State of the Art in Interactive Ray Tracing

Ingo Wald and Philipp Slusallek

Computer Graphics Group, Saarland University

Abstract

The term ray tracing is commonly associated with highly realistic images but certainly not with interactive graphics. However, with the increasing hardware resources of today, interactive ray tracing is becoming a reality and offers a number of benefits over the traditional rasterization pipeline.

The goal of this report is to provide a better understanding of the potential and challenges of interactive ray tracing. We start with a review of the problems associated with rasterization based rendering and contrast this with the advantages offered by ray tracing. Next we discuss different approaches towards interactive ray tracing using techniques such as approximation, hybrid rendering, and direct optimization of the ray tracing algorithm itself. After a brief review of interactive ray tracing on supercomputers we describe implementations on standard PCs and clusters of networked PCs. This system improves ray tracing performance by more than an order of magnitude and outperforms even high-end graphics hardware for complex scenes up to tens of millions of polygons. Finally, we discuss recent research towards implementing ray tracing in hardware as an alternative to current graphics chips.

This report ends with a discussion of the remaining challenges and and the future of ray tracing in interactive 3D graphics.

1. Introduction

Today interactive rendering is almost exclusively the domain of rasterization-based algorithms that have been put into highly integrated and optimized graphics chips. Their performance increased dramatically over the last few years and now even exceeds that of (what used to be) graphics supercomputers. At the same time this hardware can be provided at low cost such that most of today’s PCs already come equipped with state-of-the-art 3D graphics devices.

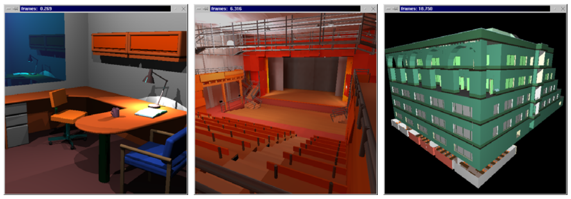

Figure 1: *Interactive ray tracing: The office, conference and Soda Hall models contain roughly 40k, 680k, and 8 million triangles, respectively. Using a software ray tracing implementation running on a single PC (Dual Pentium-III, 800 MHz, 256MB) at a resolution of* 512^2 *pixels, these scenes render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

Even though the performance increase of 3D graphics hardware has been tremendous over the last few years, it is still far from sufficient for many applications. In particular e-commerce applications would benefit from a significant higher level of realism than is possible with current graphics hardware. Computer games and virtual studio applications have strong real time constraints and are too limited in the complexity of their scene geometry. In many areas we observe a trend towards the need to handle more and more complex environments. It simply becomes too difficult and time consuming to select and preprocess the relevant parts out of a large geometric data base. Prime examples are large architectural and engineering design projects such as entire power plants, airplanes, or cars.

Current graphics hardware has seen a number of difficulties in this respect because it exhibits essentially linear cost in the number of polygons. In order to achieve interactive rendering performance sophisticated preprocessing if required for reducing the number of rendered polygons per frame.

This can be accomplished through techniques such as geometric simplification, level-of-detail, occlusion culling, and many others. It has also been difficult to efficiently scale the graphics pipeline to parallel processing. The main issues have been the communication requirements between parallel units and the need to avoid redundant processing and data storage, e.g. for textures [8].

Other weaknesses of the current graphics devices are related to realism. One one hand its feature set for shading has been increased significantly with multi-texturing, programmable vertex processing, programmable fragment processing, dependent texture lookups, and many more. On the other hand it became increasingly difficult for applications to actually take advantage of these features as they are hard to program and even simple shading effects are difficult to express in terms of these extensions. We have seen the formation of a whole new field in computer graphics research dealing with how to best approximate known algorithms with the features of the latest graphics hardware.

Interestingly enough there has been another rendering algorithm in wide use for many years that solves some of the problems mentioned above – ray tracing. Ray tracing is famous for its ability to generate high-quality images but is also well-known for long rendering times due to its high computational cost. This cost is due to the need to traverse a scene with many rays, intersecting each with the geometric objects, shading the visible surface samples, and finally sending the resulting pixels to the screen. Due to the cost associated with ray tracing the technique is perceived almost exclusively as an off-line technique for cases where image quality matters more than rendering speed.

On the other hand ray tracing offers a number of benefits over rasterization-based algorithms that would make it an interesting alternative.

Flexibility Ray tracing allows us to trace individual or unstructured groups of rays. This provides for efficient computation of just the required information, e.g. for sampling narrow glossy highlights, for filling holes in image-based rendering, and for importance sampling of illumination [44]. Eventually this flexibility is required if we want to achieve interactive global illumination simulations based on ray tracing.

Occlusion Culling and Logarithmic Complexity Ray tracing enables efficient rendering of complex scenes through its built in occlusion culling as well as its logarithmic complexity in the number of scene primitives. Using a simple search data structure it can quickly locate the relevant geometry in a scene and stops its front to back processing as soon as visibility has been determined. This approach to *process geometry on demand* stands in strong contrast to the *"send all geometry and discard at the end"* approach taken by current triangle rasterization hardware.

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

Simpler Shader Programming Programming shaders that create special lighting and appearance effects has been at the core of realistic rendering. While writing shaders (e.g. for the RenderMan standard [4]) is fairly straightforward, adopting these shaders to be used in the pipeline model of rasterization has been very difficult [29]. Since ray tracing is not limited to this pipeline model it can make direct use of shaders [10, 36].

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

Parallel Scalability Ray tracing is known for being "trivially parallel" as long as a high enough bandwidth to the scene data is provided. Given the exponential growth of available hardware resources, ray tracing should be better able to utilize it than rasterization, which has been difficult to scale efficiently [8]. However, the initial resources required for a hardware ray tracing engine are higher than those for a rasterization engine.

Coherence is the key to efficient rendering. Due to the low coherence between rays in traditional recursive ray tracing implementations, performance has been rather low. However, as we show in this report, ray tracing inherently offers considerable coherence that can be exposed by new algorithms in order to to speed up rendering to interactive levels even on a standard PC.

It is due to this long list of advantages that ray tracing is an interesting alternative even in the field of interactive 3D graphics. The challenge is to improve the speed of ray tracing to the extent that it can compete with rasterization-based algorithms. This report will mainly concentrate on existing software implementations for interactive ray tracing. In addition we provide a brief discussion on current research work towards ray tracing hardware, as it seems that some dedicated hardware support will eventually be needed.

At this point is is worth noting that the use of the ray tracing algorithms goes well beyond graphics applications such as rendering. Ray casting is a fundamental task that is at the core of a large number of algorithms from other disciplines. Examples are the simulation of neutron transport in physics, radio wave propagation [7], and diffusion [33]. If we can accelerate ray tracing to interactive performance in rendering applications this will have significant effects also for these other disciplines.

1.1. Document Organization

In the following, we will first introduce and classify the different techniques for accelerating ray tracing. In Section 3 we give a brief survey of acceleration techniques based on approximations before concentrating on techniques to improve the performance of the ray tracing algorithms itself. In Section 4, we discuss the first interactive ray tracing systems that have been realized on large supercomputer systems. This is followed by a more detailed discussion of interactive ray tracing systems implemented on inexpensive PCs and clusters of workstations in Sections 5 and 6.

An brief overview of ongoing research towards the design of ray tracing hardware is given in Section 7 and Section 8 presents the most important open research issues. The report ends with a brief summary, and an outlook on the potential future of interactive or even realtime ray tracing.

2. Accelerated Rendering of Ray Tracing Effects

There are several options to achieve ray tracing effects at interactive rates. The basic approach is to speed up the ray tracing algorithms itself and its implementation. This will be the main focus of this paper. However, there have also been a number of other approaches that work with approximations and hybrid rendering techniques. These techniques typically trade image quality for rendering performance. We can classify all these approaches broadly into the following four categories. We describe some of them in more detail in the next section.

Rasterization-Based Approximation These techniques combine the high rendering speed of todays rasterization hardware with the superior image quality of ray tracing. Early approaches only tried to accelerate ray tracing using rasterization by precomputing index structures such as vista-buffers and light-buffers.

Newer approaches use ray tracing to augment rasterization-based images with ray tracing effects: triangle rasterization is used to compute a low-quality frame at interactive rates, and ray tracing is then used to add effects that are difficult to achieve with other means.

However, due to the approximations these approaches typically show artifacts. Even more importantly, they are based on the assumption that rasterization is faster than ray tracing. As we will show later, this is no longer the case for more complex scenes.

Image-based Approximation These techniques mix ray tracing with image-based rendering techniques. One approach is to approximate a new frame with information from previous frames and augment this by tracing some new rays in order to obtain new information as time allows. Examples include techniques like the *Holodeck* [19] or the *RenderCache* [43].

For static situations this method converges to the same image as a traditional ray tracer as all pixels of a frame will eventually be computed by ray tracing. However, this approach suffers from the fact that traditional ray tracing only allows for computing a limited number of rays per frame, which leads to noticeable artifacts in dynamic situations.

Ray Tracing Based approximation These techniques aim at reducing the cost per pixel by approximations in the ray tracing process. For example, adaptive sampling methods like *pixel-selected ray tracing* [2] can be used to reduce the average number of rays per pixel: color interpolation in image space is used in image regions with a smooth change in colors. Only image regions with high contrast will be sampled with higher accuracy. A more advanced approach using interpolation in ray space with error bounds has been presented by Bala [5].

Accelerating Ray Tracing Due to its high computational cost researchers have been looking for improving the performance of ray tracing since the introduction of ray tracing. Much research has focussed on acceleration structures and their traversal algorithms like BSP trees, octrees, (hierarchical) grids, bounding volume hierarchies, hybrids of these, and many more.

Little research has been done to deal with the recursive nature of the algorithm, which is well-known to be ill-suited for today's CPU and memory architectures [37]. Exposing coherence by reformulating or creating new algorithms is an essential tool for achieving efficiency on current architectures. Then the implementation needs to be optimized to the features of todays CPUs such as its cache, memory bus, pipeline, and commonly also SIMD extension in order to exploit this coherence.

