Efficient Ray Tracing Kernels for Modern CPU Architectures

Valentin Fuetterling[1,2], Carsten Lojewski[1], Franz-Josef Pfreundt[1], and Achim Ebert[2]

[1]Fraunhofer ITWM [2]University of Kaiserslautern

Figure 1. Various scenes (SPONZA, DRAGON, FAIRY, R8, HAIRBALL, and POWERPLANT) rendered with diffuse global illumination (GI). Our large-packet traversal for coherent rays can calculate primary visibility for the scenes up to 4.1 times faster than Embree 2.7.1, and our ordered ray stream traversal accelerates diffuse GI ray queries up to 59% compared to the best competing method, dynamic ray stream traversal.

Abstract

The recent push for interactive global illumination (GI) has established the 4-ary bounding volume hierarchy (BVH4) as a highly efficient acceleration structure for incoherent ray queries with single rays. Ray stream techniques augment the fast single-ray traversal with increased utilization of CPU vector units and leverage memory bandwidth for batches of rays. Despite their success, the proposed implementations suffer from high bookkeeping cost and batch fragmentation, especially for small batch sizes. Furthermore, due to the focus on incoherent rays, optimization for highly coherent BVH4 ray queries, such as primary visibility, has received little attention. Our contribution is twofold: For coherent ray sets, we introduce a large packet traversal tailored to the BVH4 that is faster than the original BVH2 variant, and for incoherent ray batches we propose a novel implementation of ray streams which reduces the bookkeeping cost while strictly maintaining the preferred traversal order of individual rays. Both algorithms are designed around a fast traversal order look-up mechanism. We evaluate our work for primary visibility and diffuse GI and demonstrate significant performance gains over current state-of-the-art implementations.

 ISSN 2331-7418

1. Introduction

The coherence of a ray set can be defined as the tendency of the individual rays to follow the same traversal path through an acceleration structure and to intersect the same primitives. Perfectly coherent rays perform identical operations throughout the entire traversal routine. Thus, ray coherence maps perfectly to the single instruction multiple data (SIMD) units of modern CPU architectures, ranging from four (SSE) to 16 (AVX-512) vector elements. Primary visibility queries, for example, produce well-characterized coherent ray sets with a common origin and tightly bundled ray directions. In contrast, the rays spawned by diffuse GI calculations can originate anywhere on the scene surface and have their directions randomly distributed. While a set of such rays is commonly labeled as incoherent, usually *hidden* coherence exists, which is just difficult to characterize a priori. Instead, an efficient traversal algorithm aims to extract the hidden coherence on the fly to maximize the utilization of SIMD units.

For coherent rays, SIMD packet traversal was first introduced for the BSP acceleration structure [Wald et al. 2001] and later extended to bounding volume hierarchies (BVH). Packet traversal traces n rays in parallel, where n is the number of SIMD elements. As long as all rays follow the same path through the acceleration structure, full SIMD utilization is achieved. Otherwise, divergent rays need to be masked and the calculations on the corresponding vector elements are wasted. For tightly bundled coherent rays, identical paths are the common case, and a significant speed-up is observed for packet tracing. A traversal technique for packets larger than the SIMD size was proposed by Wald et al. [Wald et al. 2007]; that technique employs arithmetic culling and speculation strategies to accelerate ray traversal beyond SIMD size scaling.

The efficiency of packet traversal has motivated algorithms that attempt to extract coherent subsets out of incoherent ray batches, such as stream filtering and related methods [Overbeck et al. 2008], where ray batches are intersected with the current node in a breadth-first manner, and coherency is extracted implicitly during tree traversal. While this algorithm potentially allows high SIMD utilization, it requires expensive gather-and-scatter operations. An entirely different approach has been proposed by several papers concurrently [Wald et al. 2008; Ernst and Greiner 2008; Dammertz et al. 2008]. Instead of testing n rays against one node, a single ray is intersected with n nodes of a n-ary BVH simultaneously, where branching factors of 4–16 have been investigated. Aside from efficient SIMD utilization, higher branching factors reduce the depth and memory footprint of a BVH. On the negative side, early culling opportunities are reduced because some nodes are intersected that would not have been visited in a tree with a lower branching factor. This leads to wasted operations and bandwidth. Combining the strengths of ray streams and BVH4 yields a simple and efficient solution [Tsakok 2009]. Like previous streaming algorithms, a major drawback is a common traversal order that is enforced for every ray in the batch. This restriction was relaxed by the introduction of dynamic ray stream traversal (DRST) [Barringer and Akenine-Möller 2014], where, in the case of BVH4, each ray can follow almost the same traversal order that would result from single ray traversal. Despite its high performance, two major drawbacks are apparent:

- First, the flexible traversal order results in a large number of bins leading to fragmentation of the ray batch and increased bookkeeping overhead. Accordingly, the method achieves its full potential only for large ray batches.
- Second, despite the flexibility, the number of possible traversal orders is still limited to 8 (2!*2!*2!) out of 24 (4!).

In the following, we introduce two novel algorithms for the BVH4, which further accelerate ray traversal of coherent and incoherent rays from 35% to 4.1× compared to previous results. Section 2 describes the construction and memory layout of the BVH4 and introduces a fast look-up mechanism for ordered traversal. In Sections 3 and 4, the algorithms for coherent and incoherent ray transport are described, respectively. Our results are compared to Embree and DRST in Section 5, and the conclusion is presented in Section 6.

2. BVH4 Implementation

2.1. BVH4 Construction

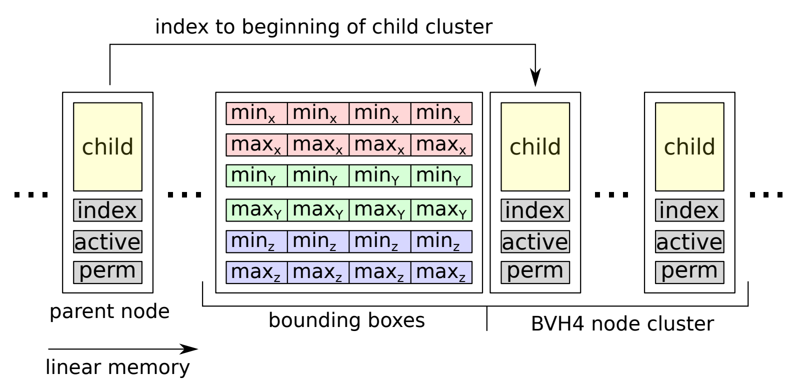

Figure 2. Memory layout of the BVH4. A BVH4 node contains four records: 'child' is an index to a cluster of child nodes, 'index' is the relative position of a node within its own cluster, 'active' is a bit mask that has a set bit for every active child node, and 'perm' encodes the geometric configuration of the child nodes which is used in the ordered traversal look-up. If the node is a leaf, 'child' indexes a list of primitives and 'perm' holds the primitive count. Every node cluster is directly preceded by a SOA structure of four bounding boxes independent of the cluster size to allow vectorized access.

