DOI: 10.1111/j.1467-8659.2012.03073.x

COMPUTER GRAPHICS *forum*
*Volume 31 (2012), number 6 pp. 1936–1947*

# Improving Data Locality for Efficient In-Core Path Tracing

J. Bikker

NHTV University of Applied Sciences, Breda, The Netherlands
bikker.j@nhtv.nl

**Abstract**

*In this paper, we investigate the efficiency of ray queries on the CPU in the context of path tracing, where ray distributions are mostly random. We show that existing schemes that exploit data locality to improve ray tracing efficiency fail to do so beyond the first diffuse bounce, and analyze the cause for this. We then present an alternative scheme inspired by the work of Pharr et al. in which we improve data locality by using a data-centric breadth-first approach. We show that our scheme improves on state-of-the-art performance for ray distributions in a path tracer.*

**Keywords:** ray tracing, path tracing, ray divergence, data locality

**ACM CCS:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism Raytracing

## 1. Introduction

Over the past decade, CPU ray tracing performance (relative to absolute performance) has greatly increased. In their 1999 paper [PMS*99], Parker *et al*. report 1M rays per second (for figure 16) on a 24Gflops SGI Origin 2000 system. Boulos *et al*. achieve 3M for (almost) the same scene [BEL*07], on a 2Gflops machine, a 36× improvement. The algorithms that enabled these advances depend on coherent ray distributions, where rays have similar origins and directions, to work well. In the context of Monte Carlo path tracing, this coherency is mostly unavailable: directional coherency is typically already lost for the first diffuse bounce, and although these rays still have similar ray origins, this is not the case for the second diffuse bounce. Beyond this point, ray distributions are essentially random. This leads to a highly random access pattern of scene data, and, as a consequence, poor utilization of caches and SIMD hardware.

In this paper, we investigate the work that has been done to improve data locality in the context of ray tracing. We show that the extend to which existing approaches improve data locality is limited in the context of path tracing, and analyse the cause of this. For divergent rays, we propose a traversal scheme that uses breadth-first traversal and batching to improve the performance of a ray packet traversal scheme. Our scheme is based on the work by Pharr *et al*., which targeted out-of-core rendering. Unlike their system, our scheme targets the top of the memory hierarchy. Our system consistently outperforms single-ray traversal through a multi-branching bounding volume hierarchy (BVH).

## 2. Path Tracing and Data Locality

The game developer and optimisation specialist Terje Mathisen once stated that "almost all programming can be viewed as an exercise in caching" [Abr97]. With this remark, he points out the importance of caches on modern computer architectures, especially when algorithms deal with large amounts of data. Considering the vast gap between the rate at which a processor can execute commands, and the rate at which memory can supply data for these commands, caching is usually the optimisation with the greatest effect.

Caches are part of the memory hierarchy [BO10] (see Figure 1). Small, but fast caches hide the latency of larger, but slower memories, assuming a certain level of data locality exists [KA02]. In computer science, data locality is subdivided in temporal locality, spatial locality and sequential locality. In modern systems, caches benefit from the first two, while the latter is exploited by instruction prefetching.

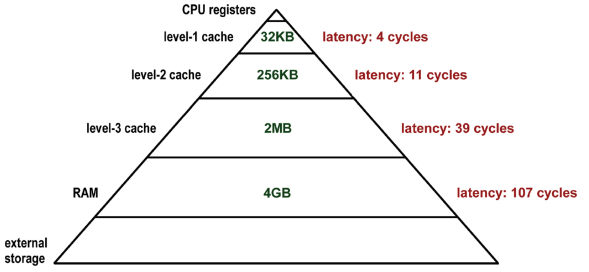


**Figure 1:** *The memory hierarchy: smaller, but faster caches hide the latency of larger but slower memories. Shown cache sizes and latencies are for our test system (Intel Xeon X5670), per core.*

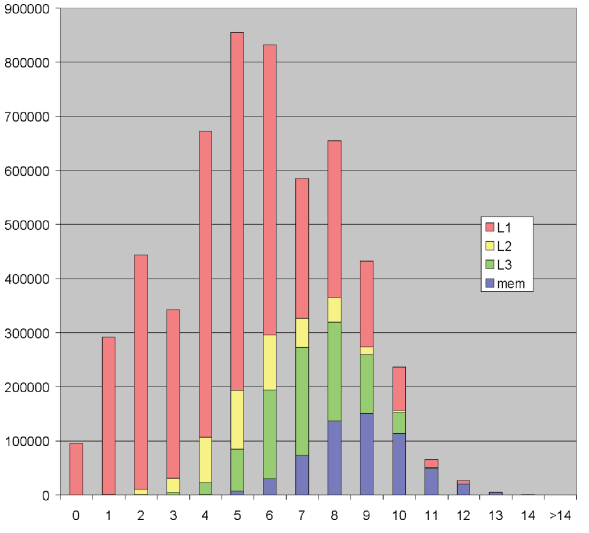


**Figure 2:** *Access of the levels of the memory hierarchy, scaled by access cost (in CPU cycles), for the first 14 levels of an acceleration structure. Measured for single ray traversal, for the Soda Hall scene.*

Optimal data locality in an algorithm is achieved when the number of times that the same datum is loaded into the caches is one. In other words: all the work that involves a particular datum is carried out, after which the datum will not be accessed again. Note that uniform streaming algorithms, where one kernel is applied to all elements of an input stream, naturally reach this optimum.

For algorithms that perform tree traversal, data locality tends to decrease with tree node depth. While nodes high in the tree are repeatedly accessed for successive queries, deeper levels are often evicted before being accessed again. This is shown in Figure 2. The graph shows how the cost of accessing the various levels of the memory hierarchy is distributed over the levels of a four-wide multi-branching BVH, traversed by the rays of an incoherent ray distribution. This data was gathered using a cache simulator, which is described in Section 4. The L1 cache is able to handle the majority of the transfers. At deeper levels, more queries fall through to L2, L3 and memory. While L1 is mostly ineffective for the deepest levels, the total amount of traffic at these levels is small, and contributes little to overall memory access cost.

The overall cost of memory access can be reduced by improving data locality. Better data locality keeps data higher in the cache hierarchy, and reduces the total number of memory transfers, by using the same data for more rays.