Another way of accelerating ray tracing is to exploit the inherently parallel nature of ray tracing by parallel processing. However, scalability over a large number of machines in an interactive environment requires good load balancing, hiding of latencies, and careful optimization of communication overhead. This is especially true when using a cluster of networked PCs based on commodity network technology.

Eventually, we also have to consider hardware solutions in order to improve the performance of ray tracing.

3. Approximative Ray Tracing Techniques

Ever since the development of ray tracing, people have sought to accelerate ray tracing by approximations and by mixing it with other technologies. For many applications, fast visual feedback during interactions is more important than correctness of the image. Trading image quality for interactivity makes sense in those cases. Approximative techniques can typically provide visual feedback at a rate faster than the renderer could generate complete frames — at the cost of producing approximate images during camera and object motion.

3.1. Combining Ray Tracing and Rasterization

When fast graphics hardware became available researchers have tried to exploiting such hardware for accelerating ray tracing. In the simplest form they used rasterization to obtain visibility information by rasterization of geometry where colors encode the id of the rendered object. This allows for creating vista buffers (for accelerating primary rays), shadow buffers (for point light sources), and similar data structures. This allowed for resolving visibility for primary and shadow rays by a simple texture lookup at the cost of minor discretization artifacts. Variations of these techniques are still in use in many renderers today.

More recently, Stamminger, et al. have introduced *corrective textures* [38], a technique to combine the rendering speed of polygon rasterization with the high image quality of ray tracing: All non-specular lighting effects (calculated with a radiosity) are rendered with rasterization hardware. Specular effects are then progressively added in a second step. Information from the rasterization pass (i.e. information on which pixels are covered by specular objects) is used to determine where a-posterior corrections must be applied.

The specular component is then rendered with a conventional ray tracer, and saved in texture maps. These so-called corrective texture maps can then be used by the rasterization hardware to add the specular effect to the object (see Figure 2). As a conventional ray tracer can not deliver all the corrective information in real time, the approach generates artifacts. However, it is progressive in the sense that the image quality converges to that of ray tracing as soon as the view is static.

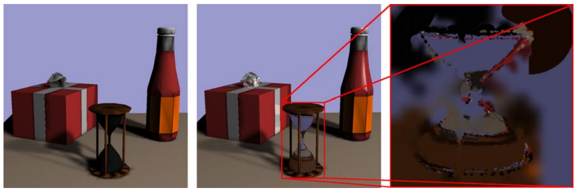

Figure 2: *Corrective Textures: Left: The scene without corrective information as rendered by the rasterization hardware. Middle: The same scene with corrective textures applied. Right: The corrective texture for the hourglass after having shot only a few samples.*

Recently, Haber et al. [11] have extended the approach to include perceptual information: This is used to focus the sampling effort towards perceptually important regions, thus getting visually pleasing images in less time. Corrective information is no longer stored in textures, but is splatted directly into the image. Here too, the image quality improves progressively as more and more corrective samples are computed (see Figure 3).

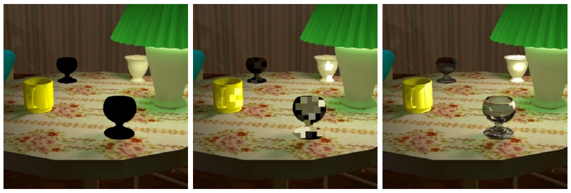

Figure 3: *Perceptually Guided Corrective Splatting: A rasterized image gets progressively better as more ray traced corrective information is applied. Left: Without corrective information. Middle: After having splat only few corrective samples. Right: Converged image after a few frames.*

However, accelerating ray tracing with rasterization hardware is only useful as long as rasterization hardware is considerably faster than the ray tracer. This is definitely true for small models as they are typically used in games or other interactive applications. For complex models however, interactive ray tracers are already faster than even high-performance graphics accelerators (see Section 5.5).

3.2. Image-Based Approximation: The RenderCache

Another approach to accelerate ray tracing is to reduce the number of rays that have to be traced for an image. With the *RenderCache* [43] information from previous frames is reused for successive frames by reprojecting the information already computed to the new camera position. After a ray for a pixel has been traced and shaded, it is stored in a cache of active samples (the so-called *RenderCache*) together with information on the ray, the hit-position, the color computed for that ray, and so on.

When a new frame is to be rendered, all samples in the render cache are reprojected to the new camera position, and stored in the frame buffer. As this reprojection step can result in artifacts (e.g. holes in the image, disocclusion, or warping several samples to the same new pixel position), several heuristics have to be applied to reduce such artifacts. Several different samples per pixel can simply be resolved by z-buffering. Holes, disocclusion, and other artifacts can often be detected and fixed by other simple heuristics (like comparing depth and color contrast of neighboring pixels). In a third pass, an error value is computed for each pixel also based on simple heuristics, which specifies the necessity of re-tracing certain pixels. In the last step, a small number of rays is traced based on the error value of pixels. It is shaded and inserted into the render cache, replacing some of the old samples.

With this approach, interactive rendering speed can be achieved for moderate resolutions (up to 15 frames per second for resolutions of 320x320 pixels) [43]. As not all pixels are traced every frame, the method creates visually objectionable artifacts. However, it is progressive in the sense that image quality improves towards the correct image for static views as all pixels are eventually re-traced.

The RenderCache is especially interesting for achieving interactive frame rates when the cost for computing a pixel is very high, for example when using a full global illumination solver instead of a simple interactive ray tracer. This allows the RenderCache to even render a scene with full path tracing at interactive rates (see Figure 4).

However, the efficiency of the RenderCache relies on the fact that it is much faster to reproject the samples and perform the error analysis than to trace new rays. With interactive ray tracers this not obvious anymore if only simple shading is performed. The RenderCache also performs non-local operations in image space, which complicates a parallel implementation in a distributed environment.

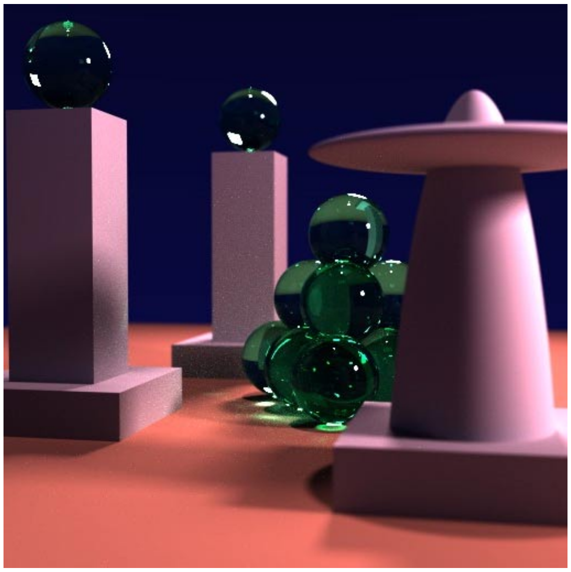

Figure 4: *Path-Tracing in Kajiyas original example scene, rendered interactively with the RenderCache.*

3.3. Light-field like Methods: The Holodeck

Similar to the RenderCache, the *Holodeck* [19] achieves interactive speeds of ray traced environments by reusing old ray samples. The Holodeck subdivides the entire scene into a three-dimensional grid of cells in which previous illumination samples (rays with their corresponding radiance) are stored. When a new image is generated, these illumination samples are used to generate the image in a light-field like approach [20, 9].

Similar to the render cache, new illumination samples are computed with a non-interactive, traditional ray tracer (e.g. the *Radiance* system [45]). These samples are computed in several parallel threads and inserted into the current cell as soon as they are available. Contrary to the RenderCache, the Holodeck progressively builds up an object space representation of the light field. Old samples are never discarded, but written out to disk if they are no longer needed by the current view. If the user reenters a previously visited cell, the corresponding old samples can be read back from the file. In order to hide i/o latencies, prefetching is used for obtaining the samples for nearby viewing cells even before they are visited.

The problem with all the above approaches is the reduced image quality at least during interactions such as camera or object movements. It would instead be very valuable to be able to do full ray tracing on an entire image at interactive rates.

4. Interactive Ray Tracing on Supercomputers

For a long time researchers have pointed out the advantages of ray tracing over rasterization. They argued that the sub-linear complexity and the better scalability of ray tracing should eventually make ray tracing more efficient than polygon rasterization. However, it took almost two decades until the first interactive ray tracing systems have actually been built. They were first realized on highly parallel, shared-memory supercomputer systems.

4.1. The Interactive Ray Tracing System by Muuss

The first big step towards interactive ray tracing is due to Muuss [24, 25]. He required interactive frame rates for simulating the airplane and missile sensors and their automatic target recognition system. His models were highly detailed, realistic outdoor scenes modeled with CSG primitives. Due to the high model complexity, brute force triangle rasterization was impossible even on the most expensive rasterization hardware: The models used in their simulations were made up of several hundred thousands of CSG primitives, corresponding to several million polygons when tessellated.

He used a traditional ray tracing system to directly render the CSG models, saving both the memory overhead as well as the compute time to generate and store the tesselated objects. In order to achieve interactive framerates, he used a highly parallel, 96-processor SGI PowerChallenge array computer and achieved rates of 1-2 frames per second for video resolution.

4.2. The Utah Parallel Ray Tracing System

At the University of Utah, Parker et al. [28] have built an interactive ray tracing system also based on a shared-memory supercomputer. Like the Muuss system, their approach is a ’brute force’ implementation in that it uses the same ray tracing algorithms as used by conventional ray tracers. However, their system is carefully optimized for the available system resources. It takes advantage of fast synchronization available on the SGI Origin 2000 supercomputer with its fast interconnection network (see Figure 5).

