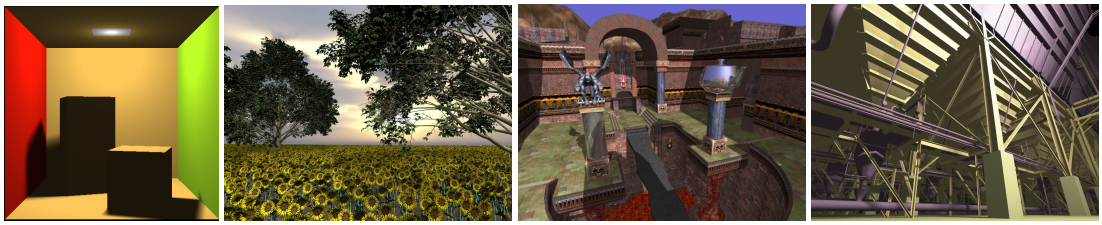
Realtime Ray Tracing and its use for Interactive Global Illumination



Ingo Wald† Timothy J.Purcell‡ Jörg Schmittler† Carsten Benthin† Philipp Slusallek†

{wald,schmittler,benthin,slusallek}@graphics.cs.uni-sb.de
†Computer Graphics Group
Saarland University

tpurcell@graphics.stanford.edu
‡Computer Graphics Group
Stanford University

Abstract

Research on realtime ray tracing has recently made tremendous advances. Algorithmic improvements together with optimized software implementations already allow for interactive frame rates even on a single desktop PC. Furthermore, recent research has demonstrated several options for realizing realtime ray tracing on different hardware platforms, e.g. via streaming computation on modern graphics processors (GPUs) or via the use of dedicated ray tracing chips. Together, these developments indicate that realtime ray tracing might indeed become a reality and widely available in the near future.

As most of todays global illumination algorithms heavily rely on ray tracing, this availability of fast ray tracing technology creates the potential to finally compute even global illumination – the physically correct simulation of light transport – at interactive rates.

In this STAR, we will first cover the different research activities for realizing realtime ray tracing on different hardware architectures – ranging from shared memory systems, over PC clusters, programmable GPUs, to custom ray tracing hardware. Based on this overview, we discuss some of the advanced issues, such as support for dynamic scenes and designs for a suitable ray tracing API. The third part of this STAR then builds on top of these techniques by presenting algorithms for interactive global illumination in complex and dynamic scenes that may contain large numbers of light sources. We believe that the improved quality and the increased realism that global illumination adds to interactive environments makes it a potential "killer application" for future 3D graphics.

1. Introduction

The ray tracing algorithm is well-known for its ability to generate high-quality images, making it the de-facto standard for high-quality rendering and for almost all lighting simulation systems. On the other hand, ray tracing is well-known for its long rendering times, often taking minutes to hours for a single frame. Therefore, ray tracing is usually only applied in an offline context.

Recently, however, algorithmic, implementation, and hardware improvements have made it possible to speed up ray tracing for interactive use, at least for moderate resolutions and frame rates. These recent developments suggest that realtime ray tracing might indeed be ubiquitously available in the near future.

The availability of realtime ray tracing would offer a number of interesting benefits for computer graphics in general: Among others, ray tracing offers physical correctness, ease-of-use for users and developers, efficient handling of complex models, and support for advanced algorithms like global illumination (Figure 1).

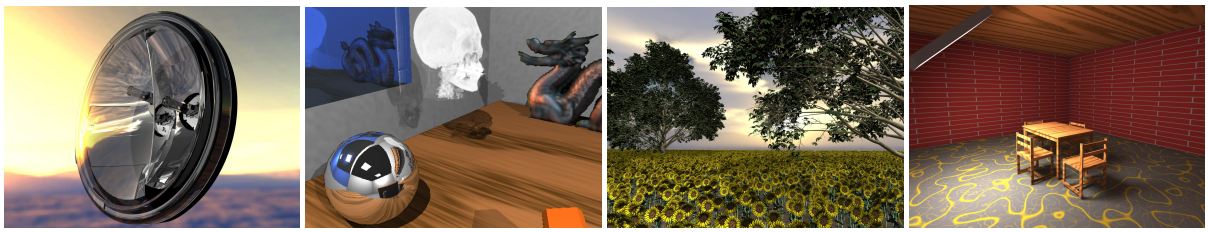


Figure 1: *Several application demonstrating the benefits of realtime ray tracing: a.) Physical correctness: accurate simulation of reflection and refraction in a car headlight. b.) Ease of use: easily combining different shaders. Shown is an office scene with textures, procedural shaders, shadows, reflections, and with volume and lightfield objects. c.) Massively complex geometry: the Sunflowers scene consists of roughly one* billion *triangles and is rendered with shadows and semi-transparent leaves. d.) Support for advances lighting algorithms: interactively simulating global illumination effects. All these applications run interactively today and will be discussed in more detail later in this report.*

Image quality and physical correctness: Ray tracing closely models the physical process of light propagation (in reverse) and thus 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 Figure 1a).

Ease of use: 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 take care of correctly rendering the resulting combinations of shading effects (see Figure 1b). This feature is essential for robust industrial applications.

Efficient handling of complex geometries: Ray tracing efficiently supports huge models with billions of polygons showing a logarithmic time complexity with respect to scene size. Additionally, ray tracing features inherent pixel-accurate occlusion culling and demand-driven and output-sensitive processing that computes only visible results. For example, shadows and reflections are only calculated for actually visible points. Taken together, this allows ray tracing to be highly efficient even for massively complex environments (see Figure 1c).

Support for advanced lighting algorithms: The ability to quickly trace millions of rays from arbitrary positions into arbitrary directions is a prerequisite for many advanced rendering effects including interactive global illumination (see Figure 1d). The flexibility of tracing such arbitrary rays is a unique advantage of ray tracing.

For a more in-depth discussion of these advantages and disadvantages of ray tracing, also see the 2001 STAR on interactive ray tracing [92]. Figure 1 shows some example applications that run interactively on todays realtime ray tracing engines. These applications will be discussed in more detail below.

1.1. Outline

This report is organized into three parts. Part 1 first discusses the different approaches to realizing realtime ray tracing: Section 2 summarizes purely software-based approaches. As this topic has already been addressed in a previous STAR on interactive ray tracing [92], we will concentrate on the most interesting and most recent improvements since then. Section 3 reports on the use of programmable graphics hardware for ray tracing. Finally, Section 4 discusses the option of designing specialized hardware for realtime ray tracing.

Part 2 discusses advanced topics of realtime ray tracing, such as support for dynamic scenes (Section 6), issues for future realtime ray tracing APIs (Section 7), and potential and implications for future applications (Sections 8 and 9).

Finally, Part 3 covers the question how realtime ray tracing can best be used for achieving interactive global illumination. We briefly summarize approaches that are not based on realtime ray tracing, but focus on the specific impact of realtime ray tracing on interactive global illumination.

PART ONE

Realizing Realtime Ray Tracing

Today, there are three different hardware platforms on which realtime ray tracing can be realized:

CPUs run highly optimized and parallelized software implementations of the classical ray tracing algorithm.

Programmable GPUs are used as massively parallel, powerful streaming processors, that run a specialized software ray tracer.

Special-purpose hardware architectures are explicitly designed VLSI chips to achieve maximum performance for realtime ray tracing.

Software-based systems essentially run fast implementations of the traditional ray tracing algorithm. However, they

have specifically been optimized for speed rather than for quality and flexibility. Additionally, they often use parallel or distributed processing to achieve interactive frame rates. This parallelization can be realized on both shared memory multiprocessor-machines [54, 55, 64, 62, 63], as well as on loosely-coupled clusters of commodity PCs [90, 93, 13].

Recently, Purcell et al. [69] have shown that ray tracing can also be realized on programmable graphics hardware. In his work, Purcell has exploited the programmability of todays GPUs by using the graphics card as a massively parallel, highly efficient streaming processor. Here, the recursive ray tracing algorithm is first reformulated as a stream processing task by expressing the core algorithms of ray tracing – i.e. traversal, intersection, and shading – as small "kernels" that operate on a stream of pixels and textures, where each pixel corresponds to exactly one ray. The different kernels can then be implemented using pixels shaders, and can be executed by applying the respective pixel shader to a screen-aligned quad.

Finally, the third alternative to realizing ray tracing is the design of custom hardware that is specialized for ray tracing. In that approach, the whole ray tracing algorithm is embedded in hardware. Given todays hardware resources, Schmittler et al. [75] have recently shown that this approach is feasible. In fact, it apparently can be realized using less hardware resources than used in a modern GPU, and promises to achieve full-screen ray-traced images at interactive rates even on a single graphics chip.

Today, all of these three options are being actively pursued, and will be briefly described in the following sections.

2. Realtime Ray Tracing in Software

In order to reach realtime ray traced frame rates with a software system, one has to focus on two different aspects: First, the system has to be built on a highly optimized ray tracing kernel that optimally uses the CPU. Second, as even the fastest CPUs today cannot deliver the performance needed for practical applications using a software based system also requires to combine the resources of multiple CPUs by using parallel or distributed ray tracing.

Interactive Ray Tracing on Shared Memory Systems

Though ray tracing itself trivially lends itself to parallelization, special care has to be taken in an interactive setting where only a minimum amount of time can be spent on communication and synchronization. Generally, these issues – fast inter-processor-communication and synchronization – can best be handled on shared memory computers.

Thus, it is not surprising that interactive ray tracing has first been realized on massively parallel shared memory supercomputers. These systems provided the required floating point power and memory bandwidth, and combined the performance of many CPUs with relatively little effort.

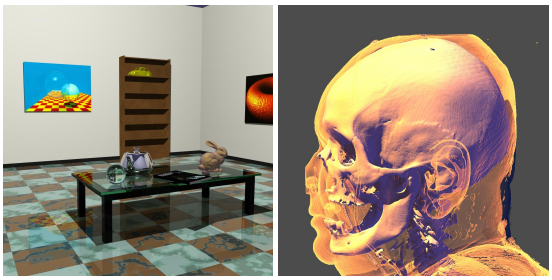


Figure 2: *Two examples from the Utah Interactive Ray Tracing System. Left: A typical ray-traced scenes with parametric patches, shadows and reflections. Right: Complex volume rendering. (Image courtesy of Steve Parker)*

The first to achieve interactive frame rates on such platforms were Muuss et al. [54, 55] who used interactive ray tracing to simulate radar systems in highly complex CSG (Constructive Solid Geometry) environments that would otherwise have been impossible to be rendered interactively.

On a similar hardware platform, Parker et al. [64, 62, 63] were the first to show a full-featured ray tracer with shadows, reflections, textures, etc (see Figure 2a). Additionally, their system allows for high-quality volume rendering [62] and iso-surface visualization [63] (see Figure 2b).

Interactive Ray Tracing on PC clusters

Today, a more cost-effective approach to obtain high compute power is the use of a clusters of commodity PCs. Such systems are already widely available and usually cost only a fraction of a shared memory machine while providing equivalent performance. However, PC clusters do have certain drawbacks as compared to a shared memory supercomputer, i.e. they do not offer hardware-supported inter-processor communication, and they have less memory, less communication bandwidth, and higher latencies.

The Saarland RTRT/OpenRT Engine

In 2001, Wald et al. [90, 93] showed that interactive ray tracing can also be realized on such low-cost hardware. Their system – the Saarland University's RTRT/OpenRT ray tracing engine† – combines a fast ray tracing core with sophisticated parallelization on a cluster of commodity PCs. In the meantime, this system has been extended to a complete rendering engine featuring a fast ray tracing core, efficient parallelization, support for dynamic scenes, and a flexible and powerful API.

† Note: "RTRT" refers to the "RealTime Ray Tracing" core of the engine, while "OpenRT" refers to the API through which this engine is driven (see Section 7)

The Utah "Star-Ray" Architecture

Just recently, the above-mentioned "Utah-system" [64, 62, 63] (now called "Star-Ray") has also been ported to run on a PC clusters [13]. It too consists of a sophisticated parallelization framework around a highly optimized ray tracing core. In its core, the new system uses the same algorithms as on the original system on the Onyx [63]: Highly efficient traversal of the volume data set that quickly skips uninteresting regions, efficient data layout using bricking to improve caching (reported to bring up to a tenfold performance improvement on certain architectures [76]), optimized algorithms for analytic ray-isosurface intersection computation, and efficient parallelization in the image plane.

While certain of the systems aspects – i.e. the distribution framework and optimization for memory accesses – are similar to the RTRT/OpenRT engine, the system has been optimized mainly for the interactive visualization of volumes and isosurfaces, and does not primarily target polygonal scenes and lighting simulation.

Due to the above-mentioned drawbacks of using PC clusters – less memory, less communication bandwidth, and higher latencies – the parallelization and communication layer of the PC-based Star-Ray system had to be adapted [13]. Similar to the Saarland System, they now use a client-server approach, in which the server controls the clients via TCP/IP by sending them image tiles to be computed. Using the same number of nodes, their cluster-based system achieves roughly the same performance as the original, shared memory based system on the Onyx (see Figure 3).

The new Star-Ray system is also able to handle massively complex volume data sets by implementing a software layer offering a distributed shared memory architecture: Volume data is separated into disjoint regions that are kept distributed among the different machines. If a client needs access to remote data, this software layer transparently fetches and caches the required data. Additionally, they perform several optimization to reduce the bandwidth for transferring these tiles. Their system for handling massive volume data is similar to the approach that Wald et al. have taken for rendering massive polygonal data sets [93], but uses a better, distributed scheme of storing the data. While this distributed data storage costs roughly half the performance of their system, it allows them to render an eight gigabyte dataset (of a Richtmyer-Meshkov instability) at interactive rates with high-quality analytic isosurfaces, as shown in Figure 3.

As can be seen, both the Saarland Engine as well as the new Utah engine have concentrated on similar issues: First, a highly optimized kernel that especially considers memory effects. Second, sophisticated parallelization with special emphasis on handling the bandwidth and latency issues of a PC cluster.

While volume rendering obviously requires different algorithms and optimizations than ray tracing in polygonal

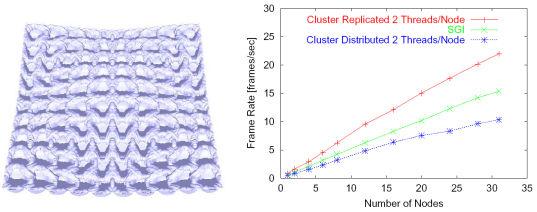


Figure 3: *Left: One frame from an eight gigabyte volume data set rendered interactively with analytic isosurfacing. Right: Performance comparison of their new, cluster based system is comparison to the Onyx. Using the same number of nodes, their cluster based system provides roughly the same performance as the Onyx system.*

scenes, many of the concepts are still similar. As the scope of this STAR is more on polygonal rendering, we will in the following concentrate on the Saarland engine. Before discussing the actual parallelization aspects in Section 2.2 we briefly summarize the most recent developments in realtime ray tracing.

2.1. The RTRT Realtime Ray Tracing Kernel

The RTRT "RealTime Ray Tracing" kernel concentrates mostly on efficient data layout to minimize memory access, and on optimally exploiting processor caches. These techniques are essential for good performance on todays CPUs, which often waste most of their time waiting for data from memory. Additionally, the RTRT system leverages the SIMD extensions of modern CPUs that perform several floating point operations in a single operation. This is only possible due to algorithmic changes that expose the coherence of the ray tracing algorithm. These algorithmic changes concentrate around rearranging the ray tracing algorithm to trace, intersect, and shade *packets* of rays instead of recursively tracing individual rays. This amortizes memory access over several rays and enables the use of SSE instructions [33] by performing operations on four rays in parallel.

Both general design and algorithmic aspects of the RTRT kernel have already been covered in depth in the original STAR on interactive ray tracing [92]. Since then, however, this kernel has been gradually improved, including both significant performance improvements in ray traversal and intersection, as well as improvements in shading and flexibility.

2.1.1. Ray Traversal and Intersection

The RTRT software ray tracing kernel still builds on "Coherent Ray Tracing" [90], but has been completely reimplemented to gain even higher performance. Additionally, the new kernel employs algorithmic improvements on BSP construction using advanced cost prediction functions [31] to achieve even higher performance. Even when traversing single, incoherent rays (i.e. *without* using the SSE instruction set) the new

kernel is slightly faster than the originally published system tracing packets of rays. Exploiting the full performance of the SIMD code then achieves an additional performance improvement of 2–3 when shooting coherent rays. It is important to note that the RTRT kernel does not use any approximations to achieve this speedup. It still performs at least the operations of a traditional ray tracer.

The improvements have also been supported by better compilers. Even though experiments with automatic SIMD code generation by the compiler have been disappointing, modern compilers offer increasingly powerful tools for writing optimized software. For example, the recent GNU gcc and the Intel C/C++ compilers now support "intrinsics" that allow for writing easily maintainable low-level SIMD code in a C-style manner that can then be tightly integrated with standard C/C++ code [32]. Using intrinsics also allows the compiler to perform automatic low-level optimizations such as loop unrolling, instruction reordering, constant propagation, register allocation, etc., which a compiler can do much better than a programmer.

Figure 4: *The "ERW6" and "soda" scenes (800 and 2.5 million triangles, respectively) rendered at* 1024×1024 *pixels on a single Pentium-IV 2.5GHz CPU using the RTRT kernel. Including shading, these scenes run at 2.3 and 1.8 frames per second, respectively. Only tracing the rays – i.e. without shading – RTRT achieves 7.1 respectively 4.1 frames per second, see Table 1.*

These recent improvements - better implementation, compilers, and BSPs – allow the new kernel to achieve significant improvements over earlier data [90] by roughly a factor of 2.5 to 3 even when normalized by the speed of the processors (see Figure 4).

Additionally, CPU speed has increased by roughly a factor of 4 since the original publication – from the 800MHz Pentium-IIIs used in [90] to almost 3 GHz Pentium-IVs available today. This combination of algorithmic improvements with higher CPU speed now allows for tracing several million rays per second even on a single CPU, as can be seen in Table 1.

CPU/scene \ RT & shading	SSE none	SSE simple	non-SSE simple
Pentium-IV 2.5 GHz			
ERW6 (static)	7.1	2.3	1.37
ERW6 (dynamic)	4.8	1.97	1.06
conf (static)	4.55	1.93	1.2
conf (dynamic)	2.94	1.6	0.82
soda hall	4.12	1.8	1.055
AthlonMP 1700+ (1.5GHz)			
ERW6 (static)	3.7	1.55	0.9
ERW6 (dynamic)	2.54	1.29	0.7
conf (static)	2.5	1.25	0.77
conf (dynamic)	1.7	1.0	0.58
soda hall	2.11	1.14	0.67

Table 1: *Ray casting performance in million rays per second on a* single *CPU at a resolution of* 1024×1024 *pixels using a 2.5 GHz Intel Pentium-4 notebook (top) and on an AMD AthlonMP 1700+ (1.5 GHz, bottom). Though ray tracing scales nicely with scene complexity, even simple shading can already cost more than a factor of two given our current ray tracing performance! The above numbers directly correspond to the achievable frame rate on a* single *CPU at full-screen resolution (*1024×1024 *pixels) . The ERW6, soda hall, and conference scenes can be seen in Figures 4a, 4b, and 26b, respectively.*

2.1.2. Shading

Compared to the original system presented in [90], the new engine also offers shader plug-ins to support arbitrary shading computations — generating initial camera rays, computing the scattering of light at surfaces, sampling light sources, and processing pixel data.

