Dynamic Ray Stream Traversal

Rasmus Barringer
Lund University

Tomas Akenine-Möller
Lund University and Intel Corporation

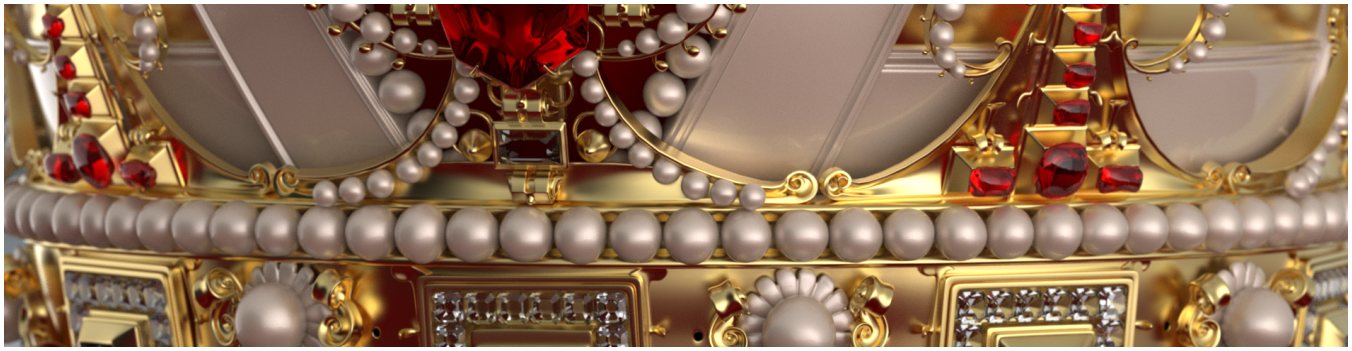


Figure 1: *A close-up of the crown scene rendered using Embree 2.0 modified with our novel traversal algorithm. This scene renders, including shading, frame buffer updates, and all other overhead, about 22% faster with our algorithm on a Haswell-based ultrabook. The performance improvement comes from our traversal, which is 36% faster than the fastest traversal algorithm in Embree for this scene. For other scenes, our traversal algorithm can be over 50% faster.*

Abstract

While each new generation of processors gets larger caches and more compute power, external memory bandwidth capabilities increase at a much lower pace. Additionally, processors are equipped with wide vector units that require low instruction level divergence to be efficiently utilized. In order to exploit these trends for ray tracing, we present an alternative to traditional depth-first ray traversal that takes advantage of the available cache hierarchy, and provides high SIMD efficiency, while keeping memory bus traffic low. Our main contribution is an efficient algorithm for traversing large packets of rays against a bounding volume hierarchy in a way that groups coherent rays during traversal. In contrast to previous large packet traversal methods, our algorithm allows for individual traversal order for each ray, which is essential for efficient ray tracing. Ray tracing algorithms is a mature research field in computer graphics, and despite this, our new technique increases traversal performance by 36-53%, and is applicable to most ray tracers.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture;

Keywords: ray tracing, traversal, coherency, shading

Links: DL PDF

ACM Reference Format
Barringer, R., Akenine-Möller, T. 2014. Dynamic Ray Stream Traversal. ACM Trans. Graph. 33, 4, Article 151 (July 2014), 9 pages. DOI = 10.1145/2601097.2601222 http://doi.acm.org/10.1145/2601097.2601222.

DOI: http://doi.acm.org/10.1145/2601097.2601222

1 Introduction

Ray tracing [Whitted 1980] is a flexible tool that forms the basis for state of the art photo-realistic rendering algorithms. Recently, it has also seen increasing use in real-time applications. One of the core operations in a ray tracer is determining the closest visible surface along a ray. The performance of this operation is critical and is usually accelerated by building a spatial data structure over the scene. One of the most popular spatial data structures is the bounding volume hierarchy (BVH). The time required to trace a set of rays against a BVH is dependent on the size of the scene, the distribution of the rays, the quality of the BVH, and the performance of the ray traversal algorithm. Our goal is to create a high-performance ray traversal algorithm that is less sensitive to the size of the scene and the distribution of the rays, compared to other packet tracing approaches.

Modern CPUs employ a variety of techniques to improve performance by rearranging programs to exploit inherent parallelism [Hennessey and Pattersson 2011]. Superscalar CPUs can execute multiple instructions simultaneously on different functional units and pipelining is used to issue a new instruction each clock cycle. This makes it possible to realize high throughput with many independent instructions, even though the latency of different instructions may differ. When the CPU is unable to issue a new instruction, resources are wasted. Therefore, instruction scheduling is performed both by the compiler and the CPU. However, efficient scheduling is possible if and only if the algorithm contains enough independent instructions at any given time. Our goal is to construct a ray traversal algorithm that aligns well with current hardware, by providing enough parallel work within a single instruction stream.

Data level parallelism attempts to make the most out of each instruction by performing the same computation on several data items using single instruction multiple data (SIMD) execution. The extra performance scales linearly with SIMD width for suitable algorithms and is very power efficient. Current trends indicate that SIMD width will continue to increase in the future. For example, contemporary CPUs have a SIMD width of 256 bits, the many-core Xeon Phi architecture features 512-bit SIMD, and most GPUs have a width of 1024 or 2048 bits. Algorithms need to have low instruc-

tion level divergence in order to efficiently exploit increasing SIMD width. Divergence leads to underutilization since SIMD lanes will need to be masked out. This is not a trait generally attributed to depth-first traversal. Even when packets of rays are traced in a SIMD fashion, rays usually diverge quickly when incoherent, such as for diffuse interreflections, for example.

As compute increases (wider SIMD and more cores), one of the key challenges involves the external memory bandwidth and cache design. To reduce latency and bandwidth, CPUs are equipped with a large cache hierarchy in combination with sophisticated hardware prefetchers that detect memory access patterns, and prefetch memory into a cache before it is needed. Our algorithm is designed to have a predictable memory access pattern with high data coherence, which substantially reduces the amount of memory bandwidth usage in our tests.

Over the last 30 years or so, the topic of ray tracing has been researched thouroughly, and can in many ways be considered a mature research field. As a consequence, it is increasingly difficult to develop algorithms that improve performance, and in particular so for the core methods, such as traversal, intersection, and shading. Despite this, our results show that total ray tracing performance can be improved by 22–37%, while traversal alone is increased by 36–53%, which is rather remarkable. In addition, we expect that our techniques can be applied to a wide range of existing ray tracers, since the BVH is the most popular spatial data structure.

2 Previous Work

As mentioned above, a substantial amount of research has been devoted to finding new and improved ray traversal algorithms in order to make the entire ray tracing process faster. Here, we can only review a small number of these papers, and we chose the ones that are most relevant for our work.

