Realtime Ray Tracing

Meinrad Recheis*
Vienna University of Technology

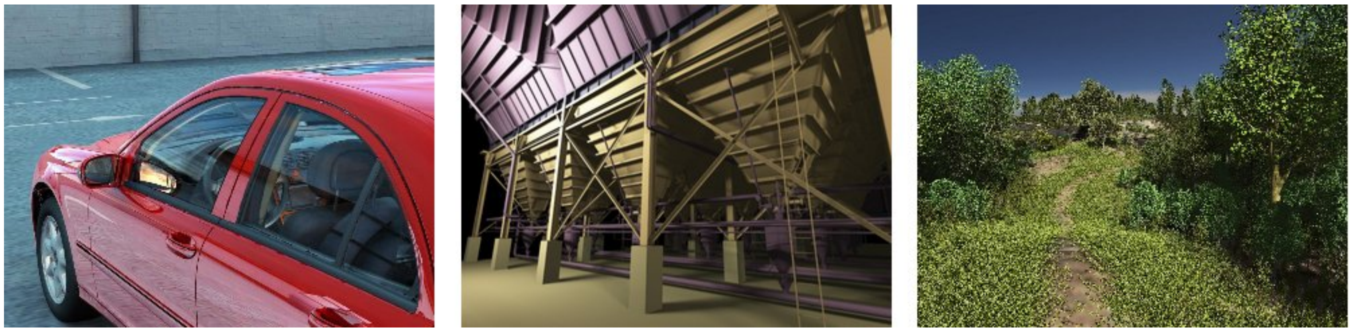

Figure 1: Images rendered in realtime with OpenRT on PC clusters at resolution 640×480. **a)** A Mercedes C-Class model consisting of 320.000 bezier patches and 200.000 triangles for the environment renders between 4.8 and 20 fps on 30 CPUs with an average of 2.4 GHz [Wald et al. 2006]. **b)** Global illumination simulation for a model of a power plant with 50 million triangles renders with 2 fps (when viewed from the inside) on 48 CPUs with 2.4 GHz [Benthin et al. 2003]. **c)** A global illumination simulation for 365.000 individual plant instances with all in all over 1.5 billion polygons renders at 1.7 fps on 48 CPUs with 2.4 GHz and 8GB RAM [Dietrich et al. 2005].

Abstract

Realtime ray tracing produces high quality images at interactive frame rates. While the realtime rendering domain is still dominated by raster graphics, realtime ray tracing recently has become much more powerful. This paper addresses the essence of realtime ray tracing: the various acceleration techniques which yield interactive performance, how coherence between rays can be exploited and effective parallelization. In addition to that, a comprehensive comparison of realtime ray tracing versus modern raster graphics hardware is given. Compared to raster graphics, ray tracing allows highly realistic image synthesis and most importantly, it scales well for massively complex scenes. Considering those advantages and the fact that realtime ray tracing software performance on PCs has finally surpassed the performance of dedicated rasterization hardware for highly complex models, all indications are that ray tracing may replace rasterization based realtime rendering for scientific visualization and games in the near future.

CR Categories: I.3.7 [Computer Graphics]: Ray Tracing—Three-Dimensional Graphics and Realism;

Keywords: realtime ray tracing, realtime rendering, interactive global illumination

1 Introduction

Ray tracing [Appel 1968] is an algorithm that supports highly realistic image synthesis. For this purpose rays are cast from the eye point through the pixels of an image plane and intersected with the scene geometry. The shortest distance from the image to an intersection determines the visibility of the scene through that pixel. The brightness of that pixel is an estimate of the irradiance emitted from the found intersections through that pixel towards the eye point. It is gathered by shooting more rays from that point into the scene i.e. towards all light sources and weighting their energy by the point's bidirectional reflectance distribution function (BRDF). We distinguish between two major methods for the estimation of the irradiance from a point of the scene: *Whitted-style ray tracing* (simply referred to as ray tracing) [Whitted 1980] and *global illumination*. For ray tracing the estimate only includes direct illumination, perfect reflections and refractions. Even more realistic than that are global illumination simulations which account for the diffuse inter-reflections from the complete scene arriving in that point, as described by the rendering equation stated by Kajiya [1986].

Due to the high intrinsic computational cost of ray tracing it has not been considered for realtime rendering for a long time. Nevertheless, with the increasing computation power of modern CPUs, interactive ray tracing and even interactive global illumination have become reality and offer a number of benefits over traditional rasterization based algorithms.

There are many approaches to achieve interactive frame rates for ray tracing. The most successful ones accelerate those parts of the ray tracing algorithm which have the highest computational cost. Moreover, the inherently parallel nature of ray tracing can be massively exploited by using SIMD instructions on modern CPUs, by distributing workload in PC clusters or by using dedicated highly parallel ray tracing hardware. The effectiveness of parallelization can even be increased by taking advantage of coherence between rays. All these techniques combined with cache-awareness can improve the performance of a ray tracer by several orders of magnitudes.

Section 2 explains the computational cost of different parts of a ray tracing algorithm. All optimizations presented in this paper are based on this knowledge. A very important class of optimizations are represented by the spatial acceleration structures which are discussed in section 3. Section 4 introduces coherent ray tracing and its application for interactive global illumination. To effectively take advantage of coherence, parallelization and caching, explained in section 5, are necessary. Last but not least, section 6 discusses the advantages and limitations of realtime ray tracing in comparison to rasterization.

*e-mail: meinrad.recheis@gmail.com

2 The Computational Complexity of Ray Tracing

In order to maximize optimization profits one needs to carefully analyze the computational complexity of the ray tracing algorithm. The most expensive operation is the search for ray-geometry intersections. To reduce the number of primitives to be intersected acceleration structures like kd-trees are used. They typically reduce the complexity for finding a small set of potentially intersecting geometric primitives to be $O(\log n)$ if the number of primitives is n. Yet, the tree traversal operations are the most frequent operations per pixel. A carefully chosen acceleration structure and a scene traversal algorithm that is optimized for that structure are most essential.