As shading has traditionally been cheap compared to casting rays, most optimizations in the RTRT engine so far have focussed on the core ray tracing computations, i.e. on BSP traversal and triangle intersection. With the recent improvements in ray tracing performance, however, shading is now becoming the bottleneck. In contrast to other ray tracers most of the time is now spent in shading calculations, and tracing rays usually takes significantly less than 50% of total rendering time even for complex scenes and simple shading. This is mainly due to the need to split up coherent packets of rays and to feed them to shaders, which can currently only operate on single rays. Even though this is still faster than tracing all rays individually (see Table 1), the overhead can cut performance in half !

However, the flexibility of shader plug-ins is essential for making a ray tracing engine a general tool that enables the unique applications discussed in later sections, so this penalty is currently unavoidable. Still, these results indicate that in the future, more effort should be concentrated on fast

and efficient shading computations and in particular on an efficient interface between ray tracing and shading. The efficient shading of packets of coherent rays in a streaming manner requires closer investigation. This has been shown to work well for fixed lighting models [8], but has to be adapted to general shading operations.

2.2. Distribution Issues of the RTRT Engine

Even though the performance of the RTRT kernel allows some limited amount of interactive ray tracing on a single processor, one CPU alone still cannot (yet) deliver the performance required for practical applications, which require complex shading, shadows, and reflections. Achieving sufficient performance on todays hardware requires to combine the computational resources of multiple CPUs. In the medium term, it is likely that small-scale multiprocessor shared-memory systems will become available for the PC market. Until then however the most cost-effective approach to compute power is the use of a distributed-memory PC cluster.

2.2.1. General System Design

In the following we briefly discuss the main issues of high-performance parallelization in a distributed cluster environment, by taking a closer look at the distribution framework of the RTRT/OpenRT interactive ray tracing engine.

Screen Space Task Subdivision: Effective parallel processing requires breaking 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 optimizes throughput, it is not applicable to a realtime setting, where only a single frame is to be computed at any given time.

For realtime ray tracing, there are basically two approaches: *object space* and *screen space* subdivision [70, 11]. Object space approaches store the scene database 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.

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 us to render models that are larger than the individual clients' memories. Usually, 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 [93, 74]). Using this approach, models larger than each client's memory can be rendered, as long as the combined memories of all clients are large enough to hold the working set of the model.

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 rectangular "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 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 the cost of tasks would lead to bad client utilization and would result in low scalability.

Therefore, RTRT combines the tile-based approach with a dynamic load balancing scheme: Instead of assigning all tiles in advance, the clients follow a demand-driven strategy and ask for work: As soon as a client has finished a tile, it sends its results back to the master, which automatically requests the next unassigned tile.

2.2.2. Performance Issues on PC Clusters

Screen space parallelization and dynamic load balancing are both well-known and are applied in similar form in many different parallel ray tracing systems (for an overview, see e.g. [11]). However, the need for communication with the different client machines – together with the high network latencies of commodity PC hardware – require very careful optimizations and several additional techniques to achieve realtime performance and good scalability.

Efficient communication: Most standardized libraries such as MPI [19] or PVM [21] cannot provide the required level of flexibility and performance that we are faced with in an interactive environment. Therefore, all communication in the RTRT/OpenRT engine has been implemented from scratch with standard UNIX TCP/IP calls. This ensures a minimum of communication latency, and extracts the maximum performance out of the network.

Task prefetching: Upon completion of a task, a client sends its results to the server, and – in dynamic load balancing – has to wait for a new task to arrive. This delay (the network round-trip time) is usually the worst problem in dynamic load balancing, as it may result in the clients running idle waiting for work.

To cope with this problem, we have each client "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 back to the server. Currently, each client is usually prefetching about 4 tiles. This however depends on the ratio of compute performance and tile cost to network latency and might differ for other configurations.

Frame interleaving: Another source of latency is the interval between two successive frames, in which the application usually changes the scene settings before starting the next frame. 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 $N-1$. Note, that this is similar to usual *double buffering* [73], but with one additional frame of latency.

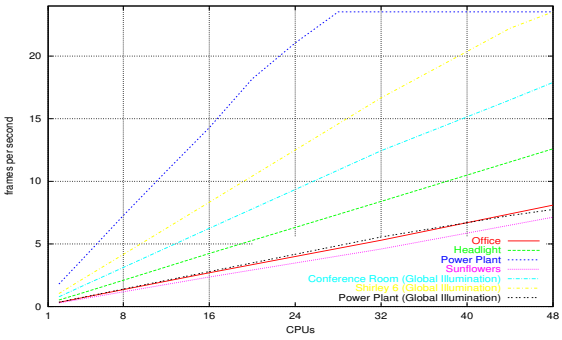
Differential updates: For realistic scenes the network bandwidth obviously is not high enough for sending the entire scene to each client for every frame. Thus, only differential updates are sent between subsequent frames: Only those settings that have actually changed from the previous frame (e.g. the camera position, or a transformation of an object) will be sent to the clients. These updates are sent to the clients asynchronously: The server already streams partial updates of frame N while the application continues specifying the differences and while the clients are still working on frame $N-1$. Of course, this requires careful synchronization via multiple threads on both clients and server.

Multithreading: Due to a better cost/performance ratio, each client is a dual-processor machine. Using multithreading on each client then allows sharing of most data between these threads, amortizing the communication cost for scene updates over two CPUs.

2.2.3. Results

In its standard configuration, the RTRT/OpenRT engine runs on a cluster of up to 24 dual processor AMD AthlonMP 1800+ PCs with 512 MB RAM each (48 CPUs total). The nodes are interconnected by a fully switched 100 Mbit Ethernet using a single Gigabit uplink to the master display and application server to handle the large amounts of pixel data generated in every frame. Note that this hardware setup is not even state of the art, as much faster processors are already available. It seems reasonable to assume that the ray tracing performance of this setup will be commonly available on the desktop only a few years from now.

The master machine is responsible for communicating with the application (see Section 7) and centrally manages the cluster nodes as described above. Given the ray tracing performance shown in Section 2.1, efficient load balancing requires having enough tasks with a high enough cost available in order to offset the high communication latency of



CPUs	2	4	8	16	24	32	48
PP/S	1.26	2.36	5.06	9.52	13.8	18.2	22.2
PP/IGI	0.61	1.03	2.18	4.3	∼6	∼8	11.1
SF	0.3	0.59	1.18	2.45	3.36	4.87	7.01

Table 2: *Scalability of our distributed ray tracing engine for different scenes (PP/S: "Power Plant" scene with simple shading, PP/IGI: power plant with instant global illumination [91], SF: Oliver Deussens "Sunflowers"). The impact of the cost per pixel can be seen in the power plant scene: For simpler shading we start to see load balancing problems at 24 CPUs because at a resolution of 640x480 we no longer have enough jobs to keep all clients busy. For complex computations like global illumination this problem occurs later. For the respective scenes see Figures 18, 20, and 26.*

Ethernet. For simple scenes with simple shading, it becomes a problem to have enough tiles available to keep all clients busy. However, using more and smaller tiles increases the network load and decreases the available coherence within each tasks. For a given number of clients and compute-to-latency ratio there is a tile size that optimizes the achievable frame rate. While this optimal tile size depends on the actual settings, 16×16 pixels has shown to be reasonably good for most scenes.

As seen in Table 2 load balancing works fairly well for reasonably complex scenes and a good computation to latency ratio. Fortunately, many interesting applications — such as global illumination — require costly computations per pixel and thus scale well to 48 processors and more (see Table 2 and Part 3). The distribution process is completely transparent to both applications and shaders. The application runs only on the master machine and interacts with the rendering engine only through the OpenRT API (see Section 7). The shaders are loaded dynamically on the clients and compute their pixel values independently of the application.

The achieved performance allows the RTRT/OpenRT engine to be used in many practical applications already today [86], including the visualization of massively complex models, interactive lighting simulation, high-quality rendering, and even interactive global illumination. An overview of these applications will be given in Section 8.

3. Ray Tracing on Programmable GPUs

For the last several years, graphics hardware has seen a faster rate of performance increase than CPUs. Modern CPU design is optimized for rapid execution of serial code. It is becoming increasingly difficult to realize performance benefits by adding extra transistors. The GPU on the other hand, is optimized for massively parallel vertex and fragment shading code [51]. Transistors spent on additional functional units directly improve performance. As such, GPUs are able to utilize extra transistors more efficiently than CPUs, and GPU performance gains will continue to out pace CPU performance gains as semiconductor fabrication technology advances.

Recently, GPUs have become programmable in an effort to expand the range of shading effects they can produce. This programmability has enabled several algorithms to be ported to the GPU [9, 30, 43, 46], many of which run at rates competitive with or faster than a CPU-based approach. The ubiquity and low cost of graphics processors, coupled with their performance on parallel applications, makes them an attractive architecture for implementing realtime ray tracing.

Graphics algorithms like ray tracing can benefit from a GPU-based implementation in two other ways. First, when an algorithm executed on the GPU finishes running, the data meant for display is already on the graphics card. There is no need to transfer data for display. Second, graphics algorithms can work as hybrid algorithms, supplementing the capabilities of the GPU, and leveraging existing GPU rendering capabilities. For example, a ray tracer could be used to add global illumination effects like shadows, reflections, or indirect lighting to a polygon renderer.

We will examine two different approaches to using the GPU for ray tracing. Both utilize the high computational throughput of the GPU to obtain rendering rates comparable to those obtained by the fastest software-only ray tracers. Section 3.2 describes the work done by Carr et al. [10] in configuring the GPU as a ray-triangle intersection engine. Section 3.3 then describes the work by Purcell et al. [69] in mapping the entire ray tracing computation to a programmable GPU. Before discussing either implementation, we review the modern programmable graphics pipeline in Section 3.1.

3.1. Modern Graphics Pipeline

Figure 5 shows an abstraction of the graphics pipeline used by GPUs like the ATI Radeon 9800 Pro [3] or the NVIDIA GeForce FX 5900 Ultra [57]. The vertex and fragment stages are implemented with programmable engines that execute user defined programs. Modern GPUs have support for floating point computation throughout most of the pipeline, and have floating point frame buffer and texture memory. The two GPU-based ray tracing systems examined in this report only use fragment programs, so we will not consider the vertex engine further.

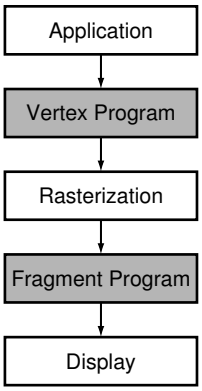


Figure 5: *The programmable graphics pipeline. The gray boxes show the programmable vertex and fragment engines available on modern GPUs. Older GPUs have fixed function vertex and fragment processing stages.*

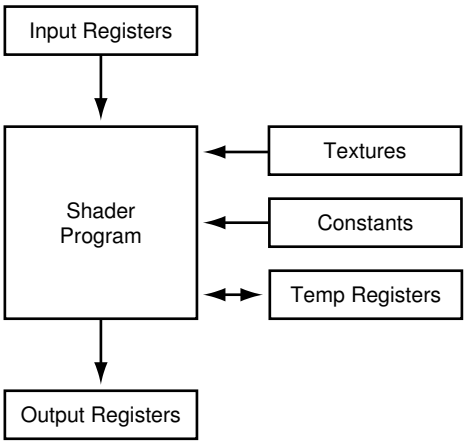


Figure 6: *The programmable fragment processor. A shader program can read data from input registers, constants, texture memory, and temporary registers. Temporary registers store intermediate calculations, and the output registers store the final color values for the fragment.*

The execution model for the fragment engine is shown in Figure 6. Fragment programs are written in a 4-way SIMD assembly language [60, 58], which includes common operations like add, multiply, dot product, and texture fetch. Fragment programs are currently limited to 64 or 1024 instructions, depending on the specific chip being used. These limits are likely to increase with future generations of GPUs.

Current GPUs do not allow data dependent looping or branching within fragment programs, though this limitation is likely to be removed in upcoming generations. Data dependent texture fetches are allowed, however. A dependent texture fetch is simply a texture fetch at an address that has been computed, unlike a standard texture fetch where the

address is determined by interpolated texture coordinates. Modern hardware allows for at least four levels of dependent texture fetches.

Finally, modern graphics hardware provides the NV_OCCLUSION_QUERY extension [59]. An occlusion query simply returns a count of the number of fragments that were drawn to the frame buffer between the start and end of the query. In order to hide the latency and minimize pipeline flushes the results of a query can obtained later after more geometry has been rendered.

The ray engine requires only floating point computation to work correctly. The streaming ray tracer, on the other hand, relies on all the described features of modern GPUs. While each system takes a fairly different approach to GPU-based ray tracing, both initiate computation in a similar fashion, and both configure the GPU as a high performance parallel compute engine.

3.2. The Ray Engine

The ray engine [10] implements a ray-triangle intersection routine as a fragment program on programmable graphics hardware. Batches of rays are sent down to the GPU from a CPU-based rendering task. A series of triangles is then streamed down to the GPU where they are intersected with all the rays in the batch. Finally, the results of the intersection tests are read back to the host to be used in subsequent rendering stages.

The ray engine is set up to allow it to integrate into existing applications that utilize ray-triangle intersection. Monte Carlo ray tracing, photon mapping, form factor computation, and general visibility preprocessing all use ray-triangle intersection routines. The ray engine could replace the software ray-triangle intersection routine, freeing host CPU cycles for shading or other tasks.

3.2.1. Implementation

The ray engine is designed to accept batches of rays and triangles from a host application and return the nearest triangle intersection (hit) point for each ray. Rays are downloaded to the GPU as two screen sized textures: one texture for the ray origin, and one texture for the ray direction.

Triangles are distributed to each ray by drawing a screen sized quadrilateral. The triangle data is stored as vertex interpolants (e.g. color and texture coordinates). Each vertex has an identical interpolant value, meaning interpolation during rasterization distributes an identical copy of the triangle data to each pixel.

Ray-triangle intersection happens one triangle at a time over all the downloaded rays. A pixel shader implements ray-triangle intersection between the ray parameters fetched from texture memory and the triangle data stored in interpolant memory. Output from the intersection test is stored in the frame buffer. The alpha value indicates whether a given ray found an intersection point. The color buffer stores the triangle id of the closest hit. The z-buffer is used to store the ray-triangle intersection distance. The built in z-test ensures that the nearest intersection point is always stored. The host reads back the hit information contained in the color and depth buffers once all triangles have been sent through the pipeline.

Intersecting all rays against the entire scene database would reduce the ray engine to a brute force ray-triangle intersector. Instead, triangles and rays are batched up into coherent batches, and these batches are downloaded to the GPU and results are read back. Sufficiently incoherent rays are intersected directly on the host.

3.2.2. Results

The ray engine was originally implemented on the Radeon 8500 [1]. This early programmable GPU was missing several features found on more recent GPUs, most notably floating point textures and floating point math operations in fragment programs. Floating point is essential for avoiding artifacts in a ray tracer, as evidenced by the teapot rendering shown in Figure 7.

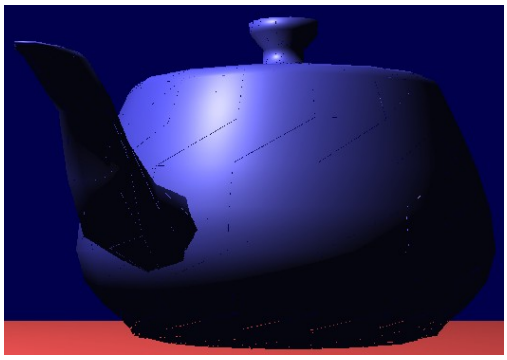


Figure 7: *Teapot rendered by the ray engine on the Radeon 8500. Ray tracing computations require floating point fragment programs and textures to eliminate precision artifacts.*

Though the Radeon 8500 implementation of the ray engine did not produce usable images, it served to provide an estimate of how fast ray-triangle intersection on the GPU could be. The ray engine could perform 114M intersection tests per second, a number nearly three times the rate of the fastest software ray tracer (implemented on an 800 MHz Pentium III [90]).

The ray engine was also simulated on a GPU with floating point capabilities. The simulated performance was between 100K and 200K rays per second. Images generated by the simulator are shown in Figure 8. The office scene was rendered with classic ray tracing using multiple point light sources. The Cornell box with teapot scene was rendered with a Monte Carlo ray tracer.

Figure 8: *Images generated by the ray engine for a simulated GPU with floating point capabilities. The office scene was rendered with classic ray tracing, and the Cornell box with teapot scene was rendered with Monte Carlo ray tracing.*

The overall performance of the ray engine is determined by the amount of hit readback to the host relative to the number of triangles intersected. For a small number of triangles, reading back the hits can be the performance limiting factor. For large numbers of triangles, the readback cost is amortized over the increased number of intersection tests. Since triangles are sent to the GPU in coherent batches, scenes lacking ray-triangle coherence will not be able to take advantage of the high ray-triangle intersection rates of the GPU.

3.2.3. Summary

The ray engine uses the GPU as a ray-triangle intersection co-processor. The host rendering process downloads batches of rays and triangles to the GPU, and the GPU returns the nearest hit for each ray over the set of triangles. The raw ray-triangle intersection rate achieved by the ray engine is much faster than CPU-based ray-triangle intersection rates, yet overall rendering performance is limited by the amount of hit data read back by the host.

3.3. Streaming Ray Tracer

The streaming ray tracer described by Purcell et al. takes a different approach toward GPU-based ray tracing. The ray engine only mapped ray-triangle intersection onto the GPU. The streaming ray tracer maps the entire ray tracing computation to the GPU including ray generation, acceleration structure traversal, triangle intersection, and shading.

A system flow diagram for the streaming ray tracer is shown in Figure 9. Each of the boxes represents a separate fragment program (or computation kernel), and the arrows represent the data flow (streams) between different stages of the execution. As with the ray engine, the computation for each kernel is initiated by drawing a screen filling quad. However, unlike the ray engine, triangles are stored in texture memory and are accessed through a uniform grid acceleration structure and not sent to the GPU as vertex interpolants.

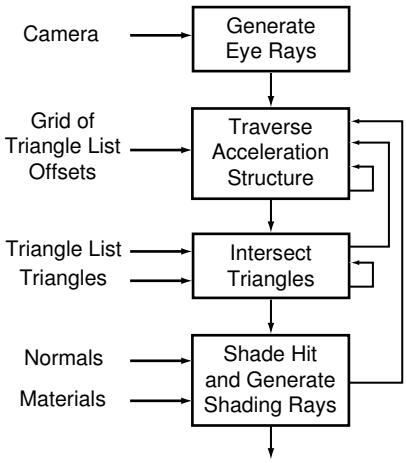


Figure 9: *The streaming ray tracer.*

3.3.1. Implementation

The streaming ray tracer is implemented as four separate kernels: the eye ray generator, the uniform grid traverser, the ray-triangle intersector, and the shader. These kernels are run as a sequence of rendering passes with the host controlling which kernel gets run at each pass. Each pass consists of binding the appropriate fragment program and drawing a screen sized quadrilateral to initiate computation. Data is streamed between kernels via texture memory. Each ray stores state indicating which kernel it needs to execute next.

Kernels

Eye Ray Generator The eye ray generation kernel takes camera parameters, including viewpoint and a view direction, and computes an initial viewing ray for each screen pixel. This kernel also tests rays against the scene bounding box and terminates rays that fail the bounding box test. Rays that pass the bounding box test are sent to the traverser.

Traverser The uniform grid traversal kernel reads rays and steps them through the grid using a 3D-DDA algorithm [20]. Grid cells are loaded from static texture memory. Rays loop through this kernel until they exit the grid (and are terminated), or they enter a voxel containing triangle data. The ray data and voxel address are passed along to the intersection kernel.