Usually, a stack is maintained that contains the next node to be processed during ray traversal. The technique has been combined with various forms of ray sorting and tracing of whole packets [Wald et al. 2001] of rays to improve performance. Sorting generally incurs some overhead and is usually a heuristic that hopes to improve data locality and minimize divergence. Packets of rays are often quite small and inevitably leads to divergence in all but the most coherent workloads (such as primary visibility). Stack-less ray traversal [Hughes and Lim 2009; Laine 2010; Hapala et al. 2011; Barringer and Akenine-Möller 2013] is a more recent endeavor that was, at least initially, motivated by the high overhead of maintaining a traversal stack on previous generations of GPUs. More recently, uses have been postulated to include cheap ray suspension and transfer on distributed systems or custom hardware [Hapala et al. 2011], though no real demonstration of such system exists, to the best of our knowledge. Áfra and Szirmay-Kalos [2013] present optimized stack-less traversal algorithms for CPUs, MIC, and NVIDIA GPUs using multi-BVHs [Wald et al. 2008; Ernst and Greiner 2008; Dammertz et al. 2008], where a BVH node usually have four or more children in order to improve SIMD efficiency in ray vs. many bounding volumes and intersection tests.

While both stack and stack-less traversal generally performs a depth-first traversal with a small number of rays, Hanrahan [1986] introduced breadth-first ray tracing to increase the number of cache hits for geometry in a beam tracer. Stream filtering [Wald et al. 2007; Gribble and Ramani 2008; Tsakok 2009; Ramani et al. 2009] builds upon the ideas of breadth-first ray tracing to test a large number of rays against the same BVH node or triangle, which may allow high utilization of very wide SIMD. The downside of their approach is mainly that all rays need to traverse the BVH in the same order. Mora [2011] traverses large packets of rays against a set of triangles, but perform traversal in such a way that a BVH is not needed.

When traversing a binary BVH at a certain node, and it is determined that a ray intersects both children, the algorithm must determine which node to traverse next, and which to postpone (usually by pushing it to a stack). A *very important* optimization in ray tracing is to let each ray start traversing the child node that is most likely to occlude the ray, thus potentially making traversal into the other child unnecessary. The order is often based on some heuristic, e.g., start with the node closest to the ray origin. One major disadvantage in existing stream filtering approaches is that all rays must start with the same node, meaning that, for highly divergent rays, about half of the rays will start with a suboptimal node. Our approach combines the strengths of stream filtering, in terms of SIMD efficiency and memory locality, with the flexibility of a depth-first traversal so that any ordering heuristic is possible for each individual ray.

Boulos et al. [2008] use ray packets of arbitrary size in order to exploit coherence otherwise hidden in separate smaller packets. They combine BVH packet traversal (interval arithmetic), SIMD packet tracing, and breadth-first ray tracing. However, they do not attempt to allow for dynamic descent direction for each individual ray during traversal, i.e., all rays in a packet must take the same path in the tree. Tsako et al. [2009] combine multi-BVH traversal with stream tracing of large packets in order to reduce the memory bandwidth associated with single-ray tracing against an n-way BVH, while eliminating the cost of filtering rays at each traversal step. They maintain one stack of rays per SIMD lane that are referenced by a single task stack. Again, the traversal order is the same for all rays.

Garanzha and Loop [2010] target GPU architectures and improve performance by efficiently sorting rays for coherence and then traversing coherent packets of rays using frustums in a breadth-first manner. They use an Octo-BVH and traverse using an ordering heuristic based solely on the average ray direction within a frustum.

3 Overview

As previously mentioned, a stack, which contains nodes that are still to be tested against the ray, is usually maintained during single-ray traversal. If multiple rays are traversed simultaneously, rays will typically have some of the same entries in their node stacks. This observation forms the basis for our algorithm. The key idea is that if multiple rays are to be tested against the same node, this can be performed in a very efficient manner. First, memory fetches for that node can be amortized over multiple rays. Second, since a single node is tested against multiple rays that are completely independent, we gain instruction level parallelism and have additional opportunities for vectorization. This means that 1) SIMD execution can be used, and 2) dependency chains that may stall the pipeline can be avoided. In fact, if enough rays are tested against the same node, vectorization can scale to any SIMD width.

We therefore form a *ray stream*, which is essentially just a collection of rays, e.g., the initial eye rays from a 16×16 pixel tile, or some light paths in a global illumination renderer. When traversing a ray stream against a BVH, our goal is to group rays that take the same path in the tree, without restricting the path of each individual ray. As such, we allow for a flexible traversal order but utilize coherence when present.

The subsequent section describes the details of the algorithm and then follows implementation details in Section 5. Results are presented in Section 6 and finally, we offer some conclusions and ideas for future work.

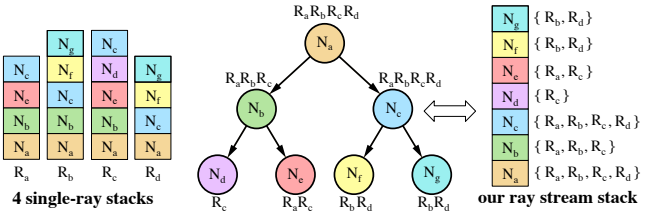


Figure 2: *Example traversal state showing the relationship between multiple single-ray stacks and our combined ray stream stack. $N_{a..f}$ represents nodes of the BVH and $R_{a..d}$ represents rays traversed against it. Left: four single-ray stacks – note that since a stack is a data structure that changes over time, we have chosen to just visualize which nodes are visited by each ray. Middle: here the BVH is shown, and next to each node, all the rays that visit that node are listed. Right: our ray stream stack is visualized here, and it is straightforward to confirm that those two stack representations include exactly the same information. One part of our new traversal algorithm is that we perform ray vs. node tests in the order provided by the ray stream stack to increase SIMD efficiency.*

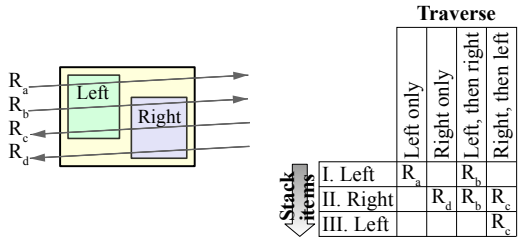


Figure 3: *To the left, a parent box is shown with a left and right child, and four rays, $\mathrm{R_a}$, $\mathrm{R_b}$, $\mathrm{R_c}$, and $\mathrm{R_d}$, which illustrate the different types of paths that can be taken into the children of the parent node. As mentioned in the text, our traversal algorithm needs three stack items, here enumerated* I, II, *and* III. *Depending on which path (left, right, left then right, or right then left) a ray takes, the ray needs to be put into one or more stack items. For example, ray $\mathrm{R_a}$ hits the left child only, and is therefore put into* I. *$\mathrm{R_c}$ hits the right and then the left child, and is therefore put into* II *and* III. *Note that the order is important here, i.e., stack items are visited in the order given by the enumeration (*I, II, *and* III*).*

4 Traversal Algorithm

In order to extract coherence from individual rays in a ray stream, we remove the per-ray node stacks typically used in single-ray traversal. We instead define a single *ray stream stack* that is shared by all rays in a ray stream. In order to keep track of which rays should be tested against a particular node in the ray stream stack, each item in the stack has a list of pointers to associated rays. The relationship between multiple single-ray stacks and our ray stream stack is shown in Figure 2.