Computing ray-geometry intersection points is also very costly because it happens quite often per pixel. The number of intersections to be calculated depends on the spatial resolution of the search structure. The higher the resolution the deeper the search tree. In other words, there is always a trade-off between the number of traversal operations and the number of intersection calculations which can be controlled by adjusting the maximum number of geometric primitives in a volume element of a scene acceleration structure (see section 3). Typically the number of intersection calculations is less than the number of acceleration structure traversal operations for a given pixel.

In contrast the computational complexity of texture mapping and shading is insignificantly compared to the complexity of the other operations since it has to be done only once for a found intersection point. From this analysis one can tell that optimization of the scene traversal and reduction of ray-geometry intersections yields the most performance payoff.

3 Spatial Acceleration Structures

Spatial acceleration structures have the most impact on rendering performance, because they restrict the potentially visible set for a given ray to a very small number of primitives independently of the overall size of the scene. This makes visibility queries by shooting rays into the scenes very efficient.

A spacial indexing structure is a tree of voxels[1] or bounding volumes. Each volume element in the tree may contain some subelements (commonly referred to as *children*) and a list of geometric primitives and their shading information. Intersecting a ray with the scene means traversing the acceleration structure by intersecting the ray with the children of a bounding volume recursively until the ray hits an opaque primitive or an empty leaf of the tree.

The following sections discuss various acceleration structures which have been successfully used for realtime ray tracing.

3.1 The Kd-Tree

The kd-tree (short for k-dimensional tree) [Jansen 1986] is a space partitioning data structure for organizing objects in a k-dimensional space. It is a special case of the BSP tree (binary space partition tree). A kd-tree uses only splitting planes that are perpendicular to one of the coordinate system axes. This differs from BSP trees, in which arbitrary splitting planes can be used. Building a static kd-tree from n points takes $O(n \log n)$ time and orthogonal range search takes $O(\log n)$ time where n is the number of cells in the kd-tree. Along with clever heuristics for subdivision, kd-trees have been successfully used for realtime ray tracing [Wald 2004, Havran 2001, Reshetov et al. 2005].

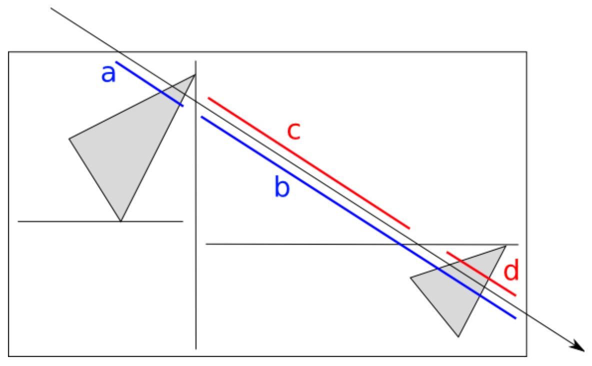

Figure 2: Ray intersection with a kd-tree. The ray intersects with the top level splitting plane and is therefore split into the ranges a and b. First, the left tree is checked recursively against a. The right tree is checked against b only if a does not hit any triangle. In that case b is split up to c and d. Since the upper cell is empty c is immediately discarded and d is tested against the triangle in the lower cell.

Because of his axis aligned splitting planes the range search for a given ray can be implemented very easily and is very fast. Since all normals of the subdivision planes coincide with one of the coordinate axes, scalar products and object volume element intersection tests are numerically robust and efficient. In addition to that, the runtime of ray shooting against a kd-tree in a realtime renderer can be greatly improved by careful tree construction [Havran 2001]. That construction is based on a greedy algorithm with a cost function called the *surface area heuristic* [MacDonald and Booth 1989; Havran 2001]. The performance of a kd-tree is further improved by a traversal algorithm which allows to traverse entire packets of rays into the scene [Wald et al. 2001a].

One problem common to all spatial partitioning schemes is that objects can reside in more than one volume element. This problem can be reduced by allowing only split planes which intersect just vertices of an object. However, as a consequence of the a priory unknown number of references in the kd-tree, memory management becomes an issue during the construction of the hierarchy. For efficient creation of the data structure complete pre-allocation would be most efficient. There are heuristics to predict memory size of a kd-tree but they do not work well for arbitrary scenes. When the heuristics are too pessimistic far too much memory is allocated, or even worse, when the predicted size is too small several times of expensive reallocation may be the consequence. Implementations that do not pre-allocate the memory but use dynamic data structures instead suffer from memory fragmentation. There are acceleration structures which do not suffer from this problem, i.e. the bounding interval hierarchy.

Another general disadvantage of the kd-tree is its inability to be adjusted efficiently for dynamic scenes. Complete or partial rebuilding of a kd-tree for every frame is too costly for complex scenes. However, it is possible to overcome this problem by using a two-level kd-tree. The scene acceleration structure is divided in low-level kd-trees representing the objects which reside in a high-level kd-tree for the scene [Wald et al 2003]. Usually large parts of a scene are static and therefore do not need to be updated. The low-level structures for animated objects can be rebuilt if necessary. In

[1]A voxel (a fusion of the words *volumetric* and *pixel*) is a volume element on a regular grid in three dimensional space.

some cases where the animation of whole objects can be described by an affine transformation, there is no need to rebuild the low-level tree. Instead the intersecting rays are transformed with the inverse transformation [Lext and Akenine-Möller 2001]. This trick does not help in the case of unstructured motion. Here the local low-level trees have to be rebuilt for every frame and the high-level tree might be invalidated too. The costly rebuilding of a local tree can be done on demand, i.e. when a ray hits such object. This way only the local acceleration structures of visible objects need to be rebuilt. It is easy to see that this approach works very well for highly occluded environments like indoor scenes but will fail for open scenes where lots of objects exhibiting unstructured motion are visible.