Like previous approaches [Dammertz et al. 2008], our BVH4 is obtained directly from an existing BVH2 tree. While a native BVH4 construction potentially leads to higher-quality structures, the conversion process is simpler to implement in available BVH2 frameworks. For the creation of the initial BVH2, we use our own optimized implementation of the SBVH algorithm [Stich et al. 2009]. The conversion proceeds from top to bottom, always collapsing the two intermediate nodes in between the current node and its grandchildren. If one or both of the intermediate nodes are leaves, the resulting BVH4 node may contain only two or three children.The memory layout of the BVH4 structure is shown in Figure 2. We conceptually separate the BVH4 node from its geometric bounding box. BVH4 nodes are always grouped in clusters with capacities of two or four nodes (never three due to 16-byte alignment for vectorized access). Every cluster is adjacent in memory to four bounding boxes in structure-of-arrays (SOA) format (96 bytes) which are located in front of the cluster. With this definition, a BVH4 node occupies only eight bytes, with a five-byte index to the child-node cluster, one byte for the active child mask, one byte for the node's relative index within its own node cluster, and one byte for traversal order look-up. In case of a leaf, the active child mask is set to zero, the five-byte index points to a list of triangles, and the number of triangles is saved in the look-up byte. This somewhat flexible structure allows a saving of about 5% of memory compared to the standard 128-byte layout [Dammertz et al. 2008; Ernst and Greiner 2008] and also performs marginally faster during traversal.

2.2. Ordered Traversal

A ray that overlaps with multiple child nodes may trace the corresponding subtrees in any order and always produce the correct final result. However, for maximum performance, it is crucial to maintain a front-to-back ordering so that a successful primitive intersection can efficiently cull more distant nodes. If nodes overlap, a strict front-to-back ordering can not be determined a priori, but a heuristic can be employed to make a good guess. Commonly implemented heuristics are based on the distance of the ray origin to the node bounding box (DIST-H) or on the signs of the ray direction (SIGN-H). The DIST-H corresponds to a strict front-to-back ordering if the ray does not cross a region of overlapping nodes. The ray is intersected with all bounding boxes, and the distance values are sorted to determine the traversal order. If multiple rays are traced simultaneously, average distance values can be used to steer the entire batch [Tsakok 2009]. The main drawback of DIST-H is the computationally expensive sorting step, especially for branching factors larger than two. A heuristic that can be evaluated more efficiently is SIGN-H, as it does not depend on the dynamics of the traversal. For a BVH2, the traversal order is chosen based on the axis along which the child nodes overlap the least and on the sign of the corresponding ray-direction component. The axis is stored in the node data structure to quickly index the ray sign

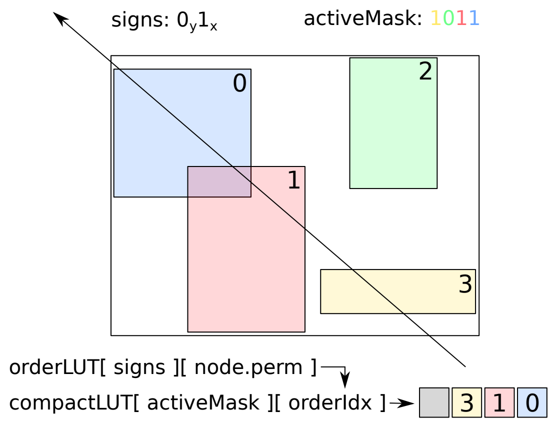

Figure 3. Visualization of the traversal order look-up mechanism. The colored boxes represent child nodes that are labeled according to memory order. A ray with a negative x-sign and a positive y-sign intersects child nodes 3, 1, and 0, and the corresponding color-coded 'active mask' is shown. The geometrical relations of the child nodes are captured in the parent node's 'perm' field. The 'perm' value together with the ray signs produces an 'order index' from the order look-up table, which may represent any permutation of the child nodes. The 'order index' together with the 'active mask' are used for a second look-up to retrieve a sorted, compact list of intersected child nodes.

during traversal. The heuristic has been applied to the BVH4 [Dammertz et al. 2008], so that the traversal order of the underlying BVH2 is reproduced. Somewhat surprisingly, the authors report that SIGN-H results in slightly less visited nodes compared to DIST-H, even for single rays. Despite the good results, the presented SIGN-H implementation carries some overhead and does not allow for all possible 24 permutations of the traversal order, only those eight that can be mapped to a balanced BVH2 subtree. In the following, we present a SIGN-H look-up mechanism that removes both limitations (Figure 3).

Our BVH4 node layout (Figure 2) contains a *perm* field, which is used in conjunction with the ray *sign mask* to obtain an *order index* from the two-dimensional order look-up table (orderLUT). The ray *sign mask* contains the signs of the ray components in the lowest 3 bits, and the *order index* represents one of the 24 possible traversal orders. The *order index* together with the *active node mask* forms a second index pair to obtain the actual *order* from the two-dimensional compact look-up table (compactLUT). The *order* contains 2-bit indices to the child nodes in a front-to-back order, where the 2 lowest bits are occupied by the most distant node. The *active node mask* maps the lowest 4 bits to the child nodes according to the memory position and has a set bit for every intersected node. As a result, the indices obtained from the *order* are already compacted to reference only child nodes that are actually intersected, so that the traversal can directly proceed without additional checks. The complex logic implemented by the two look-ups is very efficient and would require significantly more instructions otherwise. The remainder of this section explains the details of the *perm* index structure.