Ordinary stack-based tree traversal progresses in steps by testing a single ray, or a small packet of rays, against a node at a time. Our algorithm works in exactly the same way, only that, in our case, a variable sized set of rays is tested against the node at each traversal step. Testing the set of rays should ideally be performed in a loop over the set of rays with few dependencies between loop iterations, so that the loop can be unrolled to provide for instruction level parallelism and opportunities for vectorization. At the same time, we desire to group rays that continue along the same path so coherence can be continued to be exploited in the next traversal step.

Assume that an internal node contains pointers to its children, as well as the bounding volumes of the children. A high level description of our algorithm can be found below:

1. Build a ray stream of, e.g., 4096 eye rays.
2. Set *active rays* to be all rays in the ray stream. This is essentially a list of ray pointers or indices, referencing the ray stream.
3. Fetch the BVH root node, and make it *current node*.
4. Using SIMD, test all active rays against the current node:
   (a) If the node is internal, then test all rays against the BVs of the children (two in the case of a binary BVH).
   (b) If the node is a leaf, then test all rays against the triangles in the leaf.
5. According to the result from step 4a, push stack items to the ray stream stack, each with a BVH node and a corresponding ray pointer list of associated rays.
6. If the stack is empty, we are done. Otherwise continue with the next step.
7. Pop a stack item and make the item's BVH node the current node. Also make the associated list of rays the active rays.
8. Go to 4.

This algorithmic description is fairly complete, but omits an important optimization that switches to single-ray traversal as the active rays becomes few. How this optimization is applied is discussed in more detail in Section 5. The most interesting step in the algorithm is arguably step 5 that deals with appending rays to the ray stream stack. In fact, how to efficiently manage the ray stream stack and its associated ray pointer lists during traversal is the topic of the remaining part of this section.

When testing a set of rays against an internal node (Step 4a), the outcome of the tests determines what stack items are added to the ray stream stack in Step 5. The maximum number of stack items that can be added, and to which child they each refer to, depends on how many children each interior node of the BVH has and the number of possible order configurations that are allowed for continued paths. For example, for BVH2 (Section 4.1), we need to push a maximum of $n = 3$ stack items to the ray stream stack. The specifics for BVH structures with 2 and 4 children are discussed in Sections 4.1 and 4.2.

In order to support a streaming approach where each ray is tested against the internal node once, we need an efficient way to append rays to the stack. The upper limit on the number of stack items that can be added, allows us to reserve space for all of them prior to testing any rays, i.e., before Step 4. After testing a specific ray (Step 4a) and determining how traversal should progress for that ray, a pointer to the ray is appended to all stack items that make up that particular path continuation. Once all rays have been tested, the result is a set of stack items that reference zero or more rays. Only those items that actually reference any rays are pushed to the ray stream stack in Step 5.

The limited number of stack items added at a traversal step means that memory for the ray pointer lists can be managed in a very efficient manner. If the maximum number of items is n, it is enough to allocate n large lists for ray pointers at program start. At each traversal step, ray pointers can simply be appended to the end of these lists by incrementing a counter. Because the ray stream stack always processes items in stack order, memory can be reclaimed from the n lists by subtracting the counter of each list after processing a stack item is complete. Note that we are usually reading ray pointers from one of the lists that we are also appending to. Given that we can never append more ray pointers than what we have read, a careful implementation can read and append new ray pointers in-place without any extra copying.

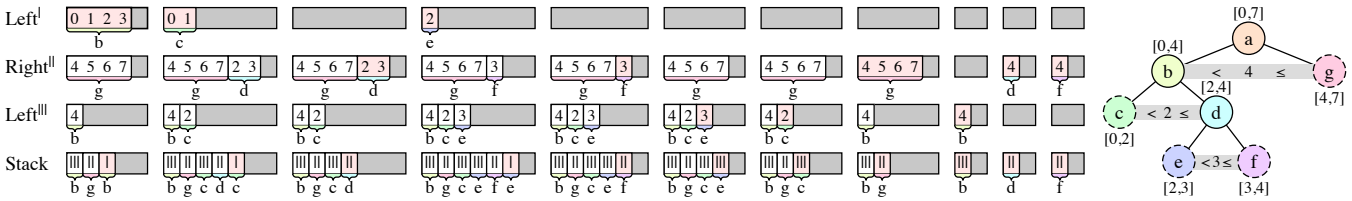


Figure 4: *Rays $\{0, 1, ..., 7\}$ traversing the tree shown to the right. The nodes in the tree are named and leaf nodes are indicated using a dashed outline. For illustrative purpose, traversal is based on ray index, i.e., the interval within brackets indicates what rays will enter a node and the inequality between two siblings represents the ordering heuristic. No occlusion occurs in this example, so all paths are explored. The states of the pre-allocated ray pointer lists are shown from left to right as traversal progresses and the numbers within a range indicate individual rays, while the letter below indicates the associated node. At the bottom, the ray stream stack is shown, referencing ranges in the pre-allocated ray pointer lists. The range that is next to be processed is highlighted in light red. The initial state, to the far left, is the result of testing all rays against the immediate children of the root node. The strength of our algorithm is that all rays within a range can be efficiently processed using SIMD execution.*

4.1 BVH2

In this subsection, we describe how our traversal algorithm works for a BVH2. Each internal node in a BVH2 structure has two children, namely one left and one right. The following outcomes are thus possible during the intersection test:

1. The ray misses both left and right children → do nothing.
2. Traverse into left child.
3. Traverse into right child.
4. Traverse into left child, then right child.
5. Traverse into right child, then left child.

We now need to represent these cases using stack items, each referring to the left child or right child. The first case means that the path is terminated, so it does not need a stack item. To accommodate the following two cases, it is enough to have one stack item for each node and the order of them is not important. However, in order to accommodate the two possible orders when hitting both children, we need to be able to order the left path before the right path in some cases, as well as order the right path before the left path in other cases. For this to work, three stack items are needed that reference the nodes left-right-left or right-left-right. This is explained in Figure 3 for left-right-left. Note that these cases can be accommodated using two different configurations. Which configuration is chosen is generally not important, however, the node that appears a single time may have sightly better SIMD efficiency because of the higher chance of maintaining larger packets as traversal continues.

As described previously, for efficient allocation of ray pointer lists to the different stack items, we pre-allocate one large list of ray pointers for each stack item that may be added during a traversal step. The pre-allocated ray pointer lists need to be able to hold all rays in the worst-case scenario. At most, each ray can exist in a single stack item at each level of the tree, so, if the maximum depth of the BVH is $d = 32$ and the number of rays traversed simultaneously is $m = 1024$, the maximum size of a pre-allocated ray pointer list is $m \times d = 32,768$.

A high level description of the traversal algorithm for BVH2 is shown below. In this case, the left-right-left configuration was chosen.