Intersector The ray-triangle intersection kernel reads rays and voxel addresses sent by the traverser and performs ray-triangle intersection. Triangles are fetched from static texture memory. Rays are processed against all the triangles in the voxel. If a ray-triangle intersection (hit) occurs, the hit information is passed along to the shading kernel. If no hit is found after all triangles in the voxel are tested, the ray is passed back to the traversal kernel.

Shader The shading kernel evaluates the color contribution of a given ray at the hit point. The shading kernel is also responsible for computing secondary rays (such as shadow

rays and reflection rays) and passing those back to the traversal kernel. Shading data is stored in texture memory and indexed by triangle id.

Memory Layout

The streaming ray tracer takes advantage of the dependent texturing capabilities of GPUs to traverse a uniform grid acceleration structure on the hardware. The memory layout for the scene database and acceleration structure is shown in Figure 10.

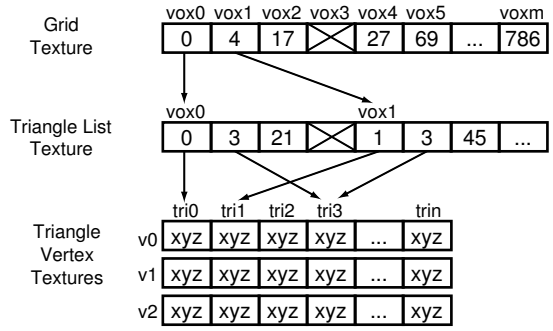


Figure 10: *Texture memory layout for the streaming ray tracer. Each grid cell contains a pointer to the start of the list of triangles for that grid cell, or a null pointer if the cell is empty. The triangle lists are stored in another texture. Each entry in the triangle list is a pointer to a set of vertex data for the indicated triangle. Triangle vertices are stored in a set of three separate textures.*

The grid is stored as a single component floating point texture. Each grid cell either contains a pointer to the start of the list of triangles for that grid cell, or a null pointer. The triangle list texture stores the triangle id for every triangle inside a given grid cell. The triangle id is used to locate the actual triangle data. Triangle geometry data is stored as a set of three floating point textures, one texture per vertex. Triangle normals are stored as another set of three floating point textures, and triangle vertex colors are stored as yet another set of three textures.

The streaming ray tracer was designed to render static scenes. The uniform grid acceleration structure is built offline, and the scene geometry along with the grid are downloaded to GPU texture memory once before rendering starts. Dynamic scenes could be implemented by downloading a new set of triangle and grid textures every frame.

Data passed between kernels is also stored in texture memory. These textures are generated in the same manner as intermediate outputs during a traditional multipass rendering. The frame buffer is copied to texture memory at the end of each kernel (save). Data is retrieved in the next pass by doing a non-dependent texture lookup at each pixel (restore). To avoid precision artifacts, the data saved and restored each pass by the streaming ray tracer requires floating point texture and frame buffer memory.

Flow Control

Ray tracing inherently has data dependent loops. Each ray can access a different number of triangles and a different number of grid cells before finding a hit point. Looping in the streaming ray tracer is accomplished through the use of the NV_OCCLUSION_QUERY extension.

An occlusion query is issued around the screen sized quad rendered at each step in the streaming ray tracer. Fragment programs are set up such that they do not write to the frame buffer when the ray they are computing is not in the currently executing stage. For example, if an intersection pass is being run but a given ray is in an empty voxel and needs to be traversed further, it will not update the contents of the frame buffer. The fragment is not counted by the occlusion query, and the value returned by the query indicates how many rays actually performed the computation each pass. The decision of which kernel to run next is determined by the value returned by the query.

Modern hardware makes executing looping with the occlusion query slightly more efficient with early-z occlusion culling. With early-z occlusion culling, fragments with depth values that are guaranteed to fail the depth test can be discarded right after rasterization. This frees up fragment processor resources for pixels that need it. The streaming ray tracer takes advantage of early-z occlusion culling by setting the depth of fragments based on which fragment program they need to execute next. This enables the hardware to skip over rays that are not executing the current fragment program.

3.3.2. Results

Purcell et al. implemented the streaming ray tracer in simulation for their paper. They were able to demonstrate a prototype system running on a Radeon 9700 Pro [2] when it was released. Figures 11 and 12 show two scenes rendered using the streaming ray tracer.

Figure 11 shows a Cornell box scene rendered on the GPU with an area light source. The random sample positions for

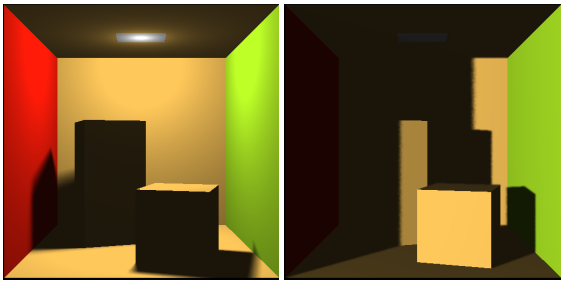


Figure 11: *Cornell box scene ray traced with soft shadows on the GPU.*

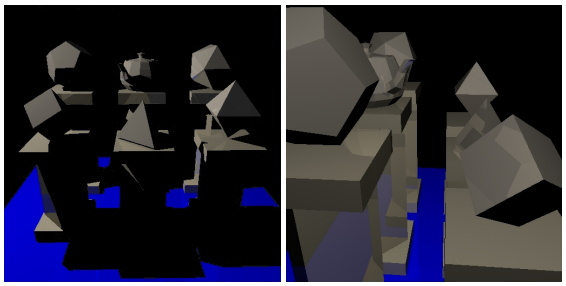


Figure 12: *"Teapotahedron" scene ray traced with shadows and reflections on the GPU. The right image is rendered with reflections only.*

the area light are pre-computed and stored in a texture. Figure 12 shows a simplified "teapotahedron" scene [23]. This scene demonstrates that the streaming ray tracer can handle reflections and shadows together.

The streaming ray tracer also runs in a hybrid mode, combining ray tracing effects with standard pipeline rendering. Figure 13 shows a level from Quake 3 that has been rendered with the standard pipeline only, and the same scene rendered with the standard pipeline plus ray traced shadows.

All of the preceding scenes rendered interactively at 256×256 pixels. The rendering rates for all the scenes are summarized in Table 3.

Scene	Frame Rate
Cornell Box (Soft Shadows)	15 fps
Teapotahedron (Refl., Shadows)	3 fps
Teapotahedron (Refl. only)	5 fps
Quake 3 (Hybrid)	5 fps

Table 3: *Rendering rates for the streaming ray tracer demonstration scenes. Scenes were rendered at* 256×256 *pixels.*

3.3.3. Summary

The streaming ray tracer takes advantage of the capabilities of modern graphics hardware to map the entire ray tracing computation onto the GPU. A sequence of multiple rendering passes process rays through the various stages in a ray tracer and generate the final display image. System performance is dictated by the efficiency of early-z occlusion culling to eliminate rays from processing. Even with limited hardware functionality, the prototype system was able to achieve high frame rates for interesting scenes.

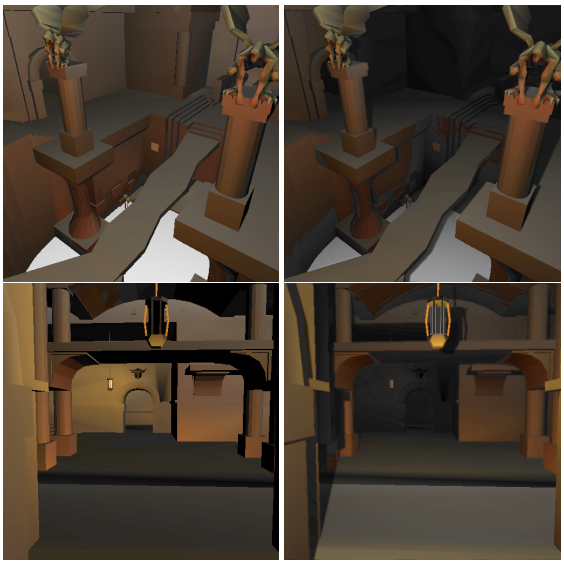


Figure 13: *Quake 3 rendered with standard feed-forward pipeline shading (left column) and with shadows (right column) added through a ray tracing shader.*

3.4. Conclusions

Each of the methods for ray tracing on GPUs still have several limitations that need to be addressed. The ray engine is particularly sensitive to GPU texture download and frame buffer readback performance. The readback path on current GPUs is not particularly fast, reducing the utility of any system designed around reading back data from the GPU.

The streaming ray tracer achieves relatively low utilization of the computational resources on the GPU. The GPU can only execute a single fragment program over all the fragments generated in a given pass. That means any rays that are intersecting triangles when a traversal pass is run would ideally be idle. Unfortunately, every fragment generated runs through the shader, but the outputs are simply masked for those rays not participating in the pass. Early-z occlusion culling helps reduce this overhead, but is not user controllable.

Despite these limitations, the ray engine and the streaming ray tracer both utilize programmable GPUs for high performance ray tracing. The inherently parallel nature of fragment programs, coupled with the rate of performance increase of GPUs make them an ideal candidate for implementing real time ray tracing. Early implementations of two different ray tracing systems on first generation programmable hardware is able to match performance of single CPU-based ray tracing systems.

4. Dedicated Realtime Ray Tracing Hardware

In previous sections we discussed realtime ray tracing implementations utilizing general purpose mainstream PC-processors (Section 2) or the processor on the graphics card available in most PCs (Section 3).

As an alternative to these software solutions it is highly interesting to also analyze special purpose chips that accelerate parts or even the whole ray tracing process in dedicated hardware. Although the development of hardware can be very costly and time consuming, it allows for the most efficient use of hardware resources and thus can potentially offer the highest performance given currently available hardware technology. This efficiency makes it very interesting for industrial applications that require highest performance together with advanced and quantitative visualization results. Dedicated hardware is also relevant in academia for finding a lower bound on the hardware resources required for realtime ray tracing.

The long rendering times of former ray tracing systems has led to many research efforts to speed up ray tracing using special purpose hardware. At the beginning, only the ray-triangle intersection was accelerated using several different special purpose hardware approaches (for a detailed survey see [25]). However, all approaches of accelerating only parts of the ray tracing algorithm suffered from the same general problem: the required bandwidth between the different parts is far too high to be efficiently handled unless all parts of the ray tracing system are located in the same chip.

The first full ray tracing systems built in hardware were ray casters for the visualization of volume data sets [53, 66, 65]. These ray casters used only primary rays and did not recursively spawn new rays to calculate lighting or secondary optical effects. These volume ray casters already delivered interactive frame rates and even became available commercially. For the more common application of ray tracing polygonal geometry only a hardware system accelerating offline ray tracing was ever developed [80, 28].

Last year, Schmittler et al. [75] published the first hardware architecture for full featured ray tracing of polygonal geometry aimed at realtime frame rates. His *SaarCOR* architecture shows that it should be possible to build a PC graphics engine for ray tracing at a hardware cost comparable to current rasterization chips. Such a system would deliver comparable performance while using significantly less off-chip bandwidth than current graphics technology.

Later, Schmittler et al.[74] added virtual memory support to their architecture that allows ray tracing-based graphics cards to render scenes many times larger than the on-board memory. This virtual memory support is completely transparent to the ray tracing core and to the application, allowing fully automatic memory management for any scene with hardly any performance impact. This architecture overcomes the hard restrictions of previous ray tracing systems, which required that the entire scene is stored in local memory.

The next sections provide a more detailed overview of the SaarCOR architecture and its use of virtual memory for scene management. We analyze the approach, provide simulation results for the expected performance, and discuss remaining issues and potential research directions.

4.1. The SaarCOR Architecture

The SaarCOR hardware architecture (see Figure 14) consists of a custom ray tracing chip connected to several standard SDRAM chips, a separate frame-buffer, and a bridge to the system bus all placed on a single board. The bus bridge is used to transfer all scene data from the host memory under the control of the virtual memory subsystem. The SDRAM chips are used as second level caches storing the current working set of the scene including its geometry, the spatial index structures for fast ray traversal, material data, and textures. The image is rendered into a dedicated frame buffer and is displayed via a standard VGA port.

The architecture is divided into three main components: The ray-generation controller (RGC), possibly multiple ray tracing pipelines (RTP), and the memory interface (MI). Each RTP consists of a ray-generation and shading unit (RGS) and the ray tracing core (RTC). The RGC tells each RGS which primary rays to generate. These primary rays are handed over to the RTC for computing the ray triangle intersections. Within the RTC, the traversal unit traverses the ray through the spatial index structure (a kd-tree in our case) until a leaf-node is reached. Leaf-nodes store lists of triangle addresses, which are then fetched by the list unit. The intersection unit then loads the data of a triangle and performs the intersection computation. Its results are sent back to the traversal unit, which either continues ray traversal through the kd-tree or sends the intersection results back to the RGS.

The RGS is responsible for shading the ray, which might recursively generate new rays. Currently only an extended Phong reflection model has been simulated that can access two textures: a standard image texture and a bump map. In addition this shader implements shadows, reflection, and refraction effects by spawning new rays as needed. Please note that this fixed Phong shader is only used to approximate the current use of shading until support for full programmable shading has been integrated into the architecture.

All memory requests by the pipelines are handled by the unified memory interface. This unit contains four different first-level caches, one for each type of functional unit. Since traversal units only read kd-tree nodes from memory, each memory item fetched is 8 bytes wide. Therefore the cache lines of the traversal cache (trav-cache) are also 8 bytes wide. Similarly since triangle data fetched by the intersection units consists of 36 bytes, cache lines of the intersection cache (int-cache) contain exactly these 36 bytes. The RGS units

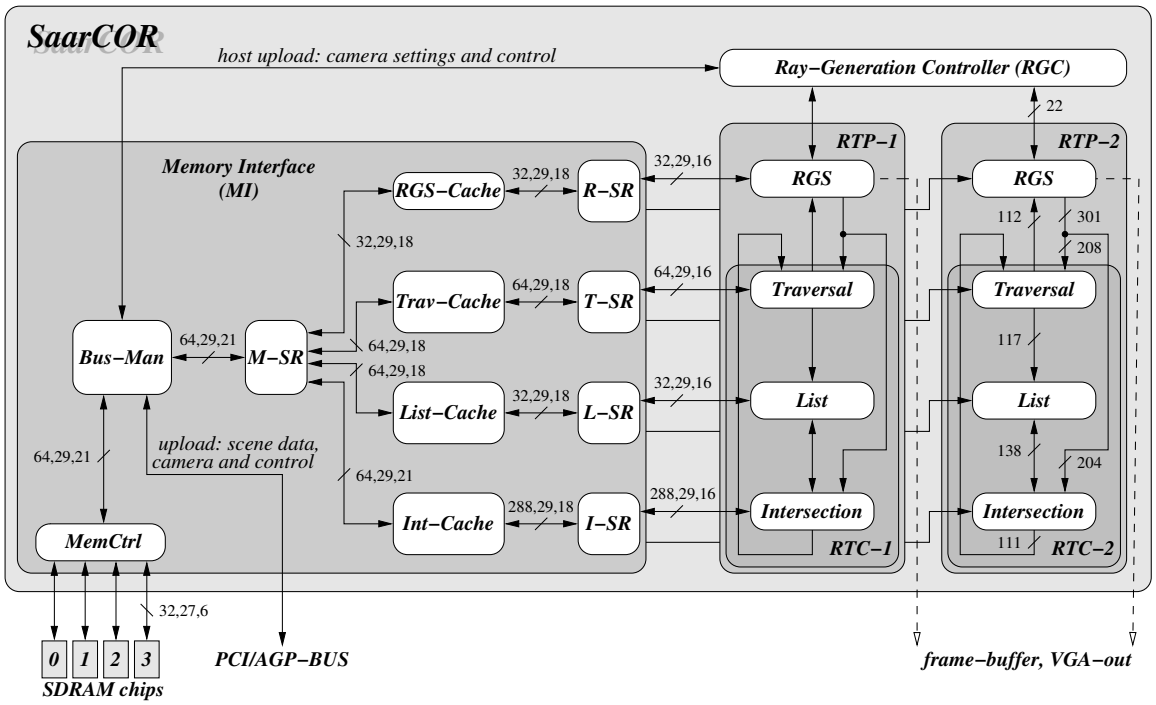


Figure 14: *The SaarCOR architecture consists of three components: The ray-generation controller, multiple ray tracing pipelines (RTP) and the memory interface. Each RTP consists of a ray-generation and shading unit (RGS) and the ray tracing core (RTC). Please note the simple routing scheme used: it contains only point-to-point connections and small busses, whose width is also shown separated into data-, address- and control-bits.*

and the list units operate only on four bytes of data per memory request. But since the memory bus is 64 bits wide, the cache lines of the RGS cache and the list cache have been extended to eight bytes.

All functional units of the same type share their cache. This works well since all ray tracing and shading operation are performed on packets of rays instead of single rays. This significantly reduces the number of memory requests from any unit and allows to scale the performance simply by increasing the number of RTPs.

In order to keep the pipelines busy all the time, memory access latencies are hidden by using multi-threading with the RTP simultaneously working on several independent packets of rays. Even a small number of threads suffices to achieve high utilization. Section 4.4 provides further details of this technique.

4.2. The Virtual Memory Architecture

All previous approaches to build a hardware support for ray tracing required the entire scene to be stored in local memory. This limited the complexity of the scenes that could be rendered as they needed to fit into the local memory and made the hardware very costly as it had to be equipped with large amounts of on-board memory.

In order to minimize the amount of local memory and maximize the performance, the scene database has to be explicitly managed such that only those parts needed to render the current image are stored in on-board memory.

Traditionally, data management was done by the application. However, this is a non-trivial task because the application needs to find out which parts of the scene are visible in the current view. With ray tracing this task becomes even harder due to scene data required by secondary shadow, reflection, or refraction rays.

Schmittler et al. [74] propose a fully automatic architecture to manage the scene data in hardware. This architecture is completely transparent to the ray tracing core as well as to the application. Even though they used the SaarCOR ray tracer for their research the design should easily transfer to any hardware based ray tracer.

The concept utilized by the VMA is as simple as effective: The scene data is stored in host memory only and the ray tracing card accesses this memory by DMA transfers via the PCI or AGP bus. To minimize bandwidth requirements on the system bus, the local memory on the graphics card

is used to cache the current working set. The mechanism used to manage this second level cache is the same as is used by the first level caches: it utilizes a standard four-way set-associative scheme with an increased cache lines size of 128 bytes. Additionally address-hashing is used to reduce the probability of collisions, i.e. of different addresses mapping to the same cache entry.

The size of these cache lines is a trade-off between larger cache lines resulting in a coarse subdivision of the memory and smaller cache lines that require significantly more memory to store the cache tags and additional bits for management purposes. This meta data already requires 640 KB to manage 16 MB of on-board memory with cache lines of 128 bytes each. It is therefore desirable to store the meta data off-chip in local memory. But because parts of this meta data is required for every memory access, another first-level cache is added to hold be least-recently used entries. This TLB-like approach increases the speed while needing little on-chip memory.

Measurements of several benchmark scenes [74] show that this two-level approach is very well suited to manage even highly complex scenes with several millions of multi-textured triangles with little loss in performance. Even with a standard PCI-bus the overhead due to the virtual memory processing is between 1% and 10% compared to a graphics board with enough memory to hold the entire scene. This is especially interesting as roughly 20% of all accesses to local memory are for fetching and updating the cache's meta data.