The *perm* index is constructed as a hierarchy of three subdivisions to partition the four child nodes. There are exactly two unique hierarchy types: One is the balanced type, where the first subdivision creates two sets of two nodes, and the second is the unbalanced type, where the first subdivision creates one set of one node and one set of 3 nodes. All possible subdivision variations of the balanced type are symmetric and can be accounted for by adjusting the relative position of the child nodes in memory accordingly. For the unbalanced type, four unique topologies exist. Thus, the first part of the *perm* index identifies one of the five possible topologies. The second part defines the axes for each of the three subdivisions. The three possible axes (x, y, z) result in $3 * 3 * 3 = 27$ variants for each topology. Thus the *perm* index is computed as:

$$\mathrm{axis}_{1^{\mathrm{st}}\mathrm{split}} + \mathrm{axis}_{2^{\mathrm{nd}}\mathrm{split}} * 3 + \mathrm{axis}_{3^{\mathrm{rd}}\mathrm{split}} * 9 + \mathrm{topologyId} * 27$$

Since a ray can have $2^3 = 8$ possible sign combinations, 8 sets of the topology variants are required, resulting in a orderLUT size of $5 * 27 * 8 = 1080$ bytes which corresponds to 17 64-byte cache lines. The compactLUT has a size of $24 * 16 = 384$ bytes (24 possible traversal orders and 16 possible active mask combinations), so that a moderate total of 26 cache lines is reserved for the look-up tables. If a BVH4 is derived from a BVH2, the original BVH2 traversal order is maintained by transferring the axis values (defining the axis along which the two children of a BVH2 node overlap the least) and setting the topologyID to the balanced type. Consider a BVH2 treelet with a root node, two intermediate nodes and four child nodes, which is flattened into a single BVH4 node. Then, $\mathrm{axis}_{1^{\mathrm{st}}\mathrm{split}}$ receives the axis value of the root node, and $\mathrm{axis}_{2^{\mathrm{nd}}\mathrm{split}}$ and $\mathrm{axis}_{3^{\mathrm{rd}}\mathrm{split}}$ receive the axis values of the left and right intermediate nodes, respectively. The four child nodes are arranged in memory according to the traversal order of a ray with only positive direction components. Example code for the generation of the look-up tables is provided in Appendix A.

3. Coherent Large Packet Traversal

A very efficient traversal algorithm for large packets of coherent rays exists for the BVH2 [Wald et al. 2007], where a single search ray out of the packet is intersected with the current BVH node, speculatively descending the entire packet on hit. Otherwise, a conservative frustum or interval test attempts to reject the node completely. Only then are the individual rays tested in groups of the SIMD-size and, upon the first intersection (if any), the search ray is updated and the traversal continues down the tree. The implementation combines conservative early miss, speculative early hit, ordered traversal, active ray tracking, and a packet test of last resort in a single unified traversal step. Efficiently adapting this method to the BVH4 presents two challenges:

(1) The complexity of the traversal logic increases significantly. While during a BVH2 traversal step only a single bounding box is processed (the current node's), in the case of the BVH4, a variable number of up to four child bounding boxes needs to be considered simultaneously.

(2) Maintaining the culling efficiency of the BVH2 algorithm can be tricky and requires special care.

We solve challenge (1) with our efficient look-up mechanism introduced in Section 2.2 and challenge (2) with a combination of a deferred packet test of last resort and early-hit pruning. Conceptually our algorithm is very similar to the original BVH2 variant described above. The pseudocode of the complete algorithm is given in Algorithm 1, and the various techniques are discussed in detail below.

Algorithm 1. Coherent large packet traversal. Vector symbols are postfixed with a 4, as are functions operating on vector data. Individual rays are accessed with R(ray index), SIMD ray packets with R4(packet index). Sign4() concatenates the sign bits of vector elements to a 4 bit integer, IsectBox4x1() intersects 4 child nodes with one ray, IsectBox1x4() intersects one node with a SIMD packet, FirstSetBit() returns the index of the lowest set bit, BitSet() returns the value of a bit at the given bit index, and CountBits() counts the number of set bits.

1: $stack[]$
2: $sP \leftarrow 0$
3: $node \leftarrow root$
4: $activeRID \leftarrow 0$
5: **loop**
6:   **if** $node$ is inner node **then**
7:     $activeMsk \leftarrow$ IntersectBox4($node, intRay$)
8:     $activeMsk \leftarrow activeMsk$ **and** $node.activeMsk$
9:     **if** $activeMsk = 0$ **then**
10:       **goto** line 42
11:     **end if**
12:     $(hitMsk, t4) \leftarrow$IsectBox4x1($node$, R($activeRID$))
13:     $savedRID \leftarrow activeRID$
14:     $orderIdx \leftarrow orderLUT[sign][node.perm]$
15:     $order[] \leftarrow compactLUT[orderIdx][activeMsk]$
16:     $o \leftarrow 0$
17:     **for** $cnt \leftarrow$ CountBits($activeMsk$)-1 **to** 0 **do**
18:       $o \leftarrow order[cnt]$
19:       **if** BitSet($hitMsk, o$) **then**
20:         **goto** line 31
21:       **end if**
22:       **for** $p \leftarrow activeRID/4$ **to** $numPackets$ **do**

```
23:             msk ← IsectBox1x4(node,R4(p))
24:             if msk ≠ 0 then
25:                 activeRID ← p ∗ 4+FirstSetBit(msk)
26:                 goto line 31
27:             end if
28:         end for
29:     end for
30:     goto line 42
31:     for i ← 0 to cnt − 1 do
32:         so ← order[i]
33:         n ← root + node.child + so
34:         stack[sP] ← (n, savedRID, t4[so])
35:         sP ← sP + 1
36:     end for
37:     node ← root + node.child + o
38:     continue
39:   else
40:     Leaf intersection
41:   end if
42:   while sP ← sP − 1 ≥ 0 do
43:     (node, activeRID, t) ← stack[sP]
44:     if cast2uint(t) < cast2uint(R(activeRid).t) then
45:         goto line 5
46:     end if
47:     for p ← activeRID/4 to numPackets do
48:         msk ← IsectBox1x4(node, R4(p))
49:         if msk ≠ 0 then
50:             activeRID ← p ∗ 4+FirstSetBit(msk)
51:             goto line 5
52:         end if
53:     end for
54:   end while
55:   break
56: end loop
```

Conservative early miss. This test attempts to quickly reject nodes that do not overlap with any rays in the packet. We use interval arithmetic (IA) [Hayes 2003], where intervals on the inverse ray directions are precomputed prior to traversal and evaluated with a standard ray-box intersection (line 7, Algorithm 1). While the alternative frustum plane test [Assarsson and Möller 2000] is less conservative, the increased computational cost does not pay off, especially because IA shares some calculations with the speculative early hit (line 12, see below). The IA is evaluated for all four child nodes simultaneously, resulting in a 4-bit active mask which is further superimposed with the active mask stored in the node data structure (Figure 2). If the active mask is zero at this point, all active child nodes are missed and traversal can continue with a node from the stack (lines 42–54).