1. Build a ray stream of size m.
2. Allocate ray pointer lists $Left^{|}$, $Right^{||}$, and $Left^{|||}$, each with $m \times d$ entries, where d is the maximum depth of the BVH.
3. Initialize counters $cnt^{|}$, $cnt^{||}$, and $cnt^{|||}$ to 0.
4. Set *active rays* to be all rays in the ray stream.
5. Fetch the BVH root node, and make it *current node*.
6. Using SIMD, test all active rays against the current node:
   (a) If the node is internal, then test all rays against the BVs of the two children. The outcome determines what to do next:
      i. If the right child was hit, append a pointer to the ray to $Right^{||}$ at position $cnt^{||}$, and increment $cnt^{||}$.
      ii. If the left child was hit and it was closer than the right child, append to $Left^{|}$ and increment $cnt^{|}$.
      iii. If the left child was hit and it was farther than the right child, append to $Left^{|||}$ and increment $cnt^{|||}$.
   (b) If the node is a leaf, then test all rays against the triangles in the leaf.
7. Create stack items for all ray pointer lists where rays were added. Each item references the added range of ray pointers as well as the corresponding node (left or right). Then, push applicable stack items in the following order: $Left^{|||}$, $Right^{||}$, and $Left^{|}$, so that $Left^{|}$ is popped next.
8. If the stack is empty, we are done. Otherwise continue with the next step.
9. Pop a stack item and make the item's BVH node the current node. Also make the associated list of rays the active rays. To free up memory from the pre-allocated ray pointer lists, simply subtract the cnt variable this stack item refers to so that it points to the beginning of the current stack item. Added ray pointers will then overwrite the ones that were just read in the next traversal step.
10. Go to 4.

A detailed example of a few rays traversing a simple BVH2 is shown in Figure 4. Note how the ray stream stack consistently references the last range of rays in the pre-allocated ray pointer lists.

4.2 BVH4

In the case of a BVH with four children per internal node, i.e., a multi-BVH [Wald et al. 2008; Ernst and Greiner 2008; Dammertz et al. 2008], the number of possible outcomes increases. In order to reduce that number, we restrict the ordering to that of two levels of a BVH2. Specifically, let $\{c_0, c_1, c_2, c_3\}$ be the children of an internal node of a BVH4. Conceptually, these nodes can then be put into a BVH2 where the first level has children $\{\text{left}, \text{right}\}$ and $\text{children}(\text{left}) = \{c_0, c_1\}$ and $\text{children}(\text{right}) = \{c_2, c_3\}$. This approximation allows us to accommodate all cases using 9 stack

items, i.e., 3 were required by a BVH2 which indicates that 3×3 are sufficient for two levels in that tree.

Depending on how the BVH4 was created, this restriction can have little to no performance impact since the order flexibility is comparable to that of a BVH2. This would be the case if the BVH4 was built by collapsing nodes in a BVH2. If the BVH4 is instead built by iteratively splitting nodes with the best SAH cost, for example, care should be taken to group nodes that are in the same vicinity.

Analogous to Section 4.1, multiple configurations of stack items will be able to fulfill the ordering requirements. We used the following sequence of nodes for the stack items in our implementation: c_0-c_1-c_0-c_2-c_3-c_2-c_0-c_1-c_0.

For a BVH4, each ray may end up in three stack items at each level of the tree, in the worst case. However, looking at a single ray pointer list, it is sufficient to reserve space for all rays in all levels, since every stack item at the same depth level corresponds to a different ray pointer list. Therefore, each pre-allocated ray pointer list must have a size of $m \times d$, where m is the number of rays, and d is the maximum depth of the BVH, which is the same size required for BVH2 per list. The number of lists increase for BVH4, but on the other hand, the depth of the tree will decrease as well.

5 Implementation

As starting point for our implementation, we use Embree 2.0 [Woop et al. 2013], which is a collection of highly optimized ray traversal kernels and spatial data structures, and is widely used in the industry. The traversal interface of Embree was extended to include support for ray streams. The example path tracer [Kajiya 1986] that comes with Embree, modified with a few performance improvements, was used to render all images, and it was also extended to support our algorithm. Most notably, in addition to the original recursive single-ray render loop, we added a separate render loop that builds ray streams, in order to trace many rays simultaneously. The new render loop eliminated recursion entirely and instead stored the required light path data for each ray, intersected all rays, and resumed each path in a loop. This approach is of similar spirit to what Laine et al. [2013] did to separate material evaluation and path extension from ray casting in a GPU path tracer. Care was taken so that our new render loop traced the same number of rays and generated exactly the same end result as the original path tracer.

The path tracer is parallelized over multiple CPU cores by letting multiple threads work steal tiles of the final image. We made use of coherence within a tile by building a ray stream containing all primary rays generated from that tile. The initial tile size was 16×16 pixels and we used 16 samples per pixel, resulting in 4096 initial rays in a ray stream. Tile size variations are studied further in Section 6, however. Note that the number of rays in a ray stream decreases as light paths are terminated, so efficiency may be lowered after a few bounces. It could be possible to fill up the ray stream with rays from a neighboring tile, for example, but for simplicity and clarity of analysis, we did not attempt to do that.

Our traversal algorithm was optimized for the Intel Haswell architecture which provides support for 8-wide SIMD instructions for floating-point operations. The ray stream stores a single ray in two 32-byte structures, specifically, one used during BVH traversal and one used for triangle intersection. Each of these structures can be fetched using a single 256-bit AVX load. The BVH version of the ray holds the inverse of the ray direction to speed up slab tests. In addition, the ray origin is premultiplied with the inverse ray direction. Hit point information was also stored in a separate 32-byte structure, only accessed when an actual hit is registered.

When implementing the ray stream vs. internal BVH node test, different SIMD strategies were tested. The simplest and most parallel version was to test 8 rays at a time with Struct of Arrays (SoA) style SIMD. However, we found that the register pressure became too high in this case. The winning strategy turned out to be testing 2 rays at a time by running a 4-wide single-ray SIMD box-test over two rays simultaneously. Our ray pointer lists were implemented as arrays of 16-bit integers that index into the ray stream during traversal. For each loop iteration over the ray stream, we thus have to fetch two 16-bit indices and two 32-byte rays. Due to the frequent access, they are likely to be resident in the L1 or L2 cache for reasonable ray stream sizes. Appending ray indices to the ray pointer lists was done using scalar stores. We make use of fused multiply-adds to perform the ray against bounding-box test, which is a high throughput, high latency operation. During this latency, we fetch the rays for the next loop iteration. In order to avoid fetching invalid rays for the next iteration, we always pad with valid ray indices at the end of each stack item, by duplicating the last ray index in the item. The details of our implementation for BVH2 is shown in Appendix A. While the tests for BVH2 and BVH4 are similar, the benefits of BVH4 are twofold: 1) ray loads are amortized over more bounding box tests, and 2) the capability to hide instruction latency increase due to an increased number of slab tests.

As an optimization, our algorithm switches to an optimized single-ray code path whenever there are less than n active rays. This code path is very similar to the single-ray code in Embree, but has been changed to work well with our 32-byte ray structures. For BVH2, n was set to 8, which is a fairly low number. For BVH4, n was instead set to 16, indicating that the overhead for this version is higher, which is expected since up to 9 stack items are added at each traversal step.