The Utah system has also been adapted to accomplish

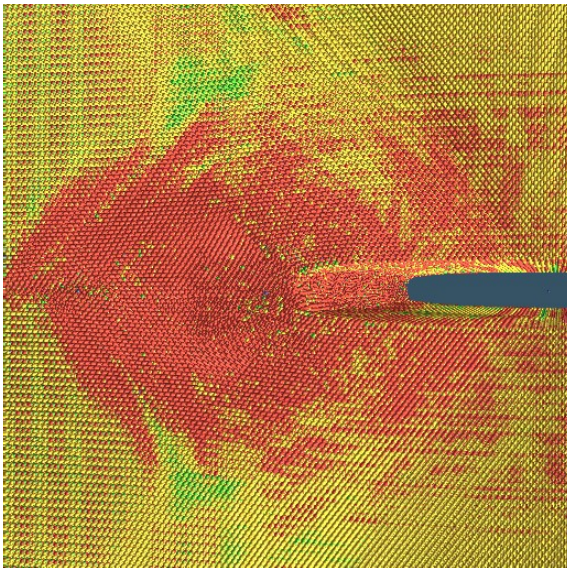

Figure 5: *Simulation of crack propagation visualized with the Utah interactive ray tracer using 35 million spheres. The image at 512 by 512 pixels runs at approximately 15 frames per second on 60 CPUs.*

other interactive visualization tasks, like interactive rendering of volumetric data and for high-quality iso-surface rendering. Examples are shown in Figures 6 and 7. Due to the flexibility of ray tracing, simple changes to the core of the algorithm suffice to directly ray trace such data without having to first generate a triangular representation. Similarly, the system was easily extended to also support realistic shadows. This yields a much better three-dimensional appearance of the volumetric model as shown in Figure 6.

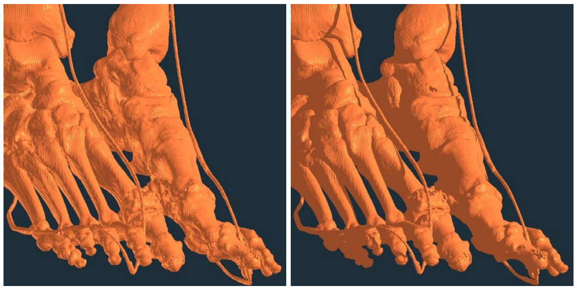

Figure 6: *Example images from the Utah Interactive Ray Tracing System when rendering a complex volume data set. Left: without shadows, right: with raytraced shadows.*

The Utah system supports both synchronous as well as asynchronous operation with frameless rendering. In synchronous operation, synchronization overhead is minimized with an optimized static load-balancing scheme of variable-sized jobs. In order to achieve good caching behavior (for the pixels as well as for the job tiles), the job sizes are multiples of the caching granularity. With this load balancing scheme, the Utah system showed near-linear scalability for up to 128 processors.

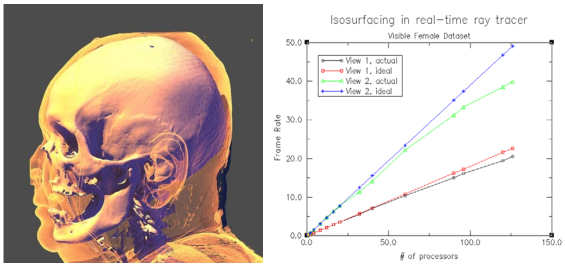

Figure 7: *Scalability of Utah ray tracing engine: The graph on the right shows almost linear scaling behavior for up to 128 CPUs.*

5. Interactive Ray tracing on Desktop PCs

The Muuss and Utah ray tracing systems already achieve interactive frame rates with high image quality even for highly complex scenes. However, their approaches are based on expensive, shared-memory supercomputers, which are not generally accessible. On the other hand, standard PCs are readily available in almost every lab loosely connected using commodity Fast-Ethernet technology. Even dedicated PC clusters typically cost only a fraction of a supercomputer, and thus are accessible to a wider range of people.

Thus, the goal was to design a ray tracer that efficiently runs on these PC architectures and exploits as much of the available performance as possible. Additionally, it should be possible to run on loosely-coupled cluster systems, in order to fully exploit the compute power that is available in many environments.

The low performance of ray tracing on todays computers is strongly affected by the structure of the basic algorithm. It is well-known that the recursive sampling of ray trees neither fits with the pipeline execution model of modern CPUs nor with the use of caching to hide low bandwidth and high latency when accessing main memory [37]. These optimizations become more and more important as processor pipelines get longer and the gap between processor speed and memory bandwidth and latency opens further.

In order to exploit all the performance offered by todays CPUs, we have to address issues such as reducing code complexity, optimizing cache usage, reducing memory bandwidth, and prefetching data. The Utah ray tracer has already shown that such issues are important for parallel shared-memory systems. However, due to the typically much poorer architecture of PC systems (i.e. cache size, memory speed, slower networks, higher latencies, lack of shared memory architecture), they become even more crucial for PC architectures.

The *coherent ray tracing* algorithms presented below consists of four main ingredients:

- We designed a new algorithm for *coherent ray tracing* by reformulating the traditional depth-first recursive algorithm and reordering ray evaluations. Using packets of rays the algorithms operates in a partial breadth-first order exposing significantly more coherence.
- This coherence is used by *optimization techniques* to better adapt the algorithms to the properties of today's CPUs and memory systems. In particular we make sure that the ray tracing algorithm runs almost completely in the caches of the CPU. While these optimization techniques are well-known they can only be effective due to the increased coherence exposed by the new algorithm.
- Next we exploit the *SIMD extensions* now commonly available on CPUs, such as the Intel's SSE extension to the x86 instruction set. This allows us to operate on up to to four data values in parallel during ray traversal, intersection, and shading.
- Finally, we use the coherence available in the new ray tracing algorithm for *efficient distributed computing* on a number of loosely coupled machines in a network. Taking advantage of the available coherence we store the scene data only once without replicating it to each client. Instead we use the memory of the client machines as caches for the scene data that is fetched on demand. Again, careful attention is payed to proper use of these third level caches.

5.1. Coherent Ray Tracing

In the standard recursive ray tracing algorithms relatively little work is done for each recursive invocation of the traversal, intersection, and shading operations. This does not run efficiently on the long pipelines of today's CPUs. Additionally, adjacent primary rays operate on almost the same data during traversal, intersection and shading. The same is true to a somewhat lesser degree for secondary rays, such as shadow rays to the same light source, or reflection rays off of spatially close intersection points from coherent primary or even secondary rays. However, these operations are far apart in time due to the depth-first recursive traversal of traditional ray tracing.

By grouping related, coherent rays into packets of rays and traversing, intersecting, and shading them in parallel, we expose significantly more coherence. We can make good use of this coherence for the general optimizations, for performing data parallel operations in SIMD fashion, and for cache coherent demand loading of scene data in the distributed case. It is the combined effect of the better algorithm and the optimizations that allow us to perform interactive ray tracing on standard PCs.

5.2. Optimizing Code and Memory Access

Even though modern processors have hardware features for performing branch prediction, instruction reordering, speculative execution, and others optimizations, their success is fairly limited for complex code paths. Therefore, we prefer simple code that contains few conditionals, and organize it such that it can execute in tight inner loops. Such code is easier to maintain and can be well optimized by the programmer as well as by the compiler (also see [37]).

Contrary to general opinion a ray tracer is not bound by CPU speed, but is usually bandwidth-limited by access to main memory. Especially shooting rays incoherently, as done in many global illumination algorithms, results in almost random memory accesses and bad cache performance. On current PC systems, bandwidth to main memory is typically up to 8-10 times less than to primary caches. Even more importantly, memory latency increases by similar factors as we go down the memory hierarchy.

We employ an optimized memory layout for the most commonly used data structures, like BSP nodes, triangle intersection data, and even ray and hit point data structures. We keep data together if and only if it is used together: E.g. only data necessary for a triangle intersection test (plane equation, etc.) are stored in our geometry structures, while data that is only necessary for shading, such as vertex colors and normals, shader parameters, etc., is stored separately. Because we intersect many triangles before we find an intersection that requires shading, we avoid loading shading data that will not be used.

Since data transfer between memory and cache is always performed in cache lines of typically 32 bytes, the *effective* cost when accessing memory is not directly related to the number of bytes read, but the number of cache line transfers. Thus we need to carefully align data to cache lines: This minimizes the additional bandwidth required to load two cache lines just because some data happens to straddle a cache line boundary. However there are trade-offs: our triangle data structure requires 37 bytes. By padding it to 48 bytes we trade-off memory efficiency and cache line alignment.

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

With such carefully designed data structures, we reduce bandwidth, cache misses, and false sharing of cache lines as much as possible. This allows to render even large scenes with good caching behavior, and enables the CPU to run at full performance most of the time.

5.3. Using SIMD extensions for Ray Tracing

In addition to the more implicit use of coherence to better use caches, we explicitly exploit coherence between several rays by traversing, intersecting, and shading *packets of rays* in parallel. With this approach, we can reduce the number of operations by using SIMD instructions on the data of multiple rays in parallel, reduce memory bandwidth by requesting data only once per packet, and increase cache utilization at the same time.

SIMD extensions (such as Intel's SSE [13], AMD's 3D-Now! [1], and IBM/Motorola's AltiVec [23]) are offered by several modern microprocessor architectures. These extensions allow to execute the same floating point instructions in parallel on several (typically two or four) data values, yielding a significant speedup for floating point intensive applications.

These SIMD extensions can be used by either exploiting the instruction-level parallelism in already existing algorithms, or by explicitly designing new, data-parallel algorithms. Our experiments have shown that the performance gains obtained by instruction level parallelism are too small to be of significant impact [42]. Data-parallelism can be extracted in two ways: One could traverse one ray and intersect it with four triangles in parallel or traverse four rays in parallel.

Intersecting one ray with four triangles would require us to always have four triangles available for intersection to achieve optimal performance. However, voxels of acceleration data structures contain only few triangles on average (typically 2-3).

In contrast it is much simpler to bundle four rays and intersect them with a single triangle. This approach requires us to always have a bundle of four rays available together, which requires the new scene traversal algorithm. The data-parallel ray-triangle intersection computation corresponds almost exactly to the original algorithm and is straightforward to implement in SSE and shows almost ideal speedup.