Speculative early hit. The purpose of this test is descending into a child node early if it overlaps with the current active ray (see active ray tracking below), omitting bounding box intersections for the rest of the rays, which are assumed to take the same traversal path. While this speculation can be harmful in the case of divergent rays, for highly coherent rays the performance gains can be huge. The intersection of the active ray is performed for all child nodes in parallel, and an early-hit mask along with the intersection distance is produced (line 12). The intersection distance is set to a value with all bits equal to one if no intersection exists.

Ordered traversal. As described in Section 2.2, the *perm* field of the current node, together with the signs of the active ray, represent an index into the orderLUT, which yields the *order index* (line 14). The *order index* in conjunction with the *active mask* form another index into the compactLUT, which returns an ordered list of active child-node indices (line 15). The list is represented by a single byte and the elements are 2-bit values.

Deferred packet test of last resort. The ordered child nodes are iterated from front to back (lines 17–29). First, the early-hit mask of the current child node is checked and if the corresponding bit is set, control flow exits the loop (lines 19–21). Otherwise, the packet test of last resort is executed (lines 22–28). If a new active ray can be found, control flow returns to the early-hit path (lines 24–27). If not, the loop continues with the next child node until all have been processed without success (line 30). At this point, traversal can continue with a node from the stack. The first child node with a ray overlap (if any) becomes the new current node (line 37). The remaining unprocessed child-node indices are pushed onto the stack in a back-to-front order, together with the near distance value from the early-hit test and the corresponding active ray (lines 31–36). This way, the packet test of last resort is deferred for stacked nodes until the moment of their retrieval (lines 43–53), maximizing culling efficiency.

Active ray tracking. Active ray tracking is essential for the early-hit test, because if the test fails once it will fail for the entire subtree. Instead, a new active ray (if any) is found during the packet test of last resort (line 25). As a bonus, all rays that come before the active ray are guaranteed to miss the current subtree and require no further processing. In our implementation, the rays of a large packet are stored continuously in memory in a SOA format befitting the SIMD width (SIMD packet), so that all rays in a SIMD packet can be intersected with one node simultaneously (line 23). The active ray is the first ray in memory order, which has an overlap with the current node. The index of the active ray is composed from its SIMD packet index and its position within the SIMD packet (lines 25 and 50).

Early-hit pruning. A node popped from the stack requires a deferred packet test of last resort if the early-hit test has failed, but also if, in the meantime, the active ray has detected a primitive intersection closer than the saved early-hit distance. This is important to increase culling efficiency. Since the saved distance value is either a positive real number due to a successful early hit or a value with all bits set to one otherwise, both cases are evaluated correctly by comparing the bit patterns of the saved distance and the active ray distance as unsigned integers (line 44).

The combination of the presented techniques forms an efficient algorithm for BVH4 large packet traversal of coherent rays. The overhead of the traversal logic is minimized due to our order + compaction look-up mechanism, and a culling efficiency identical to the original BVH2 traversal is achieved by the combination of the deferred packet test of last resort and early-hit pruning. The notable advantages of the BVH4 algorithm are full SIMD utilization during early-hit and early-miss tests and significantly reduced traversal steps resulting from the shallower tree.

4. Ordered Ray Stream Traversal

Ray streams have emerged as an efficient approach to extract hidden coherence from incoherent ray batches, reducing memory bandwidth and increasing SIMD utilization compared to single ray traversal. The dynamic ray stream traversal (DRST) [Barringer and Akenine-Möller 2014] appears to be the fastest algorithm so far, deriving its performance from the ability to grant each ray its individual traversal order to increase culling efficiency. In the BVH4 case, however, the possible permutations of the traversal order are limited to eight out of 24 (compare balanced type, Section 2.2). In addition, during every traversal step, rays are mapped to nine different bins, producing considerable maintenance overhead and leading to batch fragmentation. In the following, we propose a ray stream implementation, which like the DRST allows each ray in the batch to traverse the tree in its preferred order. Unlike the DRST however, a ray may follow any of the 24 possible permutations. In addition, no fragmentation of the ray batch happens during traversal (only once initially) and the number of bins is reduced to four, minimizing overhead.

Algorithm 2. Ordered ray stream traversal. Vector symbols are postfixed with a 4, as are functions operating on vector data. For the definition of functions see Algorithm 1.

$rayStack[8][]$
$rayStart[8] \leftarrow 0$
$sP \leftarrow 0$
for $rayID \leftarrow 0$ **to** $|R|$ **do**
    $signs \leftarrow$ Sign4(R($rayID$))
    $rayStack[signs][rayStart[signs]] \leftarrow rayID$
    $rayStart[signs] \leftarrow rayStart[signs] + 1$

```
end for
for i ← 0 to 8 do
    if rayStart[i] > 0 then
        taskStack[sP] ← (root, rayStart[i], i, i)
        sP ← sP + 1
    end if
end for
while sP > 0 do
    sP ← sP − 1
    (node, numRays, lane, sign) ← taskStack[sP]
    if node is inner node then
        rayStart[lane] ← rayStart[lane] − numRays
        numActive4 ← rayStart4
        for r ← 0 to numRays do
            rayID ← rayStack[lane][rayStart[lane] + r]
            msk4 ← IsectBox4x1(node, R(rayID))
            for i ← 0 to 4 do
                rayStack[i][rayStart[i]] ← rayID
            end for
            rayStart4 ← rayStart4 − msk4
        end for
        activeMsk ←Sign4(rayStart4 > numActive4)
        numActive4 ← rayStart4 − numActive4
        orderIdx ← orderLUT[sign][node.perm]
        order[] ← compactLUT[orderIdx][activeMsk]
        cnt ← CountBits(activeMsk)
        for i ← 0 to cnt do
            o ← order[i]
            n ← root + node.child + o
            taskStack[sP] ← (n, numActive[o], o, sign)
            sP ← sP + 1
        end for
    else
        Leaf intersection
    end if
end while
```