Another optimization we tried was to bin rays according to ray direction, which is a technique used by many before us (see Eisenacher et al.'s work [2013] for a review of previous work). This was accomplished by creating 8 initial stack items in the ray stream stack, one for each possible sign configuration of x, y, and z. By sweeping over all rays once and appending each ray to the appropriate item, based on ray direction, we can ensure that all rays within a stack item share the same directional sign. We can then move any direction-dependent code out of the loop over the ray stream, which incidentally reduced register pressure too. Binning rays did not pay off for BVH2 as the lowered SIMD efficiency reduced performance, and the instructions eliminated due to constant ray sign partially resulted in more pipeline stalls. However, it was worthwhile for BVH4 as the register pressure in this case could be lowered by holding fewer node bounds at the same time. Furthermore, the BVH4 implementation has more parallel work, meaning that removed instructions gave a noticeable performance boost.

The renderer makes use of explicit light source sampling using shadow rays at each bounce, which warrants special treatment for these rays. When traversing a shadow ray against a BVH, the nearest hit point is of no interest. Instead, we just want to determine if anything is between a point and a light source. Because of this, it is no longer clear that traversing into the closest node is the best choice. Instead, traversal should continue into the child with the highest chance of occluding a shadow ray [Ize and Hansen 2011]. However, altering the BVH with a new cost metric would be detrimental to the comparison to previous algorithms in Embree, so they are therefore considered out of scope in this paper. Embree instead relaxes the ordering requirement for shadow rays to avoid some overhead. We do the same and implement shadow rays so that all rays in the ray stream take the same path in the tree, which decreases divergence and increases SIMD utilization. At each traversal step, we have a single stack item for each node for shadow rays. The order of these stack items are then determined by a simple heuristic:

the stack item with most rays should be processed first. We have found this to work well in practice.

Embree also includes a set of packet traversal kernels [Wald et al. 2001], as well as hybrid kernels that starts with packets and switches to single-ray traversal as utilization becomes low [Benthin et al. 2012]. To use these kernels, it is the responsibility of the renderer to manage ray packets. The example renderer that supports these kernels constitutes a total rewrite of the single-ray renderer using *ISPC* [Pharr and Mark 2012], which makes the entire renderer vectorized. This makes it a bit difficult to compare performance directly with our algorithm and single-ray traversal. For example, in the case of incoherent rays, shading and path extension get lower utilization as well. On the other hand, in the case of coherence between rays, shading would benefit from vectorization. We note this difference but still make comparisons in Section 6. Also note that with our algorithm, it may be possible to select between a scalar or vectorized version of a shader based on groups of rays created during traversal. However, we have chosen to keep the scalar shaders intact for simplicity and to make the comparison with single-ray traversal easier.

As ray-triangle test, all algorithms use parallel variations of the Möller-Trumbore test [1997]. Our algorithm, as well as the single-ray test, make use of an 8-wide SoA SIMD implementation, intersecting 8 triangles against a single ray at a time. The packet traversal kernel instead tests 8 rays against 4 triangles at a time. The hybrid traversal kernel switches between 8 rays against 4 triangles and 1 ray against 4 triangles.

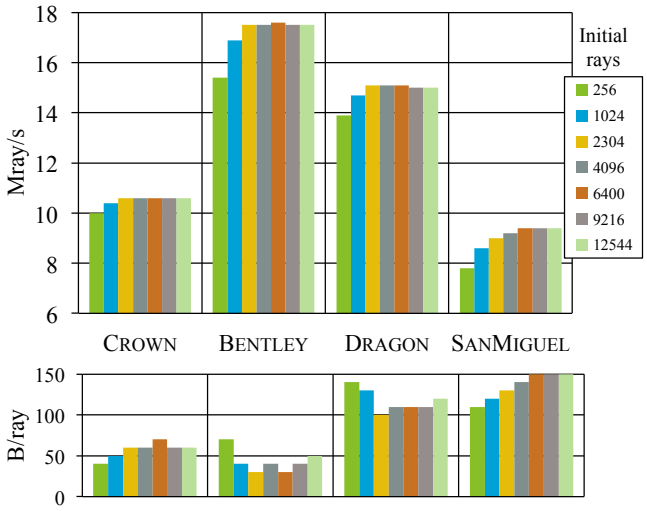


Figure 5: *Performance and bandwidth usage of Our BVH4 when changing the number of initial rays in a ray stream. In order to avoid changing the ray distribution, samples per pixel was fixed at 16, and tile size was varied from* 4×4 *to* 28×28 *pixels. The number of shadow rays that is traversed together is limited to the same number. However, since more shadow rays may be generated in the render loop (up to one per eye ray and light source), multiple batches of shadow rays may be traversed to test all of them.*

6 Results

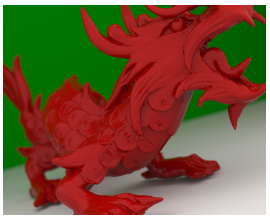
We compare our algorithm to the single-ray, packet, and hybrid traversal kernels available in Embree 2.0. When possible, we compare using both BVH2 and BVH4 as spatial data structures, i.e., BVHs with two and four children, respectively. The BVH is built using the standard settings in Embree, which uses an SAH-binned construction algorithm [Wald 2007]. Since our traversal algorithm was designed to generate exactly the same results as previous traversal algorithms, the generated images are exactly the same, and therefore, an image quality comparison is omitted. The algorithms were evaluated on an ultrabook laptop with a 4-core Intel Haswell Core i7-4750HQ CPU clocked at 2.0 GHz and with a turbo frequency of 3.2 GHz. This chip has 4×32 kB L1 instruction cache, 4×32 kB data cache, 4×256 kB L2 cache, 6 MB L3 cache, and 128 MB L4 EDRAM cache. The highest turbo frequency can only be reached when a single core is active and by monitoring the CPU frequency during rendering, we have found it to be stable at 2.8 GHz on all cores. The thermal design power (TDP) of the CPU and GPU combined is 47 W. Our modified version of Embree 2.0 was compiled with Intel C++ Studio XE 2013 Update 1 for Windows (compiler version 14.0.1) and all tests were performed in Windows 8. The packet renderer was compiled with ISPC version 1.6.0.

The evaluation is based on four test scenes, namely, CROWN, BENTLEY, DRAGON, and SANMIGUEL. The scenes and measured performance are shown in Table 1. As can be seen, our algorithm with BVH4 is the fastest for all scenes, while our BVH2 is the next to fastest. Of the competing traversal methods, there is no clear winner: Hybrid BVH4 is best for BENTLEY, while Single BVH4 is best for the other. The last row reveals that total performance (including even frame buffer updates) is between 22–37% higher with our algorithm compared to the best of the competing methods. The last row also reveals if we only look at the time it takes to perform ray queries, i.e., including traversal and intersection testing (also including the additional overhead of the new render loop that builds ray streams), our traversal algorithm is 36–53% faster. Since our method is consistently and substantially faster for all test scenes, we believe that this is a very encouraging result.