4.3. Algorithmic Issues

Section 2 gave a detailed description of the Saarland RTRT/OpenRT software ray tracer. The SaarCOR hardware ray tracer uses essentially the same algorithms as the software version that have been modified to better suit a hardware design. These modifications started by separating all steps of the algorithms into small independent kernels or functional units that can be efficiently implemented in hardware. The small modules increase the utilization of the chip in total, because the independent modules can be kept busy most of the time thus maximizing hardware usage.

More fundamental modifications have been made to turn the recursive spawning of new rays by the shader into independent iterations. For instance during shading of a primary ray a shader may generate two secondary rays: a shadow and a reflection ray. Traditionally, in a ray tracing software, the shader for the primary ray interrupts itself to first shoot the shadow ray and after completion continues shading until it spawns the reflection ray. This causes another interruption to shoot the reflection ray, with possibly even more interruptions, if the reflection ray spawns more rays, e.g. shadow rays. It is obvious that this approach is not well suited for hardware.

In the SaarCOR hardware, a primary ray gets fully shaded without the contribution of any secondary ray. The shader generates all secondary rays which are fully self-contained: Each secondary ray includes the coordinates of the pixel it contributes to as well as the accumulated weight of its contribution. In the example above, when a primary ray is shaded a newly generated shadow ray is a assigned a weight based on the weight of the incoming ray times the weight for light arriving from the direction of the new ray. If the shadow ray does reach the light source its weighted contribution is directly added to the final pixel value. Other rays spawned by a shader are handled similarly.

4.4. Pipelining and Latencies

One big advantage of the SaarCOR architecture is that it consists mainly of small independent units that allow for an easy placement on the chip and high utilization of the functional units. Furthermore careful optimizations have been made in order to simplify routing schemes using narrow busses (see Figure 14).

Pipelining and multi-threading are used extensively throughout the architecture. Pipelining allows very high computational density as long as the pipeline can be fed with valid inputs and no stalls occur. Multi-threading increases the possibility of having valid inputs for the pipeline and the use of small pipelined functional units helps in avoiding situations where the pipeline would need to stall.

Another important issue of the SaarCOR hardware architecture is the hiding of latencies that are unavoidable, e.g. due to memory access and in case of computational dependencies. Latency is addressed by multi-threading where each packet of rays forms a thread. Multiple independent threads allow for keeping the functional units throughout the chip busy. Due to its massively parallel nature, ray tracing can be trivially parallelized and enough independent packets are always available.

Simulations [75] show that the memory access latencies of slow SDRAM can be efficiently hidden by using only a few threads per RTP. Sixteen threads results in 60%-80% utilization while 32 threads increase the utilization by another 10 percent. Other simulations with the virtual memory architecture [74] show that even the high latencies of accessing host memory using a standard PCI-bus can be mostly hidden with hardly any loss in performance.

More detailed simulations of various sources of latency in the different stages in the RTC, the shaders, and the memory interface demonstrate that latency hiding works well for almost any source and amount of latency if the number of threads per pipeline can increased accordingly. However, threads require additional on-chip memory which limits the amount of latency that can be hidden efficiently.

4.5. On-Chip Bandwidth Issues

In order to limit the required on-chip bandwidth from the RTC to the caches, all operations are performed on packets of rays instead of single rays. This is similar to the Saarland RTRT/OpenRT software that utilizes SSE for efficient packet handling. In contrast to the software version, packets used in SaarCOR contain significantly more rays, which allows for decreasing the required bandwidth [75] due to data reused with a packet. On the other hand, larger packets have a larger working set that would decrease the hit rate of fixed size caches. As a compromise SaarCOR uses packets of 64 individual rays organized as 8×8 pixels.

In the software version the use of SSE offers a speed-up close to a factor of four. Unfortunately, sometimes packet need to be split into individual rays which is very costly. This would limit the usability of larger packets. In contrast, SaarCOR was explicitly developed to efficiently handle even large packets with only a minor performance drawback even when not all rays of the packet are active. In these cases, however, bandwidth requirements are higher because of limited data reuse. This packet handling allows SaarCOR to efficiently render even incoherent rays, which results in an almost constant cost per ray independent of the type of the ray, i.e. whether it is a primary or a secondary ray for shadows, refractions, or reflections [75].

4.6. Arithmetic Complexity

Schmittler et al. [75] compare the arithmetic complexity of a SaarCOR-chip to that for standard rasterization hardware and show that for comparable performance SaarCOR requires only half the number of floating point units. What makes this comparison difficult is that no exact architectural details on the floating point power of commercial rasterization hardware are publicly available. Because SaarCOR utilizes only streamlined floating point units that have been cut down to the minimum required for ray tracing and vendors of rasterization hardware claim support for full featured single-precision IEEE floating point operations, this comparison seems to be rather on the conservative side.

In fact, a floating-point multiplier of the SaarCOR-chip has less than a third of the size of a standard single-precision IEEE floating-point multiplier. This reduced complexity is achieved by careful optimizing the required floating-point precision at various stages of the pipeline. These optimizations include simulations of various floating-point formats. Please note that the term floating point unit stands for a circuit that can perform exactly one type of operation, i.e. addition, subtraction, or multiplication. It should not be confused by the term floating point ALU that consists of several arithmetic circuits and therefore is necessarily much larger in size.

4.7. Results

The performance of the SaarCOR-system has been analyzed and evaluated using cycle-accurate simulations on the register transfer level. These simulations include all parts of the system, i.e. the RTCs, full shading, the SDRAM, and finally the system bus with its long latencies. The configuration of the standard SaarCOR-chip used for these measurements results in roughly half the floating-point power of a traditional GPU in 2001 [75].

The standard configuration consists of four pipelines with a core frequency of 533 MHz and a 128-bit wide SDRAM memory running at 133 MHz delivering a theoretical bandwidth of 2 GB/s. The L1-caches are 4-way set-associative and their size is 400 KB split into 64 KB for shading, 64 KB for kd-tree nodes, 64 KB for triangle addresses, 144 KB for triangle data, and 64 KB for L2-cache meta data. The on-board memory contains 64 MB and a standard PCI-bus is used as the connection to the host. Sixteen threads have been used per RTP.

The on-chip caches are rather large but this should not be an issue: even for L1-caches high latencies can be tolerated due to multi-threading allowing to optimize the cache design. In addition many of today's CPUs already include up to 1 MB of on-chip cache. Further indication that the assumptions are on the conservative side are given by the fact that current graphics cards also run at 500 MHz and are equipped with up to 256 MB memory delivering a 10-times higher memory bandwidth than standard SaarCOR.

Table 4 and Figure 15 provide data about some of the scenes used to benchmark the SaarCOR architecture. This selection of benchmark scenes is motivated by the following results: For a fixed scene the performance of a ray tracing system scales linearly with the number of rays used, but is mostly independent on the type of the ray [75]. Thus given the performance p of a system and a scene S with no light and no reflections it is possible to estimate the performance p' for the same scene with n lights and where r % of all primary rays are reflected as

$$p'(S) = \frac{p(S)}{\frac{100+r}{100} \cdot (1+n)}$$

Arguments in [75] suggest and measurements on multi-textured scenes in [74] show that texturing has hardly any impact on the performance of the ray tracing system.

The benchmark scenes listed in Table 4 require up to 500 MB of storage on hard-disk but can be rendered with only 64 MB on-board memory. For most scenes even as little as 8 MB are sufficient. However, in order to simplify measurements, all scenes were rendered using 64 MB of local memory.

The SaarCOR architecture and the simulated configuration are the same as in [75, 74]. However, we also applied algorithmic improvements for building more efficient kd-trees.

Although the kd-tree used in the measurements were optimized for the software ray tracer, they also resulted in significantly higher speeds on the SaarCOR hardware (up to 3-times). It should be possible to increase performance even more by building more suitable kd-trees. The analysis in [75] shows that for a given hardware architecture a kd-tree can be built that best utilizes the resources in tree-traversal and ray-triangle intersection.

The side effect of using kd-trees that were optimized for a different architecture can be seen by analyzing the results listed in Table 5. Unbalanced workloads are likely to show higher variations in performance if architectural parameters are changed. For instance the BQD-2 scene utilizes ten times as many traversal than intersection operations. However, the architecture is designed for 4:1 ratio.

The new kd-trees also reduce the working set and increase the hit rate in the various caches — especially in more complex scenes. The performance measurements on the Cruiser scene in [75, 74] showed that the working set on triangles was too large such that a triangle-cache with 576 KB became necessary. The new kd-tree reduce the working set such that the scene can be efficiently rendered using only the same small cache (144 KB) as for all other scenes.

Figure 15: *Some of the scenes used for benchmarking.*

scene	#triangles	#lights with shadows	reflection-depth	textures
Cruiser	3 637 101	0	0	–
BQD-1	2 133 537	0	0	–
BQD-2	2 133 537	0	0	–
Quake	39 424	0	0	bilinear
Conference	282 000	2	0	–
Office	33 952	3	3	–

Table 4: *The scenes used for benchmarking.*

	std. SaarCOR		SaarCOR+VMA		
scene	fps	mem	fps	mem	PCI
Cruiser	170	6.1	121	7.7	0.14
BQD-1	137	1.9	135	2.5	0.03
BQD-2	59	26.6	42	34.1	0.91
Quake	129	9.4	126	11.4	0.01
Conference	77	8.5	68	10.8	0.09
Office	44	2.1	43	2.6	0.02

Table 5: *Performance measurements of a standard SaarCOR chip without and with VMA. Columns labeled with* fps *state the performance measured in frames per second, while columns labeled with* mem *list the amount of off-chip memory transfers per frame in MB. The column* PCI *shows the memory transfer over the PCI bus in MB.*

4.8. Conclusion

The SaarCOR system is the first hardware architecture showing that a ray tracing system can be built with hardware resources comparable to rasterization-based systems. It delivers essentially the same performance while requiring significantly less off-chip bandwidth. Of course, it also allows for all the advanced shading effects ray tracing is known for, like physically correct reflections, refractions, and lighting.

Using the virtual memory architecture, fully automatic and transparent management of the scene data is achieved. This removes the burden of explicit scene management from the application and allows to render scenes many times larger than the on-board memory with only a small loss in performance even in configurations based on slow standard SDRAM and a slow PCI bus.

Thus the SaarCOR architecture demonstrates the potential of hardware-based ray tracing systems for realtime applications. Although this first approach is already very powerful, it still leaves plenty of room for future work and improvements. While the OpenRT API (see Section 7) should directly be usable also with this hardware architecture no support for dynamic scenes has been integrated yet.

Also, the complexity estimates given in [75] take into account only the required floating point power and the amount of memory used for caches and register files. It does not account for routing and the overhead required for pipelining. Even though the SaarCOR-architecture uses a simple and narrow routing scheme, these issues might still eat up large amounts of chip area. However, a more exact analysis can only be performed after the architecture has been fully synthesized.

Currently, SaarCOR has no support for programmable shading. Besides the issue of generating new rays, shading is independent of the algorithm that is used to calculate visibility. Therefore programmable shading computations for ray tracing are not more complex than for rasterization. There

full featured programmable shading is commonly available even today. Further work is required on how to efficiently parallelize generation of new rays and efficiently forming packets for secondary rays.

The virtual memory architecture removed the necessity to store the scene in local memory but the entire scene still has to reside in host memory. A solution was suggested in Schmittler et al. [74]: Some geometry objects would be marked as "proxy" geometry and the hardware ray tracer would report the number of rays hit with such an object to the application. If the number of hits increases beyond a certain threshold the application can use this information for instance to replace the current proxy by more detailed geometry. This concept is similar to a level-of-detail approach but has the advantage that only objects need to be managed where the visibility is already known. This strategy seems very promising but further work is needed for evaluating its practical relevance.

5. Summary and Conclusion

In the previous sections, we have discussed the several different options for realizing realtime ray tracing, including software ray tracing on commodity CPUs, ray tracing on programmable graphics hardware, and the design of special purpose ray tracing hardware.

So far, the above systems are mostly prototype systems that are not yet widely used for practical applications. Still, they already achieve impressive performance and enable applications that have been impossible with different technologies: Walkthroughs of massive models without the need for approximations, interactive visualization of massively complex scientific and volumetric data sets, interactive simulation of complex reflection patterns, etc. As shown for example in Section 4, this increased flexibility, quality, and performance can even be realized with less resources than used in today's technology.

The above approaches to realtime ray tracing are fundamentally different and offer different advantages and disadvantages: Software ray tracing offers the largest flexibility because commodity CPUs impose no restrictions on the program they are running. On the other hand, software systems suffer from the fact that most CPUs concentrate on accelerating single-threaded applications, and thus cannot optimally exploit the inherent parallelism in ray tracing.

In contrast, the programmable GPUs can leverage the superior performance, bandwidth, and parallelism of modern chip design for ray tracing. Additionally, GPUs have become cheap and are commonly available in most PCs providing a cost-effective approach to fast ray tracing on the desktop. However, the programming model of GPUs still imposes severe restrictions on the algorithms that can be implemented efficiently. While some of these restrictions will eventually disappear, it seems unlikely that GPUs will ever be as flexible as CPUs.

Finally, special-purpose hardware promises optimal performance given current hardware technology, but it obviously more expensive and time-consuming to realize. However, the performance of special-purpose hardware is the yard-stick against which the other implementations will have to judged.

The presented approaches cover a wide spectrum in terms of flexibility (software ray tracing), cost efficiency (GPUs), and performance (SaarCOR). While it is not yet clear which one will be most successful, they *all* have the potential of eventually providing realtime ray tracing performance on every desktop.

PART TWO

Realtime Ray Tracing – Advanced Issues and Applications

In the first part of this STAR, we have argued that realtime ray tracing for future 3D graphics is both desirable and possible. We have shown that there are several ongoing developments towards realizing realtime ray tracing, and that these approaches all have the potential of bringing realtime ray tracing performance to the desktop in the near future.

However, even though all these approaches bear the same potential to be "the" future platform for realtime ray tracing, it is also clear that they will eventually all face similar problems: *All* the approaches discussed above have mainly focussed on accelerating the core ray tracing algorithms, i.e. traversing, intersecting, and shading of rays.

While concentrating on these core issues is the obvious first step towards realtime ray tracing, there are several more subtle – though not less important – issues that have to be addressed as next steps in making ray tracing a practical alternative for future graphical applications:

Support for dynamic scenes: Future ray tracing systems have to support dynamic scenes, as real interaction can only take place once the user can interactively modify the scene.

API Issues: Ubiquitous realtime ray tracing requires a unified and simple, yet flexible and powerful API to make ray tracing available to a wider class of users. Of course, API issues also have implications for higher software layers, e.g. by influencing the design of future scene graph APIs.

New Applications: The availability of a new technology enables new applications that have not previously been possible. We need to explore and practically analyze this set of applications in order to better understand the real value for the end user and future developments. While

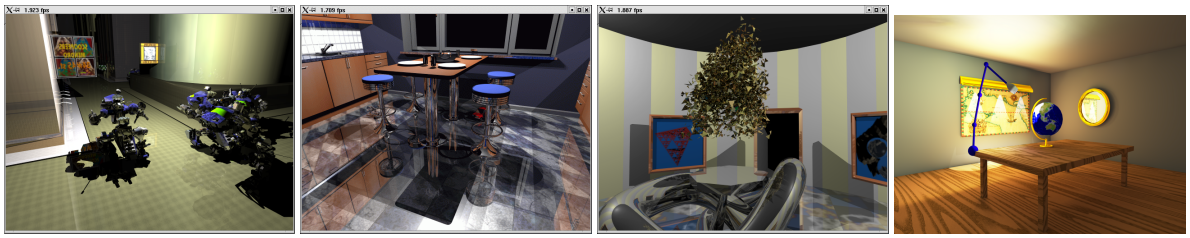


Figure 16: *Example screenshots from animated scenes. From left to right: a.) BART Robots, b.) BART Kitchen, c.) BART Museum, d.) Animated VRML scene with global illumination shader. At video resolution, all these scenes can be rendered at several frames per second. For more complex examples, also see Figure 18 that has also been computed with this technique.*

lighting simulation and the visualization of massive models are obvious but very important new applications, it seems that the potential of realtime ray tracing is much larger than that.

In the following sections, we will discuss some of the currently ongoing work in these respective areas. While this discussion is focused on the software ray tracing side it should apply similarly to the other architectures.

6. Ray Tracing in Dynamic Environments

Probably the biggest challenge for ray tracing is handling dynamic environments. Before realtime ray tracing, the time used for building an index structure such as kd-trees was insignificant compared to the long rendering times. Thus it has attracted only little research so far [23, 71, 49].

Some methods have been proposed for the case where predefined animation paths are known in advance (e.g. [22, 26]). These however are not applicable to our target setting of totally dynamic, unpredictable changes to the scene. Little research is available for truly interactive systems.

Recently new results on ray tracing in dynamic environments have obtained by Lext et al. with the BART project [50]. They provide an excellent analysis of the problems with dynamic scenes. Based on this analysis, they proposed a representative set of test scenes designed to stress the different aspects of ray tracing dynamic scenes (also see Figure 16a–c). Thus, the BART benchmark provides an excellent tool for evaluating and analyzing a dynamic ray tracing engine.

In their research, the behavior of dynamic scenes was classified into two inherently different classes: One form is *hierarchical motion*, where a whole group of primitives is subject to the same affine transformation. The other class is *unstructured motion*, where each triangle moves without relation to all others. For a closer explanation of the different kinds of motion, see the BART paper [50].

In a first step, Parker et al. [64] kept moving primitives out of the acceleration structure and checked them individually for every ray. This of course is only feasible for a small number of moving primitives.

Another approach would be to efficiently update the acceleration structure whenever objects move. Because objects can occupy a large number of cells in an acceleration structure this may require costly updates to large parts of the acceleration structure for each moving primitive. To overcome this problem, Reinhard et al. [71] proposed a dynamic acceleration structure based on a hierarchical grid. In order to quickly insert and delete objects independently of their size, larger objects are kept in coarser levels of the hierarchy. As a result, objects always cover approximately a constant number of cells, thus updating the acceleration structure in constant time. However, their method resulted in a rather high overhead, and also required their data structure to be rebuilt once in a while to avoid degeneration. Furthermore, their method mainly concentrated on unstructured motion, and is not well suited for hierarchical animation.

Recently, Lext et al. [49] proposed a way for quickly reconstructing an acceleration structure in a hierarchically animated scene by transforming the rays to the local object coordinate systems instead of transforming the objects and rebuilding their acceleration structures. To our knowledge, they have never applied their method to an interactive context.

At the same time, Wald et al. [88] have proposed a method that is motivated by the same observations as Lext et al. [49] of how dynamic scenes typically behave. Large parts of a scene often remain static over long periods of time. Other parts of a scene undergo well-structured transformations such as rigid motion or affine transformations. Yet other parts are changed in a totally unstructured way. This common structure within scenes can be exploited by maintaining geometry in separate *objects* according to their dynamic properties, and handling the different kinds of motion with different, specialized algorithms that are then combined into a common architecture.

Each object can consist of an arbitrary number of triangles. It has its own acceleration structure and can be updated independently of the rest of the scene. Of course, an additional top level acceleration structure must then be maintained that accelerates ray traversal between the objects in a scene. Each ray then first starts traversing this top level

structure. As soon as a voxel is found, the ray is intersected with the objects in the leaf by simply traversing the respective object's local acceleration structures.

For *static objects*, ray traversal works as before by just traversing the ray with the usual, fast traversal algorithm.