### 2.1. SIMD efficiency and data locality

Modern CPUs strongly depend on *single instruction multiple data* (SIMD) technology to achieve optimal compute efficiency. SIMD operates on vectors, rather than on scalars. Assuming that multiple streams of data are available for which the same instructions are to be executed, SIMD hardware processes these streams in parallel. The elements of the vectors used to operate on these streams are typically referred to as *lanes*. CPUs operate on four lanes (Intel/SSE [TH99], AltiVec [DDHS00]), eight lanes (Intel/AVX [Lom11]) or sixteen lanes (Intel/Larrabee [SCS*08]). Similar technology on the GPU simultaneously processes 32 lanes [Dog07, Gla09].

SIMD is effective when all lanes require the same instructions. When this is not the case (e.g. due to conditional code), operations can be masked, or processed sequentially. In both cases, SIMD utilisation decreases.

SIMD efficiency is also affected by scatter/gather operations: loading data into vector registers is faster if the required addresses are sequential. In fact, on many SIMD architectures, this is a requirement; sequential code is used when this requirement is not met. At the same time, sequential data access reduces the total number of cache lines that is read from memory, as sequential data typically resides in the same cache line.

Efficiency of the memory hierarchy and SIMD efficiency are tightly coupled: optimisations that aim to improve data locality will often also lead to better SIMD utilisation.

### 2.2. Previous work on improving data locality in ray tracing

Several authors recognise the importance of data locality for the performance of the ray tracing algorithm.

**Ray Packets** Zwaan, Reinhard and Jansen propose to use ray packet traversal to amortize the cost of cache misses over the rays in the packet [vdZRJ95, RJ97]. By traversing ray packets (referred to as *pyramids* in their papers) rather than single rays, acceleration structure nodes are fetched once for a number of rays. The authors report improved data locality for coherent ray distributions. Wald *et al.* uses SIMD to traverse

a kD-tree with a ray packet containing four rays [WSBW01], and achieves interactive frame rates on a cluster of PCs. Smittler *et al*. propose a custom hardware architecture, SaarCOR [SWS02], that traces packets of 64 rays. They hide the latency of cache misses by swapping between ray packets, using a technique similar to *multi-threading* [PBB*02]. Later, the concept of ray packet traversal is generalised to arbitrarily sized ray packets by Reshetov [Res07] and to other acceleration structures [WBS07, WIK*06].

**Reordering** Based on the observation that packets of secondary rays often exhibit little coherence, reordering schemes aims to regain coherence by reordering the secondary rays from multiple packets into more coherent sets. Mansson *et al*. [MMAM07] investigated several reordering methods for secondary rays. They aim to create coherent packets of secondary rays by batching and reordering these rays. They conclude that due to the cost of reordering none of the heuristics improves efficiency when compared to secondary ray performance of the Arauna system [Bik07], which does not attempt to reorder secondary rays. Overbeck *et al*. propose a ray packet traversal scheme that is less sensitive to degrading coherence in a ray packet [ORM08]. Their partition traversal scheme reorders the rays in the packet in-place by swapping inactive rays for active rays and by keeping track of the last active ray. This scheme is less efficient for primary rays, but performs better for secondary rays.

**Hybrid schemes** Taking into account the inefficiency of ray packets for divergent ray distributions, Benthin *et al*. proposed a hybrid scheme for the Intel MIC architecture [Int10] that traces packets until rays diverge, after which it switches to efficient single ray traversal [BWW*11].

**Breadth-first** A typical traversal scheme uses an outer loop that iterates over a set of rays, and an inner loop that processes acceleration structure nodes. Hanrahan proposed to swap these loops [Han86]. By using the inner loop to iterate over rays rather than objects, access to objects stored on disk is minimized. In their 2007 study, Wald *et al*. investigated breadth first ray tracing with reordering at every step [WGBK07]. They conclude that breadth-first ray tracing reduces the number of acceleration structure nodes that is visited, but also that the high reordering cost may not justify this. Boulos *et al*. continue this work [BWB08]. In their paper, they show that the performance gains of demand-driven reordering out-weight the overhead. For diffuse bounces, these gains drop below 2**x** however. On the GPU, Garanzha and Loop propose breadth-first traversal of rays [GL10]. Their scheme sorts the set of rays into coherent packets and then performs a breadth-first traversal of a BVH. On the GPU, they claim a 3x improvement over depth-first implementations for soft shadows cast by large area lights.

**Batching** Several authors propose to use a form of batching to improve data locality. In these schemes, traversal of a single ray is broken up in parts; rays are batched in nodes of the acceleration structure, and advanced when such a node is scheduled for processing. Pharr *et al*. [PKGH97] describe a system, Toro, for out-of-core ray tracing where objects are subdivided using regular grids. Rays are batched in the voxels of a secondary regular grid. This system is discussed in more detail in Section 3. Kato and Saito schedule shading separately from ray traversal and photon map look-up to hide latency on a cluster of PCs in their Kilaeua system [KS02]. Budge *et al*. [BBS*09] perform out-of-core rendering on hybrid systems by breaking up data and algorithmic elements into modular components, and queuing tasks until a critical mass of work is reached. Navratil *et al*. propose a system that actively manages ray and geometry states to provide better cache utilisation and lower bandwidth requirements [NFLM07]. As in the Toro system, rays in their system progress asynchronously. While Pharr *et al*. apply ray scheduling at the bottom of the memory hierarchy, Navratil *et al*. aim to reduce RAM-to-cache data transport. They claim a reduction of RAM-to-L2 cache transport up to a factor 7.8 compared to depth-first packet traversal. Hanika *et al*. propose a two-level ray tracing scheme, in which rays are first traced through a top-level structure over the bounding volumes of free form patches, and then through a lazily constructed bottom-level structure over the micro-polygons of the tessellated patches [HKL10]. Between these steps, rays are reordered according to patch ID to increase data locality.

**Streaming** Breadth-first ray traversal combined with a filtering operation that partitions the set of rays into active and inactive subsets effectively transforms ray traversal into a streaming process, where one traversal step provides the input stream for the next traversal step. Gribble and Ramani [GR08] propose an approach that during traversal sorts a stream of rays into a set of active rays (rays that intersect the current node) and inactive rays. They implement this on a custom hardware architecture that supports wide SIMD processing. For a stream of rays, their approach bears resemblance to breadth-first ray traversal [Han86]. Tsakok [Tsa09] proposes a streaming scheme, MBVH/RS, that benefits from coherency if this is present, and falls back to efficient single-ray traversal using an multi-branching BVH for divergent rays. For divergent ray tasks on x86/x64 CPUs, this scheme is currently the best performing approach.