Bandwidth to and from DRAM was measured in gigabytes per second using Intel VTune for an extended number of frames. Specifically, we let the renderer run for 30 seconds without measurements and then measured bandwidth during a 60 second interval. This number was divided by the number of rays per second because a higher throughput of rays would increase the bandwidth usage over a given time. The resulting value we use for comparison is thus bytes per ray, also shown in Table 1. Note that this is not the storage, but rather the average number of bytes needed to transfer to and from DRAM per ray. Our traversal methods use a lower amount of bandwidth per ray, or the same, as the other traversal algorithms. This is likely one of the sources to our performance improvement.

In Figure 5, we have evaluated the performance our algorithm with respect to different tile sizes. Recall that the number of samples per pixel have been fixed to 16, and then we vary the tile size, in order to build ray streams of different size. With a tile size of 12×12, the number of rays in a stream is $12 \times 12 \times 16 = 2304$, for example. As can be seen, 6400 initial rays in a ray stream performs the best, and this is equivalent to a tile size of 20×20 pixels. However, the performance is relatively stable for tile sizes between 16×16 and 24×24 pixels. The number of bytes per ray (B/ray), reported in Figure 5, has some variation among the scenes. Simply from our recursion-less render loop, one can expect a reduction in memory bandwidth because of better data locality. For example, ray queries is performed at the same time for all eye rays, and similarly for all rays after each bounce, which should increase locality of accesses in the BVH. Furthermore, shading happens without interference from traversal, which should improve texture caching. In addition, traversing large ray streams may also be beneficial for memory bandwidth if the reduction in node fetches happens to increase cache hits. This is, however, not certain, and the amount of cache lines the ray stream occupies in the cache may actually decrease cache hits for the BVH. From the plot we see that a reduction






	CROWN		BENTLEY		DRAGON		SANMIGUEL	
# triangles	4.9 million		2.3 million		7.3 million		7.9 million	
Resolution	1280 × 1024		1280 × 1024		1280 × 1024		1280 × 1024	
Ray queries[a]	61%		64%		64%		79%	
	Speed	**BW**	**Speed**	**BW**	**Speed**	**BW**	**Speed**	**BW**
Our BVH2	9.9 Mray/s	60 B/ray	15.7 Mray/s	40 B/ray	13.3 Mray/s	120 B/ray	7.8 Mray/s	130 B/ray
Our BVH4	**10.6** Mray/s	60 B/ray	**17.5** Mray/s	40 B/ray	**15.1** Mray/s	110 B/ray	**9.2** Mray/s	140 B/ray
Single BVH2	7.8 Mray/s	90 B/ray	11.7 Mray/s	80 B/ray	9.9 Mray/s	180 B/ray	5.6 Mray/s	150 B/ray
Single BVH4	*8.7* Mray/s	100 B/ray	12.8 Mray/s	70 B/ray	*11.9* Mray/s	120 B/ray	*6.7* Mray/s	180 B/ray
Packet BVH2	5.2 Mray/s	250 B/ray	8.3 Mray/s	100 B/ray	8.5 Mray/s	220 B/ray	4.0 Mray/s	290 B/ray
Packet BVH4	5.4 Mray/s	220 B/ray	8.9 Mray/s	120 B/ray	8.3 Mray/s	200 B/ray	4.0 Mray/s	330 B/ray
Hybrid BVH4	8.3 Mray/s	160 B/ray	*13.1* Mray/s	90 B/ray	11.5 Mray/s	170 B/ray	6.1 Mray/s	220 B/ray
Total | Traversal	22%	36%	34%	53%	27%	42%	37%	47%

[a]The fraction of rendering time spent on BVH traversal and triangle tests was measured using single-ray BVH4.

Table 1: *Performance (speed) numbers and bandwidth usage (BW), using a laptop CPU, for different algorithms when rendering four scenes. All images were generated by accumulating 16 samples per pixel each frame. The numbers include all work the renderer does, including generation of eye rays, shading, path extension, updating the frame buffer, and presenting the result on the screen. The overhead is approximately linear with respect to the number of rays per second, because higher performance means higher frame rate and more updates to the screen. The last row shows the total performance improvement to the left, and the speedup in terms of just the traversal (but including the additional overhead of the new render loop that builds ray streams for Our) to the right, for each scene. Those numbers are the ratios between our best method (Our BVH4) and the best competing technique (single, packet, hybrid), whose best result is marked in italics.*

in memory bandwidth happens for BENTLEY and DRAGON, but not to a great extent for CROWN and SANMIGUEL, for the ray stream sizes that we investigated.

We also evaluated how our traversal algorithm behaved with varying field of view. The results can be seen in Figure 6, where it is clear that the advantage of our algorithm is relatively large for all angles, except for 8 degrees. In this particular case, only perfectly reflective surfaces are visible within the view, which makes all rays very coherent. As expected, Hybrid traversal performs extremely well in this case and wins by about 10%. The advantage that Hybrid has over our algorithm is twofold. First, it does not spend any time building ray streams and reordering rays, which is unnecessary overhead when rays are fully coherent. Second, the render loop and material evaluation is fully vectorized in this case due to the ISPC renderer and the same material covers the entire view. However, it is clear that this benefit disappears as soon as diffuse surfaces enters the view, and for 32 degrees, our total performance is about 43% higher compared to the fastest of Hybrid and Single BVH4. A nice property of our algorithm is that it does very well for various scenarios without any surprising performance cliffs. If anything, the diagram in Figure 6 reveals that the weaknesses of our algorithm is when all rays are coherent and they hit a reflective surface (so the rays continue to be rather coherent), or they hit a single large surface. However, it should be pointed out that our algorithm is rather fast in these cases, but the Hybrid may be faster.

In order to evaluate exactly where our performance improvement comes from, we measured the fraction of time spent in the ray query kernels (BVH traversal + intersection testing) using VTune during an extended period of time for our test scenes. The measured fraction was then multiplied by the ray throughput to get s/Gray (seconds per gigaray). The results are shown in Table 2. These numbers tell us exactly how much time ray queries take, and therefore, how much our traversal algorithm improves performance, disregarding the new render loop. This is very interesting since results including the render loop already were reported in Table 1. The cycles per instruction (CPI) shows how much the CPU pipeline stalls during traversal, and it is clear that our algorithm has low CPI. To highlight this fact, we show the same numbers (bottom row), but only for our optimized loop, which tests ray streams against the BVH. The rest of the ray query time in our algorithm is actually spent doing triangle intersection or single-ray traversal once coherence is low. Therefore, the optimized loop is the sole source for our increased traversal performance. The CPI value for that loop is very close to the optimal value for our target architecture, Haswell, which can issue two instructions per clock. So, its ideal CPI is 0.5, and the lower the reported CPI numbers are, the better.

For completeness, we also investigated the percentage of all ray vs. BVH node tests that was performed inside the optimized loop. We got the following results for BVH4 for the test scenes: CROWN: 76%, BENTLEY: 88%, DRAGON: 86%, and SANMIGUEL: 83%.