3.2 The Bounding Volume Hierarchy (BVH)

A bounding volume hierarchy [Rubin and Whitted 1980] is a tree of bounding volumes where each bounding volume stores references to child nodes and each leaf node has a list of geometric primitives. In addition to that, the bounding volume is guaranteed to entirely enclose the bounding volumes of its descendants. Each geometric primitive is exactly one leaf. Bounding volumes can be represented by arbitrary shapes but axis-aligned boxes are the best choice for realtime rendering.

In contrast to spatial subdivision structures such as the kd-tree, the bounding volume hierarchy divides the object hierarchy, and a given object hierarchy is more robust than a given subdivision of space. Bounding volume hierarchies differ from space partitioning data structures like kd-trees, in that overlapping bounding volumes are allowed and empty space is not explicitly represented. Due to the fact that overlapping bounding volumes are allowed, a bounding volume hierarchy can be quickly updated for every frame without invalidating the whole structure. This advantage makes them well suited for dynamic scenes. Moreover, bounding volume hierarchies can be built very efficiently because the memory requirements can be bounded a priori.

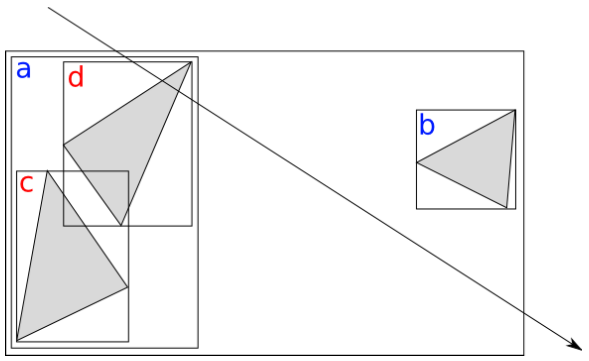

Figure 3: Ray intersection with a traditional BVH. The ray has to be intersected with both boxes a and b because they are not ordered. Since the ray overlaps with a, it is tested against c and d too.

The biggest disadvantage of bounding volume hierarchies is, that unlike space partitioning structures the child volumes are not spatially ordered, therefore it is not possible to abort the intersection procedure early on first hit (see Figure 3). Even if a ray hits a child, it is not guaranteed that there are no other sibling volumes which would yield intersections in front of it. Consequently all children of a parent volume have to be tested against a ray that hits that volume. This is why bounding volume hierarchies in their original form perform worst of all spatial acceleration structures for visibility queries. Despite this, recent research shows that bounding volume hierarchies can perform as well as kd-trees. Even better, they are well suited for fully dynamic scenes [Wald et al. 2007]. To make bounding volume hierarchies more effective, the same surface area heuristic as for kd-trees can be applied to find an optimal tree of bounding volumes for a given set of geometric primitives [Wald and Havran 2006]. While it has substantial impact on rendering times, it does not make bounding volume hierarchies as effective as kd-trees. It is the clever combination of ordering the children of a bounding volume along an axis to allow *early exit on first hit* and *early exit on miss* strategies, and a special traversal algorithm that determines the order of the bounding volumes from properties of the ray[2]. Most important, the traversal algorithm can traverse the bounding volume hierarchy for a large number of rays in parallel (usually 8×8 or 16×16 rays) with very few ray-box intersection tests. This is achieved by testing the boxes against a frustum representing a whole packet of rays, not against the single rays [Reshetov et al. 2005; Wald et al. 2007]. The frustum traversal algorithm is another advantage of bounding volume hierarchies over kd-trees which can hardly use this optimization because they have to compute the entry and exit distances to the box for all rays in the packet [Wald 2004].

3.3 The Bounding Interval Hierarchy

The bounding interval hierarchy is a crossover of space partitioning and bounding volume hierarchies combining the advantages of both. Unlike bounding volume hierarchies which store a full axis-aligned bounding box for each child, the idea of the bounding interval hierarchy is to only store two parallel planes perpendicular to one of the axes of the parent volume. Given a bounding box, an axis and the geometry inside the box, the first plane defines a child volume that contains all primitives that are on the left side of the box with respect to the axis. The second plane defines a volume that contains all primitives that are on the right side of the box with respect to the given axis. The two volumes may overlap. If not a third volume which is empty by definition resides between the two child volumes. The bounding interval hierarchy as acceleration structure for realtime ray tracing has been shown to outperform other fast acceleration structures significantly [Wächter and Keller 2006]. Also Wächter and Keller point out in their paper that bounding interval hierarchies can be combined with frustum traversal but they have not yet presented an implementation and performance measures for it. Unfortunately, a performance comparison between bounding interval hierarchy and the new efficient application of bounding volume hierarchies (see section 3.2) has not yet been published at the time of this writing. Therefore it is not clear which approach is the fastest.

The efficiency of the bounding interval hierarchy is due to the advantages inherited from space partitioning structures, namely ordered traversal which allows early exit (see Figure 4). Opposite to space partitioning, bounding interval hierarchies have a fixed preallocatable size depending on the number of primitives. The advantages inherited by bounding volume hierarchies are that the volume elements can overlap and thus allow efficient update of the structure for dynamic scenes. But the key to the exceptional performance of bounding interval hierarchies is a new global heuristic for partitioning and the idea that the exact structure of untouched volumes can be computed on demand. The global heuristic differs from the conventional approaches which use a greedy algorithm in combination with the surface area heuristic. Instead it is a non-greedy heuristic which is cheap to evaluate because it does not explicitly analyze

[2]Note that this modifications of the bounding volume hierarchy yields a spatial indexing structure that is very similar to the bounding interval hierarchy discussed in section 3.3