The key feature of our algorithm is the traversal order look-up mechanism presented in Section 2.2. The order produced by this method is exactly the same for all rays with identical signs. Thus, sorting a batch into the octants generated by the possible sign combinations elegantly groups the rays by common traversal order, resulting in up to eight disjunct batches which are then processed successively. The rays included in a batch are tracked by a list of ray indices (short integers). During node intersection, a new list for each of the four nodes is created which contains the indices of all the rays intersecting the respective node. The four lists are pushed to four separate stacks aligned with the four nodes, and the algorithm proceeds with the stack entry corresponding to the node chosen by ordered traversal. Pseudocode is listed in Algorithm 2, and details are discussed below.

At the start, the ray indices of the batch are pushed onto one of eight ray-index stacks depending on the ray sign, each stack corresponding to one of the octants (lines 4–8, Algorithm 2). A ray stack is also called a lane, and the first four lanes are simultaneously mapped to the four child nodes during traversal. Then, a task is generated for each octant that is not empty and pushed onto the task stack (lines 9–14). The core loop begins with popping a task (line 17), checking whether the current node is an inner node (line 18), and if not jumping to the leaf intersection (line 41). Otherwise, the corresponding rays are intersected with the current child nodes in parallel (lines 21–28). The variable *rayStart4* is aliased with the first four lanes of the *rayStart* stack indices. The ray index is copied to the first four lanes associated with the child nodes (lines 24–26), and *rayStart4* is increased for every lane that matches a successful child-node intersection (line 27). Since *msk4* has all bits set if the corresponding node is intersected, the value is subtracted from *rayStart4* in order to increase the stack indices by one. After all active rays have been tested, the traversal order is obtained from the look-up tables, where the bits of the active ray mask (Section 3) are set for all lanes with at least one child-node intersection (lines 29–32). Finally, a new task is generated for all active child nodes and pushed onto the task stack in back-to-front order (lines 33–39). The algorithm is simple and efficient, maximizes coherence, and enables high SIMD utilization.

5. Results

We compare our coherent large packet traversal (CLPT) to our corresponding BVH2 implementation and to Embree [Wald et al. 2014] (version 2.7.1), and our ordered ray stream traversal (ORST) to DRST [Barringer and Akenine-Möller 2014]. An exhaustive comparison between Embree and DRST for incoherent rays can be found in the original work. The benchmarks are performed for six different scenes, the well-known SPONZA, FAIRY, DRAGON, HAIRBALL, and POWERPLANT, and, in addition, for the Audi R8, representing a typical industrial design application.

We generated camera fly-through paths for every scene in order to capture the full scene complexity and avoid view point specific variations. While ORST and DRST both run in our own code base on the same BVH4 (Section 2), Embree has a separate code base and generates its own BVH4 (set to highest quality, spatial splits enabled, optimized for coherent rays). The variations among the acceleration structures might introduce a small scene-dependent bias for either Embree or CLPT.

The hardware platform is a dual socket Intel Xeon E5-2680v3 Haswell (24 cores / 48 threads total at 2.5GHz) which allows Embree and DRST to use AVX2, while our own implementations target only the AVX instruction set. Finally we evaluate the DIST-H and SIGN-H traversal order heuristics for coherent packets and single ray traversal.

5.1. Primary Visibility

The results are presented in Table 1. The measurements include ray generation, intersection, and simple shading (n-dot-eye type, no secondary rays), which only amounts to a small fraction of the total run time. Embree is configured to use packet traversal (*BVH4Triangle4Intersector8ChunkMoellerNoFilter*). The BVH2 entry is our own optimized implementation of Wald et al.'s algorithm [Wald et al. 2007], the starting point for our CLPT implementation. Both algorithms coexist in the same code base and use a tile size of 8×8 pixels. CLPT is able to outperform Embree significantly, from 1.6× for the HAIRBALL at 1 spp to 4.1× in SPONZA at 16 spp. Our algorithm aggressively exploits the high coherence in the ray sets, saving many instructions compared to Embree due to conservative early miss, speculative early hit, and efficient culling (Section 3). This can be demonstrated in two ways, either by increasing the pixel samples or by considering, for example, the HAIRBALL and SPONZA results. HAIRBALL is a chaotic model with many small triangles, while SPONZA has a number of large surfaces composed of few triangles so that rays are more likely to hit the same triangle. In both situations, rays tend to have higher coherence, which shows in the relative performance increase. The comparison of the HAIRBALL and SPONZA results also bring up a limitation and a corresponding optimization opportunity for CLPT, which shares the performance characteristics of the original BVH2 algorithm: For scenes with very small triangles (compared to the tile size) performance degrades disproportionately compared to Embree due to the overestimating nature of the speculative early hit. In this case a switch from large packets to a finer granularity could be beneficial. Comparing CLPT to its BVH2 ancestor, a speed-up from 1.4× up to

	SPONZA		DRAGON		FAIRY		R8		HAIRBALL		POWERPLANT	
# triangles	66k		871k		174k		795k		2.9M		12.8M	
# frames	4000		2600		3000		4000		2000		2600	
	Mray/s		Mray/s		Mray/s		Mray/s		Mray/s		Mray/s	
	1 spp	16 spp	1 spp	16 spp	1 spp	16 spp	1 spp	16 spp	1 spp	16 spp	1 spp	16 spp
CLPT (AVX)	1192	**1552**	594	**1247**	819	**1282**	883	**1362**	178	**452**	583	**1018**
BVH2 (AVX)	655	736	427	717	493	636	577	752	113	212	286	424
Embree (AVX2)	349	383	293	409	322	385	387	463	110	161	250	307
	Speed-up		Speed-up		Speed-up		Speed-up		Speed-up		Speed-up	
BVH2	1.8×	**2.1**×	1.4×	**1.7**×	1.7×	**2.0**×	1.5×	**1.8**×	1.6×	**2.1**×	2.0×	**2.4**×
Embree	3.4×	**4.1**×	2.0×	**3.1**×	2.5×	**3.3**×	2.3×	**3.0**×	1.6×	**2.8**×	2.3×	**3.3**×

Table 1. Results for primary rays. Pictures of the test scenes are presented in Figure 1. The measurements are taken during a full camera fly-through with the total number of frames listed in the table. The performance numbers include ray generation, traversal, intersection, and simple shading. The resolution is 1280×1024 with one and 16 samples per pixel (spp).