For *hierarchical animation*, the ray in world-space has to be intersected with an object that has been transformed with an affine transformation. In this case we can more efficiently transform the ray with the inverse transformation [49]. This slightly increases the per-ray cost (e.g. for the transformation), but totally removes the reconstruction cost for hierarchically animated objects.

To enable this scheme, all triangles that are subject to the same set of transformations (e.g. all the triangles forming the head of the animated robot in Figure 17) must be grouped by the application into the same object. Transforming such an object then simply requires updating its transformation matrix. This way, large groups of triangles can be transformed *en-bloc* without modifying their acceleration structure at all. Only the top-level acceleration structure has to be updated to reflect the new position of the object.

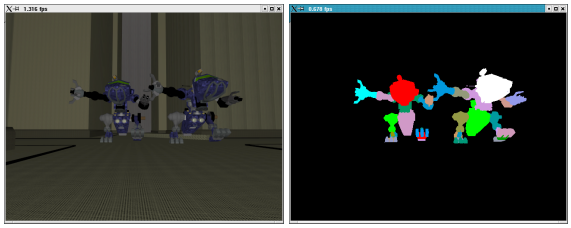


Figure 17: *One frame from the BART Robots scene (left), with color-coded objects (right). Triangles of the same color belong to the same object.*

For non-affine changes to an object (like the incoherently moving triangles in Figure 16c), the local index structure of the object must be rebuilt or updated. In this case only the parent object must be notified of any changes to the bounding box of an object and must in turn update its local index structure. Thus, modifications to the scene can be localized to only those parts of the scene that are actually affected by it, without the need to rebuild the index for the whole scene every frame [88].

The resulting speedup significantly depends on how an application organizes its scene data into individual objects. In that respect the effects are similar to OpenGL where a different scene structure — e.g. the use of display lists or the grouping of materials — can also result in significant performance differences. However, the "optimal" organization of the scene structure for OpenRT is different from the optimal organization in OpenGL. This carries strong implications for the design of future applications and scene graph libraries, at least if those should efficiently support a realtime ray tracing engine.

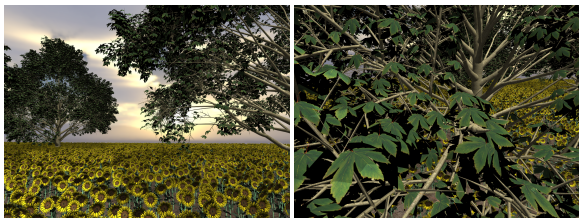


Figure 18: *Instantiation: The "Sunflowers" scene consists of roughly 28,000 instances of 10 different kinds of sunflowers with 36,000 triangles each together with several multi-million-triangle trees. The whole scene consists of roughly one billion triangles. The center image shows a closeup of the highly detailed shadows cast by the sun onto the leaves. All leaves contain textures with transparency which increase the number of rays needed for rendering a frame. The whole scene renders at roughly 7 fps on 24 dual PCs at video resolution. All objects including the sun can be manipulated interactively.*

In their implementation [88], Wald et al. use a simple flat organization of objects. Thus, their entire top-level index structure has to be rebuilt for every frame even if the bounding box of only a single object has been changed. However, Wald et al. have shown that rebuilding a top-level index structure can be performed in a few milliseconds even for thousands of instances as shown in Figure 18 [88]. Additionally, it seems likely that index creation can eventually be performed completely in hardware as the algorithms are rather simple.

As a side effect, the presented approach also allows for "instantiation", i.e. using multiple instances of the same object: Figure 18 shows a slight modification of Oliver Deussens "Sunflowers" scene, which consists of several large trees with millions of triangles each plus 28,000 instances of 10 different sunflower models with roughly 36,000 triangles each. Roughly one billion triangles are potentially visible, many of which are also partly transparent due to alpha mapped textures for the leaves. The entire scene is rendered with detailed pixel-accurate shadows that are clearly visible on the leaves of the tree.

Every object can be manipulated interactively while the scene renders at about 7 fps on 24 dual processor PCs at video resolution (see Table 2). Figure 16 shows screenshots from several animated scenes (including the BART benchmark scenes [50]) that can be rendered interactively using this technique in the Saarland RTRT/OpenRT engine.

Of course, support for dynamic scenes does not come entirely for free. While it is easy to construct cases where the method breaks completely, in practice this overhead is in the range of only a few percent (see Table 1): While overhead can result in a factor of up to two for extremely simple scenes (ERW6) without shading, it reduces to roughly 20% for the conference scene with shading. This seems to

be a reasonable price compared to the increased flexibility of handling dynamic scenes. The relative overhead is smaller with more complex scenes and more costly shading. While the technique is currently used only in the Saarland software ray tracing engine, porting it to the SaarCOR hardware is straight-forward.

In conclusion, the support for dynamic scenes in ray tracing is still limited but already good enough for a large class of applications (also see the applications in Section 8 and Part 3 that all use this technique). Furthermore, support is likely to improve as more researchers start looking at this largely ignored problem.

7. The OpenRT Interactive Ray Tracing API

Once the ability of handling dynamic scenes is combined with one of the mentioned fast ray tracing systems the essential *technical* prerequisites for realtime ray tracing on the desktop are fulfilled. However, a key issue for reaching the scenario of realtime ray tracing on the desktop is widespread application support that requires a standardized API. OpenGL [56] is used by many graphics applications and is well-known to developers. Ideally, one would simply adopt it for ray tracing, in which case any OpenGL application could transparently render its images using ray tracing.

Unfortunately, OpenGL and similar graphics APIs are too closely related to the rasterization pipeline. Their level of abstraction is too low and reflects the stream of graphics commands that is fed to the rasterization pipeline. In contrast to OpenGL, RenderMan [67] offers a more suitable high-level API that also supports ray tracing. However, it does not offer support for interactive applications.

Another option would be the use an existing high-level scene graph library such as Performer, OpenInventor, OpenSG, or others [72, 99, 61]. However, their level of abstraction is too high for a generic ray tracing API and these scene graphs libraries are too application specific. A low-level API in the spirit of OpenGL seems more appropriate and would allow for layering such scene graph APIs on top of it.

Due to the above issues with existing APIs, it became necessary to design a new API for realtime ray tracing. Ideally, such an API for realtime ray tracing should be designed with the following guidelines in mind:

- It should be as low-level as possible in order to be able to layer higher-level scene graph APIs on top of it.
- It should be syntactically and semantically as similar to OpenGL as possible, in order to facilitate porting of existing applications and for leveraging programmers' experiences.
- It should be as powerful and flexible as RenderMan for writing shaders in order not to restrict the shader writers.

Based on these observations, Wald et al. [87] and Dietrich et al. [14] have recently proposed the OpenRT API. While OpenRT has been first implemented on top of the Saarland RTRT/OpenRT system, it has been specifically designed to abstract from their distributed setup. It was designed for also driving other architectures such as the SaarCOR architecture [75] or an implementation based on GPUs.

7.1. The OpenRT Application Programming Interface

For application programming, OpenRT has been designed to be as close to OpenGL as possible, due to OpenGLs popularity and wide application support. As a rule of thumb, OpenRT offers the same calls as OpenGL wherever possible (albeit using "rt" as a prefix instead of "gl" for interface functions), and only uses different calls where a concept of ray tracing has no meaningful match in OpenGL (or vice versa).

In particular any calls for specifying geometry, transformations, and textures have identical syntax and semantics as OpenGL. This simplifies porting of applications where large parts of the OpenGL code can be reused without changes. OpenRT only differs from OpenGL in four key areas: Support for retained objects, programmable shaders, frame semantics, and handling of the resulting images.

7.1.1. Objects and Instances

The main issue with using OpenGL for ray tracing is the fact that no information is available about the changes between successive frames. In OpenGL, even unchanged display lists can be rendered differently in successive frames due to global state changes in between the frames. This however does not map well to a ray tracing engine, where such information is essential for interactive performance (see Section 6).

Instead of display lists OpenRT offers *objects*. Objects encapsulate geometry together with references to shaders and their attributes. In contrast to display lists, objects may not have any side effects. The appearance of objects can only be changed by redefining shaders that are referenced by its geometry. Objects are defined using an *rtNewObject(id)/rtEndObject()* pair. Each object is assigned a unique id that is used to instantiate it later by a call to *rtInstantiateObject(id)*. Note how this is similar to OpenGLs way of handling display lists (i.e. *glNewList(id)*,*glEndList()* and *glCallList(id)*). An instance consists of a reference to an object, together with a transformation matrix to place the instance in the scene.

In order to support unstructured motion, each object can be redefined any time by calling *rtNewObject* with the same object ID. Note that this API functionality perfectly matches the requirements of the previously proposed method to handle dynamic scenes as outlined in Section 6.

7.1.2. Shading and Shaders

In order not to be limited by the fixed reflectance model of standard OpenGL, OpenRT supports *programmable shaders* similar to RenderMan [67]. Shaders provide a flexible "plug-in" mechanism that allows modifying almost any functionality in a ray tracer, e.g. the appearance of objects, the behavior of light sources, the way that primary rays are generated, how radiance values are mapped to pixel values, or what the environment looks like. In its current version, OpenRT supports all these kinds of programmability by offering support for "surface", "light", "camera", "pixel" and "environment" shaders, respectively.

In terms of the API, shaders are named objects that receive parameters and are attached to geometry. The syntax and functionality are essentially the same as proposed in the Stanford shader API [68, 52]: Shaders are loaded and instantiated by calls to *rtShaderFile()* and *rtCompileShader()* and are bound to geometry via *rtBindShader()*. Arbitrary shader parameters can be specified by a generic *rtParameter()* call. Depending on the scope in which a shader parameter is defined, it is attached to an object, a primitive, or to individual vertices. These different ways to specify parameters allow for optimizing shaders and minimize storage requirements.

7.1.3. Semantic Differences

As mentioned before, OpenRT has been explicitly designed to be as similar to OpenGL as possible. Still, ray tracing is inherently different from rasterization, and therefore has different requirements on an API. Thus – even given the syntactical similarity – there are also certain semantic differences between OpenGL and OpenRT.

For example, OpenRT differs from the semantics of OpenGL when binding references. OpenGL stores parameters on its state stack and binds references immediately when geometry is specified. This is natural for immediate-mode rendering, but does not easily fit to a ray tracer. OpenRT instead extends the notion of identifiable objects embedding state, similar to OpenGL texture objects. However, it binds them only during rendering once the frame is fully defined.

This approach significantly simplifies the reuse of objects across frames but means that any changes to such objects might also affect the appearance of geometry defined earlier. For example, changing a shader parameter may automatically change the appearance of all triangles that this shader is bound to, even if those triangles have been specified before this call. These semantics are natural for a ray tracer but require careful attention during porting of existing OpenGL applications. More research is still required to better resolve the contradicting requirements of rasterization and ray tracing in this area.

Finally, some OpenGL functions are meaningless in the new context and consequently are not supported in OpenRT. For instance, fragment operations, fragment tests, and blending modes are no longer useful and can be better implemented using surface and pixel shaders. Traditionally ray tracing writes only a single "fragment" to each pixel in the frame buffer after a complete ray tree has been evaluated. Thus the usual ordering semantics of OpenGL and its blending operations that are based on the submission order of primitives are no longer meaningful.

Instead of writing the pixels to a hardware frame buffer OpenRT returns them in buffers provided by the application. This, however, is only due to the current hardware setup that uses a software implementation, and is likely to change for more dedicated ray tracing hardware.

For writing shaders OpenRT currently provides a simple C/C++ interface that is close modeled the RenderMan shader API [67] in terms of semantics and names of functions. A separate shading language compiler for RenderMan or some other language would help programmers already familiar with these languages and would allow for better optimizing the shader code.

OpenRT has been implemented as a separate but closely related API changing the prefix of functions from "gl" to "rt". This allows applications to simultaneously make use of both APIs. Eventually it should be possible to port the changes in OpenRT to true OpenGL extensions. The main issue in such an implementation would be the proper handling of frame semantics.

The OpenRT API has been used exclusively in all examples and applications throughout this paper. That includes a VRML-97 browser [6] that has been ported from OpenGL to the new API (see Section 7.3). Using the OpenRT API allows the application to completely abstract from the underlying implementation. Currently, both a single-CPU version as well as a distributed version are available and can be exchanged without the application even noticing it. The obvious next step consist in evaluating the practicability of using the OpenRT API to also drive other hardware platforms such as the SaarCOR architecture [75] or a GPU-based ray tracing engine [69] (see Sections 4 and 3, respectively)

7.2. A Simple Example

After having outlined the basic concepts of the OpenRT API here is a simple example of an OpenRT program.

Though this is but a very simple example of an OpenRT program, it already uses most of the above discussed concepts: Loading and using shaders, building objects, and multiple instantiation. Though 'real' OpenRT programs can be quite more complex, they all follow the same pattern.

Note that we have omitted the code for the shader, which in this case is trivial. The shader is written in C++ (with an API similar to RenderMan), complied to a shared library ("libPhong.so"), and loaded from the application.

```
// EightCubes.c:
// Simple OpenRT example showing
// eight rotating color cubes
#include <rtut/rtut.h>
#include <openrt/rt.h>

#define PHONG_ID      0
#define PHONG_FILE_ID 1

RTint createColorCubeObject()
{
  // Create an object for our
  // vertex-colored cube

  // Step1: Load the shader
  srtShaderFile(PHONG_FILE_ID,
          "Phong","libPhong.so");
  srtCreateShader(PHONG_ID);
  srtBindShader(PHONG_ID);

  // Step2: Define the object
  RTint objId = rtGenObjects(1);
  rtNewObject(objId, RT_COMPILE);
   // Step3: Issue geometry
    rtBegin(RT_POLYGON);
      rtColor3f(0, 0, 0);
      rtVertex3f(0, 0, 0);
      rtColor3f(0, 1, 0);
      rtVertex3f(0, 1, 0);
      rtColor3f(1, 1, 0);
      rtVertex3f(1, 1, 0);
      rtColor3f(1, 0, 0);
      rtVertex3f(1, 0, 0);
    rtEnd();
    // other cube sides ...
  rtEndObject();
  return objId;
}

int main(int argc, char *argv[]) {
  // Init, open window, etc.
  rtutInit(&argc, argv);
  rtutInitWindowSize(640, 480);
  rtutCreateWindow("Eight Cubes");

  // set Camera
  rtPerspective(65, 1, 1, 100000);
  rtLookAt(2,4,3, 0,0,0, 0,0,1);

  // generate object *once*
  objId = createColorCubeObject();
  for (int rot = 0; ; rot++) {
    // re-instantitate object
    // for every frame
    // with different transformation
    rtDeleteAllInstances();
    for (int i=0; i<8; i++) {
      int dx = (i&1)?-1:1;
      int dy = (i&2)?-1:1;
      int dz = (i&4)?-1:1;
      // position individual objects
      rtLoadIdentity();
      rtTranslatef(dx,dy,dz);
      rtRotatef(4*rot*dx,dz,dy,dx);
      rtScalef(.5,.5,.5);
      rtInstantiateObject(objId);
    }
    rtutSwapBuffers();
  }
  return 0;
}
```

After opening a window, the “main” function first generates a vertex-colored RGB cube with a shader that just displays the interpolated vertex color. Afterwards, the “for”-loop creates eight rotating instances of this cube by re-instantiating each of the eight instances with a different transformation in subsequent frames.

Being similar to OpenGL, this example should be easy to understand – and extend – by any slightly experienced OpenGL programmer. Of course, this is but a very simple example, and real programs will be considerably more complex. For example, a real program also has to load textures, light shaders, parameterize the shaders, etc. Still, using advanced ray tracing effects in OpenRT is significantly simpler than generating the same effect in an OpenGL program: For example, rendering a scene once with global illumination effects and once without only requires to load a different shader – e.g. changing the shader name in “srtShaderFile” from “Phong” to “InstantGlobalIllumination” (see Part 3) – without having to touch any other code in the program.

7.3. Extended VRML Browser with Ray Traced Effects

In order to demonstrate that OpenRT can also be used for realistically complex applications, we have ported a VRML-97 browser [6] from OpenGL to OpenRT. Using the OpenRT API allows the browser application to completely abstract from the underlying ray tracing implementation.

While the browser supports the full VRML97 specification – including VRML animations and even with a scripting interface – the ported browser also supports typical ray

Figure 19: *A VRML-97 browser with ray traced effects. Left: A typical VR application, showing a car model with shadows and physically-correct reflections even on curved surfaces. Right: A VRML-97 animation with an interactive global illumination shader.*

traced effects. By extending the VRML "`Appearance`" node with a "`shader`" entity, any ray tracing shader can be assigned to any of the VRML primitives. This allows the VRML browser to also compute shadows and reflections, and even to render standard VRML scenes with full global illumination (see Figure 19).

In summary, OpenRT is a simple yet flexible API for realtime ray tracing. It is simple to use, flexible enough to support all typical ray tracing effects, and similar enough to OpenGL (for application programming) and RenderMan (for writing shaders) to be easy to learn even for novice ray tracing users.

8. Applications and Case Studies

Based on the previously outlined advances in ray tracing, the scenario of ubiquitous realtime ray tracing draws closer. In this section, we will briefly summarize the state-of-the-art and thereby give an overview of some of the most obvious future applications of realtime ray tracing.

Interactive Visualization of Complex Models

One of the most obvious applications of ray tracing is visualizing massively complex models. Since complexity of ray tracing is logarithmic in scene size, we can efficiently ray trace scenes with tens of millions of individual triangles. This allows us to render complex objects such as the 12.5 million triangle UNC "PowerPlant" scene without the need for geometric simplifications (see Figure 20). Note that the technique proposed in Section 6 even allows interaction with the power plant by moving different parts of the model.

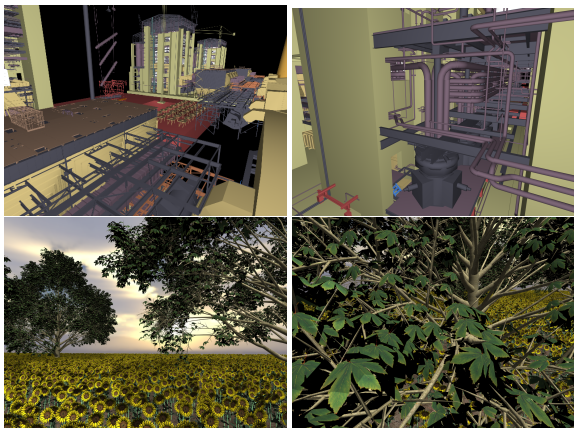


Figure 20: *Complex models: a) Three power plants of 12.5 million individual triangles each, rendering interactively at 23 fps. b) A closeup on the highly detailed geometry. c) An outdoor scene consisting of roughly 28,000 instances of 10 different kinds of sunflowers with 36,000 triangles each together 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.*

Additionally, instantiation allows to further increase scene complexity. For example, the "Sunflowers" scene (see Figure 20) contains 28,000 instances of several kinds of sunflowers, plus several highly detailed trees, totalling one *billion* triangles. Note that even a single tree in this scene contains more than a million individual triangles.

Note, that no simplifications or level-of-detail have been used in these scenes but that the original geometry has been rendered directly. This eliminates costly preprocessing and avoids any artifacts. Also, the geometric complexity does not limit the kinds of effects that can be computed. For example, the sunflowers scene is rendered including shadows and transparency from semi-transparent plant leaves. Of course, the objects and the sun can be moved around interactively. Note that advanced effects can also be computed in the power plant, as can be seen in Figure 26.

