Faster Incoherent Rays: Multi-BVH Ray Stream Tracing

John A. Tsakok*
Intel Corporation

Figure 1: *BMW Scene (598 K polygons), Conference Room (280 K polygons), Ferrari Scene (100 K polygons) rendered with distribution ray tracing with soft shadows, depth of field and glossy reflections. Rendered using 16 ray paths per pixel.*

Abstract

High fidelity rendering via ray tracing requires tracing incoherent rays for global illumination and other secondary effects. Recent research show that the performance benefits from fast packet traversal schemes that exploit high coherence are lost when coherency is low due to inefficient use of the CPU's SIMD units. In an effort to solve this problem, methods have been proposed which try to extract the remaining coherency from secondary rays through ray sorting, reordering and streaming. Another category of traversal methods have also been proposed which ignore coherency altogether and use a higher order tree branching factor while tracing single rays at a time. These single ray methods not only target applications with incoherent rays but are also scalable with larger SIMD widths. This paper combines ideas from both categories to form a new traversal method which extracts coherency from a group of rays through simple filtering while still providing a fast single ray traversal in cases where there is no coherency present. This new algorithm does not depend on the use of packets which cleanly decouples traversal from shading and is scalable for larger SIMD widths. Results show that overall performance benefits are obtained on a current generation CPU architecture.

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

Keywords: real time, ray tracing, MBVH RS, MBVH, QBVH, ray traversal, incoherent rays, path tracing, global illumination

1 Introduction

Ray tracing of primary rays has reached interactive rates through the use of traversal techniques such as packets [Wald et al. 2001] and frustums [Reshetov et al. 2005; Wald et al. 2007a]. These techniques exploit the high coherence between rays which allow for a high utilization of a CPU's SIMD units while also reducing memory bandwidth through the reduction of node visits within the scene's acceleration structure. Unfortunately, primary rays account for a small percentage of the overall number of rays traced for high fidelity rendering which trace many secondary rays via Whitted [Whitted 1980] or stochastic sampling algorithms [Kajiya 1986; Veach and Guibas 1997; Cook et al. 1984; Jensen 2001] for capturing reflected, refracted and global illumination effects. Ray packets that result from perfectly specular reflections via Whitted ray tracing have been shown to lose coherency after several bounces [Reshetov 2006]. Ray packets using stochastic sampling have been shown to yield even less coherence than Whitted sampling [Boulos et al. 2007] due to its rendering of non-singular effects. However, when these secondary packets are carefully built based on "Ray Type" to exploit coherence, a decent speed up over single rays has been reported [Boulos et al. 2007].

Additionally, the requirement of carefully built secondary packets makes the software implementation of a packet based renderer unattractive. One reason behind this is that shaders are no longer independent of the traversal algorithm since secondary rays must be grouped coherently for performance benefit. This can get complicated [Georgiev and Slusallek 2008; Boulos et al. 2007] when a packet hits different objects with different materials since either: the shaders must generate a coherent group of secondary rays independently; secondary rays from multiple shaders must be grouped together; or both.

Another motivator not to use packets is the trend of wider SIMD units on future CPU architectures such as Intel's Larrabee [Seiler et al. 2008] which features 16-wide SIMD. With a wider SIMD, the efficiency due to inactive rays in packets is even lower than with current architectures.

This paper introduces a new traversal method called "Multi-BVH Ray Stream Tracing" (MBVH RS), which is targeted at the tracing of incoherent or partially coherent rays and is scalable for wider SIMD architectures. This method does not depend on ray packets and thus allowing for a simpler software implementation within the shaders and the rest of the rendering architecture. Background information describing past techniques which led up to design of this method are first discussed. Next, the tree construction and traversal

*e-mail: john.a.tsakok@intel.com

algorithm are detailed with an additional C code listing. Finally, results of several tests are discussed analyzing several important features of the algorithm. Real world performance on current generation CPUs are also summarized with an insight into the behavior of the results.

2 Background

Unlike primary rays, secondary rays are not guaranteed to be highly coherent but they may have either "hidden" coherence or no coherence at all. Hidden coherence within a group of rays can be extracted through the sorting and reordering of rays. The goal of sorting methods is to partition a group of rays into sub packets which contain coherent rays which are then traced as usual. Boulos et al. [2007] report back superior performance over single ray traversal by sorting rays based on "Ray Type". Sorting can also be done over all or some of the five dimensions of each ray: 2D for direction and 3D for origin. Mansson et al. [2007] summarize several different sorting algorithms based on some of these dimensions which were shown to all have inferior performance to that of standard packet tracing with no sort. Despite the fact that sorting was shown to increase coherence, the act of sorting resulted in too many additional operations.