### 2.3. Interactive rendering

Interactive ray tracing poses specific challenges for efficient ray tracing schemes. In an interactive context, many schemes exhibit overhead that exceeds the gains. Because of this, schemes developed for out-of-core and offline rendering often do not transfer to interactive rendering.

Of the approaches targeted at improving data locality in ray tracing, ray packets have been by far the most successful for interactive ray tracing. Using ray packets, the cost of data access is amortized over multiple rays. To work efficiently, ray packet traversal schemes require that the rays in the set visit a similar set of acceleration structure nodes. Benthin

defines this *traversal coherence* in his Ph.D. thesis as "the ratio between the number of spatial cells traversed by all rays and the sum of cells traversed by any ray" [Ben06]. This ratio is low when rays travel in a similar direction, and have a similar origin. Without this coherence, ray packet traversal schemes fail to improve on naive, depth-first single ray traversal.

Ray packet schemes, which have proven to be successful for interactive rendering of primary rays and shadow rays, show degrading efficiency for secondary rays. Although some authors report reasonable results for interactive Whitted-style ray tracing [ORM08], for path tracing, the overhead of these schemes makes them slower than single ray traversal. Some authors therefore suggest to abandon ray packets altogether [WBB08], and to focus on efficient single ray traversal [DHK08, EG08].

To better understand the impact of ray coherence and the overhead of schemes, we have implemented three schemes that target interactive performance, which we compare against base-line performance of single ray traversal. We use the terminology of Overbeck *et al*. [ORM08] for the naming of masked traversal, ranged traversal and partition traversal. We refer the reader to their paper for a detailed description of these schemes.

**Single ray traversal** For baseline performance, we chose single-ray, depth-first traversal of a *multi-branching BVH* (MBVH or QBVH [DHK08, EG08, WBB08]), rather than the more commonly used 2-ary BVH. Compared to a 2-ary BVH, the 4-ary BVH performs 1.6 to 2.0 times better [DHK08].

**Ranged traversal** This scheme is based on the packet traversal scheme introduced by Wald *et al*. (*masked traversal*, [WSBW01]), where a node of the acceleration structure is traversed if any ray in the packet intersects it. Ranged traversal improves on masked traversal by storing the first and last active ray in a packet. Rays outside this range are not tested against the nodes of the acceleration structure, reducing the number of ray-AABB (*axis aligned bounding box*) tests. Like masked traversal, this scheme performs best for primary rays. For secondary rays, the range may contain a considerable amount of inactive rays, reducing efficiency.

**Partition traversal** Designed for secondary rays in a Whitted-style ray tracer, this scheme partitions the rays in the packet in-place by swapping inactive rays for active rays and by keeping track of the last active ray. Compared to ranged traversal, this scheme is less efficient for primary rays, but it performs better for secondary rays, assuming some coherence is still available. Partition traversal operates on groups of $N$ rays (where $N$ is the SIMD width) to reduce overhead.

**MBVH/RS** Tsakok's Multi-BVH Ray Stream tracing scheme, designed for divergent ray distributions. For each MBVH node, the scheme intersects a list of rays with the four child nodes, generating new lists for each of them. Like other packet traversal schemes, MBVH/RS amortizes the cost of fetching a node over all active rays. Unlike in partition traversal, the generated lists do not contain any inactive rays. This makes MBVH/RS more efficient for divergent ray distributions, where many MBVH nodes are traversed by a small number of rays.

**Table 1:** *Performance of four traversal schemes, in* $10^6$ *rays per second: single ray traversal through a MBVH, partition traversal, ranged traversal, and MBVH/RS. Measured for five scenes, for primary rays and 1, 2 and 3 diffuse bounces, on a single core of a 3.8Ghz Intel Xeon processor. Bold figures denote the best performing scheme for each scene and depth.*

| Scheme | Scene | Primary | 1st | 2nd | 3rd |
|---|---|---|---|---|---|
| Single | Modern | 3.032 | **1.874** | **1.608** | **1.531** |
| Partition | | 6.400 | 1.157 | 0.790 | 0.723 |
| Ranged | | **9.102** | 0.707 | 0.448 | 0.399 |
| MBVH/RS | | 3.539 | 1.793 | 1.421 | 1.324 |
| Single | Sponza | 2.788 | 1.973 | **1.926** | **1.890** |
| Partition | | 5.595 | 1.231 | 0.941 | 0.856 |
| Ranged | | **7.897** | 0.800 | 0.520 | 0.476 |
| MBVH/RS | | 3.424 | **2.174** | 1.765 | 1.673 |
| Single | Lucy | 2.899 | **1.746** | **1.645** | **1.591** |
| Partition | | 4.145 | 0.879 | 0.636 | 0.605 |
| Ranged | | **5.310** | 0.516 | 0.341 | 0.313 |
| MBVH/RS | | 3.180 | 1.626 | 1.282 | 1.251 |
| Single | Conference | 3.528 | 2.141 | 1.718 | 1.583 |
| Partition | | 5.937 | 1.435 | 1.102 | 0.956 |
| Ranged | | **6.976** | 1.037 | 0.802 | 0.715 |
| MBVH/RS | | 4.959 | **2.492** | **1.887** | **1.631** |
| Single | Soda Hall | 3.542 | 2.787 | 2.527 | 2.477 |
| Partition | | 5.349 | 1.098 | 0.755 | 0.682 |
| Ranged | | **8.821** | 0.788 | 0.459 | 0.430 |
| MBVH/RS | | 4.151 | **3.058** | **2.734** | **2.631** |

All traversal scheme implementations are hand-tuned for optimal performance.

Table 1 shows performance figures for the five scenes shown in Figure 3. Using a 4-ary BVH, baseline performance slowly degrades for each diffuse bounce. For primary rays, ranged traversal outperforms all other traversal methods by a significant margin. After one diffuse bounce, only MBVH/RS outperforms single ray traversal, for some scenes, and by a small margin. After three diffuse bounces, ranged traversal only achieves 17–26% of single ray traversal performance, while partition traversal achieves 27–60% of single ray traversal performance. MBVH/RS achieves between 79% and 106%.