2.4× is observed. To our knowledge, this is the first time a BVH4 traversal for primary rays is demonstrated to be faster than the best competing BVH2 algorithm. Both implementations perform exactly the same number of triangle intersections, because the traversal order produced by the SIGN-H heuristic is identical in both cases, as is the culling efficiency due to the deferred packet test of last resort. At the same time, CLPT requires fewer traversal steps due to the higher branching factor of the BVH4 and also allows efficient vectorization of the conservative early-miss and speculative early-hit tests.

5.2. Diffuse Global Illumination

The benchmark results are presented in Table 2. All frames are rendered in 17 iterations, where each iteration consists of a single primary ray per pixel spawning 16 diffuse rays from its scene-intersection point. The diffuse rays are distributed randomly over the hemisphere around the surface normal according to the Lambertian reflectance model. They can bounce up to four times and are terminated earlier only if they hit the background. In total up to $17 \times 16 \times 4 = 1088$ rays are traced per pixel and frame, and care is taken that the rays are exactly the same for all competing algorithms. The primary ray traversal is performed using CLPT with a tile size of 8×8 pixels and all diffuse rays corresponding to the same tile (up to 1024) are placed in a single batch for the first bounce. Each bounce then produces a new batch

	SPONZA	DRAGON	FAIRY	R8	HAIRBALL	POWERPLANT
# triangles	66k	871k	174k	795k	2.9M	12.8M
# frames	20	13	15	20	10	13
Ray queries	76%	84%	73%	81%	91%	86%
	Mray/s	Mray/s	Mray/s	Mray/s	Mray/s	Mray/s
SR (SSE4)	75	73	72	75	27	74
ORST (AVX)	**117**	**117**	**115**	**118**	**35**	**119**
DRST (AVX2)	84	86	85	85	27	84
	Speed-up	Speed-up	Speed-up	Speed-up	Speed-up	Speed-up
Total	39%	35%	35%	39%	31%	41%
Traversal	**59%**	**47%**	**61%**	**53%**	**35%**	**53%**

Table 2. **Results for diffuse rays**. Pictures of the test scenes are presented in Figure 1. The measurements are taken during a full camera fly-through with the total number of frames listed below. The performance numbers include the entire rendering process, while 'ray queries' lists the percentage of the total time spent in traversal and intersection as measured for single rays (SR). For each frame 17 primary samples and up to 272 diffuse rays with a maximum bounce depth of 4 are generated per pixel at a resolution of 1280×1024 (see text). Speed-up percentages are given once for the total rendering process and once only for the traversal and intersection part.

that is equal or smaller in size compared to the previous batch. An optimized single ray implementation (SR) based on the traversal look-up mechanism is provided for reference. The measurements include random number and ray generation, traversal, intersection, and shading (with textures).

Both ORST and DRST are AVX/AVX2 implementations that always intersect two rays simultaneously with four nodes or four triangles. DRST switches to single ray traversal once the batch size drops below a certain threshold which is optimally set to 16, whereas for ORST a switch to single ray traversal does not increase overall performance. ORST is able to outperform DRST by 37% on average with respect to the total run time. If only ray traversal and intersection is considered, ORST is about 51% faster. One notable result is the complex HAIRBALL scene where DRST offers no advantage over single ray traversal, while ORST can maintain a 31% lead. The throughput increase from DRST to ORST can be attributed to lower overhead (node ordering, stack operations, lane management) and reduced batch fragmentation. The difference in overhead is already apparent from the single ray threshold (0 vs. 16) for which either algorithm achieves its maximum performance. Thus the implementation of the ORST traversal logic is nearly as efficient as that for pure single ray traversal.

	SPONZA	DRAGON	FAIRY	R8	HAIRBALL	POWERPLANT
	avg. rays node / leaf	avg. rays node / leaf	avg. rays node / leaf	avg. rays node / leaf	avg. rays node / leaf	avg. rays node / leaf
Bounce 1						
Initial	1024	1007	1024	1021	984	1000
ORST	27.6 / 8.69	17.3 / 6.19	23.5 / 8.61	35.3 / 14.7	8.84 / 4.03	38.1 / 13.6
DRST	88% / 93%	85% / 88%	85% / 87%	84% / 84%	86% / 88%	87% / 88%
Bounce 2						
Initial	1009	294	617	599	705	603
ORST	8.06 / 3.15	3.29 / 1.41	6.29 / 2.93	6.62 / 2.90	2.94 / 1.55	9.25 / 3.93
DRST	85% / 91%	88% / 93%	86% / 87%	82% / 86%	84% / 90%	88% / 89%
Bounce 3						
Initial	991	131	376	556	621	497
ORST	5.02 / 2.17	2.19 / 1.16	3.93 / 2.08	4.13 / 1.88	2.13 / 1.21	5.13 / 2.35
DRST	86% / 93%	90% / 96%	88% / 89%	85% / 90%	86% / 94%	89% / 91%
Bounce 4						
Initial	968	78	258	552	553	440
ORST	4.09 / 1.90	1.87 / 1.10	3.16 / 1.84	3.38 / 1.63	1.86 / 1.11	3.85 / 1.83
DRST	87% / 93%	93% / 97%	88% / 90%	86% / 92%	88% / 95%	90% / 93%

Table 3. Performance counts for diffuse rays corresponding to the benchmarks presented in Table 2. For each bounce, the average effective batch size (number of active rays in a batch) during traversal and intersection is listed. The batch size for ORST is presented in absolute numbers, while for DRST the results are expressed relative to ORST. In addition the average initial batch size is given. The switch to single ray traversal for DRST is disabled.

Table 3 presents a direct measurement of the batch fragmentation by averaging the batch size maintained by each algorithm throughout traversal and intersection, which can be compared to the average initial batch size. The data is broken down by bounce number to demonstrate the effect of increasingly chaotic ray distributions. For DRST the switch to single ray traversal is disabled to allow a direct comparison to ORST, otherwise the results for the effective batch size would be significantly lower. The numbers show clearly that ORST can always maintain larger batches for both traversal and intersection, up to 19% during the first bounce for the R8 scene. On average, ORST requires only 85% of the traversal steps and 95% of the triangle intersections performed by DRST.