Plug 'n Play Shading

The second obvious advantage of ray tracing is the ability to support plug'n play shading: Shaders for certain specialized effects can be written independently of all other shaders, and can still automatically and seamlessly work together with the rest of the scene. For example, Figure 21 shows a typical office scene with several advanced shaders, like procedural wood and marble shaders, procedurally bump-mapped reflections on the mirror, reflections on the metal ball, and even shaders performing light field [48] and volume rendering [62].

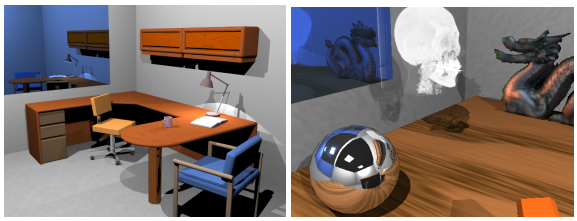


Figure 21: *Easy combination of different shading effects. Left: A typical office scene with correct shadows and reflections, and with programmable procedural shaders. Right: The same scene with additional volume and light field objects. Note how the volume casts transparent shadows, the light field is visible through the bump-mapped reflections on the mirror, etc.*

Increasingly Realistic Games

The ability to support advanced shading effects is especially interesting for game applications. Games can profit from the ability to compute advanced visual effects that can tremendously increase the realism and better immerse the player in

a game scene (see Figure 22). Eventually, it may be possible to render even games with global illumination effects for the ultimate realism is combined with detailed geometry.

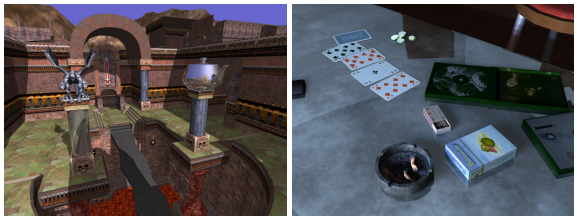


Figure 22: *Increased realism in games: Ray traced shadows and reflections in two typical game environments.*

Physically Correct Lighting Simulation

Of course, ray tracing not only allows shaders for nice images, but also supports for physically-correct visualization. Benthin et al. [7] have shown how interactive ray tracing can be used for a quantitative simulation of the reflection and refraction behavior of a car headlight (see Figure 23).

Using the Saarland RTRT/OpenRT engine together with appropriately written shaders allowed for simulating up to 25 levels of reflection and refraction even in an 800,000 triangle model at interactive rates.

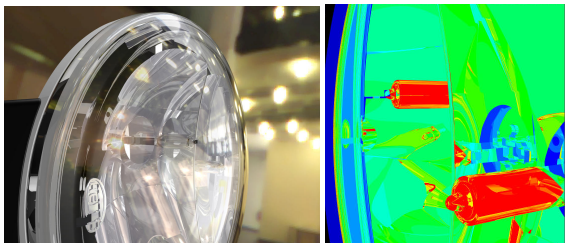


Figure 23: *Interactive Simulation of Reflection and Refraction in a car Headlight. Left: The headlight model consists of 800,000 triangles, and renders interactively with up to 25 levels of reflection and refraction. Right: False-color image visualization of number of reflection levels per pixel (black: 0, red: 25+).*

Similarly, ray tracing could be used for interactive simulation of reflections in dashboards [82, 83] or in car paint (see Figure 19). This kind of realism is especially important for industrial applications that depend on reliable and quantifiable results.

Interactive Global Illumination

Finally, ray tracing allows for interactive global illumination, i.e. the physically-correct simulation of light transport at interactive rates (see Figures 19 and 26). This will be covered more closely in Part 3.

9. Implications for Future Application

These example applications clearly demonstrate the potential of using ray tracing for practical applications. Note that the above examples run interactively on the Saarland RTRT/OpenRT engine even today. It seems very likely that the technology for these applications may be commonly available in the near future, which would offer tremendous opportunities for future graphical applications.

For example, it may soon be common to have effects like shadows, reflections, and refractions in many 3D applications. Even effects like global illumination might eventually become a standard feature of applications like virtual reality, or games. Furthermore, the potential to move from purely "good-looking" applications to applications producing reliable and verifiable results opens up completely new possibilities for the simulation, CAD, Virtual Reality, and design industry.

However, even though this potential for future applications is huge, there are still several issues that have to be solved: For example, using ray tracing for interactive applications is inherently different from the way that interactive applications work today. This bears several implications on the way that applications have to be designed.

One such implication is the design of scene graph libraries: Today, most applications are built on scene graphs like VRML [85, 6], Performer [72], OpenInventor [99], OpenSG [61], etc., all of which are designed to best match the capabilities of OpenGL. For example, most scene graphs are organized into groups of triangles with the same material, in order to minimize OpenGL state changes. In a ray tracer, this is not necessary and can even be counter-productive — instead of organizing a scene graph into groups of triangles with the same material, triangles should better be grouped depending on the way they are moving in a scene (see Section 6). Similarly, todays scene graphs often do not care which objects are static. Essentially, a scene graphs "cost function" is different for ray tracing and rasterization.

Another issue is that many of the advantages of ray tracing are simply not supported by todays applications and scene graph. For example, scene graphs whose material descriptions offer reflection coefficients or refraction indices simply have not been adopted yet. Essentially the same is true for complex models: While the ray tracer can easily handle millions of polygons, most applications available today are too heavy-weight to even load such models.

In order to fully exploit the above-mentioned potential of realtime ray tracing, future applications and scene graphs have to be designed with these issues in mind. Though this problem is neglected by many researchers and developers, it is an essential prerequisite for widespread use of realtime ray tracing.

PART THREE

Using Realtime Ray Tracing for Interactive Global Illumination

Even though classical ray tracing considers only direct lighting effects it already allows for highly realistic images that make 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 as a standard feature of 3D graphics.

Global illumination algorithms account for the often subtle but important effects of indirect illumination in a physically-correct way [12, 17] 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.

As traditional ray tracing has historically been too slow even for interactively ray tracing images with simple shading, the full recomputation of a global illumination solution has been practically impossible. Now, with the advent of realtime ray tracing, it should become possible to also compute full global illumination solutions at interactive rates.

10. Alternative Approaches towards Interactive Global Illumination

There are two different approaches to global illumination: Point-sampling techniques (ray- or path-based) [38, 44, 45, 84, 37], and finite-element-based techniques [29, 12, 78, 5]. While point-sampling techniques usually provide higher quality, finite element techniques have the advantage that the discretization into polygonal patches can directly be used for display via commonly available graphics hardware.

10.1. Radiosity Based Approaches

Thus, the first approaches to interactive global illumination have been based on finite-element techniques using rasterization hardware to display the computed finite element solution. In its most trivial form, this consisted of a simple interactive display of a precomputed radiosity solution of the scene. However, relying on precomputated radiosity values only allows for simple walkthroughs, as any interactive change to the environment would require recomputing the radiosity solution. Except for trivial scenes, recomputing the whole solution from scratch every frame is not feasible.

Interactive Radiosity using Line Space Hierarchies

In order to avoid this full recomputation, Drettakis and Sillion [16] have proposed to incrementally update a radiosity solution using a line space hierarchy. This hierarchy is generated by augmenting the hierarchical radiosity links with "shafts", each of which represents all the lines that pass through the two connected hierarchy elements. Traversing this data structure then allows to easily identify the links that are affected by a dynamic change to the scene, and thus allows to quickly update the radiosity solution. Additionally, it simultaneously allows to clean up subdivisions in the hierarchical radiosity solution that are no longer required any more after an update to the scene (e.g. a for representing a disappearing shadow border).

However, the algorithms are quite complex. Additionally, like all radiosity systems the proposed system is limited to diffuse light transport, suffers from tesselation artifacts, and does not easily scale to complex geometries.

Instant Radiosity

In 1997, Keller presented a totally different approach to interactive radiosity. In "Instant Radiosity" [39], a small set of virtual point lights (VPLs) is computed using a quasi random walk from the light sources. These VPLs are then used to illuminate the scene using a shadow algorithm. In instant radiosity, most computations (including the shadow generation) can be performed on the graphics card. For moderately complex scenes, this allows for fully recomputing lighted images at interactive rates. Additionally, instant radiosity avoids many of the typical tesselation artifacts: Instead of discretizing the geometry, instant radiosity performs discretization by using only a small number of discrete virtual point light positions. However, relying on rasterization graphics hardware, instant radiosity is limited to purely diffuse scenes and lacks performance in realistically complex scenes.

10.2. Approximating Techniques

Though radiosity based systems already interactively render dynamic scenes with indirect lighting, they all suffer from similar problems: First, they are rather slow and often do not scale to reasonably complex scenes. Additionally, radiosity based systems are inherently limited to purely diffuse light transport, which often gives them a rather 'flat' and unrealistic appearance.

To avoid these limitations, most off-line global illumination systems use ray- and path-based techniques. These, however, usually require tracing of hundreds of rays for each pixel, which is not affordable at interactive rates. Interactivity can still be achieved by using approximating techniques. The basic idea behind these techniques is to approximate the full solution (e.g. by computing only a fraction of all pixels) and using a separate display thread for interactively reconstructing an image from the this information.

As the display process usually runs much faster than the sampling process (usually 10 – 100 times as fast), the display thread has to cope with coarse approximations of the image, which often results in artifacts.

Render Cache

One of the first systems to use this approach was Walter et al.'s "Render Cache" [96, 95]. The render cache stores a cache of previously computed illumination samples – roughly twice as many as pixels in the image. For each new frame, these are reprojected to the new camera position, and stored in the frame buffer. As this can result in artifacts (e.g. holes in the image or disocclusion) several heuristics have to be applied to reduce such artifacts.

The rendering thread runs asynchronously and decoupled from the display and reconstruction thread, and simply generates new illumination samples as fast as possible. Such generated samples are then inserted into the render cache (thereby replacing some old samples) and can be used for future frames. The main limitation of the render cache is the speed with which it can reproject the old samples and reconstruct the new frame. The high cost for these operations – which is usually linear in both frame rate and number of pixels – has limited the original system to only moderate resolutions. Though recently proposed optimizations [95] allowed for significant speedups, generating full-screen images is still too costly for realtime frame rates.

The render cache becomes beneficial for rendering algorithms where the cost for computing a new sample is very high. For a fast ray tracer with a simple illumination model it is often cheaper to simply trace a new ray than reprojecting and filtering old samples. Similar techniques have also been proposed in the form of Wards Holodeck [47], and Simmons et al.'s Tapestry [77] system.

Shading cache

A similar approach has recently been proposed by Tole et al [81]. Instead of using an image-based technique for reconstructing the image from previously computed samples, they use an object-based technique that uses triangles for representing the illumination samples. Image reconstruction and interpolation between shading samples can then be performed using graphics hardware, which allows to produce realtime frame rates at full-screen resolutions.

However, their approach suffers from similar problems as the render cache: Depending on old (and thus potentially outdated) samples results in disturbing delays until an change to the scene has any effect: While some geometric object can be moved in realtime, the global effects caused by this object – e.g. its shadow – will be computed much slower. They will "follow" the object with an objectionable latency of several seconds or more and show visible artifacts during the transition phase.

This makes it hard to use the shading cache in totally dynamic scenes with constantly changing illumination from dynamic lights, scenes, and materials. Finally, the shading cache requires as least one sample per visible polygon, and thus is not suitable for highly complex scenes [4].

Edge-and-Point Images (EPIs)

The newest variant of this idea of interactively generating high-quality images based on using sparse samples has recently been proposed by Bala et al. [4]. The "edge and point image" (EPI) essentially is an extension of the render cache (Section 10.2) that uses – and preserves – analytic discontinuities during reconstruction of the image.

The EPI analytically finds the most important shading discontinuities – in its current form these are object silhouettes and shadow boundaries – using efficient algorithms and data structures, and then respects these discontinuity edges during reconstruction. This allows for high-quality reconstruction even including anti-aliasing at interactive rates.

10.3. Hybrid Techniques

To avoid the qualitative limitations of pure radiosity approaches, several approaches have been undertaken to augment radiosity solutions with specular effects such as reflections and refractions, for example by using corrective textures [79] or corrective splatting [27]. These techniques are similar to the above-mentioned subsampling-approaches, and have already been covered in greater detail in the previous STAR on interactive ray tracing, see [92].

Fast Global Illumination including Specular Effects

Another interesting way of combining radiosity with specular effects has been proposed by Granier et al. [24]. They augment a radiosity solution with specular effects – like e.g. caustics – using particle tracing. In its core, their system uses hierarchical radiosity [29] with clustering [78] for quickly and efficiently computing the diffuse light transport. Non-diffuse effects are computed by integrating particle tracing into the gather step of hierarchical radiosity. Particle tracing is performed only where necessary, by shooting particles only over links that connect to specular surfaces.

In simple scenes, their system allowed for interactive viewing of up to 2 frames per second for global diffuse illumination and caustic effects. However, interactive viewing required using graphics hardware, thereby limiting the system to diffuse plus caustics only. Reflection and refraction could not be supported in interactive mode resulting in a "dull" appearance [24]. Though combining their technique with the render cache [96] (cf. Section 10.2) allowed to push frame rate and add reflections and refractions, the render cache itself introduced other artifacts.

Selected Photon Tracing

A different way of interactively computing global illumination solutions has recently been proposed by Dmitriev et al [15]: In "selective photon tracing", radiosity values of triangular patches are computed by shooting photons from the light sources into the scene and depositing their energy in the

vertices of the triangular mesh. The lighted triangles can then be displayed interactively with graphics hardware. To hide some of the tesselation artifacts, direct illumination from point lights is computed separately on the graphics hardware (see Figure 24).

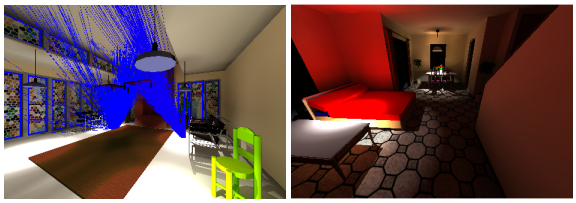


Figure 24: *Selected photon tracing. Left: Once some pilot photons have detected a change in the scene geometry, several similar photons (in blue) are selectively retraced. Right: An example frame while interactively changing the scene.*

Due to the limited speed of the photon tracer, only a certain amount of photons can be retraced every frame. Therefore, selected photon tracing uses a clever way of determining which photon paths have probably been affected by a scene update: A small subset of photon paths (called "pilot photons") is shot into the scene to determine which parts of a scene have been updated. If a pilot photon hits an object that it did not hit in the previous frame (or vice versa), selected photon tracing selectively re-traces similar photons by generating only photon paths that are similar to the original path (see Figure 24). Similar photons are generated by exploiting correlation in the quasi random number generator used for determining the photon paths.

Being a hybrid of radiosity and subsampling based techniques combines many of the advantages of both techniques, and allows to interactively display scenes with global illumination while allowing for interactive manipulations to the scene.

However, selected photon tracing also inherits some of the inherent problems of radiosity and subsampling: First, the speed decreases linearly in the number of patches, allowing only a rather coarse tesselation that does not capture high-frequency details such as caustics or indirect shadow borders. Second, as only a subset of all photons is re-traced every frame, drastic changes in the illumination caused by user interaction can take several seconds to take effect.

11. Using Realtime Ray Tracing for Interactive Global Illumination – Issues and Constraints

All the approaches discussed above had to be undertaken because compute power and ray tracing performance did not suffice to compute full ray tracing based global illumination solutions at interactive rates. With the recent availability of realtime ray tracing, however, it should eventually become possible to compute such images interactively: As most global illumination algorithms spend most of their time tracing rays, combining such an algorithm with a realtime ray tracer should theoretically result in interactive global illumination performance.

However, global illumination algorithms are inherently more complex than classical ray tracing and thus not all algorithms will automatically benefit from a much faster ray tracing engine. In order to take maximum profit from the availability of fast ray tracing, appropriate global illumination algorithms have to be designed to meet several constraints:

Parallelism: Future realtime ray tracing engines will exploit the inherent parallelism of ray tracing. For classical ray tracing, parallelism can be exploited easily by computing pixels separately and independently. Many global illumination algorithms, however, require reading or even updating global information, such as the radiosity of a patch [12], entries in a photon map [37], or irradiance cache entries [98]. This requires costly communication and synchronization overhead between different ray tracing 'units', which quickly limits the achievable performance.

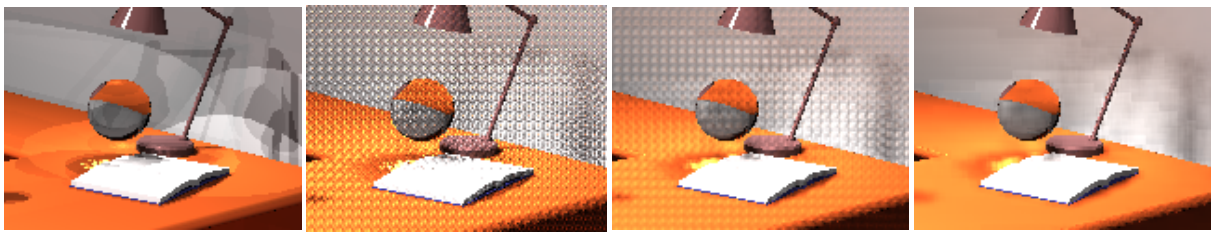
Efficiency: Even the ability to shoot millions of rays per second leaves a budget of only a few rays per pixel in order to stay interactive at non-trivial frame resolutions. Thus, an algorithm must achieve sufficiently good images with a minimum of samples per pixel. Given the performance of current realtime ray tracing engines, only an average of about 50 rays per pixel is affordable. Thus, the information computed by each ray has to be put to the best possible use.

Realtime capabilities: For real interactivity – i.e. arbitrary and unpredictable changes to the scene made by a user (including changes to geometry, materials, and light sources) – algorithms can no longer use extensive preprocessing. Preprocessing must be limited to at most a few milliseconds per frame and cannot be amortized or accumulated over more than a few frames as the increased latency of lighting updates becomes noticeable.

Focus on Ray Tracing: The availability of a realtime ray tracing engine can only save time previously spent on tracing rays. Thus, performance must not be limited by other computations, such as nearest-neighbor queries, costly BRDF evaluations, network communication, or even random number generation.

Independence From Geometry: In order to fully exploit the ability of ray tracing to scale to complex geometries (see Section 2.1), the global illumination algorithm itself must be independent from geometry, and may not store information on individual patches or triangles.

Within the above constraints most of todays global illumination algorithms cannot be implemented interactively on a realtime ray tracing engine: All radiosity style algorithms [12, 29, 78, 16, 24] require significant preprocessing of global data structures which seems impossible to implement under these constraints. In principle, it should be possible to use the ray tracer for augmenting one of the above interactive



a)no interleaved sampling no discontinuity buffer | b)5×5 interleaved sampling no discontinuity buffer | c)5×5 interleaved sampling 3×3 discontinuity buffer | d)5×5 interleaved sampling 5×5 discontinuity buffer

Figure 25: *Interleaved sampling and the discontinuity buffer: All close-ups have been rendered with the same number of rays apart from preprocessing. In a) only one set of point light sources and caustic photons is generated, while for b)-d) 25 independent such sets have been interleaved. Choosing the filter size appropriate to the interleaving factor completely removes the structured noise artifacts.*

radiosity systems with specular effects and accurate shadows in the spirit of [94]. This, however, would not easily fit into the distributed framework that the ray tracer is running on, and would still suffer from tesselation artifacts.