	C code	SSE	speedup
min	78	22	3.5
max	148	41	3.7

Table 1: *Amortized cost (in CPU cycles) for the different intersection algorithms. 41 cycles correspond to roughly 20 million intersections per second on a 800 MHz Pentium-III.*

The algorithm for traversing four different rays through a BSP tree works essentially the same way: For each BSP node, we use SSE operations to decide which subtrees of a node a packet of rays needs to traverse. This decision is based on the requirements of all rays: If any ray requires traversal of a subtree, then the entire packet will traverses that subtree. Of course this introduces some overhead, as some rays might traverse parts of the tree that they would not have traversed otherwise. In practice, this overhead is relatively small, less than a few percent, especially for small packet sizes of two by two rays and large image resolutions in the order of 1K by 1K. The traversal operation has a higher memory access to computation ratio such that the speedup is limited to roughly a factor of two.

Similar to the efficient data-parallel intersection and traversal code, the same benefits also apply for shading computations. We can shade four rays in parallel but since the four hit points may have different materials, we may have to re-arrange them first in order to expose more coherence. Although this results in some overhead, the following shading operations can be very efficiently implemented in SSE, yielding good utilization of the SSE units.

Texturing has shown to be relatively cheap. Even an unoptimized implementation has reduced frame rates by less than 10 percent, even in large, highly textured models as shown in Figure 8. This shading cost could probably be reduced even more as there is a large potential for prefetching and parallel computations that we currently do not take advantage of. As shading typically still makes up for less than 10 percent of total rendering time, more complex shading operations could easily be added without a major performance hit.

Implementing traversal, intersection, and shading with SSE operations gives a overall speedup of 2 to 2.5 as compared to a highly optimized C implementation.

5.4. Comparison with Other Ray-Tracers

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

In absolute terms, we achieve a rendering performance from about 200,000 to almost 1.5 million primary rays per second for the SSE version of our algorithm and for scenes of different complexity on a single Pentium-III 800 MHz.

In order to evaluate the relative performance increase of our optimized ray tracing engine we tested it against a number of freely available ray tracers, including POV-Ray [27] and Rayshade [17]. We have chosen the set of test scenes such that they span a wide range regarding the number of triangles and the overall occlusion within the scene. Unfortunately, both other systems failed to render some of the more complex test scenes due to memory limitations even with 1GB of main memory, even though RTRT was used with only 256 MB of memory for all test cases.

The numbers of the performance comparison for the case of primary rays are given in Table 2. It demonstrates clearly that our new ray tracing implementation improves performance consistently by a factor between 11 and 15 (!) compared to both POV-Ray and Rayshade. The test also indicate that the performance gap widens for more complex scenes, which indicate that the caching effects get even more pronounced in these cases.

The numbers show that paying careful attention to caching and coherence issues can have a tremendous effect on the overall performance, even for well-known and well-analyzed algorithms such as ray tracing.

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

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

5.5. Comparison with Rasterization Hardware

Up to now it was widely believed that ray tracing is not yet competitive with graphics hardware. By comparing our interactive ray tracing engine to the performance offered by the fasted available rasterization hardware, we show that the performance crossover point has already been reached. We outperform rasterization hardware even with a software ray tracer running on a single CPU – at least for complex scenes and moderate screen resolutions.

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

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

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

Table 3: *OpenGL rendering performance in frames per second with SGI Performer on three different graphics hardware platforms compared with our software ray tracer at a resolution of* 512^2 *pixels on a dual processor PC. The ray tracer uses only a single processor, while SGI Performer actually uses all available CPUs.*

These results show clearly that the software ray tracer running on a single CPU outperforms the best hardware rasterization engines for scenes with a complexity of roughly 1 million triangles and more and is already competitive for scenes of about half the size. Note that the ray tracing numbers can be scaled easily by adding more processors — just enabling the second CPU on our machines doubles our RTRT numbers given in Table 3. Due to the scalability problems of the rasterization pipeline this scaling is not easily possible for graphics hardware.

In order to visualize the scaling behavior of rasterization and ray tracing-based renderers, we used a large terrain scene (see Figure 8) and sub-sampled the geometry. The results are shown in Figure 8. Even though SGI Performer uses a number of techniques to reduce rendering times, we see the typical linear scaling of rasterization. Even occlusion culling would not help in this kind of scene. Ray tracing benefits from the fact that each ray visits roughly a constant number of triangles but needs to traverse a BSP tree with logarithmically increasing depth. Ray tracing also subsamples the geometry for the higher resolution terrain as the number of pixels is less than the number of triangles.

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

These test have compared ray tracing only as far as it offers the same features as rasterization hardware. Due to its much more flexible structure, advanced rendering effects

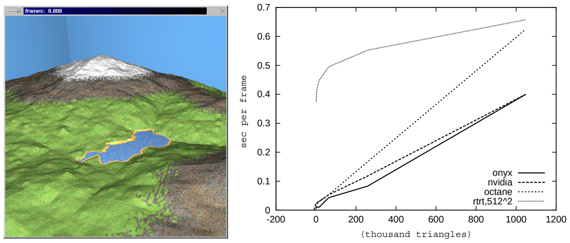

Figure 8: *This figure shows the logarithmic scaling of ray tracing with input complexity. We also show the linear scaling of different rasterization hardware. The terrain scene on the left contains one million textured triangles and was subsampled to obtain different sized scenes. Note, that this a worst-case scenario for ray tracing as there is no occlusion in this scene.*

such as shadows, reflection, complex shading effects, and may more can easily be added. This is not true for rasterization hardware.

6. Interactive Distributed Ray Tracing on PC Clusters

So far we have concentrated on simple ray tracing with primary rays only. As we add special ray tracing effects such as shadows, reflections, or even global illumination, we are confronted with the need to trace an increasing number of rays. Due to the "embarrassingly parallel" nature of ray tracing the results achievable on a single processor scale well with the use of multiple processors as long as they all get the necessary bandwidth to the scene database.

Both the system realized by Muuss, as well as Utah system have proven that this is easily realized for shared-memory multiprocessor supercomputer systems (see [24, 25, 28]). In this case all processors have the same high-speed access to the single scene database. However, on PC architecture shared-memory is not available, which has made large distributed ray tracing systems difficult to implement efficiently in the past.

In addition to high performance ray tracing through parallelization we also want to be able to handle very complex scenes. Because ray tracing can easily be run in a demand load mode in respect to accessing the scene data base, it becomes feasible to render huge data sets as long as each frame accesses only a small fraction of the data. Fortunately there is a large number of applications that fall into this category. Examples are large engineering application (e.g. construction of whole airplanes, cars, or power plants), visualization of large data sets such as urban environments, realistic rendering with detailed displacement maps, and many more.

For more and more disciplines it becomes too costly to subdivide or preprocess large models to reduce complexity before being able to display them in interactive setups. Examples are the car industry, where several man month must often be spend on remodeling a car with low polygon counts for visualizing it in a virtual reality environment.

For testing both high performance through distributed rendering as well as the handling of large data sets we have chosen a well-known reference model, which has been used for similar purposes before by Aliaga et al. [3]. This "UNC power plant" model contains 12.5 million individual triangles.

We have chosen the model since it has been used to demonstrate interactive walkthrough using rasterization hardware [3]. However, this required excessive preprocessing that was estimated to take up to three weeks for the entire power plant model. Advanced preprocessing techniques were required in order to reduce the number of polygons that had to be rendered per frame. The techniques included textured depth-meshes, triangle decimation, level-of-detail rendering, and occlusion culling. Ray tracing can render the same scene without any of these advanced and still only semi-automatic techniques. It only requires simple spatial indexing of the scene database.

For stress testing the ray tracing algorithm we also created a larger model by replicating the power plant four times generating a model with 50 million triangles total.

6.1. Overview

We use the classic setup for distributed ray tracing with a single master machine responsible for display and scheduling together with many working clients that trace, intersect, and shade rays. The main challenges of this approach are efficient access to a shared scene data base, load balancing, and efficient preprocessing.

We solve these issues with a novel approach that exploits coherence using the same basic ideas as described above but on a coarser level:

Explicit management of the scene cache: In a preprocessing step a high-level BSP-tree is built while adaptively subdividing the scene into small, self-contained voxels. Since preprocessing is only based on the spatial location of primitives it is simple and relatively fast. Each voxel contains the complete intersection and shading data for all of its triangles as well as a low-level BSP for this voxel. The complete preprocessed scene is stored only once and all clients request voxels on demand. Each client explicitly manages its own local cache of voxels.

Latency hiding: By reordering the computations we hide some of the latencies involved in demand loading of scene data across the network by continuing computations on other rays while waiting for missing data to arrive. This approach can easily be extended by trying to prefetch data for future frames based on rays coarsely sampling a predicted new view.

Load balancing: We use the usual task queue approach based on image tiles for load balancing. Instead of randomly assigning image tiles to clients we try to assign tiles to clients that have traced similar rays in previous frames. Reordering of computations and buffering of work tiles is used to bridge communication and voxel loading latencies, thus achieving almost perfect CPU utilization.

6.2. Data Management in a Distributed Environment

In the original implementation of coherent ray tracing as described above [42] we created a single binary file containing the model. We used the main memory layout for storing data in the file such that we could directly map the entire file into our address space using the Unix mmap-facilities. However this is no longer possible with huge model that generate files larger than the supported address space.

On the other hand we did not want to replicate the entire model of several GB on each of our client machines. This means that demand loading of mapped data would be performed across the network with its low bandwidth and large latency. While this approach is technically simple by using mmap across an NFS-mounted file system, it drastically reduces performance for large models. For each access to missing data the whole ray tracing process on the client is stalled while the operating system reads a single memory page across the network.

Even only a few milliseconds of stalling due to network latency are very costly for an interactive ray tracer: Because tracing a single ray costs roughly one thousand cycles [42], we would lose several thousand rays for each network access. Instead we would like to suspend work on only those rays that require access to missing data. The client can continue working on other rays while the missing data is being fetched asynchronously.

6.3. Explicit Data Management

Instead of relying on the operating system we had to explicitly manage the scene cache ourselves. For this purpose we decompose the models into small voxels. Each voxel is self-contained and has its own local BSP tree. In addition, all voxels are organized in a high-level BSP tree starting from the root node of the entire model (see Figure 9). The leaf nodes of the high-level BSP contain additional flags indicating whether the particular voxel is in the cache or not.