Once a batch is sorted into octants by ray direction, the corresponding subbatches never fragment, and two rays that have an overlapping traversal path are guaranteed to perform the common steps and triangle intersections together. This property is unique to ORST and explains the greater efficiency for coherence extraction compared to DRST. For higher bounces, the advantage decreases slightly due to overall decreased ray coherence.

5.3. Traversal Order

Table 4 compares the DIST-H and SIGN-H traversal-order heuristics according to the number of resulting traversal steps and intersection tests for CLPT and single ray traversal. For the large packets, intersection tests are slightly reduced in the case of SIGN-H, while traversal steps are about equal compared to DIST-H. For single rays, both heuristics perform about the same on average for intersection and traversal, with a slight preference for DIST-H. Due to the fast look-up mechanism, single ray traversal with the SIGN-H ordering is about 1% faster compared to DIST-H in our implementation, although ultimately this depends on the specifics of the rendering framework and, in general, both methods appear to be an equally good choice. The true strength of SIGN-H is the synergy with ORST, where it helps to reduce batch

	SPONZA	DRAGON	FAIRY	R8	HAIRBALL	POWERPLANT
Large packet						
# nodes	100.2%	101.2%	100.6%	99.9%	100.1%	99.7%
# triangles	104.9%	102.7%	104.2%	99.9%	100.7%	100.6%
Single ray						
# nodes	99.9%	100.0%	99.8%	100.0%	100.3%	99.8%
# triangles	103.9%	99.8%	99.7%	100.0%	100.0%	98.9%

Table 4. Performance counters for the DIST-H and SIGN-H traversal-order heuristics for coherent ray packets and single rays corresponding to the benchmarks presented in Table 2. The total number of visited nodes and intersected triangles for DIST-H relative to SIGN-H is reported.

fragmentation without compromising the traversal-order quality for individual rays compared to DIST-H.

6. Conclusion and Future Work

We have presented two novel methods for coherent and incoherent ray transport which are designed around an efficient traversal order look-up mechanism. For primary visibility, we achieve an average speed-up factor of 2.8× over Embree, which is a highly optimized ray-tracing library with many industry applications. For diffuse GI, we find an average traversal speed-up of 51% compared to the recently introduced DRST, which so far has been the fastest method known to us. Both algorithms are tailored to the BVH4, so that a single data structure is sufficient to achieve maximum performance for coherent and incoherent rays alike. For future work, we plan to adopt our implementation to AVX-512 to get ready for the upcoming Xeon Phi (Knights Landing) and future Xeon processors. Another interesting question is whether our algorithms are suited for GPU architectures.

References

ASSARSSON, U., AND MÖLLER, T. 2000. Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools 5*, 1, 9–22. URL: `http://www.tandfonline.com/doi/abs/10.1080/10867651.2000.10487517`, doi:10.1080/10867651.2000.10487517. 98

BARRINGER, R., AND AKENINE-MÖLLER, T. 2014. Dynamic Ray Stream Traversal. *ACM Transactions on Graphics 33*, 4, 9. URL: `http://doi.acm.org/10.1145/2601097.2601222`, doi:10.1145/2601097.2601222. 93, 100, 102

DAMMERTZ, H., HANIKA, J., AND KELLER, A. 2008. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum 27*, 1225–1233. URL: `http://doi.wiley.com/10.1111/j.1467-8659.2008.01261.x`, doi:10.1111/j.1467-8659.2008.01261.x. 92, 94, 95

ERNST, M., AND GREINER, G. 2008. Multi bounding volume hierarchies. In *RT'08 - IEEE/EG Symposium on Interactive Ray Tracing 2008, Proceedings*, IEEE, Los Alamitos, CA, 35–40. URL: `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4634618`, doi:10.1109/RT.2008.4634618. 92, 94

HAYES, B. 2003. A Lucid Interval. *American Scientist 91*, 6, 484. URL: `http://www.americanscientist.org/issues/pub/2003/6/a-lucid-interval`, doi:10.1511/2003.6.484. 98

OVERBECK, R., RAMAMOORTHI, R., AND MARK, W. R. 2008. Large ray packets for real-time Whitted ray tracing. In *RT'08 - IEEE/EG Symposium on Interactive Ray Tracing 2008, Proceedings*, IEEE, Los Alamitos, CA, 41–48. URL: `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4634619`, doi:10.1109/RT.2008.4634619. 92

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG'09)*, ACM, New York, NY, 7–14. URL: http://doi.acm.org/10.1145/1572769.1572771, doi:10.1145/1572769.1572771. 94

TSAKOK, J. 2009. Faster incoherent rays: Multi-BVH ray stream tracing. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG'09)*, ACM, New York, NY, 151–158. URL: http://doi.acm.org/10.1145/1572769.1572793, doi:10.1145/1572769.1572793. 92, 94

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3, 153–165. URL: http://doi.wiley.com/10.1111/1467-8659.00508, doi:10.1111/1467-8659.00508. 92

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1, 1–10. URL: http://doi.acm.org/10.1145/1189762.1206075, doi:10.1145/1186644.1186650. 92, 96, 103

WALD, I., BENTHIN, C., AND BOULOS, S. 2008. Getting rid of packets - Efficient SIMD single-ray traversal using multi-branching BVHs. In *RT'08 - IEEE/EG Symposium on Interactive Ray Tracing 2008, Proceedings*, Wiley, New York, NY, 49–57. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4634620, doi:10.1109/RT.2008.4634620. 92

WALD, I., WOOP, S., BENTHIN, C., JOHNSON, G., AND ERNST, M. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics 33*, 4. URL: http://doi.acm.org/10.1145/2601097.2601199, doi:10.1145/2601097.2601199. 102