This leads to another group of methods which perform on-the-fly filtering and reordering of rays. Wald et al. [2007b] propose a general method for handling streams of rays on a hardware architecture supporting scatter and gather operations with a wide SIMD. This method traces a large group of rays breadth-first through a BVH structure while filtering out inactive rays from the stream at each traversal step. Results show a high efficiency in SIMD utilization and recent simulator results [Gribble and Ramani 2008] show increasing performance due to higher group sizes and SIMD width. Due to the exact filtering at each step, this method extracts all the coherency out of a group of rays. If little coherency exists, the method will be unable to reach high SIMD efficiency without increasing the initial ray group size. Recently, Overbeck [2008] introduced a method called "Partition Traversal" which is a modified packet traversal with ray reordering where a list of active rays is updated via a simple partitioning scheme. His results show performance improvement over packet traversal on current generation CPUs for Whitted ray tracing using large packets.

Boulos et al. [2008] propose an adaptive reordering technique which breadth-first traces a group of rays through a BVH and only reorders when packet efficiency drops below a threshold. The method also includes fast "all miss" and "all hit" optimizations. Results show improvements over packet tracing for multiple specular, glossy and diffuse path tracing bounces. This method is also shown to benefit from a wider SIMD architecture.

Another class of incoherent ray traversal methods are those using a QBVH [Dammertz et al. 2008] or multi-BVH (MBVH) structure [Ernst and Greiner 2008; Wald et al. 2008]. These algorithms ignore coherency altogether and trace single rays at a time through a BVH which has a higher branching factor, usually equal to the SIMD width. The advantage to doing this is that the SIMD units are always working at high efficiency independent of coherency and allowing scenes to be built with shallower trees and thus less traversal steps. The disadvantage to doing this is that hidden coherency is not exploited which is important in saving memory bandwidth as coherent rays must revisit the same nodes along a common path. Further, during traversal, the SIMD units might be doing speculative work since rays are intersected against boxes which would not have been done in an equivalent binary BVH. However, real world performance seems promising as MBVH was shown to out perform packets when there is a little drop in coherence [Ernst and Greiner 2008]. Also, this method is a good fit for a production renderer as it fits nicely in current rendering architectures of tracing single rays [Christensen et al. 2006].

This paper introduces a new traversal scheme which uses techniques from ray stream tracing and MBVH traversal. The algorithm traces a large group of rays in a breadth-first manner by testing multiple boxes against single rays within the group at each traversal step. As each node is visited, the memory fetch of the node's data is amortized over the group of rays and thus saving memory bandwidth. In situations where there are very few active rays, SIMD units are still highly utilized via multi-box tests. When visiting leaf nodes, packets of rays are built temporarily from active rays and tested against single polygons and thus allowing tree construction to freely group as many or as little number of polygons within leaf nodes since no multi-polygons [Ernst and Greiner 2008] are used.

3 Tree Construction

The tree used is a 4-way BVH structure where internal nodes have up to 4 children. Actual polygon data are referenced in the leaf nodes as usual. Each internal node comprises of the structure-of-arrays (SoA) formatted bounding boxes of its 4 children to facilitate fast SIMD loading as described in [Ernst and Greiner 2008]. In this paper, these 4 boxes are referred to as a "box packet". In a standard MBVH, single rays are intersected against multiple polygons (multi-polygons) at once where there is no difference in cost between doing a test between a ray and 3 polygons versus 4 polygons since they are done simultaneously using SIMD instructions. For MBVH trees, polygons are given incentive to group together through a modified version of the classic surface area heuristic (SAH) [Ernst and Greiner 2008; Wald et al. 2008] allowing for better utilization of SIMD units during intersection. The downside to this is that it has a negative impact on the internal node costs due to an unnatural grouping of multiple polygons. For an approach using ray streams, packets of active rays can be built temporarily to intersect against single polygons and thus utilizing the SIMD units efficiently without having to create multi-polygons. This approach is used as it produces better performance for the tests done on a current generation CPU architecture. However, for future architectures with wider SIMD, a multi-polygon approach might produce better performance since the number of active rays in any leaf might not fully utilize the SIMD units. In the test system, MBVH RS trees are constructed using top down subdivision as described by Wald et al. [2008] using a traditional SAH with 4-way branching.

4 MBVH RS Algorithm

Ray stream tracing is highly dependent on the hardware's scatter and gather support [Gribble and Ramani 2008]. Since many ray packets are built during each traversal step from non-sequential memory accesses, high memory bandwidth becomes a large factor in the overall performance. However, the advantage of ray streams is that it is able to extract hidden coherence from a large group of rays and also amortizes the memory fetch of tree nodes across the group of active rays. On the other hand, MBVH traversal ignores hidden coherency and provides high SIMD utilization during box tests for incoherent rays but incurs high memory bandwidth costs due to node fetching as the fetch is not amortized over multiple rays. MBVH RS solves many of these problems by marrying these two complementing algorithms into a single algorithm.