If a missing voxel is accessed by a ray during traversal of the high-level BSP, reordering of computations if performed: the current ray is suspended and an asynchronous loader thread is notified about the missing voxel. Once the data of the voxel has been loaded into memory by the loader thread, the ray tracing thread is notified, which resumes tracing of rays waiting on this voxel. During asynchronous loading, ray tracing can continue on all non-suspended rays currently being processed by the client. More latency could still be hidden by deferring shading operations until all rays are stalled or a complete tile has been traced. We use a simple least-recently-used (LRU) strategy to manage a fixed size geometry cache.

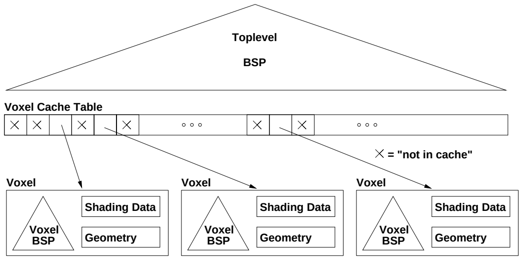

Figure 9: *The data structure used to organize the model data. Voxels are the smallest entity for caching purposes. Their average compressed size is roughly 75 KB.*

The time to load a voxel is strongly dominated by the time to transfer a voxel over the network. To reduce the transferred data volume, voxels are transferred in a compressed form and unpacked on-the-fly on the client side. For packing, we use the LZO compression library [26], that allows fast and efficient decompression of the voxels. Though this compression is more optimized towards speed, its compression ratio is approximately 3:1 for our voxel data. Decompression performance is significantly higher than the network bandwidth, taking at most a few hundred microseconds, thus making the decompression cost negligible compared to the transmission time even for compressed voxels.

6.4. Preprocessing

The total size of single copy of our reference power-plant model is roughly 2.5 GB after preprocessing including the BSP-trees and all triangle and shading data. Due to this large data size an out-of-core algorithm is required to spatially sort and decompose the initial model.

This algorithms needs to read the entire data set once in order to determine the bounding box. It then recursively determines the best splitting plane for the current BSP node, and sorts all triangles into the two child nodes. Triangles that span both nodes are replicated. Note that the adaptive decomposition is able to subdivide the model finely in highly populated areas (see Figure 11) and generates large voxels for empty space.

Once the size of a BSP node is below a given threshold we create a voxel and store it in a file that contains its data (triangles, BSP, shading data, etc.). At this stage each node is a separate file on disk in a special format that is suitable for streaming the data through the preprocessing programs.

The cost of preprocessing algorithms has a complexity of $O(n \log n)$ in the model size. Preprocessing is mainly I/O bound as the computation per triangle is minimal. We are currently using a serial implementation, where each step in the recursive decomposition is a separate invocation of a single program. The resulting files are all located on a single machine acting as the model server.

6.5. Load Balancing

The efficiency of distributed/parallel rendering depends to a large degree on the amount of parallelism that can be extracted from the algorithm. We are using demand driven load balancing by subdividing the image into tiles of a fixed size (32 by 32 pixels). As the rendering time for different tiles can vary significantly (e.g. see the large variations in model complexity in Figures 1 and 13), we must distribute the load evenly across all client CPUs. This has to be done dynamically, as the frequent camera changes during an interactive walkthrough make static load-balancing inefficient.

We employ the usual dynamic load balancing approach where the display server distributes tiles on demand to clients. The tiles are taken from a pool of yet unassigned tiles, but care is taken to maintain good cache locality in the clients. Currently, the scheduler tries to give clients tiles they have rendered before, in order to efficiently reuse the data in their geometry caches. This approach is effective for small camera movements but fails to make good use of caches for larger movements.

6.6. Implementation

Our current setup uses two servers — one for display and one for storing and distributing the preprocessed models. Both machines are connected via Gigabit Ethernet to a Gigabit switch. These fast links help in avoiding network bottlenecks. In particular we require a high bandwidth connection for the display server in order to deal with the pixel data at higher resolutions and frame rates. The bottleneck for the model data could be avoided by distributing it among a set of machines.

For our experiments we have used seven dual P-III 800-866 MHz machines as ray tracing clients. These clients are normal desktop machines in our lab but were mostly unused while the tests were performed. The client machines are connected to a 100Mbit FastEthernet switch that has a Gigabit uplink to the switch the model and display server are connected to.

We have tested our setup with the power-plant model from UNC [3] to allow for a direct comparison with previous work. This also provides for a comparison of algorithms based on rasterization versus ray tracing. The power-plant test model consists of roughly 12.5 million triangles mostly distributed over the main building that is 80 meter high and 40 by 50 meters on either side (see Figure 15).

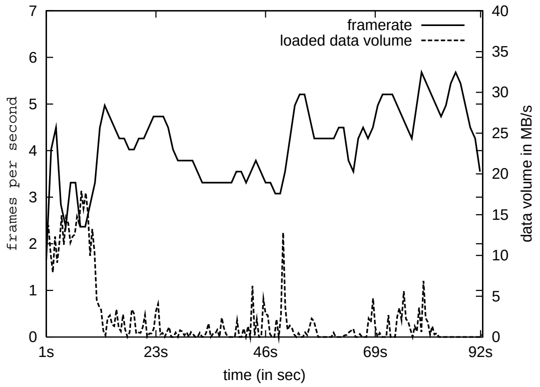

Figure 10: *Frame rate and transfered data rate after decompression during a walkthrough heading from the outside to the inside of the power-plant building. The frame rate is pretty constant around 4-5 fps unless large amounts of data are transfered (at the beginning where the whole building is visible). The frame rates have still been measured without the SIMD optimizations. The newest version runs about twice as fast at about 8–10 fps.*

6.7. Results

Figure 10 gives a compact summary of our overall results. It shows the frame rate achieved by our system as well as the amount of geometry fetched over the course of a walk through the model. The total time of the walkthrough is 92 seconds using all seven clients. Note that we only trace primary rays for this test in order to allow direct comparison with the results from [3]. We only show the results of a single walkthrough, as they closely match those from other tests.

With seven dual CPU machines we achieve an almost constant frame rate of 3–5 fps. However, all numbers are computed with plain C++ code. We have currently disabled the optimized SIMD version of our ray tracing engine because it had not yet been converted to use the new two level BSP tree. Latest experiments with the updated SSE version of the ray tracer has indeed shown the expected speedup by a factor of two and increased the framerate to roughly 8–10 fps, which is about the same frame rate achieved in [3]. Note that we still render the original model with all details and not a simplified version of it.

Figure 11 visualizes the BSP structure that is built by our preprocessing algorithm. The voxel size decreases significantly for areas that have more geometric detail.

In order to test some of the advanced features of ray tracing, we added a distant light source to the model and made some of the geometry reflective (see Figure 12). Of course, we see a drop in performance due to additional rays being traced for shadows and reflections. However, the drop is mostly proportional to the number of traced rays, and shows

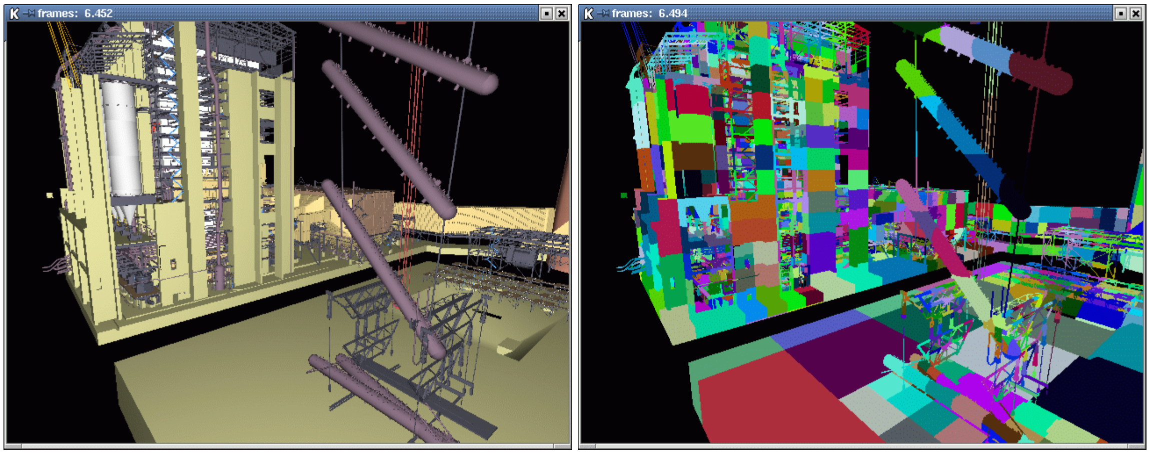

Figure 11: *Two images showing the structure of the high-level BSP tree by color coding geometry to each voxel in the image at the bottom. Voxels are relatively large for the walls but become really small in regions with lots of details.*

little effect due to the reduced coherence of the highly diverging rays that are reflected off the large pipe in the front as well as all the tiny pipes in the background.

We also tested the scalability of our implementation by using one to seven clients for rendering exactly the same frames as in the recorded walkthrough used for the tests above and measured the total runtime. The experiment was performed twice — once with empty caches and once again with the caches filled by the previous run. The difference between the two would show network bottlenecks and any latencies that could not be hidden. As expected we achieved almost perfect scalability with filled caches (see Figure 14), but the graph also shows some network contention effects with 4 clients and we start saturating the network link to the model server beyond 6 or 7 clients. Note, that perfect scalability is larger than seven because of variations in CPU clock rates.