Pure light-tracing or path-tracing [38] based approaches benefit most from fast ray tracing, but usually suffer from noise artifacts in the spatial and temporal domains. The latter is particularly objectionable in animations and interactive systems. Decent image quality would require far too many rays per pixel at least for non-trivial lighting conditions. Finally, photon mapping [35, 34, 36, 37] requires costly preprocessing for photon shooting and creation of the kd-trees as well as expensive nearest neighbor queries during rendering. It also uses irradiance caching, which imposes similar problems.

12. Instant Global Illumination

The above discussion shows that a new, appropriately designed algorithm is required to take advantage of a fast ray tracing engine for interactive global illumination computations: Such an algorithm has recently been proposed by Wald et al. [91]: Their Monte Carlo based algorithm is explicitly designed to run efficiently under the constraints mentioned above.

In its core, their method – called "Instant Global Illumination" – builds on Keller's "Instant Radiosity" [39] (see above). The illumination in a scene is approximated by a small set of "virtual point lights" (VPLs) that are generated by a brief preprocessing step by performing a few random walks from the light sources. During rendering the irradiance at a surface point is computed by casting shadow rays to all VPLs and adding their contribution.

Instant Radiosity as a core algorithm perfectly fits the above restrictions: Instant Radiosity spends most of its time evaluating visibility from point light sources, which generates highly coherent rays (allowing the ray tracer to perform best), and profits perfectly from increased ray tracing speed. Additionally, it has minimal preprocessing cost (generating only a few dozen VPLs), does not need access to global data, is totally independent of geometry, and parallelizes trivially per pixel.

Being implemented on top of a realtime ray tracing system allows Instant Global Illumination to easily and cheaply integrate important specular effects like reflections and refractions, which have traditionally been a problem in fast global illumination. Even simple caustics can be supported by a simplified version of caustic photon mapping [91], but usually lead to scalability and performance problems due to the higher preprocessing cost for shooting the photons.

In order to remain interactive, only a small number of shadow rays to VPLs is affordable in each pixel. Therefore, instant global illumination uses randomized Quasi Monte Carlo sampling [42] and a combination of Interleaved Sampling [40] together with an image based filtering technique in order to get sufficient image quality at such small sampling rates (also see Figure 25): Instead of using the same set of n VPLs for every pixel, the algorithm generates 3×3 such sets (of n VPLs each), which are interleaved between pixels: Neighboring pixels use different sets, and the same set is used every 3×3 pixels. Using 3×3 different sets of VPLs effectively increases the number of VPLs used for computing an image by a factor of 9. This improves the visual appearance of the image, at the expense of getting structured noise in the image, as can be seen in Figure 25b.

This structured noise is then removed by "discontinuity buffering", a technique that removes structured noise by filtering the image. It uses several heuristics to avoid filtering across image discontinuities. If the filter size is chosen to exactly match the size of the interleaving pattern the structured noise can be entirely removed (see Figure 25b–d). However, some structured noise can remain in areas where the heuristics do not permit filtering, e.g. on highly curved surfaces.

The combination of interleaved sampling and discontinuity buffering improves the performance by roughly an order

of magnitude and achieves interactive frame rates even if all illumination is recomputed every frame.

12.1. Distribution Design Issues

In most aspects, implementing Instant Global Illumination on top of a distributed ray tracing system is rather simple. Most computations are spent on shooting highly coherent shadow rays, which perfectly fits the underlying ray tracing engine, and which can be easily implemented as a programmable shader. The most critical design decision for a distributed implementation is to decide where the discontinuity buffering is to be performed.

If this final filtering pass is performed on the final image (i.e. on the server), it is possible to assign only pixels with the same interleaving pattern to each client. Thus, each client only has to generate the VPLs and photons for its respective interleaving set. If this preprocessing phase is quite costly – e.g. if several thousand rays have to be shot for producing convincing caustics – this efficiently avoids replicating the preprocessing cost on all clients, and significantly improves scalability. Essentially, each client only has to compute one ninth of all VPLs and photons. However, this strategy implies that clients do not have access to neighboring pixel values – as these would have to be computed by different interleaving sets – and that therefore filtering has to be performed on the server. This however can create a scalability bottleneck, as the server can only receive and filter a quite limited number of pixels per second.

The alternative strategy is to perform the filtering on the clients. This effectively avoids the server bottleneck, but requires clients to have access to neighboring pixels, thereby incurring some overhead at tile boundaries, and – even worse – requiring each client to generate all VPLs and photons for all interleaving patterns. While generation of the VPLs is cheap enough to be replicated on each client, generating a sufficient number of caustic photons is no longer affordable in this strategy.

12.2. The Original Instant Global System

Therefore – in order to be able to compute caustics – the original IGI system [91] was performing filtering on the server. As could be expected, this created a bottleneck that limited performance to roughly 5 frames per second at 640×480 pixels. However, the original system still scaled very well in image quality: Using twice as many CPUs, allows for twice the number of VPLs maintaining the same frame rate. The increased number of VPLs considerably improving the resulting image quality.

Filtering on the server also allowed this system to generate reasonably good caustics (see Figure 27). Using a simple hashing scheme avoided the costly construction of kd-trees every frame, and provided fast and cheap photon queries.

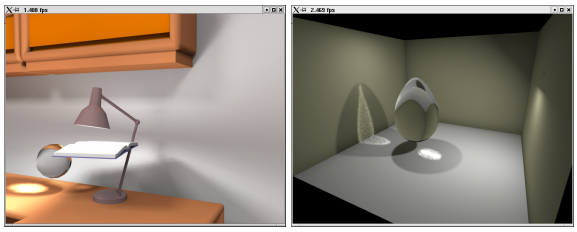


Figure 27: *Caustics in the original Instant Global Illumination system. Though the limited number of photons results in some artifacts, caustics are still convincing.*

Even though only a few thousand caustics have been affordable for interactive use, interleaved sampling and discontinuity buffering also increased the quality of the caustics.

However, the original system was mainly designed as a proof-of-concept system, and did not exploit the full performance of the ray tracer. For example, it was explicitly designed to generate coherent rays to point light sources, the original implementation did not yet use the fast SSE packet traversal code for tracing these rays. Thus, the performance of the original system was quite limited, therefore allowing only a rather small number of VPLs at interactive rates. As this resulted in visible artifacts in the image, the IGI system also allowed to progressive refine the quality of an image as soon as user interaction stopped. The refinement process then progressively increased the number of VPLs, caustic photons, and samples per pixel, thereby eventually resulting in high-quality, artifact-free, and anti-aliased images after only a few seconds of convergence.

13. Advanced Instant Global Illumination

However, due to the limited performance of the original system, this image quality could not be maintained during user interaction. Recently, Benthin et al [8] have proposed several important improvements that now allow for improved image quality and significantly higher performance.

13.1. Improved Performance and Programmable Shading

First, the system has been completely rewritten to function in a streaming manner. As all the rays shot in IGI are highly coherent shadow rays to point light sources, packets of coherent rays can be easily maintained most of the time, thereby allowing to trace these rays with the fast SIMD code as described in Section 2.1.

Furthermore these packets all undergo similar shading computations. Thus they can also be shaded with SIMD computations without the need to break them up for separate shaders, and thus without the penalty outlined in Section 2.1. This is even true for complex procedural shaders

Figure 26: *Instant Global Illumination. a.) Office scene, with soft shadows, reflections, refractions, and a caustic from the glass ball. b.) Conference scene with 280,000 triangles and 220 light sources, c.) Global Illumination in an animated VRML scene, and d.) Global Illumination in the 12.5 million triangle "power plant" scene. All scenes run interactively with several frames per second at* 640*x*480 *pixels while allowing interactive updates to the scene. Note that a.) was rendered with the older system, as caustics are currently not supported in the new version.*

like the "wood", "marble", and "brickbump" shaders that can be seen in Figure 28.

These speed improvements – together with the recent improvements in ray tracing performance mentioned in Section 2.1 – now allow the new system to significantly outperform the old system, while simultaneously achieving much higher image quality and higher frame rates.

However, with the much higher performance of the clients the scalability bottleneck on the server could no longer be tolerated. Therefore, the filtering computations have also been moved to the clients, where they can now be computed using fast SIMD code. This completely removes any computational load from the server but does no longer allow for caustic photon mapping. With the server bottleneck removed the server can concentrate on load balancing which allows the system to easily scale to more than 48 CPUs and to frame rates beyond 20 frames per second (see Figure 29). Currently, the systems performance is mainly limited by the bandwidth of Gigabit Ethernet, which cannot transfer more than roughly 25–30 frames per second at 640×480 pixels.

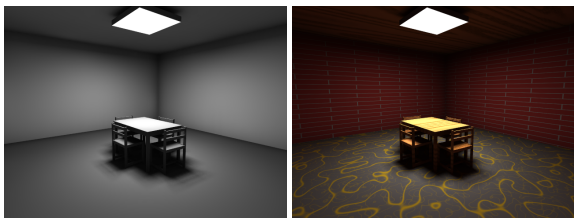


Figure 28: *Freely programmable procedural shading in a globally illuminated scene. The standard "ERW6" test scene (left) and after applying several procedural shaders (marble, wood, and brickbump). Even with shaders that make extensive use of procedural noise the performance only drops to 3.7 fps compared to 4.5 fps with a purely diffuse BRDF.*

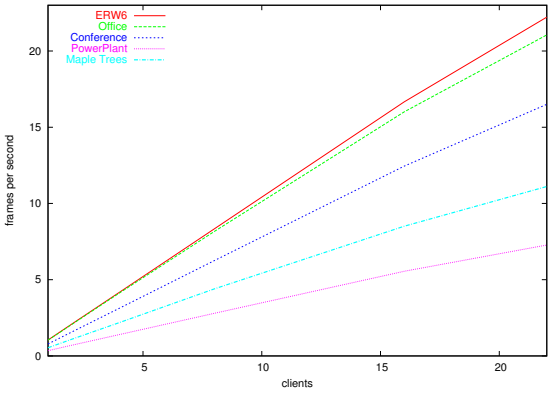


Figure 29: *Scalability of the new IGI system with the number of rendering clients: Performance is essentially linear up to 24 PCs/48 CPUs. This applies to scenes ranging from several hundred triangles (ERW6) up to the power plant with 50 million triangles (four instances). Also note how the new system scales in frame rate well beyond the original system, which was limited to at most 5 frames per second.*

13.2. Efficient Anti-Aliasing by Interleaved Super-Sampling

The original system provided high-quality anti-aliasing using progressive over-sampling in static situations but suffered from artifacts during interaction. This was caused by the low image resolution and the fact that only a single primary ray was used per pixel.

Efficient anti-aliasing is still an unsolved problem in ray tracing as the rendering time increases linearly with the number of rays traced. Anti-aliasing by brute-force super-sampling in each pixel is thus quite costly, in particular for an interactive context. On the other hand, methods like adaptive super-sampling are problematic due to possible artifacts and the increased latency of refinement queries in a distributed setup.

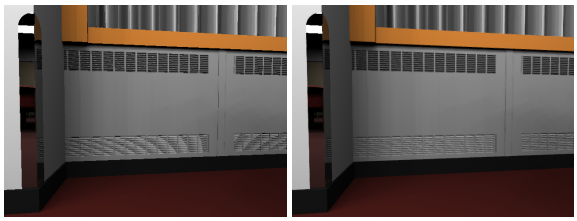


Figure 30: *Efficient anti-aliasing. Left: A single primary ray per pixel, exhibiting strong aliasing artifacts. Right: 4 primary rays per pixel, resulting in an aliasing-reduced image. Using interleaved super-sampling the performance of the anti-aliased image is only slightly lower than the left image, running at 3.2fps compared to 4.0 fps. As both images use the same total number of shadow rays per pixel, the quality of the lighting simulation is virtually indistinguishable.*

Benthin et al. [8] have also proposed an extension of instant global illumination that achieves anti-aliasing with little performance impact using a similar interleaving approach as for sampling the VPLs. Instead of connecting each primary ray to all M VPLs in the current set, the VPLs are grouped into N different subsets with roughly M/N VPLs each. We then use N primary rays per pixel for anti-aliasing, each ray computing illumination only with its own subset of VPLs.

For typical parameters (i.e. $N = 4$ or $N = 8$, and $M >= 16$, the overhead of the $N-1$ additional rays is usually in the order of 20% to 30%, which is well justified by the increase in image quality (see Figure 30).

14. Instant Global Illumination in Complex and Highly Occluded Scenes

In the way discussed above, instant global illumination was mainly designed for "typical" global illumination scenes, i.e. single rooms or scenes containing only a few different rooms. In practice , however, application often have to render complete buildings, construction sites, ships, or airplanes. Such kinds of scenes are usually highly complex, contain many different light sources and show a high degree of occlusion.

Such scenes, however, are hard to handle for virtually all of todays global illumination scenes: Geometric complexity of several million triangles alone is often an important feature for many algorithms, especially if they require storing illumination information on the triangles themselves. Additionally, most algorithms are linear in the number of light sources, thereby being unable to handle thousands of light sources at interactive rates. Finally, high occlusion requires most algorithms to waste most of their computations on unimportant and occluded parts of a scene, e.g. by tracing photons in rooms that are occluded anyway, or by wasting shadow rays for sampling occluded lights while computing direct illumination.

In its original form, instant global illumination suffers from similar problems: Being entirely based on ray tracing, the pure geometric complexity of the scene is generally less of a problem. However, the generation of the VPLs is purely light driven, resulting in the VPLs to be scattered all over the model. As described above, only a rather limited set of VPLs is affordable at interactive rates (roughly in to order of 30–100). If this small number of VPLs is distributed over dozens or hundreds of rooms, the lighting in each room will be represented by only very few VPLs, resulting in low image quality. In unlucky cases, it may even happen that a room receives no VPL at all (e.g. see the Soda Hall images in Figure 31). This, however, is not a specific limitation of the Instant Global Illumination method, but happens in similar form in virtually all other global illumination algorithms used today.

Methods to handle such kinds of scenes have been proposed by Ward [97], Jensen et al. [34], Keller et al. [41] and Bala et al. [18]. However, these approaches are either applicable only to direct illumination, or do not easily fit into a distributed interactive environment. For a more detailed discussion, see [89].

Recently, Wald et al. have proposed an extension of Instant Global Illumination that handles even such complex and highly occluded models [89]. In their method, they use a simple and crude path tracing step to estimate the contribution of each light source to the image. Using a path tracer for this step has several advantages: First, it allows for easy integration into their ray tracing system. Second, a path tracer samples each pixel independently, thereby allowing for easy parallelization and for sampling different light sources in different pixels. Finally, a path tracer samples only visible objects and never touches distant and occluded geometry. This feature is prerequisite for efficient ray tracing of complex models, as incoherent sampling of the entire model would destroy memory caches and result in very low performance [90].

While the approximation from the path tracer is rather coarse, it already gives a good estimate of the contribution of

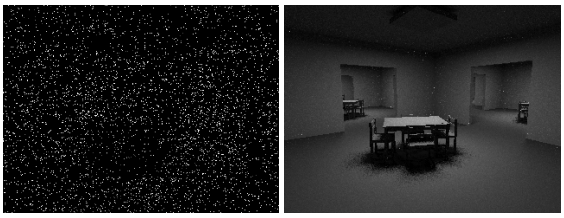


Figure 32: *Quality of the estimate in the ERW10 scene. Left: Estimate as results from a path tracer using a single sample per pixel. Right: The same image rendered with 1024 samples per pixel. Though the estimate image is hardly recognizable, the contributions of the light sources – over the whole image – are estimated correctly up to a few percent.*

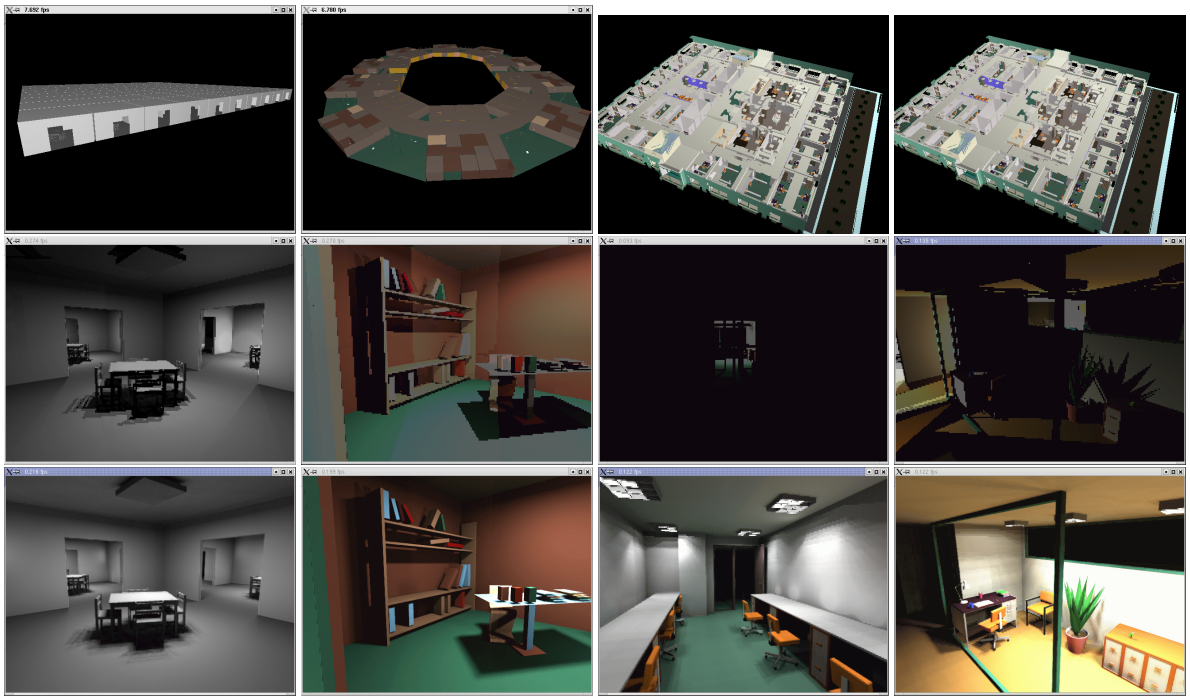


Figure 31: *Instant Global Illumination in Complex and Highly Occluded Scenes. Image quality of the new system (bottom row) versus the original system (middle row) at the same frame rate in several complex scenes (top row). From left to right: "ERW10" (80k triangles, 100 lights), "Ellipse" (19k triangles, 1,164 lights), "Soda Hall" (2.5M triangles, 23k lights), and another view of Soda hall. Running at the same frame rate, the new system achieves visibly superior image quality due to concentrating the VPLs on actually important light sources.*

the different light sources (see Figure 32). This allows for efficient importance sampling by identifying and then ignoring unimportant light sources. VPLs can then be concentrated on more important light sources, allowing for enough VPLs to be placed in the actually visible rooms. Like in the original method, each frame is computed from scratch, and no static precomputed values are being used. Therefore, the method does not limit the viewer to walkthroughs of static scenes. Instead, it allows for interactive, dynamic changes to geometry, materials, and light sources.

In highly occluded models, the proposed procedure works very well, and often results in significant savings. For example, in the Soda Hall scene the number of contributing light sources can be as low as 66, which is less than one percent of all light sources (23,256 lights total). However, the view importance influences the placement of the VPLs and introduces temporal artifacts: Even simple changes of the viewer's pose can change the importance of different light sources, thereby changing the placement of potentially all VPLs in the scene. At the small number of VPLs that are affordable in an interactive setting, this "jumping" of even a small fraction the VPLs between subsequent frames can introduce disturbing flickering of the illumination.

Even though the method uses several heuristics to reduce these temporal artifacts to a tolerable level, some artifacts still remain visible. Even so, the new method results in dramatically improved image quality if compared to the original method at the same frame rate (see Figure 31): Whereas the image quality of the original method is clearly unacceptable, the new method can reproduce most illumination features and often even produces effects light soft shadows that are not present at all in the images computed without using the proposed technique.