To understand why the traversal schemes perform so poorly for divergent ray distributions, we measured how many active rays visit the nodes of the acceleration

**Figure 3:** *The five scenes used in our experiments: Modern Room from Let there be Light (88k triangles), the Sponza Atrium (93k), Conference Room (273k), Sibenik Cathedral with the Lucy statue (603k) and Soda Hall (2.1M). Rendered with up to 1k samples per pixel, and a maximum of six diffuse bounces.*

**Table 2:** *Average number of rays per visited leaf/interior node of the BVH, per recursion depth, out of the original 256 rays in a 16×16 ray packet. Measured for the Modern Room scene.*

| | Depth | primary | 1st | 2nd | 3rd |
|---|---|---|---|---|---|
| Ranged/partition | Interior | 94.38 | 9.12 | 4.71 | 3.84 |
| | Leaf | 37.88 | 3.34 | 1.63 | 1.30 |
| MBVH/RS | Interior | 170.17 | 13.25 | 7.01 | 5.70 |
| | Leaf | 97.57 | 6.26 | 3.20 | 2.60 |

structure. Table 2 shows the average number of rays (of the original packet) that intersects each visited node. For a traversal scheme to work well, this number should be high. However, for the Modern Room scene (Figure 3a), this number drops rapidly after only one diffuse bounce. Note that this number is an average: even for random ray distributions, all rays will intersect the root node of the BVH (assuming the camera is within the scene bounds). We therefore also measured the average number of rays that intersects the leafs of the acceleration structure. After a few diffuse bounces, this number approaches one. The MBVH/RS algorithm performs better, as it uses a relatively shallow BVH.

### 2.4. Discussion

Ray tracing efficiency for divergent ray distributions is affected by the low average number of rays that is active when visiting the nodes of the acceleration structure. The result of this is that the cost of fetching data from L3 cache and memory is shared by a small number of rays. The low number of active rays also leads to poor SIMD utilisation.

The low average number is caused by the packet sizes for which existing schemes perform optimally. Ranged and partition traversal, as well as MBVH/RS, perform best for packets of 64 to 1024 rays [ORM08, Tsa09]. Although larger ray packets would lead to higher active ray counts, in practice this reduces overall efficiency of these schemes.

A scheme that is able to traverse very large ray packets, on the other hand, would exhaust the L1 cache for the ray data alone in the first nodes of the acceleration structure.

An optimal traversal scheme would operate on the same number of rays at each level of the acceleration structure. This requires batching of rays at all levels.

Although batching could improve data locality, it has some disadvantages: the batching itself may introduce considerable overhead. Batched rays must store their full state. For BVH traversal, this includes a traversal stack.

## 3. Data-Parallel Ray Tracing

In the previous section, we have shown that schemes that are designed to improve data locality in ray tracing work well for coherent ray distributions, such as those found in Whitted-style ray tracing, but fail to improve data locality for divergent ray distributions, as found in path tracing. Where Whitted-style ray tracing benefits from a task-centric approach (where a task is a ray query or a ray packet query), path tracing may benefit more from a data-centric approach.

In this section, we describe a scheme, RayGrid, that locally batches rays, until enough work is available to amortize the cost of cache misses over many rays. Our scheme is similar to the scheme described by Pharr *et al.* [PKGH97], but targets the top of the memory hierarchy. We analyse the characteristics of this scheme in the context of interactive, in-core rendering.

### 3.1. Algorithm overview

We will first describe the scheme developed by Pharr *et al.*, which was designed for out-of-core rendering in the Toro system. We will then describe our RayGrid system, which borrows from the original scheme and makes it suitable for in-core rendering.

#### 3.1.1. *Data structures used in Toro*

The Toro system uses a number of data structures. The first is a set of static regular voxel grids, which stores the scene geometry: the *geometry grid*s. One such grid is created per geometric object in the scene. Primitives that stride voxel

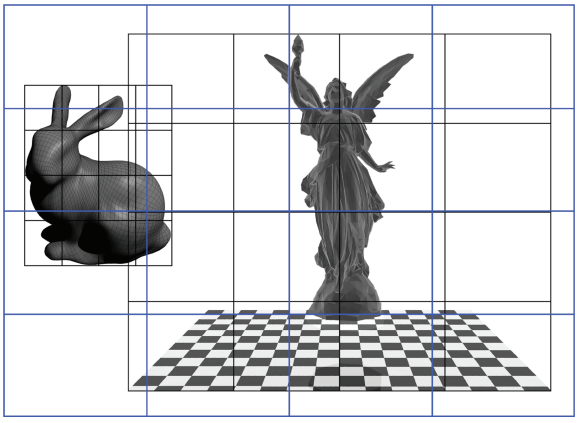

**Figure 4:** *Data structures used in the Toro system: geometry grids enclosing two geometric objects (black), and the scheduling grid, used for advancing rays through the scene (blue). The cells of the geometry grid typically contain thousands of primitives, which are stored in another grid, the acceleration grid (not shown).*

boundaries are stored in all grid cells they overlap. Geometry inside a grid cell is stored sequentially in memory, so that spatial coherence in 3D equals memory coherence. The static grids do not reside in main memory, and are accessed via a caching mechanism that loads and evicts entire grid cells at once. For this to be efficient, the grid cells must store thousands of primitives. This requires a secondary acceleration structure inside each grid cell, for which Pharr *et al.* propose another regular grid. This grid is referred to as the *acceleration grid*. Finally, rays traverse a third regular grid, the *scheduling grid*.

The data structures are shown in Figure 4. For clarity, the acceleration grid has been omitted in this figure.

#### 3.1.2. *Batching for out-of-core rendering in Toro*

Newly created rays are queued in the cells of the scheduling grid. Rays are advanced by processing grid cells from the scheduling grid. When a grid cell is scheduled for processing, each queued ray in it is tested for intersection with the geometry inside each overlapping geometry voxel, and, if no intersection occurred, advanced to the nearest neighbouring grid cell, where it awaits further processing.

The system schedules grid cells in the following order: ray queues in the scheduling grid for which all geometry data is cached are processed first. Once these are depleted, the system loads geometry into the cache for the largest ray queue. This way, the cost of loading data into the cache is amortized over as many queued rays as possible.

In this system, processing of an individual ray does not necessarily lead to completion: rays are merely advanced to the next voxel. The implication of this is that ray traversal is asynchronous: the order in which rays arrive is undefined, and the potential contribution of each individual ray to the final image must be explicitly stored with the ray.

#### 3.1.3. *Batching for In-Core rendering*

In the Toro system, the cost of loading geometry into the cache is determined by file I/O, patch tessellation, generating procedural geometry and displacement mapping. Compared to in-core rendering, where caching is used to reduce the cost of RAM to L3/L2/L1 data transfer, these costs are high, and justify considerable overhead. This explains why a similar approach has not been considered for in-core rendering, where overhead can easily nullify the potential gains of a scheme.