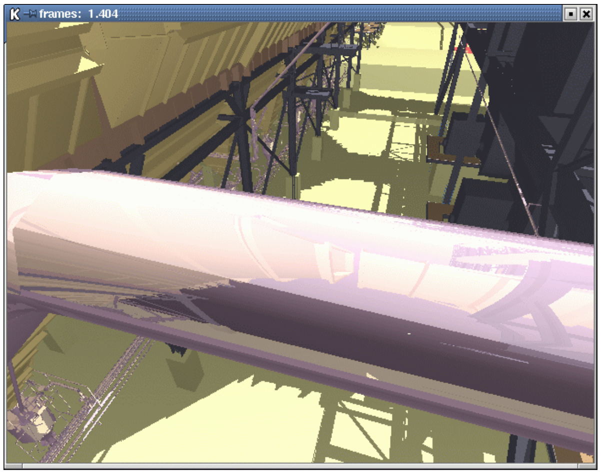

Figure 12: *Shadow and reflection effects created with ray tracing using one light source. The performance drops roughly proportional to the number of total rays traced but the size of the working set increases. Note the reflections off all the small pipes near the ground. Diffuse case: 1 ray per pixel, 4.9 fps, with shadow and reflection (multiple of 2 rays): 1.4 fps.*

Because we did not have more clients available, scalability could not be tested beyond seven clients. However, our results show that performance is mainly bound by the network

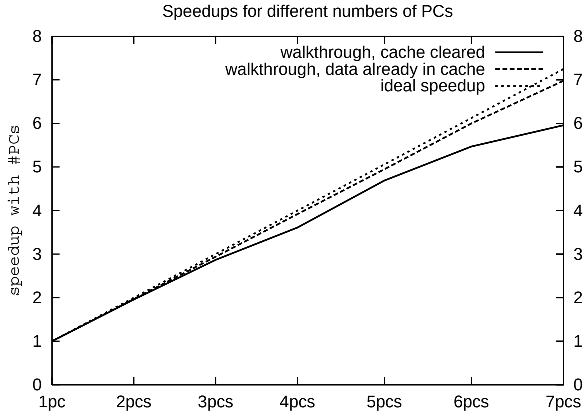

Figure 14: *Our implementation shows almost perfect scalability of from 1 to 7 dual CPU PCs if the caches are already filled. With empty caches we see some network contention effects with 4 clients but scalability is still very good. Beyond 6 or 7 clients we start saturating the network link to the model server.*

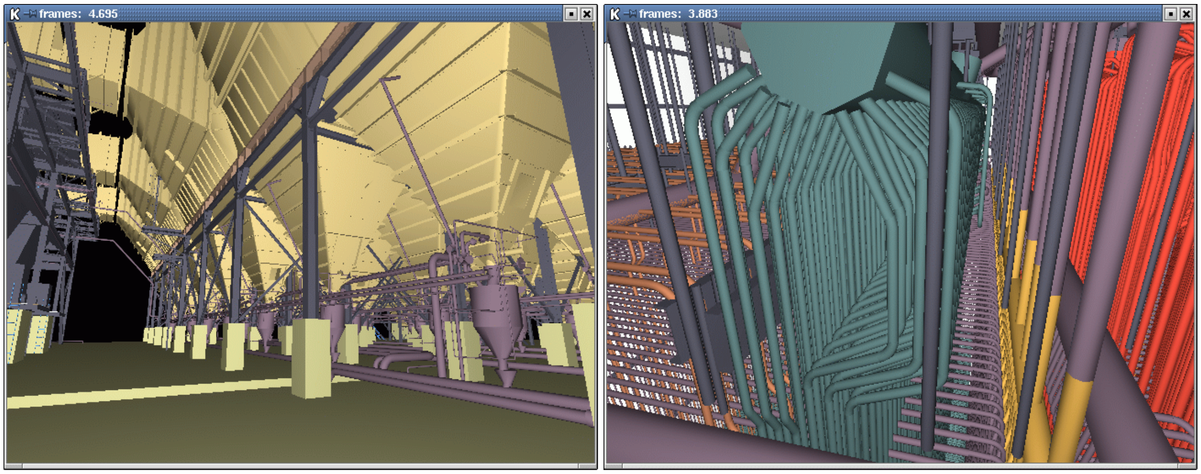

Figure 13: *Two complex view of the power-plant. Both still render at about 8 to 10 frames per second.*

bandwidth to the model server, which suggests that a distributed model data base would allow scalability well beyond the above numbers.

Figure 13 shows some other views of the power-plant showing some of the complexity hidden in this test model.

For a stress test of our system we have placed four copies of the power-plant model next to each other resulting in a total model complexity of roughly 50 million triangles (see Figure 15). Preprocessing time increased as expected, but the frame rates stay almost identical compared to the single model. Essentially the depth of the higher-level BSP tree was increased by two, which hardly has any effects on inside views.

However, for outside views we suffer somewhat from the relatively large voxel granularity, which results in an increased working set and accordingly longer loading times that can no longer be completely hidden during movements. When standing still the frame rates quickly approach the numbers measured for a single copy of the model.

7. Towards Ray Tracing Hardware

The the last two sections we have shown that interactive ray tracing is indeed possible even now. By carefully optimizing the ray tracing algorithms to take advantage of current CPU and memory architectures, we were able to speedup the software more than an order of magnitude.

While these results are encouraging it remains the fact that the computational power of current CPUs is not sufficient for interactive rendering of high-resolution, high-framerate images unless rather large number of these CPUs are employed. If ray tracing is ever to be widely used as a rendering technique for consumer PCs, some hardware support will be necessary. Thus, the question arises if one could build special purpose hardware for ray tracing that would offer better use of the available silicon and offer higher performance at a lower price point.

The first commercially viable approach towards ray tracing hardware has been taken by Advanced Rendering Technologies with their *RenderDrive* [39]. However, their system is not targeting interactive frame rates, but is rather an offline hardware accelerator for non-interactive software ray tracers.

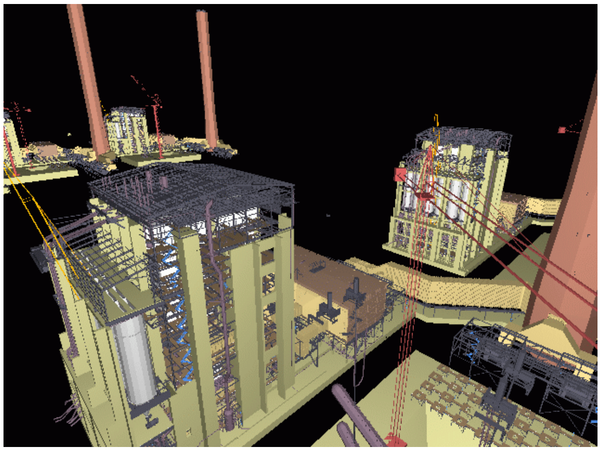

Figure 15: *Four copies of the UNC power-plant reference model with a total of 50 million triangles. In this view a large fraction of the geometry is visible. At 640x480 pixels the frame rate is 3.4 fps using seven networked dual Pentium-III PCs.*

7.1. Hardware Volume Ray Tracing

In 1999, Pfister et al. from MERL have presented the VolumePro 500 board [31], which was the first single-chip realtime volume rendering engine for consumer PCs. It implements ray casting with parallel slice-by-slice processing and renders 500 million interpolated, Phong illuminated samples per second. This is sufficient to render volumes with up to 256^3 voxels in real time. They too pointed out that coherent memory access is crucial to achieve this performance.

Similar projects have been realized as university research projects, namely the VIRIM project [41] of the University of Mannheim, and the VOGUE and VIZARD boards at the University of Tübingen [16].

As a successor to the VolumePro board, Pfister et al. have recently proposed the RAYA architecture [30]. The RAYA architecture extends the volume ray casting approach of VolumePro to a full ray tracing approach for environments consisting of both volumen and geometry primitives. The design of RAYA was inspired by Pharr's *Memory-Coherent Ray Tracing* [32] approach: It uses an explicit scene management in which the scene is subdivided into blocks.

Rays are maintained in queues for each voxels and are explicitly scheduled to one of many processing elements for intersection computations. Due to the accumulation of many rays per voxel this approach maintains coherent access to the geometry data contained per voxel. Rays that did not intersect any geometry in a voxel are forwarded to the respective neighboring voxel.

Even though detailed simulations have still to be performed preliminary results indicate that the RAYA architecture with a programmable shading unit should be realizable within a single chip [30]. This would enable real-time volume ray tracing for medium sized volumes.

7.2. Ray Tracing on Smart Memories

Last year Mai et al. have proposed the *Smart Memories* architecture [22]. A Smart Memories chip is a modular and reconfigurable architecture made up of many processing tiles, each containing local memory, local interconnect, and a processor core (see Figure 16). These components can be reconfigured to match different applications, making it possible that a wide range of architectures from stream processors to speculative multiprocessors can be mapped to this architecture.

The Smart Memories architecture is not a special ray tracing chip, but a highly integrated, general purpose multiprocessor that can be configured to provide a NUMA multiprocessor architecture on a chip. In this configuration it consists of 64 simplified RISC-microprocessors with 128 KB of local memory each and a high-bandwidth, low latency interconnection between the different processors. Tim Purcell from Stanford University is currently designing the *SHARP* ray tracing architecture that maps the ray tracing algorithms directly to the Smart Memories hardware architecture.

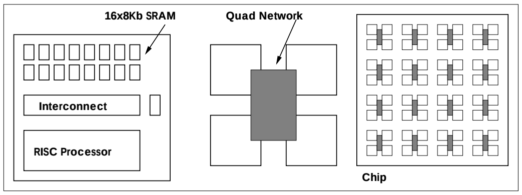

Figure 16: *The Smart Memories architecture: A chip (right) consists of 4 by 4 interconnected quads (middle), each being composed of 4 tiles. A tile consists of a fast general-purpose RISC microprocessors, together with local RAM and a fast interconnection network. Communication bandwidth is 64GB/s between units on the same quad, and still 8 GB/s to off-chip memory.*

Since Smart Memory chips are not yet available, the system is currently designed based on simulations. However, the simulations already provide a good estimate on the theoretical performance of this architecture. They show that the Sharp architecture would indeed be limited only by compute power of the Smart Memory architecture, and would deliver frame rates of 50 frames per second at resolutions of 512x512 pixels (primary rays only) for scenes with several hundred thousand triangles.

7.3. Hardware Ray Tracing on a Custom Design Chip

Custom hardware ray tracing is currently being designed at the Saarland University. This architecture is based on the experience gathered from the interactive software ray tracer already described above and uses the same techniques to reduce bandwidth and exploit parallelism. It implements the complete ray tracing pipeline including ray generation, ray traversal, intersection, and shading in a single chip solution.