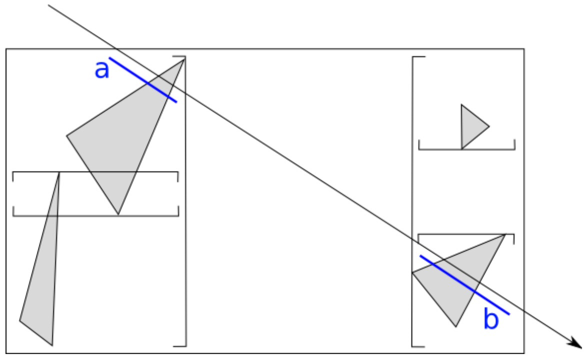

Figure 4: Ray intersection with a bounding interval hierarchy. The ray intersects both vertical split planes and is divided into the ranges *a* and *b*. *a* is tested against the left tree first by starting with the top triangle because *a* is entirely before both split planes with respect to the (vertical) axis. If that test yields no intersection point *a* will not be tested against the lower triangle. Then *b* is tested against the right tree by testing against the lower triangle immediately because *b* is entirely after the second split plane.

the objects in the scene beforehand. The heuristic uses so called *candidate planes* which divide a volume exactly in the middle of its longest side (see Figure 5). The candidate planes are used to sort the contents of the volume into left and right buckets. After that the coordinates of the primitives in the buckets are spatially sorted along the axis perpendicular to the candidate split plane. The structure of the sorting algorithm is identical to quicksort and consequently runs in $O(n \log n)$ on the average. After that, the two splitting planes are defined by the maximum coordinate of primitives from the left bucket and the minimum coordinate of the primitives of the right bucket. The clue is, that the sorting and definition of splitting planes may be omitted for volumes that are not visible, resulting in very high construction performance. As a consequence the construction of the bounding interval hierarchy data structure is faster than others by orders of magnitudes and can be done directly without the need for a preprocessing step. Thus, the application of bounding interval hierarchies for realtime ray tracing is called *Instant Ray Tracing* [Wächter and Keller 2006].

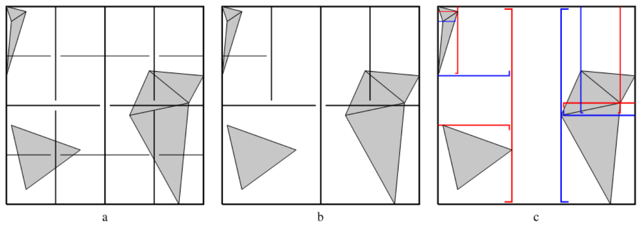

Figure 5: Illustration taken from Wächter and Keller [2006]: a) Split plane candidates from hierarchically subdividing along the longest side of the axis-aligned bounding box. b) Used candidates and c) resulting bounding interval hierarchy.

4 Coherent Ray tracing

Rays are coherent, if they point into similar directions and are relatively close to each other with respect to the size of the scene. Primary rays, (rays that are cast through the pixels of the image towards the scene) usually are coherent, especially if they are cast through adjacent pixels. The same applies to shadow rays which are used to sample the light sources to determine the intensity of a given point in the scene. Reflection and refraction rays used to sample transparent materials are usually less coherent (see Figure 6). Consequently scenes with lots of transparent or highly refractive materials perform sub-optimal in particular when such objects occupy a large area on the screen. Even worse, the rays collecting indirect illumination which are cast by Randomized Quasi Monte Carlo techniques for global illumination simulations actually are not coherent at all. This makes global illumination simulations difficult to implement with coherent ray tracing but, as described later in this section, not impossible.

All modern interactive ray tracers exploit coherence between rays to improve scene traversal and ray-geometry intersection performance. Together with good acceleration structures the traversal algorithms that take advantage of coherence between rays are responsible for the dramatic speed up that makes realtime performance possible. This section describes various techniques to utilize the coherence between rays.

4.1 Tracing Packets of Rays

A packet of rays is simply a number of rays which may be more or less coherent. The definition of a packet does not strictly require all the rays to start from a certain point and also does not require the rays to be parallel or sharing direction signs. However, for rays with a common starting point and similar directions coherence can be exploited more easily.

Consider four rays through a block of four adjacent pixels on the screen, as illustrated in figure 7. These rays are very likely to hit the same volumes of the acceleration structure and the same scene geometry. To put them in a packet and handle them together has many advantages. First, it allows to reuse the loaded data for multiple intersection computations. This is also called *effective cache utilization* which is discussed in detail in section 5. Second, the computations for the whole packet of rays could be done in parallel on an architecture that allows parallel computations for multiple data. For example, by using SIMD extensions on todays CPUs intersection computations for four rays can be done in parallel, effectively reducing the runtime of algorithms by a factor of four. While PCs can only handle packets of 2×2 [Wald et al. 2001a], on special ray tracing hardware the packet sizes may be up to 8×8 or higher, i.e. the SaarCOR ray tracing board by Schmittler et al. [2002].

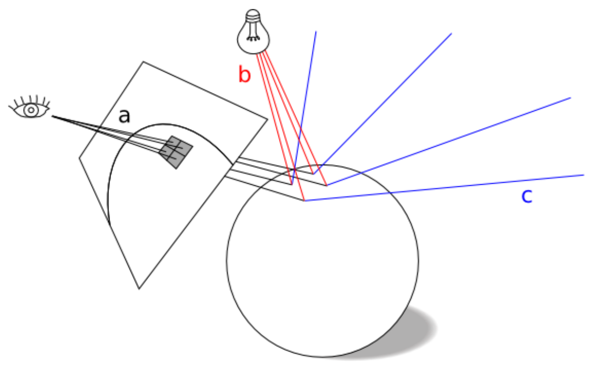

Figure 6: a) A packet of four coherent primary rays. b) Coherent shadow rays. c) Reflection rays are typically not coherent for strongly curved surfaces.

Certainly, during packet traversal it is possible to have some rays of a packet which hit different bounding volumes or space partitions. If that happens, the packet of rays either needs to be tested against both primitives with those rays masked out that do not overlap the volume, or the packet is split up and both parts are traversed independently. According to Wald et al. [2001a], the traversal overhead for packets is relatively small and is more than paid off by the overall performance improvement.