Like the Toro system, our RayGrid scheme uses a coarse spatial subdivision for geometry that is used to improve data locality for geometry data. Instead of a regular grid, we use a shallow octree structure, which adapts itself to local scene complexity. This is proposed by Pharr *et al.*, but not implemented in the Toro system. We queue rays directly in the voxels of the octree, rather than in a separate structure. Like the Toro system, we store thousands of polygons in octree nodes. To intersect these efficiently, we use an MBVH per octree node.

New rays are added to the system by placing them in ray queues, associated with octree nodes. The system then processes ray queues ordered by size.

Overhead in RayGrid has been reduced by careful data layout and code optimisation. The resulting system performs significantly better than existing approaches for divergent rays.

### 3.2. Data structures

This subsection discusses the main data structures used in RayGrid. The scheme uses a hybrid data structure, consisting of a shallow octree, which contain MBVHs for the geometry in each octree leaf.

#### 3.2.1. *Octree*

Our system uses a shallow octree to subdivide the scene geometry. The octree is extended with spatial bounds and neighbour links, to allow for stack-less ray propagation: rays that leave an octree node through a boundary plane are added to the octree node that the plane links to. If this is an interior node, the ray descents to the leaf node that contains the entry point of the ray. The octree adapts itself to local scene

complexity, while the stack-less traversal maintains the benefits of regular grid traversal.

The actual size of an octree node in memory is of little importance for the efficiency of the scheme: compared to overall memory usage for geometry, the octree node size is negligible.

#### 3.2.2. *MBVH*

An octree node typically stores thousands of primitives. We further subdivide this geometry using an MBVH, which is traversed using the MBVH/RS algorithm. The MBVH is constructed by collapsing a 2-ary BVH, as described by Dammertz *et al*. [DHK08]. Since each octree node containing geometry has its own MBVH, we refer to these as *mini-MBVHs*.

#### 3.2.3. *Ray queues*

Newly added rays are stored in ray queues. A ray queue is a container of a fixed size, that stores rays by value. Ray queues are stored in three linked lists: one for empty ray queues, one for partially filled ray queues, and one for full ray queues. Initially, all ray queues are stored in the empty ray queue list. When a ray is added to an octree node that does not yet contain any rays, the system assigns one ray queue from the empty ray queue list to the octree node, and adds the ray to this list. If the octree node already has a ray queue, the ray is added to that queue. If the queue is full after adding the ray, it is decoupled from the octree node, and added to the list of full queues.

The somewhat elaborate system for storing rays has a number of advantages:

- No run-time memory management is required to store arbitrary amounts of rays per octree node;
- Many processed ray queues are full ray queues. This amortizes the cost of fetching geometry data into the hardware caches over a large number of rays, and leads to efficient traversal using the MBVH/RS algorithm.

A non-empty ray queue stores a pointer to the octree node it belongs to. Multiple full ray queues can belong to the same octree node. Effectively, this allows us to store an arbitrary amount of rays per octree node, divided over zero or more full ray queues, and zero or one partially filled ray queues.

Rays are stored by value in the ray queues, in SoA (*"structure of arrays"*) format. The queue stores the $x$-coordinates of all ray origins consecutively in memory, then the $y$-coordinates, and so on. This data layout is more suitable for SIMD processing than the "array of structures", where full ray records are stored consecutively. Although ray queues could also store ray indices, this requires one level of indirection, which in practice proves to have a small impact on performance. We also measured the impact of moving a ray from one queue to another in SoA rather than the AoS (*"array of structures"*) format. Despite the less coherent memory writes for the SoA format (each stored float is written to a different cache line), this is not slower in practice. The SoA format does allow us to intersect the rays with the boundary planes of an octree node using SIMD code, which makes this layout the preferred one.

The optimal size for a ray queue is determined by balancing the optimal stream size for the MBVH/RS algorithm (256–1024) and the cost of having many partially filled ray queues. The cost of exchanging a full ray queue for an empty one is negligible.

#### 3.2.4. *Ray data*

Our scheme depends on the availability of large numbers of rays to run efficiently. With many rays in flight, the size of a single ray record is important. We store a ray in 48 bytes (11 floats and an integer), by storing the ray origin and direction, the nearest intersection distance, the potential contribution of the ray, and the index of the image pixel the ray contributes to. Since the contribution information is only needed during shading, this data can be stored separately per pixel, which reduces the amount of data copied per ray to 36 bytes (8 floats and an integer).

Since the ray direction is always normalised, it is possible to store only two components of the vector, and derive the third whenever it is needed. Although this reduces the ray record to only 32 bytes, this did not result in improved performance.

### 3.3. Ray traversal

Once all rays have been added into the system, typically a large number of full ray queues is available in the list of full ray queues. Ray traversal then proceeds, starting with full ray queues. Once these are depleted, partially filled ray queues are processed, until no active rays remain in the system. Each processed ray queue is returned to the list of empty ray queues. At the end of the process, this list once again contains all ray queues.

Processing a ray queue consists of two steps:

1. intersection of the rays with the primitives in the octree node, and
2. advancing rays to neighbouring nodes, if no intersection was found.

To intersect the rays with geometry, the rays in the queue are converted to the AoS format, suitable for the MBVH/RS algorithm, and then traversed through the mini-MBVH

associated with the current octree node. For this, we use an unmodified version of the MBVH/RS algorithm.

To advance a ray to a neighbouring node, we first determine the boundary plane through which the ray leaves the current octree node. The neighbour link of this plane then determines the destination for the ray. Since we use an octree, it is possible that the neighbour node is not a leaf node. If this is the case, the ray is recursively added to the child node that contains the ray entry point, until we reach a leaf.

Ray traversal in RayGrid is shown in pseudo-code in algorithm 1.

**Algorithm 1:** *Octree traversal in RayGrid. Rays are added to the octree leaf nodes that contain the ray origins. Once all rays have been added, ray queues are processed, starting with full queues, until all rays have terminated.*

```
queue_full ← {}
queue_partial ← {}
queue_empty ← allocatequeues()
node_cam ← octree.findleaf(camera.getpos())
for each ray in rays[0..N − 1]
  add(node_cam.getqueue(), ray)
do
  if not queue_full.empty()
    process(queue_full.head())
  else if not queue_partial.empty()
    process(queue_partial.head())
  else break
end

function process( Queue q )
  for each ray in q
    if intersect( ray, q.node.mbvh )
      finalize( ray )
  for each link in neighbourlinks
    for each active ray in q
      if ray.intersect( link.getplane() )
        neighbor ← link.getneighbor()
        if (neighbor.getqueue() = null)
          neighbor.setqueue( queue_empty.head() )
          queue_partial.add( neighbor.getqueue() )
        full ← neighbor.getqueue().add( ray )
        if( full )
          queue_full.add( neighbor.getqueue() )
          neighbor.setqueue( null )
  queue_empty.add( q )
end function
```

The high-level octree traversal is illustrated in Figure 5. Rays are added in octree node A, which contains the camera. Once all rays have been added to the system, the ray queue for node A is processed. Rays propagate from node A to node B. In node B, a new ray is added by the shading code. This secondary ray is advanced together with the primary rays, and arrives in node C. Once node C is scheduled for processing, this ray will be processed together with the primary rays from node A.