This architecture is based on a modular pipeline in which several traversal units are directly connected to a set of intersection units. This direct connection between sets of traversal and intersection units allows a simple, high bandwidth interconnect. The pipeline is feed from ray generators that are integrated with the shading units that compute the reflected light at intersections, generate shadow rays if required, and forward the final pixel values to the framebuffer. In contrast to the SHARP architecture, bundles of rays are traversed and intersected in lock-step thus enforcing coherence and further reducing memory bandwidth to caches and memories.

Similar to the other hardware architectures this project has not been completed yet. However, preliminary results from simulations indicate that a complete ray tracing chip would be realizable with about as much hardware effort as is being spent for todays rasterization chips or CPUs, while being able to deliver full-screen, real-time ray tracing performance of

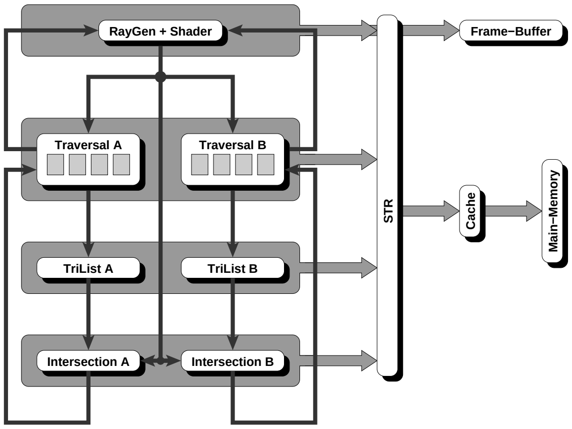

Figure 17: *The ray tracing architecture of Saarland University.*

25+ frames per second, for scenes with up to several million of primitives, three point light sources, and up to 2 secondary rays per pixel. Similar to the software architecture the performance is directly related to the number of rays that need to be processed per frame.

8. Remaining Research Challenges

In the preceding sections we have presented an overview of the current state-of-the-art in interactive ray tracing. Much has already been achieved and much more is still in progress. However, there remain quite a number of open research issues that need to be addressed. In the following we give a short overview and describe them in more detail later.

Dynamic Scenes Fully interactive applications, like virtual reality systems or computer games, require the ability to modify the scene at runtime. Ray tracing, however, requires expensive preprocessing for the computation of the underlying acceleration structure. This typically limits ray tracing to static environments and walkthrough applications. As ray tracing was typically an offline application, where preprocessing time was amortized over the rendering time, little research has been done on dynamic scene updates.

Ray Tracing API In order to be useful for a wide range of software developers, a common API has to be developed for interactive ray tracing similar to the OpenGL API for triangle rasterization. This API will be an important prerequisite for the widespread use of ray tracing based renderers as it will allow applications to be portable across different implementations.

Hardware In order to make interactive ray tracing available for a wide class of users, ray tracing hardware will eventually be needed. Such hardware is currently being designed as described above.

Extensions to Global Illumination As shooting rays is often the most time-consuming part in global illumination algorithms, interactive ray tracing brings interactive global illumination closer to reality. However, in order to exploit the full performance of interactive ray tracing systems, global illumination algorithms have to be explicitly designed to work well within those environments. In particular, large coherence between rays must be maintained in order to stay efficient. Unfortunately, the most promising algorithms are based on the Monte-Carlo technique and generate essentially random rays. Algorithms that are better suited for interactive use must still be developed.

8.1. Dynamic Environments

Because rasterization algorithms process geometry on a triangle by triangle basis the handling of dynamic scenes is fairly simple. During rendering the model is update and the new geometry is being sent to the graphics subsystem. However, this approach becomes more difficult if only the relevant parts of a model are to be send. In this case acceleration structures such as octrees or bounding boxes must be maintained in order to quickly cull entire subsets of the model. Maintenance of these data structures can be quite costly and dynamic models are often not included in these data structures.

Ray tracing depends on these data structures more strongly in order to avoid costly computations and achieve its sub-linear performance. These acceleration structures cannot currently be updated in real-time. As the cost for building the acceleration structure is at least linear in the number of scene primitives, completely rebuilding it for every frame is not feasible.

Until today, relatively few researchers have addressed the problem of dynamic environments: A first suggestion towards ray tracing of dynamic scenes has been made by Parker et al. [28]. In their system dynamic objects are being kept out of the acceleration structure and checked individually for every ray. Of course, this is only feasible for a small number of dynamic objects.

If the majority of the scene remains static, another approach would be to only rebuild those parts of the acceleration structures that are actually affected by moving objects. However, large objects are often contained in a large number of voxels, which would require to update an excessive amount of voxels for a single moving polygon. To solve this problem, Reinhard et al. [34] have proposed a dynamic acceleration structure that is based on hierarchical grids. In order to quickly insert and delete objects independent of their size, bigger objects are being kept in coarser levels of the hierarchy. Thus objects always cover approximately a constant number of voxels, making possible to update the acceleration structure in constant time.

In order to deal with the problem that arises when objects leave the bounding box of the scene, they propose a scheme in which objects are wrapped around the bounding box of the scene: If an object leaves the grid on the right side, it reenters it on the left side. If rays are wrapped around in a similar manner, objects can leave the scene bounds without having to rebuild the entire data structure. Only when the current bounding box differs too much from the size of the grid, the data structure is rebuilt. This way, the cost for rebuilding the data structure can be amortized over several frames. A sketch of this process and some example images are shown in Figure 18.

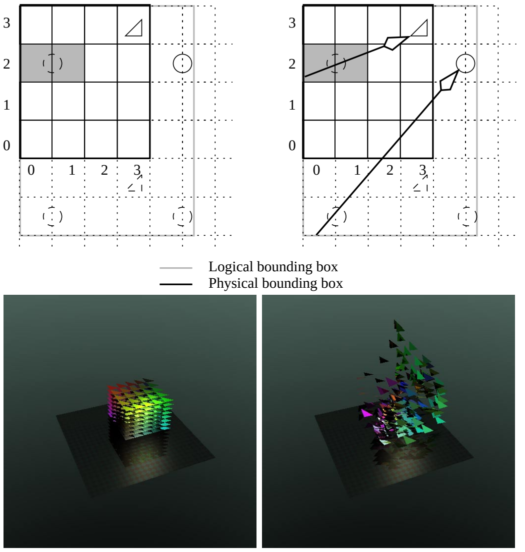

Figure 18: *Top row: The dynamic data structure proposed by Reinhard et al. Left: The sphere has moved outside the physical grid, now overlapping with voxels (4,2) and (5,2). Therefore, it is inserted at the location of the shaded voxels. The logical bounding box is extended to include the newly moved object. Right: Ray traversal through extended grid. Bottom: two example frames from an interactive animation.*

8.2. Benchmarks for Dynamic Scenes

It is obvious that more research is needed before interactive ray tracing will be able to deal with dynamically changing models. In order to evaluate the performance of different approaches a benchmark is very useful. Recently, Lext, Assarsson, and Moeller have proposed the *BART* [21] test suite as a standard testing environment for dynamic scenes. This is intended to augment the Standard Procedural Database [12] (*SPD*), which has served as a benchmark suite for traditional ray tracers, but which does not contain animated scenes.

For designing representative test scenes, they first identified and categorized the different stresses that a ray tracer has to deal with in dynamic environments: these include issues such as hierarchical versus unorganized animation, the teapot-in-a-stadium problem, lower coherence, large working sets, overlap of bounding volumes, and changing object distributions. After having identified and classified these seven stress cases, they propose three parametrically animated test scenes, each concentrating on a different set of issues [21].

8.3. Realtime Ray tracing API

Graphics APIs provide an abstraction layer above the details of the underlying rendering engine, thus enabling programmers to write 3D graphics programs that can run with any graphics boards that supports the specific API. Today's quasi-standard and cross-platform API for 3D graphics programming is OpenGL, which is mainly designed for immediate-mode polygon rasterization. Many of its features are closely related to graphics pipeline as defined by the rasterization approach.

Due to its strong ties to rasterization, OpenGL is not well suited as an API for interactive ray tracing. For example, applications send all polygons repeatedly for every frame of an animation, while ray tracing would benefit from static parts being sent only once. It seems very difficult to design a ray tracing engine that works efficiently if driven by OpenGL or other immediate-mode APIs.

As a consequence, if realtime ray tracing should ever be used by a significant set of applications, a new API has to be developed that is especially designed for the needs of interactive ray tracing. This new API should allow for specifying scene changes incrementally, by posting only those parts of the scene that actually have changed since the last frame.

Additionally, the API should enable the use of all the additional features that rasterization does not support. This specifically includes great flexibility in specifying shaders, hierarchical scene descriptions and instantiations, and tracing of flexible sets of rays. In order to allow for arbitrarily complex scenes it seems useful to support a pull model of operation, where the renderer requests those parts of the scene that are needed for further computation. It should also be possible to specify procedural objects that are only generated once they need to be accessed.

Still, such an API should be as similar to OpenGL as possible, in order to allow for easy porting of existing applications, and to make it possible for the OpenGL-experienced graphics programmer to more easily use interactive ray tracing as an alternative. A good ray tracing API should therefore use as much of the syntax and semantics from OpenGL as possible (e.g. for specifying geometric primitives) while enabling access to the advanced features of ray tracing.

Such an API is currently being designed in a joint project by the graphics groups of Saarland University and Stanford University. It will first be implemented and tested in a software interactive ray tracer, but with the goal to use the same API for hardware implementations. The main issues stem from that fact that the field of interactive ray tracing is still a moving target. Nonetheless, it seems very helpful to define at least a draft API in order to test different approaches to implementing interactive ray tracing and start writing and porting applications to use the new rendering technique.

8.4. Ray Tracing Hardware

The main future challenge for interactive ray tracing is to design, develop and build special purpose hardware. Ray tracing requires significant computational resources for scene traversal, triangle intersection, and shading that should be put onto a single chip. No detailed estimates on the size and the power requirements of such chips are available yet but current estimates indicate that the necessary hardware resources are or will soon be available.