7 Conclusions and Future Work

Over the last 10–15 years, a lot of optimization research effort has been spent on building better BVHs faster, and on parallelizing and SIMDifying the core of the ray tracing algorithm, namely, traversal, intersection testing, and shading. Embree is one of the most optimized ray tracing frameworks, for a variety of CPUs, that we know of, and it is widely used in the industry for this reason. By completely changing the interface to Embree, we have been able to

	Crown		Bentley		Dragon		SanMiguel	
	CPI	Time per ray	CPI	Time per ray	CPI	Time per ray	CPI	Time per ray
Single BVH4 Traversal	1.40	70 s/Gray	1.28	50 s/Gray	1.29	54 s/Gray	1.45	117 s/Gray
Our BVH4 Traversal	1.02	54 s/Gray	0.88	33 s/Gray	0.91	39 s/Gray	0.95	81 s/Gray
Our core loop only	**0.61**	16 s/Gray	**0.61**	13 s/Gray	**0.60**	15 s/Gray	**0.60**	31 s/Gray

Table 2: *Here, we measure the fraction of time spent in the traversal kernels per ray (in seconds per gigaray). We compare Our BVH4 against Single BVH4. In addition, we present cycles per instruction (CPI) as a measurement of how much the CPU pipeline stalls during traversal. At the bottom, we show the same measurements, but only for our optimized loop that tests a ray stream against a BVH4.*

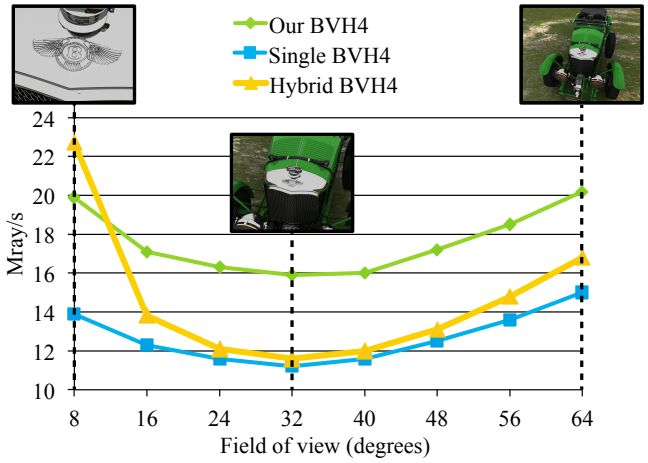


Figure 6: *In this evaluation, we changed the field of view from 8 to 64 degrees, which has similar effects as moving away from an object. As can be seen, our traversal algorithm is substantially faster for all but the smallest field of view. In the smallest field of view, only perfectly reflective surfaces can be seen which makes the rays very coherent. In this case, hybrid traversal does extremely well as expected, but quickly falls behind as diffuse surfaces enter the view. Also, when the field of view becomes very large, most of the image will contain a single large ground polygon, and therefore, the hybrid traversal slowly catches up with our performance.*

implement our novel algorithm that traverses the BVH with large ray streams using a dynamic descent in the top part of the BVH, and then switches to single-ray traversal. As we have demonstrated, when measuring only the time it takes to perform the traversal, our new algorithm is up to 53% faster. For future work, we would like to test whether a memory-mapped layout of the scene can be used to automatically render huge scenes with our traversal method. We believe this should be possible, since our algorithm is good at reducing memory bandwidth usage by visiting the same node with many rays. Furthermore, we would like to investigate whether our method is at all feasible on a GPU and for a Xeon Phi accelerator.

Acknowledgements

Thanks to Tom Piazza, David Blythe, and the whole Advanced Rendering Technology group, all at Intel. Some of Rasmus' work was done during an internship at Intel, San Francisco, hosted by Charles Lingle (thanks!). Tomas is a *Royal Swedish Academy of Sciences Research Fellow*, supported by a grant from the Knut and Alice Wallenberg Foundation. Thanks to Jim Nilsson, Ingo Wald, and Sven Woop for ray tracing discussions and proofreading, and in particular, thanks to Carsten Benthin and Jacob Munkberg for lots of help. Thanks to Martin Lubich for the Crown model (www.loramel.net).

References

Áfra, A. T., and Szirmay-Kalos, L. 2013. Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing. *Computer Graphics Forum, 33*, 1, 129–140.

Barringer, R., and Akenine-Möller, T. 2013. Dynamic Stackless Binary Tree Traversal. *Journal of Computer Graphics Techniques, 2*, 1 (March), 38–49.

Benthin, C., Wald, I., Woop, S., Ernst, M., and Mark, W. 2012. Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics, 18*, 9, 1438–1448.

Boulos, S., Wald, I., and Benthin, C. 2008. Adaptive Ray Packet Reordering. In *Symposium on Interactive Ray Tracing*, 131–138.

Dammertz, H., Hanika, J., and Keller, A. 2008. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Computer Graphics Forum, 27*, 4, 1225–1233.

Eisenacher, C., Nichols, G., Selle, A., and Burley, B. 2013. Sorted Deferred Shading for Production Path Tracing. *Computer Graphics Forum, 32*, 4, 125–132.

Ernst, M., and Greiner, G. 2008. Multi Bounding Volume Hierarchies. In *IEEE Interactive Ray Tracing*, 35–40.

Garanzha, K., and Loop, C. 2010. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum, 29*, 2, 289–298.

Gribble, C. P., and Ramani, K. 2008. Coherent Ray Tracing via Stream Filtering. In *Symposium on Interactive Ray Tracing*, 59–66.

Hanrahan, P. 1986. Using Caches and Breadth-First Search to Speed Up Ray Tracing. In *Graphics Interface*, 56–61.

Hapala, M., Davidovic, T., Wald, I., Havran, V., and Slusallek, P. 2011. Efficient Stack-less BVH Traversal for Ray Tracing. In *27th Spring Conference of Computer Graphics*, 29–34.

Hennessey, J. L., and Pattersson, D. A. 2011. *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann.

Hughes, D. M., and Lim, I. S. 2009. Kd-Jump: a Path-Preserving Stackless Traversal for Faster Isosurface Raytracing on GPUs. *IEEE Transactions on Visualization and Computer Graphics, 15*, 6, 1555–1562.

Ize, T., and Hansen, C. 2011. RTSAH Traversal Order for Occlusion Rays. *Computer Graphics Forum, 30*, 2, 297–305.

KAJIYA, J. T. 1986. The Rendering Equation. *Computer Graphics (Proceedings of ACM SIGGRAPH 86), 20*, 4, 143–150.

LAINE, S., KARRAS, T., AND AILA, T. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *High Performance Graphics*, 137–143.

LAINE, S. 2010. Restart Trail for Stackless BVH Traversal. In *High Performance Graphics*, 107–111.

MÖLLER, T., AND TRUMBORE, B. 1997. Fast, Minimum Storage Ray-triangle Intersection. *Journal of Graphics Tools, 2*, 1, 21–28.

MORA, B. 2011. Naive Ray-tracing: A Divide-and-conquer Approach. *ACM Transactions on Graphics, 30*, 5, 117:1–117:12.

PHARR, M., AND MARK, W. 2012. ispc: A SPMD Compiler for High-Performance CPU Programming. In *Innovative Parallel Computing (InPar)*, 1–13.