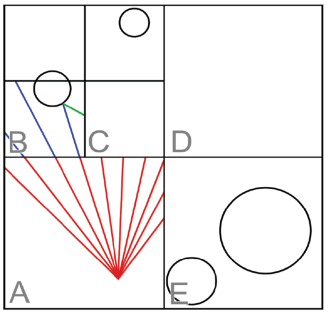


**Figure 5:** *Advancing rays through the octree structure.*

In a practical implementation, RayGrid will not handle primary rays: as discussed in Section 2.3, ray packet traversal is more efficient in this case. Secondary rays however are efficiently handled by our scheme.

### 3.4. Efficiency characteristics

As mentioned in Section 3.3, full ray queues are always processed first. Doing so may saturate partially filled ray queues belonging to neighbouring octree nodes. At some point, there are no full ray queues left, and the system starts processing partially filled ray queues. This reduces the efficiency of the algorithm: the cost of fetching geometry for an octree leaf node is amortized over fewer rays, and the MBVH is traversed with a smaller ray packet.

This tail effect can be reduced by feeding more rays in the system. There are two possible strategies for this:

- By feeding a large amount of rays before processing ray queues, the tail will be relatively short. This increases the memory requirements of the algorithm.
- Alternatively, feeding rays may be coupled to queue processing: by feeding new rays only when no full ray queues are available, memory requirements stay low. This does however require a tight coupling between the code that generates the new rays, and the ray queue scheduler.

In the context of a path tracer, where typically many passes are processed in succession to reduce variance, it is also possible to stop processing ray queues when no full ray queues are available. The partially processed rays remain in the system, and add to subsequent passes. Only when the last pass of the path tracer has completed, the tail needs to be processed. In this scenario, the scheme almost exclusively operates on full ray queues.

### 3.5. Memory use

The requirement to have many rays in flight leads to relatively high memory requirements for the proposed traversal scheme.

The size of a single ray in memory is 36 bytes. As discussed in subsection 3.2.4, rays are stored by value in the ray queues. For a set of $N$ active rays and a queue size of $M$, a minimum of $M/N$ full queues is required. However, a partially filled queue requires the same amount of memory as a full one. Since each octree leaf node may contain zero or one partially filled queues, the memory required by RayGrid is $(O \cdot M + N) \cdot 36$, where $O$ is the number of octree leaf nodes.

We found a queue size $M = 384$ and a maximum number of primitives per octree leaf of 4096 to be a good choice for most scenes. For the Conference Room, this results in $O = 386$ octree leaf nodes. For $N = 1024^2$ rays, the memory use is 41.09MB.

### 3.6. Cache use

Perhaps equally or more important is the use of cache memory for the algorithm. We measured the average amount of memory that is accessed while processing an octree leaf for the Conference Room scene. For this scene, using the proposed parameters, an octree leaf node contains 818 primitives on average. Processing a leaf involves accessing $818 \times 32$ bytes for the primitives, $384 \times 36$ bytes for a full queue, and $384 \times 36$ bytes in the queues of the neighbouring nodes. The 818 primitives are stored in a mini-MBVH of (on average) 175 nodes of 112 bytes each. The total amount of memory accessed for an octree leaf is thus 73KB on average, which is well below the size of the L2 cache in our system.

## 4. Results

In this section, we discuss the performance of the RayGrid scheme.

### 4.1. Performance

We have measured the performance of our scheme. In Table 3, we compare the performance of RayGrid against base-line single-ray performance and the original MBVH/RS algorithm.

As can be seen from these figures, our RayGrid scheme consistently outperforms MBVH single ray traversal and MBVH/RS. In many cases this is only by a small margin. However, compared to the MBVH/RS algorithm, the margin is larger.

Table 4 provides more insight in the improved efficiency. Except for primary rays, the MBVH/RS scheme is able to use a significantly larger number of rays in the visited nodes in the RayGrid algorithm.

**Table 3:** *Performance of our scheme compared to base-line single-ray MBVH traversal and MBVH/RS. Measured for five scenes, on a single core of a 3.8Ghz Intel Xeon processor. Bold percentages indicate performance relative to base-line performance.*

| Scheme | Scene | 1st | 2nd | 3rd | 4th |
|---|---|---|---|---|---|
| Single | Modern | 1.874 | 1.608 | 1.531 | 1.497 |
| MBVH/RS | | 1.793 | 1.421 | 1.324 | 1.275 |
| | | **−4.3%** | **−11.6%** | **−13.5%** | **−14.8%** |
| RayGrid | | 2.392 | 2.136 | 2.115 | 2.113 |
| | | **+27.6%** | **+32.9%** | **+38.1%** | **+41.1%** |
| Single | Sponza | 1.973 | 1.926 | 1.890 | 1.869 |
| MBVH/RS | | 2.174 | 1.765 | 1.673 | 1.636 |
| | | **+10.2%** | **−8.4%** | **−11.5%** | **−12.5%** |
| RayGrid | | 2.311 | 2.325 | 2.310 | 2.315 |
| | | **+17.1%** | **+20.7%** | **+22.2%** | **+23.7%** |
| Single | Lucy | 1.746 | 1.645 | 1.591 | 1.472 |
| MBVH/RS | | 1.626 | 1.282 | 1.251 | 1.122 |
| | | **−6.9%** | **−22.1%** | **−21.4%** | **−23.8%** |
| RayGrid | | 1.855 | 1.840 | 1.806 | 1.763 |
| | | **+6.2%** | **+11.9%** | **+13.55%** | **+19.8%** |
| Single | Conf. | 2.141 | 1.718 | 1.583 | 1.467 |
| MBVH/RS | | 2.492 | 1.887 | 1.631 | 1.499 |
| | | **+16.4%** | **+9.9%** | **+3.0%** | **+2.2%** |
| RayGrid | | 2.355 | 2.055 | 1.910 | 1.838 |
| | | **+10.0%** | **+19.6%** | **+20.6%** | **+25.3%** |
| Single | Soda | 2.787 | 2.527 | 2.477 | 2.450 |
| MBVH/RS | | 3.058 | 2.734 | 2.631 | 2.577 |
| | | **+9.7%** | **+8.2%** | **+6.2%** | **+5.2%** |
| RayGrid | | 3.482 | 3.230 | 3.188 | 3.101 |
| | | **+24.9%** | **+27.8%** | **+28.7%** | **+26.6%** |