4.2 Ray Tracing with Ray-Packet-Frustums

Even better than handling packets of rays is the relatively new idea to handle a packet-representing frustum. The frustum of a packet of rays can be a few border rays or a number of planes describing the smallest volume that contains the packet of rays. The outstanding advantage of using a frustum to represent a bundle of rays is the fact that determination of the potentially visible set can be done for a large number of rays with only a few intersection computations. In some cases it is not even necessary to know the single rays within the frustum. Similar to view-frustum-culling for rasterization based algorithms, the scene can be culled against the ray frustum to reduce the set of potentially intersecting primitives (see Figure 7). This set can then be intersected with the packet of rays inside the frustum to find out the actual intersections. Using frustums for scene traversal can reduce the number of intersection computations dramatically. Since intersection computations are costly the performance gain is significant.

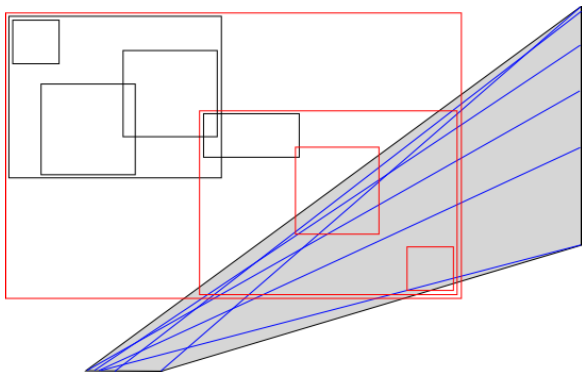

Figure 7: A frustum of coherent rays traverses a bounding volume hierarchy. The rays represented by the frustum are blue. The potentially visible set of bounding volumes is red.

4.3 Multi Level Ray Tracing (MLRT)

The multi level ray tracing algorithm by Reshetov et al. [2005] makes very effective use of coherence by traversing ray packet frustums. The idea is to tile the image into large blocks and packing all the primary rays through the pixels of one block into one large frustum. Depending on the geometric complexity of that block the frustum may be very effective, for example when a large object near the camera covers that block entirely. Or else, the block and the frustum are divided into a number of smaller ones. This procedure is performed recursively until a frustum hits only one scene acceleration volume or a minimum block size is reached. In other words, the local geometric complexity of an image area is directly derived from the scene acceleration structure. If the frustum is too large for that area, it overlaps a lot of volumes in the acceleration structure, hence the geometric complexity of that image block is too high for its frustum. In any case, it is automatically adapted to the local complexity by recursive subdivision.

Due to the fact that the multi level ray tracing algorithm automatically adapts to the local geometric complexity of image regions, it is possible to effectively budget rays. For regions of low complexity the algorithm not only shoots significantly less rays, it also does not have to perform frustum traversal for a lot of packets which altogether leads to a huge performance boost.

The multi level ray tracing algorithm is much faster than other realtime ray tracers for direct illumination evaluation without shadows. However, since the technique applies to primary rays only, it is not capable of efficiently rendering many features supported by recent realtime ray tracers including soft shadows, global illumination, reflections and refractions. To support these a multi level ray tracer would have to be extended with concepts from other recent realtime ray tracing systems.

4.4 Using Coherent Ray Tracing for Interactive Global Illumination

Interactive graphics applications have been limited for simple direct illumination that results in an artificial appearance. But, for highly realistic image synthesis global illumination simulations are indispensable. Unfortunately most algorithms rely on monte-carlo random walks for diffuse light propagation simulation which need to shoot large numbers of totally randomly oriented rays which are not coherent at all.

Wald et al. [2002] introduced an algorithm which cleverly overcomes these problems. Their ray tracer was the first that could render a global illumination simulation at interactive frame rates. They use the idea of *instant radiosity* [Keller 1997] to simulate the diffuse light transport in the scene. They generate sets of *virtual point lights* which are shot from the original light sources into the scene along monte-carlo random walks. The indirect illumination is then calculated by computing the shadows of these virtual point lights. Due to performance reasons the set of virtual point lights can not be very large leading to a noticeable discretization error in the image. Be that as it may, because successive images from the same camera position are calculated with different random numbers and can be accumulated, the quality is progressively refined for static scenes. This accumulation effectively removes the bias and converges against the true solution. The algorithm converges within a few frames, rendering it usable for interactive walkthroughs of static scenes. Yet, during interaction or animations accumulation of successive frames is not possible, that is to say, the system produces sub-standard image quality during interaction (see Figure 8).

Note that unlike traditional radiosity the instant radiosity approach is a discretization of the light transport not of the receiving geometry. Hence, tessellation of the geometry is not necessary and there are no aliasing artifacts resulting from approximation of the light transport *in the converged image* as is the case for traditional radiosity solutions.

Again, the key to interactive frame rates in this context is the effective use of coherent rays. Even for the randomized illumination simulation which yields a large number of non-coherent rays for each pixel there is a way to exploit coherence. A technique known as *interleaved sampling* facilitates coherence [Molnar 1991; Keller and Heidrich 2001]. The motivation for interleaved sampling is that generating a different set of point lights for each pixel is too costly and using the same set of point lights for each pixel causes aliasing artifacts (see Figure 8a). With interleaved sampling the image is divided into small blocks where each pixel has a different set of

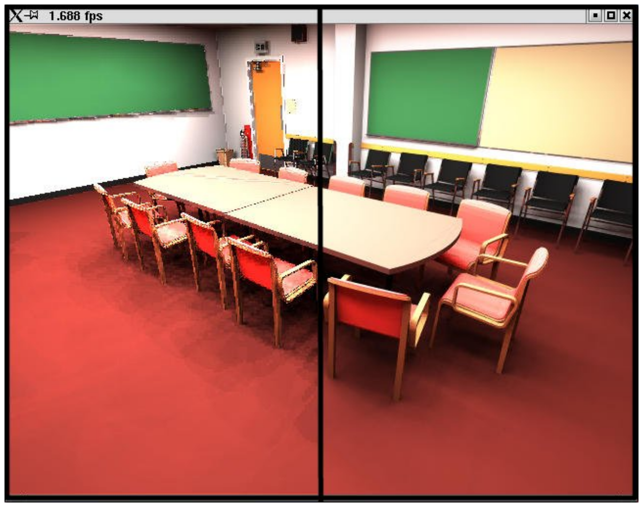

Figure 8: Interactive global illumination. Illustration made up from images taken from Wald et al. [2002]. Left side: artifacts during interaction, b) Right side: converged solution after a few frames.

virtual point lights but the same pixels of different blocks share the same set of point lights. Interestingly, this also has the side effect that by assigning the same set of point lights to the same pixels in different blocks, a number of coherent rays is generated which sample the same point lights. At the same time the number of different sets of randomly generated light sources is greatly reduced which further decreases the number of random walks to be calculated for these point lights.