A. Look-up Table Generation

C++ code for the generation of the orderLUT and compressLUT as described in Section 2.2.

```
typedef unsigned int uint;
typedef unsigned char uchar;

//order generation marco
#define P(n0,n1,n2,n3) ((n0<<6) | (n1<<4) | (n2<<2) | n3)

//arbitrary order index to order mapping
const uchar indexToOrderLUT[24] = {
  P(0,1,2,3),P(0,2,1,3),P(2,0,1,3),P(1,0,2,3),P(1,2,0,3),P(2,1,0,3),
  P(0,1,3,2),P(0,2,3,1),P(2,0,3,1),P(1,0,3,2),P(1,2,3,0),P(2,1,3,0),
  P(0,3,1,2),P(0,3,2,1),P(2,3,0,1),P(1,3,0,2),P(1,3,2,0),P(2,3,1,0),
  P(3,0,1,2),P(3,0,2,1),P(3,2,0,1),P(3,1,0,2),P(3,1,2,0),P(3,2,1,0)
};

//LUTs to fill
uchar orderLUT[8][136];
```

```
uchar compressLUT[24][16];

//helper LUT
const uchar shiftLUT[4] = {6,4,2,0};

//helper function
uchar orderToIndex(const uchar order){
  uchar idx;
  for(idx=0; idx<24; idx++)
    if(indexToOrderLUT[idx]==order)
      break;
  return idx;
}

//fill compressLUT
for(int orderIdx=0; orderIdx<24; orderIdx++){
  for(int mask=0; mask<16; mask++){
    const uchar order = indexToOrderLUT[orderIdx];
    int corder = 0;
    for(int n=3; n>=0; n--){
      const int idx = (order>>(2*n))&0x3;
      if(mask&(1<<idx)){
        corder <<= 2;
        corder |= idx;
      }
    }
    compressLUT[orderIdx][mask] = corder;
  }
}

//fill orderLUT
for(uint rs=0; rs<8; rs++){
  const uint signs[3] = {rs&1, (rs>>1)&1, (rs>>2)};

  //balanced tree
  for(uint s0=0; s0<3; s0++){
    for(uint s1=0; s1<3; s1++){
      for(uint s2=0; s2<3; s2++){
        const uint axis[2] = {s1, s2};
        const uint node0 = (2*signs[s0])^signs[axis[signs[s0]]];
        const uint node1 = node0^1;
        const uint node2 = (2*(signs[s0]^1))^signs[axis[(signs[s0]^1)]];
        const uint node3 = node2^1;
        const uint mask  = (node0<<6) | (node1<<4) | (node2<<2) | node3;
        orderLUT[rs][s0*9+s1*3+s2] = orderToIndex(mask);
      }
    }
  }

  //unbalanced tree type 1
  for(uint s0=0; s0<3; s0++){
    for(uint s1=0; s1<3; s1++){
      for(uint s2=0; s2<3; s2++){
        const uint node0 = 3*signs[s0];
        const uint node1 = (signs[s0]^1) + 2*signs[s1];
        const uint node2 = (signs[s0]^1) + (signs[s1]^1) + signs[s2];
        const uint node3 = (signs[s0]^1) + (signs[s1]^1) + (signs[s2]^1);
```

```
            const uint mask  = (0<<shiftLUT[node0]) | (1<<shiftLUT[node1]) |
                               (2<<shiftLUT[node2]) | (3<<shiftLUT[node3]);
            orderLUT[rs][s0*9+s1*3+s2+27*1] = orderToIndex(mask);
        }
    }
  }

  //unbalanced tree type 2
  for(uint s0=0; s0<3; s0++){
    for(uint s1=0; s1<3; s1++){
      for(uint s2=0; s2<3; s2++){
        const uint node0 = signs[s0] + 2*signs[s1];
        const uint node1 = signs[s0] + (signs[s1]^1) + signs[s2];
        const uint node2 = signs[s0] + (signs[s1]^1) + (signs[s2]^1);
        const uint node3 = 3*(signs[s0]^1);
        const uint mask  = (0<<shiftLUT[node0]) | (1<<shiftLUT[node1]) |
                           (2<<shiftLUT[node2]) | (3<<shiftLUT[node3]);
        orderLUT[rs][s0*9+s1*3+s2+27*2] = orderToIndex(mask);
      }
    }
  }

  //unbalanced tree type 3
  for(uint s0=0; s0<3; s0++){
    for(uint s1=0; s1<3; s1++){
      for(uint s2=0; s2<3; s2++){
        const uint node0 = 3*signs[s0];
        const uint node1 = (signs[s0]^1) + signs[s1] + signs[s2];
        const uint node2 = (signs[s0]^1) + signs[s1] + (signs[s2]^1);
        const uint node3 = (signs[s0]^1) + 2*(signs[s1]^1);
        const uint mask  = (0<<shiftLUT[node0]) | (1<<shiftLUT[node1]) |
                           (2<<shiftLUT[node2]) | (3<<shiftLUT[node3]);
        orderLUT[rs][s0*9+s1*3+s2+27*3] = orderToIndex(mask);
      }
    }
  }

  //unbalanced tree type 4
  for(uint s0=0; s0<3; s0++){
    for(uint s1=0; s1<3; s1++){
      for(uint s2=0; s2<3; s2++){
        const uint node0 = signs[s0] + signs[s1] + signs[s2];
        const uint node1 = signs[s0] + signs[s1] + (signs[s2]^1);
        const uint node2 = signs[s0] + 2*(signs[s1]^1);
        const uint node3 = 3*(signs[s0]^1);
        const uint mask  = (0<<shiftLUT[node0]) | (1<<shiftLUT[node1]) |
                           (2<<shiftLUT[node2]) | (3<<shiftLUT[node3]);
        orderLUT[rs][s0*9+s1*3+s2+27*4] = orderToIndex(mask);
      }
    }
  }
}
```

Author Contact Information

Valentin Fuetterling
valentin.fuetterling@itwm.fraunhofer.de

Carsten Lojewski
carsten.lojewski@itwm.fraunhofer.de

Franz-Josef Pfreundt
franz-josef.pfreundt@itwm.fraunhofer.de

Achim Ebert
ebert@cs.uni-kl.de

V. Fuetterling, C. Lojewski, F.-J. Pfreundt, A. Ebert, Efficient Ray Tracing Kernels for Modern CPU Architectures, *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, no. 4, 91–111, 2015
http://jcgt.org/published/0003/01/01/

Received:	2015-08-11		
Recommended:	2015-10-12	Corresponding Editor:	Peter Shirley
Published:	2015-12-26	Editor-in-Chief:	Marc Olano