To gain more insight in the performance characteristics,we gathered several statistics. For all scenes, optimal or near-optimal performance is achieved for a ray queue size of 384 and a maximum number of primitives per octree leaf node of 4096. For the Conference Room scene, these parameters result in an octree of 441 nodes (of which 386 nodes are leaf nodes). The average depth of an octree leaf node is 4.15. On average, octree leafs contain 818 primitives, for which mini-MBVHs are constructed with an average size of 175 nodes and an average leaf depth of 7.0. For comparison, we constructed an MBVH for the same scene, without the shallow octree: the average depth of leaf nodes in this structure is 10.9, which means that for this scene, the octree replaces the first 3.9 levels of the MBVH.

Profiling indicates that in the RayGrid algorithm, 47.3% is spent on octree traversal, versus 26.3% on MBVH traversal, indicating that octree traversal is considerably more expensive than MBVH traversal.

To measure the characteristics of the RayGrid algorithm in terms of caching behaviour we implemented a cache

**Table 4:** *Average number of rays per visited node of a mini-MBVH in an octree leaf when using the RayGrid algorithm. Measured for the Modern Room scene.*

| | Depth | Primary | 1st | 2nd | 3rd |
|---|---|---|---|---|---|
| RayGrid | Interior | 149.50 | 22.33 | 14.25 | 12.51 |
| | Leaf | 77.87 | 12.26 | 7.39 | 6.36 |

**Table 5:** *Detailed cache behaviour for the Conference Room scene, rendered using the RayGrid algorithm, measured for different batch sizes. Hit counts are in $10^6$ hits for a 512×512 image. Estimated cost in $10^9$ cycles, assuming 4:11:39:107 cycle latencies for L1:L2:L3:memory access. For comparison, the last column contains figures for the original MBVH/RS algorithm (without octree traversal).*

| | 64 | 128 | 256 | 512 | 1024 | MBVH/RS |
|---|---|---|---|---|---|---|
| L1 read hit | 907.1 | 856.8 | 760.1 | 665.2 | 615.2 | 1023.0 |
| L2 read hit | 10.9 | 24.7 | 89.9 | 158.6 | 185.1 | 78.5 |
| L3 read hit | 9.7 | 8.2 | 6.7 | 6.5 | 11.0 | 31.0 |
| mem read | 11.2 | 11.4 | 11.7 | 11.7 | 11.6 | 6.6 |
| L1 write hit | 428.8 | 399.2 | 351.5 | 341.0 | 338.2 | 276.1 |
| L2 write hit | 63.0 | 124.9 | 279.6 | 361.5 | 369.4 | 111.7 |
| L3 write hit | 40.5 | 33.5 | 27.7 | 36.9 | 95.3 | 110.2 |
| mem write | 41.9 | 43.7 | 45.7 | 43.9 | 42.1 | 21.4 |
| cost (est.) | 5328.8 | 5234.6 | 5536.5 | 5915.1 | 6168.0 | 6872.6 |
| L1% | 68.1 | 65.5 | 54.9 | 45.0 | 39.9 | 54.1 |
| L2% | 2.3 | 5.2 | 17.9 | 29.5 | 33.0 | 10.1 |
| L3% | 7.1 | 6.1 | 4.7 | 4.3 | 7.0 | 12.9 |
| mem% | 22.6 | 23.2 | 22.5 | 21.2 | 20.1 | 22.9 |

simulator. The simulator mimics the cache hierarchy of our test system, with a32KB L1, 256KB L2 cache and 2MB L3 cache. Note that our test system uses a 12MB shared L3 cache for six CPU cores; the simulated 2MB L3 cache is an approximation of the per-core L3 capacity when all cores run the RayGrid algorithm. The L1 and L2 caches are 8-way set associative, L3 is 16-way set associative. The caches use a pseudo-LRU eviction scheme. We use code instrumentation to record reads and writes.

In Table 5, cache behaviour of the RayGrid algorithm and MBVH/RS is compared for the Conference Room scene. We estimate the total cost of memory access by summing L1, L2, L3 and RAM accesses, multiplied by the respective latencies of each level (4, 11, 39, 107 cycles on our test platform).

Compared to the MBVH/RS, the RayGrid algorithm achieves a significant reduction in L3 cache access. Although the number of RAM transfers increased, the (estimated) overall cost of memory access is reduced.

## 5. Conclusion and Future Work

In this paper, we have investigated efficient ray tracing in the context of a path tracer.

High performance is achieved by distinguishing between primary rays and rays after one diffuse bounce. For primary rays, the ranged traversal algorithm performs best. After one diffuse bounce, it is hard to outperform MBVH single ray traversal. We proposed a data centric ray traversal scheme inspired by the work of Pharr *et al.*, that manages to improve on this base line performance, albeit by a small margin. This improvement is achieved by improving data locality, by batching rays in the leafs of a shallow octree. The improved data locality leads to improved L2 cache efficiency and SIMD utilisation.

We would like to further investigate the use of batching for improved data locality, perhaps without relying on an octree data structure, which is currently required for efficient leaf-to-leaf ray propagation. Alternatively, an implementation on an architecture that allows for efficient gather / scatter would allow for more efficient octree traversal, which might even lead to improved L1 cache efficiency.

## Acknowledgements

The Modern Room scene was modelled by students of the IGAD program of the NHTV University of Applied Sciences. The Sponza scene and the Sibenik Cathedral were modeled by Marco Dabrovik. The Bugback Toad model was modelled by Son Kim. The Lucy Statue was originally obtained from the Stanford 3D Scanning Repository. The 'Soda Hall' scene is the U.C. Berkeley Soda Hall WALKTHRU Model. The Conference Room model was created by Anat Grynberg and Greg Ward. The author wishes to thank Erik Jansen for proofreading and fruitful discussions.

This research was supported by an Intel research grant.