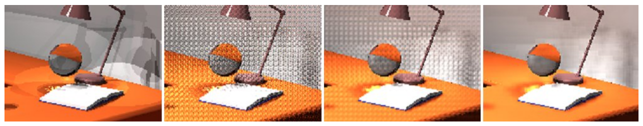

Figure 9: Interactive global illumination. Images taken from Wald et al. [2002] a) No interleaved sampling, no discontinuity buffer b) 5×5 interleaved sampling, no discontinuity buffer c) 5×5 interleaved sampling, 3×3 discontinuity buffer d) 5×5 interleaved sampling, 5×5 discontinuity buffer

While interleaved sampling increases coherence between rays and reduces the number of virtual point lights, this technique replaces the aliasing artifacts exhibited of only one set of point lights by structured noise (see Figure 8b). This noise must then be eliminated by using a filtering technique called *discontinuity buffering*. The discontinuity buffer stores the depth and a normal vector for each pixel. For example, at a silhouette the depth function is discontinuous and at a sharp edge the normal function is discontinuous. Since the the indirect illumination is a piecewise smooth function the variance can be effectively reduced by interpolating it between neighboring pixels where the geometry is continuous according to the discontinuity buffer. Of course, the indirect illumination can not be blurred at geometric discontinuities such as edges or silhouettes. If you take a close look at Figure 8d, you will see that it is extraordinary noisy on the edges. Broadly speaking, this is not very distracting, because noise that is superimposed on discontinuities is less perceivable according to Ramasubramanian et al. [1999].

5 Parallelization and Caching

Many realtime ray tracing algorithms are computationally expensive. For each frame hundreds of thousands of rays have to be traversed through acceleration trees and then tested against geometric primitives for intersections and at each intersection point custom shader code must be evaluated. Especially interactive global illumination imposes heavy computational cost on ray tracing systems due to the high complexity of the simulation of diffuse light transport.

Fortunately, it is in the nature of ray tracing to be inherently parallel. But even more important than that, it is also amazingly scalable. Ray tracing algorithm can be parallelized to an arbitrary number of processes on an arbitrary number of computers, if the necessary communication bandwidth is provided. However, the memory access bandwidth and especially the network bandwidth of ray tracing clusters are limited. To cope with these bottlenecks, effectively using the available computational resources caching is an important strategy.

5.1 Parallelism through SIMD Extensions

As pointed out before, the ability of modern CPUs to execute parallel computations enables the efficient evaluation of scene traversal and intersections for packets of rays for the same cost as a single ray. This is possible on off-the-shelf PCs by using SIMD (single instruction multiple data) extensions offered by several modern processor architectures. They allow to execute a floating point instruction in parallel on multiple data values (typically two to four), thereby yielding a significant speedup. They usually also contain instructions for explicit cache management such as prefetching.

The first application of SIMD instructions for realtime ray tracing was published by Wald et al. [2001a]. They used it to cast and intersect packets of four rays simultaneously. To make traversal and intersection tests which are the most costly operations as fast as possible, they designed their algorithm in such a way that it runs completely on the caches. Since cache memory access is more than an order of magnitude faster then RAM access, the resulting speedup of their system's ray shooting performance was enormous.

Since on a PC architecture data is transferred between memory and cache in entire cache lines of 32 bytes, the effective cost of memory access is not directly related to the number of bytes to be read but to the number of cache line transfers. According to that it is generally beneficial to carefully lay out the data structures for optimal cache usage, i.e. by aligning the structures to a multiple of 32 bytes. In addition to that, in their implementation Wald et al. keep data together if and only if it is used together, which keeps the data structures small and thereby too accounts for optimal cache utilization. They effectively hide memory access latencies almost completely by designing their algorithm to cleverly predict what memory will be accessed and prefetching it, such that it is available in a cache when it is needed for computations. To make this possible, algorithms need to be simple enough to precisely predict what memory will be accessed in the near future.

5.2 Distributed Ray Tracing

Distributed execution of parallel algorithms on a networked PC cluster is a very cheap way to increase the computing power of a system. Again, this is very easy for ray tracing due to its general scalability.

A typical distributed interactive ray tracing system, as described by Wald et al. [2001b], consists of a display server, a scene database server (multiple databases are possible) and a number of ray tracing clients connected via a high-bandwidth network. The display server executes the interactive user application and assigns jobs to the clients. In particular, the server divides the image into many small tiles and assigns them to the ray tracing clients. These need to access the scene geometry database server to get the actual volumes of the acceleration structure they need to render the tile. In the best case they already have the geometry available in their local geometry caches. Finally, when the tile is complete a compressed stream of pixels is sent back to the display server. Wald et al. note that the transfer volume of pixel-data is the limiting factor of such a realtime ray tracing cluster. When the network connection to the display server is saturated with pixel streams, the computing power of the cluster cannot be further increased by adding clients. In other words, the scalability of a cluster like this is only limited by the network bandwidth.