One major problem for ray tracing hardware is the multitude of ray tracing algorithms. Dozens of algorithms and data structures exist for scene traversal and triangle intersection. Though the performance of these algorithms seems to be similar for software implementations, their applicability to hardware implementation has not yet been thoroughly investigated.

8.5. Interactive Global Illumination

Fast global illumination algorithms have been a major research goals for quite some time. Since most global illumination systems are being build on ray tracing, improvements in ray tracing speed (like development of ray tracing hardware) should lead to a similar increase in the speed of global illumination algorithms.

However, most global illumination algorithms depend on shooting very large numbers of rays (often in the order of hundreds of rays, e.g. for Monte Carlo sampling of area light sources, for pixel-supersampling, or for final gathering steps in calculating indirect illumination). Even worse, most of these algorithms tend to generate these rays incoherently. As interactive ray tracing typically tries to exploit coherence among rays, this loss of coherence can drastically reduce the efficiency of the ray tracing algorithm. Even if the ray tracer can be scaled to cope with the increased amount of rays, interactive global illumination algorithms would have to generate a much higher degree of coherence to be efficient.

Another problem is that not all global illumination algorithms are dominated by the cost for shooting rays. For some algorithms, the cost for shading can even be more expensive than the cost for tracing rays (e.g. a photon map query can be several times as expensive as shooting a single ray). Even for mainly ray-based approaches such as path-tracing or bidirectional path-tracing [15, 18, 40], the cost for random number generation, importance sampling of BRDFs, and BRDF evaluations is not negligible.

Therefore, global illumination algorithms that aim at exploiting the advantages of interactive ray tracing should be oriented towards the following design goals:

Coherent Sets of Rays: Rays should be generated coherently. This has been shown to be crucial even today, and will become even more important in the future, as the gap between compute power and memory performance can be expected to widen, especially for hardware implementations.

Exploiting View-Importance: In order to make caching efficient, data that has no (or only minor) effect on the final image should not be touched. If an algorithm touches the entire data set of a large scene instead of only the visually important data, caching can no longer work efficiently.

Efficient Computations: The algorithms should require the computation of as few rays as possible (e.g. by approximative techniques where possible), as well as having as few other costs as possible. This is especially important for future hardware implementations, where ray shooting will most probably be several times cheaper than general-purpose operations such as sampling or BRDF evaluations.

High Degree of Parallelism: The trend towards more parallel units (e.g. by more CPUs or by SIMD extensions) is already apparent, and seems to become even stronger for the future. Future ray tracing hardware is therefore likely to employ lots of parallel ray tracing pipelines. In order to exploit this potential, a interactive global illumination algorithm should be explicitly designed to fit into such a parallel framework. In practice, that means that it should avoid communication between independent rays as much possible.

Interactive framework: If only a few milliseconds of time are available per frame, little preprocessing and global communication is possible . This poses severe restrictions on the algorithms that should work in this setup. Algorithms that require extensive preprocessing and maintenance of global data structures (such as e.g. Radiosity [6] or the PhotonMap [14]) are less suited for interactivity.

9. Summary and Conclusions

This report provides an overview of the current state-of-the-art in interactive ray tracing. We motivated this new research area by pointing out some of the problems of rasterization-based rendering and by describing the advantages that make ray tracing an interesting alternative. Some of the major advantages are the increased flexibility in image generation through the arbitrary handle arbitrary of rays, superior image quality through the simple use of advanced shading algorithms including effect such as shadows, and better scalability with model size and the number of CPUs.

Next we reviewed previous work and classified the different approaches to interactive ray tracing into two main categories, namely those that use approximations in order to reduce the number of rays that must be traced and those that concentrate on accelerating the basic ray tracing algorithms itself. Before concentrating on the latter approaches we gave a brief overview of the most important approximating algorithms.

The main part of this report then concentrated on techniques that accelerate the basic ray tracing algorithm directly. Early systems of this kind were exclusively implemented on large shared-memory supercomputers and are described briefly. We concentrated in more detail on an implementation of interactive ray tracing on low-cost, networked PCs. This implementation improves software performance by more than an order of magnitude and beats even the most expensive rasterization hardware when in comes to more complex models. We showed that even model like the UNC power plant with 12.5 million polygons can be rendered at interactive rates after some simple and fast preprocessing.

In the last part of the report we mention several ongoing research projects that design ray tracing hardware to further increase the speed of rendering with ray tracing and maybe making it competitive with rasterization hardware. Finally we discuss a number of open research issues in interactive ray tracing including dynamic scenes, the design of a ray tracing API, hardware, and the idea of interactive global illumination computations.

An interesting observation is that ray tracing and rasterization algorithms seem to slowly converge towards each other with extensions to rasterization hardware such as occlusion culling, hierarchical z-buffers, advanced shading, and others. The fundamental difference between the two techniques lies in the selection of geometry for rendering. With ray tracing the renderer selects and requests geometry from the scene data base or the application on a ray by ray basis (or packet by packet). Accordingly it can efficiently handle small sets of rays. With rasterization algorithms the application has to select objects based on an conservative estimate of their visibility across the screen and send them to the graphics subsystem to be render by all pixels (i.e. rays) that may be covered by these objects.

Even though ray tracing is still widely believed to be a high-quality but very slow rendering technique, this report clearly demonstrates that this is no longer the case. Software implementations are already available and research for hardware implementations is well under way.

We strongly believe that what we see today is only the beginning of an exciting new field of computer graphics and that it will have effects on a number of related disciplines as well. The simplicity and flexibility of ray tracing, the support for advanced shading, and the way it avoids most of the problems associated with rasterization hardware offer significant benefit for many applications. Even though it is not clear if ray tracing could ever replace rasterization algorithms, these properties ensure that ray tracing will play an important role in the future of interactive 3D graphics.

10. Acknowledgements

We would like to thank all the people that have contributed to this report, either by comments and suggestions or by making images and other material available: In particular we thank Tim Purcell from Stanford University for many fruitful discussions and for offering details of his SHARP system and the underlying Smart Memories architecture. Material was also contributed by Steven Parker, Brian Smits (now at Pixar), and Eric Reinhard from the University of Utah, and Hanspeter Pfister from MERL. Jörg Fischer from the Computer Graphics Group at Saarland University, as well as Marc Stamminger and Jörg Haber from the Max-Planck-Institute for Computer Science in Saarbrücken have also contributed with images and discussions. Thanks are also due to Carsten Benthin and Markus Wagner who have helped tremendously in building our interactive ray tracing system. Discussions with Philippe Bekaert and Alexander Keller helped us to better understand the issues related to global illumination.

References

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

2. T. Akimoto, K. Mase, and Y. Suenaga. Pixel-selected ray tracing. *IEEE Computer Graphics & Applications*, 11(4):14–22, 1991.

3. D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stürzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 199–206, Atlanta, USA, April 1999.

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

5. Kavita Bala, Julie Dorsey, and Seth Teller. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Transactions on Graphics*, 18(3):213–256, August 1999.

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

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

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

9. Steven Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael Cohen. The lumigraph. In *Computer Graphics Proceedings, Annual Conference Series*, 1996.

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

11. Jörg Haber, Karol Myszkowsky, Hitoshi Yamauchi, and Hans-Peter Seidel. Perceptually guided corrective splatting. *Proc. Eurographics 2001 (too appear)*, 2001.

12. Eric Haines. A proposal for Standard Graphics Environments. *IEEE Computer Graphics and Applications*, 1987.

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

14. Henrik Wann Jensen. Global illumination using photon maps. In Xavier Pueyo and Peter Schröder, editors, *Rendering Techniques '96*, pages 21–30. Springer-Verlag, 1996.

15. James T. Kajiya. The rendering equation. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 143–150, 1986.

16. G. Knittel. A pci-compatible fpga-coprocessor for 2d/3d image processing. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '96)*, pages 136 – 145, Napa, CA, April 1996.

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

18. E.P. Lafortune and Y.D. Willems. Bi-directional path tracing. In *Proceedings of Compugraphics '93*, pages 145–153, December 1993.

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

20. Marc Levoy and Pat Hanrahan. Light field rendering. In *Computer Graphics Proceedings, Annual Conference Series*, 1996.

21. 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/.

22. K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular recongurable architecture. *IEEE International Symposium on Computer Architecture*, 2000.

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

24. Michael J. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium '95*, June 1995.

25. 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.

26. Markus Oberhume. LZO-compression library. available at http://www.dogma.net/DataCompression/-LZO.shtml.

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

28. Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive ray tracing. In *Interactive 3D Graphics (I3D)*, pages 119–126, april 1999.

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

30. Hans-Peter Pfister. Raya irgendwas. to be presented at the SIGGRAPH 2001 course on Interactive Ray Tracing, 2001.

31. Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The VolumePro real-time raycasting system. *Computer Graphics*, 33(Annual Conference Series):251–260, 1999.

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

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

34. Erik Reinhard, Brian Smits, and Chuck Hansen. Dynamic acceleration structures for interactive ray tracing. In *Proceedings Eurographics Workshop on Rendering*, pages 299–306, Brno, Czech Republic, June 2000.

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

36. Philipp Slusallek, Thomas Pflaum, and Hans-Peter Seidel. Using procedural RenderMan shaders for global

illumination. In *Computer Graphics Forum (Proc. of EUROGRAPHICS '95*, pages 311–324, 1995.

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

38. 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.

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

40. Erik Veach and Leo Guibas. Bidirectional estimators for light transport. In *Proceedings of the Fifth Eurographics Workshop on Rendering*, pages 147–162, Darmstadt, Germany, June 1994.

41. http://www-li5.ti.uni-mannheim.de/virim.

42. Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, 20(3), 2001. available at http://graphics.cs.uni-sb.de/ wald/Publications.

43. Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. *Eurographics Rendering Workshop 1999*, 1999. Granada, Spain.

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

45. Gregory J. Ward. The RADIANCE lighting simulation and rendering system. *Computer Graphics*, 28(Annual Conference Series):459–472, July 1994.