## References

[Abr97] Abrash M.: *Michael Abrash's Graphics Programming Black Book (Special Edition)*. Coriolis Group Books, 1997.

[BBS*09] Budge B., Bernardin T., Stuart J. A., Sengupta S., Joy K. I., Owens J. D.: Out-of-core Data Management for Path Tracing on Hybrid Resources. *Computer Graphics Forum 28*, 2 (2009), 385–396.

[BEL*07] Boulos S., Edwards D., Lacewell J. D., Kniss J., Kautz J., Shirley P., Wald I.: Packet-based Whitted and Distribution Ray Tracing. In *GI '07: Proceedings of Graphics Interface 2007* (New York, NY, USA, 2007), ACM, Montreal, Canada, pp. 177–184.

[Ben06] BENTHIN C.: *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, Saarbrücken, Germany, January 2006.

[Bik07] BIKKER J.: Real-time Ray Tracing through the Eyes of a Game Developer. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), IEEE Computer Society, p. 1.

[BO10] BRYANT R. E., O'HALLARON D.R.: *Computer Systems: A Programmer's Perspective*, 2nd ed. Addison-Wesley Publishing Company, USA, 2010.

[BWB08] BOULOS S., WALD I., BENTHIN C.: Adaptive Ray Packet Reordering. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2008* (Los Alamitos, CA, USA, 2008), IEEE Computer Society, pp. 131–138.

[BWW*11] BENTHIN C., WALD I., WOOP S., ERNST M., MARK W. R.: Combining Single and Packet Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics 99*, PrePrints (2011).

[DDHS00] DIEFENDORFF K., DUBEY P. K., HOCHSPRUNG R., SCALES H., : AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro 20* (March2000), 85–95.

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum 27*, 4 (2008), 1225–1233.

[Dog07] DOGGETT M.: *AMD's Radeon HD 2900*. Tech. Rep., San Diego, August 2007.

[EG08] ERNST M., GREINER G.: Multi Bounding Volume Hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (Los Angels, CA,2008), pp. 35–40.

[GL10] GARANZHA K., LOOP C.: Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Computer Graphics Forum 29* (2010), 289–298.

[Gla09] GLASKOWSKY P. N.: *NVIDIA's Fermi: The First Complete GPU Computing Architecture*. Tech. Rep., San Francisco, CA, 2009.

[GR08] GRIBBLE C., RAMANI K.: Coherent Ray Tracing via Stream Filtering. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (Los Angels, CA, August 2008), no. 3 (2008), pp. 59–66.

[Han86] HANRAHAN P.: Using Caching and Breadth-first Search to Speed up Ray-tracing. In *Proceedings on Graphics Interface '86/Vision Interface '86* (Toronto, Ont., Canada, Canada, 1986), Canadian Information Processing Society, pp. 56–61.

[HKL10] HANIKA J., KELLER A., LENSCH H. P. A.: Two-level Ray Tracing with Reordering for Highly Complex Scenes. In *Proceedings of Graphics Interface 2010* (Toronto, Ont., Canada, Canada, 2010), GI '10, Canadian Information Processing Society, pp. 145–152.

[Int10] Intel: Intel Many Integrated Core Architecture, 2010.

[KA02] KENNEDY K., ALLEN J. R.: *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[KS02] KATO T., SAITO J.: "Kilauea": Parallel Global Illumination Renderer. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization* (Aire-la Ville, Switzerland, Switzerland, 2002), EGPGV '02, Eurographics Association, pp. 7–16.

[Lom11] LOMONT C.: Introduction to Intel Advanced Vector Extensions, 2011.

[MMAM07] MANSSON E., MUNKBERG J., AKENINE-MOLLER T.: Deep Coherent Ray Tracing. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 79–85.

[NFLM07] NAVRATIL P. A., FUSSELL D. S., LIN C., MARK W. R.: Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 95–104.

[ORM08] OVERBECK R., RAMAMOORTHI R., MARK W. R.: Large Ray Packets for Real-time Whitted Ray Tracing. In *IEEE/EG Symposium on Interactive Ray Tracing (IRT)* (Los Angels, CA, Aug 2008), pp. 41–48.

[PBB*02] PAUL W. J., BACH P., BOSCH M., FISCHER J., LICHTENAU C., Röhrig J.: Real PRAM Programming. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing* (London, UK, 2002), Euro-Par '02, Springer-Verlag, Berlin, Germany, pp. 522–531.

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering Complex Scenes with Memory-coherent Ray Tracing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 101–108.

[PMS*99] PARKER S., MARTIN W., SLOAN P.-P. J., SHIRLEY P., SMITS B., HANSEN C.: Interactive Ray Tracing. In *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics* (New York, NY, USA, 1999), ACM, pp. 119–126.

[Res07] RESHETOV A.: Faster Ray Packets - Triangle Intersection through Vertex Culling. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 105–112.

[RJ97] REINHARD E., JANSEN F. W.,: Rendering Large Scenes using Parallel Ray Tracing. *Parallel Computer 23* (July1997), 873–885.

[SCS*08] SEILER L., CARMEAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., ESPASA R., GROCHOWSKI E., JUAN T., HANRAHAN P.: Larrabee: a Many-Core x86 Architecture for Visual Computing. In *ACM SIGGRAPH 2008 papers* (New York, NY, USA, 2008), ACM, pp. 18:1–18:15.

[SWS02] SCHMITTLER J., WALD I., SLUSALLEK P.: SaarCOR: a Hardware Architecture for Ray Tracing. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 27–36.

[TH99] THAKKAR S., HUFF T.: Intel Streaming SIMD Extensions. *IEEE Computer 32* (1999), pp. 26–24.

[Tsa09] TSAKOK J. A.: Faster Incoherent Rays: Multi-BVH Ray Stream Tracing. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 151–158.

[vdZRJ95] VAN DER ZWAAN M., REINHARD E., JANSEN F. W.: Pyramid Clipping for Efficient Ray Traversal. *Rendering Techniques'95* (1995), 1–10.

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting Rid of Packets: Efficient SIMD Single-Ray Traversal using Multi-branching BVHs. In *Symposium on Interactive Ray Tracing 2008* (Los Angeles, CA, 2008), IEEE/Eurographics, pp. 49–57.

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26, 1* (2007). ACM, New York, NY, USA.

[WGBK07] WALD I., GRIBBLE C. P., BOULOS S., KENSLER A.: *SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering*. Tech. Rep. UUSCI-2007-012, SCI Institute, University of Utah, 2007.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics 25*, 3 (2006), 485–493.

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–165.