A realtime ray tracing cluster needs highly sophisticated cache management and latency hiding techniques. After rendering the first few frames, the scene geometry is evenly distributed amongst the rendering client's caches. To reduce the number of transferred geometry data over time and thereby increasing rendering speed of successive frames it is essential to effectively reuse the cached geometry. To accomplish this, the display server reprojects tiles from the next frame to the camera position of the last frame. Then it assigns the new tiles to those clients who recently rendered the matching tiles in the last frame. Moreover, the clients always queue at least one more job while they are busy in order to avoid idle times while waiting for the next job assignment. Likewise, to hide the network latency for geometry transfer from the scene database a client suspends processing of a ray which is waiting for geometry in favor of rays which can be processed immediately because the geometry they need is already available in caches.

6 Realtime Ray Tracing versus Realtime Raster Graphics

What are the essential differences between realtime ray tracing and raster graphics? This section tries to answer this question by comparing several aspects of the two rendering principles.

Realism. Rasterization accounts for direct illumination only. Shadows and reflections can be faked but their realism is very limited. In particular self reflections of a concave object are not possible. Physically correct refractions are not possible either with pure rasterization. On the contrary realtime ray tracing features correct pixel-accurate shadows from point lights, soft shadows from area lights, correct display of reflecting materials and physically correct refractions[3] (i.e. glass, water). The rasterization pipeline can handle translucent materials but requires to render them back to front which involves a sorting overhead. Altogether realtime ray tracing offers a whole lot of more realism and physical correctness for material appearance and lighting.

Scene Complexity. The most important advantage of ray tracing over rasterization is its logarithmic complexity in the number of geometric primitives, whereas the complexity of raster graphics scales only linear. As a result, massive scenes consisting of billions of triangles can still be ray traced with interactive frame rates. This is due to the logarithmic complexity of search algorithms in scene acceleration trees. Actually realtime ray tracing has built-in visibility culling. Similar techniques are used with raster graphics to reduce the workload of the GPU but the user has to take care of it. With realtime ray tracing all the advanced and highly complex visibility determination algorithms currently needed for realtime rasterization of complex scenes are no longer necessary.

[3]Wavelength dependent refraction is not yet possible in realtime

Computation power. To some extent, ray tracing requires significantly higher computation power than rasterization for scenes of low complexity because rasterization interpolates lighting information between vertices across the pixels in a triangle in image space while ray tracing evaluates lighting per pixel. Then again for very complex scenes, where most of the triangles are smaller than a pixel, raster graphics are not considered to be significantly faster. To deliver a high frame rate on a single PC it is clear, that realtime ray tracing hardware is necessary. Several prototypes have been developed but they are not yet ready for prime time [Schmittler et al. 2002; Schmittler et al. 2004; Woop et al. 2005].

Parallelization. Ray tracing is inherently parallel. Each pixel, for instance, may be calculated on a different computer without the need for any communication between them. Since rasterization interpolates lighting between vertices it cannot be parallelized to the same extent as ray tracing.

Ease of use. Realtime ray tracing relieves application programmers from a lot of tedious optimizations such as view-frustum-culling which are necessary for today's applications of raster graphics. Also, there are not so many performance limiting bottle-necks that impose restrictions on the content to be rendered as for the rasterization pipeline. On top of that, users are relieved from writing complicated algorithms that fake shadows, reflections, etc. Lastly there is an open realtime ray tracing API *OpenRT* [Dietrich et al. 2003] which is inspired by and very similar to its counterpart from the rasterization world (*OpenGL*). This API makes the application of realtime ray tracing hardware as easy as for other graphics subsystems.

Geometric scene description The only graphics primitives that can be handled by rasterization pipelines are triangles. By contrast the list of geometric primitives that are supported by realtime ray tracing systems (without triangulation or other modifications) is quite large: triangles [Wald et al. 2001a], freeform surfaces [Benthin et al. 2004], point based surfaces [Wald and Seidel 2005], bezier patches [Benthin et al. 2006], volume data in all kinds of grids [Parker et al. 1999; Marmitt et al. 2004, Marmitt and Slusallek 2006], implicit surfaces [Knoll et al. 2007] ... and so on.

Dynamics. Limited support of dynamic scenes has long been the biggest disadvantage of realtime ray tracing systems compared to raster graphics. Despite this, better solutions have been found recently to support realtime ray tracing of dynamic scenes in general (see section 3.2 and 3.3) and skinned animations in particular [Günther et al. 2006].

7 Conclusion

For a long time researchers have argued that ray tracing is superior to rasterization in various points, but nobody would have imagined that ray tracing could be applied so effectively for realtime rendering. As recent software realtime ray tracing systems become much more powerful than high-end rasterization hardware, it seems like the advent of a new age for realtime graphics. Hardware solutions are not yet as optimized and feature rich as rasterization hardware, though. Only the future can tell if realtime ray tracing hardware will eventually become good enough, so that ray tracing may supersede rasterization as the major realtime rendering paradigm.

This paper has given a an overview of the possible acceleration structures which have different advantages with respect to dynamic scenes. We have seen, that kd-trees are not primarily the fastest acceleration structures any more and that bounding volume hierarchies as well as the bounding interval hierarchies are less restrictive for animated geometry. The analysis of the typical computational complexity of a ray tracer motivates the hypothesis that the use of a well built scene acceleration structure combined with a carefully optimized scene traversal algorithm are most effective in order to increase the frame rates of realtime ray tracers.

Next, this paper pointed out the importance of coherence between rays for interactive ray tracing. Results from various implementations show that tracing packets or even frustrums of rays yields huge performance improvements. Even global illumination simulations can be rendered in realtime on a distributed coherent ray-tracing cluster by applying interleaved sampling and discontinuity buffering.

On top of that, the possibilities of parallelization have been explored especially using SIMD instructions on a single processor as well as distributing workload on a PC cluster. We have seen that parallelization always goes hand in hand with caching and sophisticated cache-utilization mechanisms. The effective utilization of the large and fast caches of modern CPUs via intelligent prefetching of data results in performance improvements of more than a magnitude.