While much work remains for improving the method and getting rid of the remaining temporal artifacts, the proposed method allows for the first time interactive global illumination in such complex and highly occluded models without restricting the user to static and precomputed environments.

15. Summary and Conclusions

As has been described in the previous sections, realtime ray tracing can be used to compute global illumination at interactive rates. Instant Global Illumination can faithfully reproduce the most important lighting effects – smooth and hard shadows, smooth direct and indirect diffuse lighting, color bleeding, reflections and refractions – while still allowing

for interactive and dynamic changes to all scene properties including geometry, materials, and lights.

Instant global illumination scales to massively complex scenes of millions of polygons and even to scenes with many light sources and high occlusion using the proposed modifications. On a PC cluster IGI scales linearly in the number of client PCs, and achieves frame rates of up to 25 frames per second at 640×480 pixels, even including procedural shaders, textures, tone mapping, and anti-aliasing.

Still, much work remains to be done: in its current form, instant global illumination is similar to radiosity methods in the sense that it is most efficient for mostly diffuse scenes. While IGI does not suffer from tesselation problems and can easily handle reflections and refractions, other specular effects – like glossiness and high-quality caustics – are still problematic. The latter effects are often less important for practical applications, but would nonetheless be interesting to support.

However, caustics are especially hard to generate. While photon mapping [37, 35, 36] presents an efficient solution to caustics in off-line computations, it is not yet clear how photon mapping could be ported to a distributed, interactive setting as described above.

As instant radiosity inherently builds on tracing packets of highly coherent shadow rays, IGI should also map easily to other ray tracing architectures like e.g. the SaarCOR architecture [75]. This approach seems promising, and is currently being evaluated. Though this has not been finally simulated, preliminary results are promising.

Final Summary

In this STAR, we have given an overview about the state-of-the-art and about the current research activities in realtime ray tracing and in its use for interactive global illumination. We have first discussed the three main approaches to realizing realtime ray tracing: software systems, GPU-based ray tracers, and dedicated ray tracing hardware. Based on the Saarland RTRT/OpenRT engine, Purcell et al.'s streaming GPU ray tracer, and Schmittler et al.'s SaarCOR engine, respectively, we have then analyzed and compared the strengths and weaknesses of these different approaches.

We have also discussed several advanced issues of ray tracing, like handling dynamic scenes and API issues, and have pointed out the potential and several implications for future applications. The results indicate that realtime ray tracing is indeed becoming feasible in the near future, and is likely to play a larger role in future graphical applications.

Finally, we have discussed how realtime ray tracing can be used to achieve interactive global illumination – the physically-correct simulation of light transport at interactive rates. Especially the latter is likely to become a "killer application" for realtime ray tracing and may become a mandatory feature of future 3D graphics similar to the introduction of realtime texture mapping a few years ago. The increased realism provided by interactive global illumination makes it highly attractive for such diverse application areas as computer games, indoor and outdoor visualization, virtual reality, and many more. Though the performance of current implementations is still somewhat limited in terms of high resolutions and frame rates, realtime ray tracing and interactive global illumination provide unique opportunities for novel applications already today, some of which have been pointed out in this report.

The research described in this report also opened up interesting new areas of research that have been mostly ignored in the past. For example the support of dynamic scenes still leaves much to be desired, some global illumination effects – such as glossy reflections and caustics – are still too costly and scale badly. It is nice to see that more and more researchers start looking into these issues.

Acknowledgements

We would like to thank all the people who have contributed to this report in whatever form possible. Especially Andreas Dietrich of Saarland University, who has been deeply involved in many applications of the OpenRT system.

The instant global illumination system has been designed and developed in conjunction with Alexander Keller and Thomas Kollig from Kaiserslautern University. Ian Buck, Bill Mark and Pat Hanrahan have contributed to the "Ray Tracing on Programmable GPUs" project, and James Percy, Pradeep Sen, and Eric Chan helped with the original Radeon 9700 ray tracing demos.

Philippe Bekaert has provided his XRML library [6], helped in writing the OpenRT based VRML browser, and provided early feedback on OpenRT. Models have been provided by Philippe Bekaert (Soda Hall), Tim Dahmen (animated globe scene), Anselmo Lastra (Power Plant), Hella Corp. (car headlight) and Oliver Deussen (Sunflowers and tree models). Images and insight into their systems have been provided by Nathan Carr, Kirill Dmitriev, Bruce Walter, Steve Parker and David DeMarle.

Finally, we would like to thank ATI, Intel Corp, and NVIDIA for their support and sponsorship.

References

1. ATI. Radeon 8500LE 128 product web site, 2001. http://mirror.ati.com/products/pc/radeon8500le128/index.html. 3.2.2

2. ATI. Radeon 9700 Pro product web site, 2002. http://mirror.ati.com/products/pc/radeon9700pro/index.html. 3.3.2

3. ATI. Radeon 9800 Pro product web site, 2003. http://mirror.ati.com/products/pc/radeon9800pro/index.html. 3.1

4. Kavita Bala, Bruce Walter, and Donald Greenberg. Combining Edges and Points for Interactive High-Quality Rendering. *ACM Transactions on Graphics*, 2003. (Proceedings of ACM SIGGRAPH 2003. 10.2

5. Philippe Bekaert. *Hierarchical and Stochastic Algorithms for Radiosity*. PhD thesis, Katholieke Universitiet Leuven, Belgium, 1999. 10

6. Philippe Bekaert. Extensible Scene Graph Manager, August 2001. http://www.cs.kuleuven.ac.be/∼graphics/XRML/. 7.1.3, 7.3, 9, 15

7. Carsten Benthin, Ingo Wald, Tim Dahmen, and Philipp Slusallek. Interactive Headlight Simulation – A Case Study of Distributed Interactive Ray Tracing. In *Proceedings of Eurographics Workshop on Parallel Graphics and Visualization (PGV)*, pages 81–88, 2002. 8

8. Carsten Benthin, Ingo Wald, and Philipp Slusallek. A Scalable Approach to Interactive Global Illumination. to be published at Eurographics 2003, 2003. 2.1.2, 13, 13.2

9. Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 2003. (Proceedings of ACM SIGGRAPH 2003). 3

10. Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proceedings of the conference on Graphics hardware 2002*, pages 37–46. Eurographics Association, 2002. 3, 3.2

11. Alan Chalmers, Timothy Davis, and Erik Reinhard, editors. *Practical Parallel Rendering*. AK Peters, 2002. ISBN 1-56881-179-9. 2.2.1, 2.2.2

12. Micheal F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Morgan Kaufmann Publishers, 1993. ISBN: 0121782700. 3, 10, 11

13. David E. DeMarle, Steve Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen. Distributed Interactive Ray Tracing for Large Volume Visualization. (submitted for publication), 2003. 1, 2

14. Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, Darmstadt, Germany, 2003. Available at http://www.openrt.de. 7

15. Kirill Dmitriev, Stefan Brabec, Karol Myszkowski, and Hans-Peter Seidel. Interactive Global Illumination Using Selective Photon Tracing. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 21–34, 2002. 10.3

16. George Drettakis and François Sillion. Interactive Update of Global Illumination using a Line-Space Hierarchy. In *Computer Graphics (Proceedings of SIGGRAPH 1997)*, pages 57–64, Aug 1997. 10.1, 11

17. Philip Dutre, Kavita Bala, and Philippe Bekaert. Advanced Global Illumination, 2001. SIGGRAPH 2001 Course Notes, Course 20. 3

18. Sebastian Fernandez, Kavita Bala, and Donald Greenberg. Local Illumination Environments for Direct Lighting Acceleration. In *Proceedings of Thirteenth Eurographics Workshop on Rendering*, pages 7–14, Pisa, Italy, June 2002. 14

19. MPI Forum. MPI – The Message Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi. 2.2.2

20. Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. ARTS: Accelerated ray tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986. 3.3.1

21. 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. 2.2.2

22. Andrew Glassner. Spacetime Ray Tracing for Animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, 1988. 6

23. Andrew Glassner. *An Introduction to Raytracing*. Academic Press, 1989. 3.3.2, 6

24. Xavier Granier, George Drettakis, and Bruce Walter. Fast Global Illumination Including Specular Effects. In *Proceedings of the 2001 Eurographics Workshop on Rendering*, pages 47–58, 2001. 10.3, 11

25. Stuart A. Green. Parallel Processing for Computer Graphics. *MIT Press*, pages 62–73, 1991. 4

26. Eduard Gröller and Werner Purgathofer. Using temporal and spatial coherence for accelerating the calculation of animation sequences. In *Proceedings of Eurographics '91*, pages 103–113. Elsevier Science Publishers, 1991. 6

27. Jörg Haber, Karol Myszkowski, Hitoshi Yamauchi, and Hans-Peter Seidel. Perceptually Guided Corrective Splatting. *Computer Graphics Forum (Proceedings of Eurographics 2001)*, 20(3):C142–C152, September 2001. 10.3

28. D. Hall. The AR350: Today's ray trace rendering processor. In *Proceedings of the Eurographics/SIGGRAPH workshop on Graphics hardware - Hot 3D Session 1*, 2001. 4

29. Pat Hanrahan, David Salzman, and Larry Aupperle. A Rapid Hierarchical Radiosity Algorithm. In *Computer Graphics (SIGGRAPH 91 Conference Proceedings)*, pages 197–206, 1991. 10, 10.3, 11

30. Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the conference on Graphics hardware 2002*, pages 109–118. Eurographics Association, 2002. 3

31. Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, 2001. 2.1.1

32. Intel Corp. *Intel C/C++ Compilers*, 2002. http://www.intel.com/software/products/compilers. 2.1.1

33. Intel Corp. *Intel Pentium III Streaming SIMD Extensions*, 2002. http://developer.intel.com/vtune/cbts/simd.htm. 2.1

34. H. Jensen and N. Christensen. Efficiently Rendering Shadows Using the Photon Map. In H. Santo, editor, *Edugraphics + Compugraphics Proceedings*, pages 285–291. GRASP-Graphic Science Promotions & Publications, 1995. 11, 14

35. Henrik Wann Jensen. Global Illumination using Photon Maps. *Rendering Techniques 1996*, pages 21–30, 1996. (Proceedings of the 7th Eurographics Workshop on Rendering). 11, 15

36. Henrik Wann Jensen. Rendering caustics on non-Lambertian surfaces. *Computer Graphics Forum*, 16(1):57–64, 1997. 11, 15

37. Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, 2001. ISBN: 1568811470. 10, 11, 15

38. James T. Kajiya. The Rendering Equation. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (Proceedings of SIGGRAPH 86)*, volume 20, pages 143–150, 1986. 10, 11

39. Alexander Keller. Instant Radiosity. *Computer Graphics*, pages 49–56, 1997. (Proceedings of ACM SIGGRAPH 1997). 10.1, 12

40. Alexander Keller and Wolfgang Heidrich. Interleaved Sampling. *Rendering Techniques 2001*, pages 269–276, 2001. (Proceedings of the 12th Eurographics Workshop on Rendering). 12

41. Alexander Keller and Ingo Wald. Efficient Importance Sampling Techniques for the Photon Map. In *Vision Modelling and Visualization 2000*, pages 271–279, November 2000. 14

42. Thomas Kollig and Alexander Keller. Efficient Multidimensional Sampling. *Computer Graphics Forum*, 21(3):557–563, 2002. (Proceedings of Eurographics 2002). 12

43. Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics*, 2003. (To appear in Proceedings of ACM SIGGRAPH 2003). 3

44. Eric Lafortune. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering*. PhD thesis, Katholieke Universitiet Leuven, Belgium, 1996. 10

45. Eric Lafortune and Yves Willems. Bidirectional Path Tracing. In *Proc. 3rd International Conference on Computational Graphics and Visualization Techniques (Compugraphics)*, pages 145–153, 1993. 10

46. E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing 2001*, page 55, 2001. 3

47. Greg Ward Larson. The Holodeck: A parallel ray-caching rendering system. *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, 1998. 10.2

48. Marc Levoy and Pat Hanrahan. Light field rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42. ACM Press, 1996. 8

49. Jonas Lext and Tomas Akenine-Moeller. Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics 2001 – Short Presentations*, pages pp. 311–318, 2001. 6

50. Jonas Lext, Ulf Assarsson, and Tomas Moeller. BART: A Benchmark for Animated Ray Tracing. Technical report, Department of Computer Engineering, Chalmers University of Technology, Goeteborg, Sweden, May 2000. Available at http://www.ce.chalmers.se/BART/. 6, 6

51. Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 149–158, August 2001. 3

52. William Mark. Shading System Immediate-Mode API, v2.1. In *SIGGRAPH 2001 Course 24 Notes – Real-Time Shading*, August 2001. 7.1.2

53. M. Meissner, U. Kanus, and W. Strasser. VIZARD II, A PCI-Card for Real-Time Volume Rendering. In *Eurographics/Siggraph Workshop on Graphics Hardware*, 1998. 4

54. Michael J. Muuss. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium '95*, June 1995. 1, 2

55. Michael J. Muuss and Maximo Lorenzo. High-Resolution Interactive Multispectral Missile Sensor Simulation for ATR and DIS. In *Proceedings of BRL-CAD Symposium '95*, June 1995. 1, 2

56. Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading, Massachusetts, U.S.A., 1993. 7

57. NVIDIA. Geforce FX 5900 product web site, 2003. http://nvidia.com/view.asp?PAGE=fx_5900. 3.1

58. NVIDIA. NV_fragment_program extension, 2003. http://oss.sgi.com/projects/ogl-sample/registry/NV/fragment_program.txt. 3.1

59. NVIDIA. NV_occlusion_query extension, 2003. http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt. 3.1

60. OpenGL ARB. ARB_fragment_program extension, 2003. http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt. 3.1

61. OpenSG-Forum. http://www.opensg.org, 2001. 7, 9

62. 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. 1, 2, 8

63. 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. 1, 2

64. 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, April 1999. 1, 2, 6

65. Hans-Peter Pfister. SIGGRAPH course on Interactive Ray Tracing, 2001. 4

66. Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The VolumePro real-time ray-casting system. *Computer Graphics*, 33, 1999. 4

67. Pixar. *The RenderMan Interface*. San Rafael, September 1989. 7, 7.1.2, 7.1.3

68. Kekoa Proudfoot, William Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Computer Graphics (Proceedings of SIGGRAPH 2001)*, pages 159–170, August 2001. 7.1.2

69. Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002. (Proceedings of SIGGRAPH 2002). 1, 3, 7.1.3

70. Erik Reinhard. *Scheduling and Data Management for Parallel Ray Tracing*. PhD thesis, University of East Anglia, 1995. 2.2.1

71. Erik Reinhard, Brian Smits, and Chuck Hansen. Dynamic Acceleration Structures for Interactive Ray Tracing. In *Proceedings of the Eurographics Workshop on Rendering*, pages 299–306, Brno, Czech Republic, June 2000. 6

72. 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. 7, 9

73. B. Schachter. *Computer Image Generation*. Wiley, New York, 1983. 2.2.2

74. Jörg Schmittler, Alexander Leidinger, and Philipp Slusallek. A Virtual Memory Architecture for Real-Time Ray Tracing Hardware. *to appear in Computer and Graphics, Volume 27, No. 5*, 2003. 2.2.1, 4, 4.2, 4.4, 4.7, 4.8

75. Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of Eurographics Workshop on Graphics Hardware*, pages 27–36, 2002. 1, 4, 4.4, 4.5, 4.6, 4.7, 4.8, 7, 7.1.3, 15

76. Peter Shirley. *Fundamentals of Computer Graphics*. AK Peters, 2001. ISBN 1568811241. 2

77. Maryann Simmons and Carlo H. Sequin. Tapestry: A dynamic mesh-based display representation for interactive rendering, june 2000. ISBN 3-211-83535-0. 10.2

78. Brian Smits, James Arvo, and Donald Greenberg. A clustering algorithm for radiosity in complex environments. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 435–442. ACM Press, 1994. 10, 10.3, 11

79. Marc Stamminger, Jörg Haber, Hartmut Schirmacher, and Hans-Peter Seidel. Walkthroughs with Corrective Textures. In *Proceedings of the 11th Eurographics Workshop on Rendering*, pages 377–388, 2000. 10.3

80. Advanced Rendering Technologies. http://www.art.co.uk/, 2002. 4

81. Parag Tole, Fabio Pellacini, Bruce Walter, and Donald P. Greenberg. Interactive Global Illumination in Dynamic Scenes. *ACM Transactions on Graphics*, 21(3):537–546, 2002. (Proceedings of ACM SIGGRAPH 2002). 10.2

82. Th. Ullmann, A. Schmidt, D. Beier, and B. Bruderlin. Adaptive Progressive Vertex Tracing for Interactive Reflections. *Computer Graphics Forum*, 20(3), September 2001. Proceedings of Eurographics. 8

83. Th. Ullmann, A. Schmidt, D. Beier, and B. Bruderlin. Adaptive Progressive Vertex Tracing in Distributed Environments. In *Proceedings of the Ninth Pacific Conference on Computer Graphics and Applications (Pacific Graphics 2001)*, pages 285–294. IEEE, October 2001. 8

84. Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1997. 10

85. VRML Consortium. VRML97 – The Virtual Reality Modelling Language, 1997. ISO/IEC 14772-1:1997. 9

86. Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek. Interactive Ray Tracing on Commodity PC Clusters – State of the Art and Practical Applications. *Lecture notes on Computer Science*, 2003. (Proceedings of EuroPar). 2.2.3

87. 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. 7

88. Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed Interactive Ray Tracing of Dynamic Scenes. In *2003 Workshop on Parallel Graphics and Visualisation (PVG) (to appear)*, 2003. Also available at http://graphics.cs.uni-sb.de/Publications. 6, 6

89. Ingo Wald, Carsten Benthin, and Philipp Slusallek. Interactive Global Illumination in Complex and Highly Occluded Environments. *Proceedings of the 14th Eurographics Workshop on Rendering*, 2003. 14, 14

90. 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). 1, 2, 2.1.1, 2.1.1, 2.1.2, 3.2.2, 14

91. 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). 2, 12, 12.2

92. Ingo Wald and Philipp Slusallek. State-of-the-Art in Interactive Ray-Tracing. In *State of the Art Reports, Eurographics 2001*, pages 21–42, 2001. 1, 1.1, 2.1, 10.3

93. 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). 1, 2, 2.2.1

94. John R. Wallace, Michael F. Cohen, and Donald P. Greenberg. A Two-pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods. *Computer Graphics*, 21(4):311–320, 1987. 11

95. Bruce Walter, George Drettakis, and Donald P. Greenberg. Enhancing and Optimizing the Render Cache. In *Rendering Techniques 2002 (Proceedings of Eurographics Workshop on Rendering)*, 2002. 10.2

96. Bruce Walter, George Drettakis, and Steven Parker. Interactive Rendering using the Render Cache. In *Rendering Techniques 1999 (Proceedings of Eurographics Workshop on Rendering)*, 1999. 10.2, 10.3

97. G. Ward. Adaptive Shadow Testing for Ray Tracing. In *2nd Eurographics Workshop on Rendering*, 1991. 14

98. Gregory J. Ward and Paul Heckbert. Irradiance Gradients. In *Third Eurographics Workshop on Rendering*, pages 85–98, Bristol, UK, 1992. 11

99. Josie Wernecke. *The Inventor Mentor*. Addison-Wesley, Reading, Massachusetts, U.S.A., 1994. 7, 9