The idea is to start off with a large group of rays which is traversed in a breadth-first manner via an MBVH structure much like ray streams. Unlike rays streams, no ray packets are built but only single rays are tested against multiple boxes (box packets) simultane-

ously which does not require any scatter/gather support and requires less memory bandwidth. This also solves the MBVH problem of memory bandwidth due to node fetching as each node fetch is amortized over the stream of active rays. Once in the leaf nodes, temporary SIMD ray packets are built from active rays like ray stream tracing and tested against each primitive. The additional computation and memory bandwidth required to build these temporary ray packets is amortized over the number of primitives within the leaf node. By using temporary packets, multi-polygons can be avoided which allows for a more optimized tree as there is no incentive for primitives to be grouped together. There are several places where hardware scatter/gather support would benefit MBVH RS but is not required to achieve real performance improvement and is not used in this test system.

The input into the algorithm is a group of ray structures which sits in a buffer and never moved. These ray structures hold the ray data and hit results. Each ray is referenced via a rayID which is an index into the buffer. A preallocated stack is required for each SIMD lane to hold streams of ray IDs. These stacks are called "activeRayStacks" and are referenced via traversal tasks. Traversal tasks hold a SIMD lane index, the number of active rays and a tree node reference. The tasks are pushed onto a traditional traversal stack called the "taskStack". In the traversal loop, a task is popped from the taskStack which is used to reference one of the activeRayStacks via its SIMD lane index. Next, the task's active ray count is used to determine the number of ray IDs to pop from the activeRayStack. Each ray ID is used to reference a single ray in the ray buffer which is used to test against the box packet of the current node. Each single ray test results in a hit result corresponding to each SIMD lane which is used to push the ray ID onto the corresponding activeRayStack if a hit occured. After each ray of the current task has finished intersecting, new tasks are pushed onto the taskStack each referencing one of the activeRayStacks and corresponding child node. The order in which these tasks are pushed is important for ordered traversal which is described further in this section.

Algorithm 1 describes the traversal algorithm of MBVH RS in more detail. Variables postfixed with a "4" are SIMD_WIDTH vectors. The input into the traversal function is a large (*numRays*) group of rays shown on line 1 which hold ray information and intersection results such as the shortest intersection distance and intersected polygon ID. *activeRayStacks* shown on line 2 is comprised of *SIMD_WIDTH* stacks which hold ray IDs of active rays. These stacks can also be viewed as ray streams [Gribble and Ramani 2008]. Each of the *SIMD_WIDTH* stacks correspond to one of the boxes within a box packet. The *taskStack* holds information about nodes that need further processing and is similar to the traditional stack used for a standard BVH traversal. Each task element on the *taskStack* is comprised of a node reference, an index corresponding to a SIMD lane and the number of active rays in the node. At the start of traversal, a new task is pushed on the *taskStack* which references the root node, has a SIMD lane index of 0 and has *numRays* active rays as shown on line 4. Next, ray IDs of all the rays in the group are pushed onto *activeRayStack[0]* which is the stack corresponding to SIMD lane 0.

Lines 8-31 handle the traversal of the internal nodes of the tree. First, a task is popped from *taskStack* on line 9. Next, a list of *task.numRays* active ray IDs are popped from *activeRayStacks*. Note, the actual implementation does not require physically copying the list of ray IDs to another temporary list as the previous ray IDs can be overwritten as described in the Traversal Implementation section. Next, *length4* and *numActive4* are both initialized to zero where they are each *SIMD_WIDTH* wide floating point and integer values, respectively.

Lines 15-24 describes the inner loop of the traversal which operates on a single ray. Each active ray is tested against a box packet contained within the current node simultaneously where the binary hit results are returned to *result4* and the hit lengths are returned to *tHit4*. In line 17, the hit lengths are added to an accumulator *length4* which is used for ordered traversal. *result4* is then used to increment the *numActive4* counter of active rays within each SIMD lane. The next step is to push the rayID onto the *activeRayStacks*. Once all the active rays have been tested against the box packet and pushed onto their corresponding stacks, child node tasks must be pushed onto the *taskStack* which refer to these active ray IDs.

The order in which these tasks are pushed determines the order in which they are traversed which is in stack (FILO/LIFO) order. Ordered traversal is designed to traverse nodes in front to back order to allow for culling if hit box distances are greater than the last intersection distances. Since there can be many active rays being traced together, there is no correct traversal order for all rays. The ordered heuristic uses an accumulated length of box intersection distances, *length4*, which are then sorted and used to traverse from shortest to longest. This heuristic makes no differentiation between a box that is hit by a few rays at a larger distance or by many rays at a short distance since they can both generate comparable accumulated lengths. However, this case is not as important as when two boxes are hit by the same subset of rays where a chance for culling can occur. This case is handled well by the heuristic since each ray contributes different distances to each node it hits. In situations when there are many active rays, the cost of sorting is amortized such that it hardly affects overall performance. This sorting is shown on lines 25-31. In the test renderer, this ordered heuristic results in a 18% increase in performance.