The final comparison of realtime ray tracing versus raster graphics showed that as far as realism and scalability are concerned realtime ray tracing is more than competitive to rasterization hardware but ray tracing hardware is not yet ready for the prime time. In near future, when realtime ray tracing has been established in the realtime rendering domain the development of interactive graphics applications will be significantly simplified because faked shadows and reflections, complicated rasterization-pipeline-aware optimizations and advanced view-frustum-culling algorithms will not be necessary any more.

References

APPEL, A. 1968. Some techniques for shading machine renderings of solids. In *Proceedings of the Spring Joint Computer Conference*, 37–45.

BENTHIN, C., WALD, I., AND SLUSALLEK, P. 2003. A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum 22*, 3, 621–630. (Proceedings of Eurographics).

BENTHIN, C., WALD, I., AND SLUSALLEK, P. 2004. Interactive ray tracing of free-form surfaces. In *AFRIGRAPH '04: Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, ACM Press, New York, NY, USA, 99–106.

BENTHIN, C., WALD, I., AND SLUSALLEK, P. 2006. Techniques for Interactive Ray Tracing of Bezier Surfaces. *Journal of Graphics Tools 11*, 2, 1–16.

DIETRICH, A., WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. The OpenRT Application Programming Interface - Towards a Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*. Available at http://www.openrt.de.

DIETRICH, A., COLDITZ, C., DEUSSEN, O., AND SLUSALLEK, P. 2005. Realistic and Interactive Visualization of High-Density Plant Ecosystems. In *Natural Phenomena 2005, Proceedings of the Eurographics Workshop on Natural Phenomena*, 73–81.

GÜNTHER, J., FRIEDRICH, H., WALD, I., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum 25*, 3 (Sept.), 517–525. (Proceedings of Eurographics).

HAVRAN, V. 2001. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

JANSEN, F. W. 1986. Data structures for ray tracing. In *Proceedings of a workshop (Eurographics Seminars on Data structures for raster graphics*, Springer-Verlag New York, Inc., New York, NY, USA, 57–73.

KAJIYA, J. T. 1986. The rendering equation. *Computer Graphics 20*, 4, 143–150.

KELLER, A., AND HEIDRICH, W. 2001. Interleaved sampling. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, Springer-Verlag, London, UK, 269–276.

KELLER, A. 1997. Instant radiosity. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 49–56.

KNOLL, A., HIJAZI, Y., HANSEN, C. D., WALD, I., AND HAGEN, H. 2007. Interactive Ray Tracing of Arbitrary Implicit Functions. Tech. Rep. UUSCI-2007-002. (submitted for publication).

LEXT, J., AND AKENINE-MÖLLER, T., 2001. Towards rapid reconstruction for animated ray tracing.

MACDONALD, J. D., AND BOOTH, K. S. 1989. Heuristics for ray tracing using space subdivision. In *Graphics Interface*, 152–163.

MARMITT, G., AND SLUSALLEK, P. 2006. Fast Ray Traversal of Tetrahedral and Hexahedral Meshes for Direct Volume Rendering. In *Proceedings of Eurographics/IEEE-VGTC Symposium on Visualization (EuroVIS) 2006*, n/a, Lisbon, Portugal, T. Ertl, K. J. Joy, and B. Santos, Eds., n/a. (to appear).

MARMITT, G., KLEER, A., WALD, I., FRIEDRICH, H., AND SLUSALLEK, P. 2004. Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, 429–435.

MOLNAR, S., 1991. Efficient supersampling antialiasing for high-performance architectures.

PARKER, S., PARKER, M., LIVNAT, Y., SLOAN, P.-P., HANSEN, C., AND SHIRLEY, P. 1999. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3, 238–250.

RAMASUBRAMANIAN, M., PATTANAIK, S. N., AND GREENBERG, D. P. 1999. A perceptually based physical error metric for realistic image synthesis. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 73–82.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM Press, New York, NY, USA, 1176–1185.

RUBIN, S. M., AND WHITTED, T. 1980. A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH*

'80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 110–116.

SCHMITTLER, J., WALD, I., AND SLUSALLEK, P. 2002. SaarCOR – A Hardware Architecture for Realtime Ray-Tracing. In *Proceedings of EUROGRAPHICS Workshop on Graphics Hardware*. available at http://graphics.cs.uni-sb.de/Publications.

SCHMITTLER, J., WOOP, S., WAGNER, D., PAUL, W. J., AND SLUSALLEK, P. 2004. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proceedings of Graphics Hardware*, 95–106.

WALD, I., AND HAVRAN, V. 2006. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 61–69.

WALD, I., AND SEIDEL, H.-P. 2005. Interactive Ray Tracing of Point Based Models. In *Proceedings of 2005 Symposium on Point Based Graphics*.

WALD, I., BENTHIN, C., WAGNER, M., AND SLUSALLEK, P. 2001. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, Blackwell Publishers, Oxford, A. Chalmers and T.-M. Rhyne, Eds., vol. 20, 153–164.

WALD, I., SLUSALLEK, P., AND BENTHIN, C. 2001. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001 (Proceedings of the 12th EUROGRAPHICS Workshop on Rendering*, Springer, S.J.Gortler and K.Myszkowski, Eds., 277–288.

WALD, I., KOLLIG, T., BENTHIN, C., KELLER, A., AND SLUSALLEK, P. 2002. Interactive global illumination using fast ray tracing. In *Proceedings of the 13th EUROGRAPHICS Workshop on Rendering*, Saarland University, Kaiserslautern University.

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*.

WALD, I., DIETRICH, A., BENTHIN, C., EFREMOV, A., DAHMEN, T., GÜNTHER, J., HAVRAN, V., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Applying ray tracing for virtual reality and industrial design. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 177–185.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1.

WALD, I. 2004. Realtime Ray Tracing and Interactive Global Illumination. *PhD thesis, Saarland University*.

WÄCHTER, K., AND KELLER, A. 2006. Instant ray tracing: The bounding interval hierarchy. In *Proceedings of EUROGRAPHICS 2006*.

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM 23*, 6, 343–349.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. Rpu: a programmable ray processing unit for realtime ray tracing. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM Press, New York, NY, USA, 434–444.