardware, and to Jörg Fischer for his help in creating the statistics.

Finally, we are especially indebted to Philippe Beakert for supplying us with nice models, and for his active support in generating the comparison to rasterization hardware.

References

1. Advanced Micro Devices. *Inside 3DNow![tm] Technology.* http://www.amd.com/products/cpg/k623d/inside3d.html.

2. 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 *1999 ACM Symposium on Interactive 3D Graphics*, pages 199–206, Atlanta, USA, April 1999.

3. Anthony Apodaka and Larry Gritz. *Advanced RenderMan.* Morgan Kaufmann, 2000.

4. A. Appel. Some techniques for shading machine renderings of solids. *SJCC*, pages 27–45, 1968.

5. Didier Badouel. An efficient ray polygon intersection. In David Kirk, editor, *Graphics Gems I*. Academic Press, 1992.

6. Kavita Bala, Julie Dorsey, and Seth Teller. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Transactions on Graphics*, 18(3), August 1999.

7. Philippe Bekaert and Frank Suykens. RenderPark – a photorealistic rendering tool. http://www.cs.kuleuven.ac.be/-cwis/research/graphics/RENDERPARK.

8. Alan Chalmers and Erik Reinhard. Parallel and distributed photo-realistic rendering. In *Course notes for SIGGRAPH 98*, pages 425–432. ACM SIGGRAPH, Orlando, USA, July 1998.

9. Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics*, 18(3):137–145, 1984.

10. DARPA. Data intensive systems. http://www.darpa.mil/ito/research/dis/index.html, 1997.

11. G.D. Durgin, N. Patwari, and T.S. Rappaport. An advanced 3D ray launching method for wireless propagation prediction. In *IEEE 47th Vehicular Technology Conference*, volume 2, May 1997.

12. Matthew Eldridge, Homan Igehy, and Pat Hanrahan. Pomegranate: A fully scalable graphics architecture. *Computer Graphics*, pages 443–454, July 2000.

13. Jeff Erickson. Pluecker coordinates. *Ray Tracing News*, 1997. http://www.acm.org/tog/resources/RTNews/html/rtnv10n3.html#art11.

14. Foley, van Dam, Feiner, and Hughes. *Computer Graphics – Principles and Practice, 2nd edition in C.* Addison Wesley, 1997.

15. Andrew Glassner. *An Introduction to Raytracing.* Academic Press, 1989.

16. Larry Gritz and James K. Hahn. BMRT: A global illumination implementation of the renderman standard. *Journal of Graphics Tools*, 1(3):29–47, 1996.

17. John L Hennessy and David A Patterson. *Computer Architecture – A Quantitative Approach, 2nd edition.* Morgan Kaufmann, 1996.

18. Intel Corp. *Intel Computer Based Tutorial.* http://developer.intel.com/vtune/cbts/cbts.htm.

19. Intel Corp. *Intel Pentium III Streaming SIMD Extensions.* http://developer.intel.com/vtune/cbts/simd.htm.

20. Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, August 1986.

21. A. Keller. *Quasi-Monte Carlo Methods for Realistic Image Synthesis.* PhD thesis, University of Kaiserslautern, 1998.

22. Craig Kolb. Rayshade home-page. http://graphics.stanford.edu/˜cek/rayshade/rayshade.html.

23. K.Sung and P.Shirley. Ray tracing with the BSP tree. *Graphics Gems III*, pages 271—274, 1992.

24. Tomas Moeller. Practical analysis of optimized ray-triangle intersection. http://www.ce.chalmers.se/staff/tomasm/raytri/.

25. Tomas Moeller and Ben Trumbore. Fast, minimum storage ray triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

26. Motorola Inc. *AltiVec Technology Facts.* Available at http://www.motorola.com/AltiVec/facts.html.

27. Persistence of Vision Development Team. Pov-ray home-page. http://www.povray.org/.

28. Steven Parker, William Martin, Peter Pike Sloan, Peter Shirley, Brian Smits, and Chuck Hansen. Interactive ray tracing. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 119–126, April 1999.

29. Steven Parker, Michael Parker, Yaren Livnat, Peter Pike Sloan, Chuck Hansen, and Peter Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Computer Graphics and Visualization*, 5(3):238–250, July-September 1999.

30. Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive ray tracing for isosurface rendering. In *IEEE Visualization ’98*, pages 233–238, October 1998.

31. Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. *Interactive Multi-Pass Programmable Shading.* ACM Siggraph, New Orleans, USA, July 2000.

32. Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Computer Graphics*, 31:101–108, August 1997.

33. Philippe Plasi, Betrant Le Saëc, and Gèrad Vignoles. Application of rendering techniques to monte-carlo physical simulation of gas diffusion. In Julie Dorsey and Philipp Slusallek, editors, *Rendering Techniques ’97*, pages 297–308. Springer, 1997.

34. David J. Plunkett and Michael J. Bailey. The Vectorization of a Ray Tracing Algorithm for Improved Execution Speed. *IEEE Computer Graphics and Applications*, 6(8):52–60, August 1985.

35. John Rohlf and James Helman. IRIS Performer: A high performance multiprocessing toolkit for real-time 3D graphics. *Computer Graphics*, 28(Annual Conference Series):381–394, July 1994.

36. Ken Shoemake. Pluecker coordinate tutorial. *Ray Tracing News*, 1998. http://www.acm.org/tog/resources/RTNews/html/rtnv11n1.html#art3.

37. George Simiakakis. *Accelerating Ray Tracing with Directional Subdivision and Parallel Processing.* PhD thesis, University of East Anglia, 1995.

38. Philipp Slusallek, Thomas Pflaum, and Hans-Peter Seidel. Using procedural RenderMan shaders for global illumination. In *Computer Graphics Forum (Proc. of EUROGRAPHICS ’95*, pages 311–324, 1995.

39. Brian Smits. Efficiency issues for ray tracing. *Journal of Graphics Tools*, 3(2):1–14, 1998.

40. Advanced Rendering Technologies. The AR250 - a new architecture for ray traced rendering. In *Proceedings of the Eurographics/SIGGRAPH workshop on Graphics hardware - Hot Topics Session*, pages 39–42, 1999.

41. Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive Raytracing on Notebooks. Technical report, Computer Graphics Group, Saarland University, 2000.

42. Greg Ward. Adaptive shadow testing for ray tracing. In *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering*, pages 11–20. Springer Verlag, New York, 1994.

43. T. Whitted. An improved illumination model for shaded display. *CACM*, 23(6):343–349, June 1980.