RAMANI, K., GRIBBLE, C. P., AND DAVIS, A. 2009. StreamRay: A Stream Filtering Architecture for Coherent Ray Tracing. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 325–336.

TSAKOK, J. A. 2009. Faster Incoherent Rays: Multi-BVH Ray Stream Tracing. In *High Performance Graphics*, 151–158.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum, 20*, 3, 153–164.

WALD, I., GRIBBLE, C. P., BOULOS, S., AND KENSLER, A. 2007. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Tech. Rep. UUSCI-2007-012.

WALD, I., BENTHIN, C., AND BOULOS, S. 2008. Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multi-Branching BVHs. In *IEEE Symposium on Interactive Ray Tracing*, 49–57.

WALD, I. 2007. On Fast Construction of SAH-based Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, 33–40.

WHITTED, T. 1980. An Improved Illumination Model for Shaded Display. *Communications of the ACM, 23*, 6, 343–349.

WOOP, S., FENG, L., WALD, I., AND BENTHIN, C. 2013. Embree Ray Tracing Kernels for CPUs and the Xeon Phi Architecture. In *ACM SIGGRAPH 2013 Talks*.

A Core Loop Implementation for BVH2

Here we list our core loop implementation for testing a ray stream against an internal node of a BVH2 acceleration structure. The code is mainly written using AVX2 intrinsic functions, but we present it using a simplified syntax to improve readability.[1] The loop tests two rays against the two bounding boxes of the children and stores ray indices into the pre-allocated ray pointer lists. The input bounds are organized as $[min_l, min_r, max_l, max_r]$, i.e., a 4-wide vector, for each coordinate axis, and are broadcast to 256 bits by duplicating the 4 floats to both 128-bit lanes. After running through the loop, stack items are created for the ray pointer lists where rays actually were added. That part has been omitted for brevity.

```
void coreLoopBvh2(
  // AABBs for left and right children.
  m256 bbX, m256 bbY, m256 bbZ,
  // Pointers to the end of the pre-allocated ray pointer lists.
  uint16* list0, uint16* list1, uint16* list2,
  // Active rays (pointing into one of the pre-allocated ray pointer lists).
  uint16* activeRays, uint16* lastActiveRay)
  {
  m256 negMask = set1(-0.0f);
  m256 posNegMask = setr(0.0f, 0.0f, -0.0f, -0.0f, 0.0f, 0.0f, -0.0f, -0.0f);

  m256 bbXneg = shuffle(bbX, bbX, M(1,0,3,2));
  m256 bbYneg = shuffle(bbY, bbY, M(1,0,3,2));
  m256 bbZneg = shuffle(bbZ, bbZ, M(1,0,3,2));

  bbX ^= posNegMask; bbY ^= posNegMask; bbZ ^= posNegMask;
  bbXneg ^= posNegMask; bbYneg ^= posNegMask; bbZneg ^= posNegMask;

  uint32 nextIndexA = activeRays[0];
  uint32 nextIndexB = activeRays[1];

  m256 dirNearOrgFarA = load(rayData + nextIndexA);
  m256 dirNearOrgFarB = load(rayData + nextIndexB);

  m256 dirNear = permute2f128(dirNearOrgFarA, dirNearOrgFarB, 0|(2<<4));
  m256 orgFar = permute2f128(dirNearOrgFarA, dirNearOrgFarB, 1|(3<<4));

  m256 orgFarNeg = orgFar ^ negMask;

  m256 dirX = shuffle(dirNear, dirNear, M(0,0,0,0));
  m256 dirY = shuffle(dirNear, dirNear, M(1,1,1,1));
  m256 dirZ = shuffle(dirNear, dirNear, M(2,2,2,2));

  uint32 mask = 0;
  uint32 indexA = 0, indexB = 0;

  for (; activeRays < lastActiveRay; ) {
    uint32 leftHit = mask >> 4;
    uint32 rightHit = (mask >> 5) & 1;
    uint32 order = (mask >> 6) & rightHit;

    m256 bbXray = blendv(bbX, bbXneg, dirX);
    m256 orgX = shuffle(orgFar, orgFarNeg, M(0,0,0,0));
    m256 bbYray = blendv(bbY, bbYneg, dirY);
    m256 orgY = shuffle(orgFar, orgFarNeg, M(1,1,1,1));
    m256 bbZray = blendv(bbZ, bbZneg, dirZ);
    m256 orgZ = shuffle(orgFar, orgFarNeg, M(2,2,2,2));

    *list0 = indexB; *list1 = indexB; *list2 = indexB;
    list0 += leftHit & (1^order);
    list1 += rightHit;
    list2 += leftHit & order;

    indexA = nextIndexA;
    indexB = nextIndexB;
    activeRays += 2;
    nextIndexA = activeRays[0];
    nextIndexB = activeRays[1];

    m256 nearFarX = fmsub(bbXray, dirX, orgX);
    m256 nearFarY = fmsub(bbYray, dirY, orgY);
    m256 nearFarZ = fmsub(bbZray, dirZ, orgZ);
    m256 nearFarRay = shuffle(dirNear, orgFarNeg, M(3,3,3,3));

    dirNearOrgFarA = load(rayData + nextIndexA);
    dirNearOrgFarB = load(rayData + nextIndexB);

    m256 nearFar = max(max(nearFarRay, nearFarX), max(nearFarY, nearFarZ));

    mask = movemask(cmple(shuffle(nearFar, nearFar, M(0,1,1,0)),
                          shuffle(nearFar ^ negMask, nearFar, M(0,0,3,2))));

    dirNear = permute2f128(dirNearOrgFarA, dirNearOrgFarB, 0|(2<<4));
    orgFar = permute2f128(dirNearOrgFarA, dirNearOrgFarB, 1|((3<<4));

    dirX = shuffle(dirNear, dirNear, M(0,0,0,0));
    dirY = shuffle(dirNear, dirNear, M(1,1,1,1));
    dirZ = shuffle(dirNear, dirNear, M(2,2,2,2));

    orgFarNeg = orgFar ^ negMask;

    leftHit = mask;
    rightHit = (mask >> 1) & 1;
    order = (mask >> 2) & rightHit;

    *list0 = indexA; *list1 = indexA; *list2 = indexA;
    list0 += leftHit & (1^order);
    list1 += rightHit;
    list2 += leftHit & order;
  }

  if (indexA != indexB) { // Checks if the last index was duplicated (padding).
    uint32 leftHit = mask >> 4;
    uint32 rightHit = (mask >> 5) & 1;
    uint32 order = (mask >> 6) & rightHit;

    *list0 = indexB; *list1 = indexB; *list2 = indexB;
    list0 += leftHit & (1^order);
    list1 += rightHit;
    list2 += leftHit & order;
  }
}
```

[1] To restore the original source, add `_mm256_` before and `_ps` after functions, replace `m256` with `__m256`, replace `M` with `_MM_SHUFFLE`, and replace the $\wedge$ operator with a call to `_mm256_xor_ps` when used with vector registers.