The final section of the algorithm located at lines 32-42 handles the traversal of the leaf nodes. Once in a leaf node, a list of active ray IDs are popped. These active rays are then loaded in *SIMD_WIDTH* packets into SoA form and tested against single polygons like in traditional packet traversal. After the intersection distances and polygon IDs have been determined, the results are unloaded back into array-of-structures (AoS) form and stored back in the *rays* list. For a SIMD width of 4, this technique of building temporary ray packets has yielded a higher performance than a multi-polygon approach as done in previous MBVH traversal.

The loading and unloading of ray packets can benefit from hardware-supported gather and scatter operations. Though expensive, the manual unloading/loading is not done for every traversal step and is amortized over the number of polygons tested within the leaf. On a side note, it is also possible to apply vertex culling [Reshetov 2007] to the active rays for further increase in performance but this has not been implemented yet.

4.1 Traversal Implementation

This section is intended to provide some more detail into the code-level implementation of the traversal loop using Intel SSE instructions. Listing 1 shows a C implementation of the traversal loop previously described in Algorithm 1 in lines 12-24. *gResult* is an accumulated hit mask that is used to determine which of the 4 child boxes have been hit. *numActive* is a 4 length array of integers that keeps track of how many ray IDs are on each of the *activeRayStacks*. *numRays* is the number of active rays within the current node and is used to calculate *rayPtr* which points to the first active ray ID on the stack. This is done so that rays are box tested and pushed on the active stack while overwriting the previous values so there is no need for copying of ray IDs to temporary lists. *BoxIntersect4* performs the multiple box test using Kay and Kajiya's popular slab test [Kay and Kajiya 1986]. Note that the ray's length

Algorithm 1 MVBH RS Traversal Pseudocode

```
 1: INPUT/OUTPUT: rays[numRays]
 2: STACKS: activeRayStacks[SIMD_WIDTH], taskStack
 3:
 4: taskStack.push(root, 0, numRays);
 5: for rayID=0 to numRays-1 do
 6:     activeRayStacks[0].push(rayID);
 7: end for
 8: while not taskStack.empty() do
 9:     task = taskStack.pop();
10:     node = task.node;
11:     if not node.isLeaf() then
12:         list = activeRayStacks[task.index].pop(task.numRays);
13:         length4 = float4_zero;
14:         numActive4 = int4_zero;
15:         for each rayID in list do
16:             (result4, tHit4) = BoxIntersect4(rays[rayID], node);
17:             length4 = length4 + AND(result4, tHit4);
18:             numActive4 = numActive4 + AND(result4, one);
19:             for i=0 to (SIMD_WIDTH - 1) do
20:                 if result4[i] then
21:                     activeRayStacks[i].push(rayID);
22:                 end if
23:             end for
24:         end for
25:         if a ray hit a box then
26:             ids[SIMD_WIDTH] = 0..SIMD_WIDTH-1;
27:             Sort4(length4, ids);
28:             for i=0 to SIMD_WIDTH - 1 do
29:                 taskStack.push(node.child[ids[i]], ids[i], numAc-
                    tive4[ids[i]);
30:             end for
31:         end if
32:     else
33:         list = activeRayStacks[task.index].pop(task.numRays);
34:         for each rayIDs[SIMD_WIDTH] in list do
35:             ray4.loadSoA(rayIDs, rays);
36:             for each poly in node.polygons do
37:                 intersect4(ray4, poly);
38:             end for
39:             ray4.storeAoS(rayIDs, rays);
40:         end for
41:     end if
42: end while
```

is passed into the slab test which represents its closest intersection which is important for culling since valid intersections will only exist between *rays[rayId].tmin* and *rays[rayId].length*. Another point to note is that the origin and inverse direction of each ray is stored in replicated, SoA format to avoid several _mm_set_ps1 instructions prior to intersection. Doing this resulted in an overall 5% speedup at the cost of additional memory usage.

After the hit lengths have been accumulated, the ray IDs are pushed onto the active stacks which can be replaced with a scatter operation on future architectures. Next, *result4* is used to increment the active stack counts. *gResult* is later used with a *_mm_movemask_ps* operation to determine if any of the child nodes have to be placed on the *taskStack*. In an additional note, if shadow rays are traversed separately, they can be filtered out early after their first intersection which is what is done in this test system.

Listing 1: *C implementation of traversal loop*

```
__m128 gResult = zero; length4 = zero;
numActive[task->index] -= task->numRays;
int rayPtr = numActive[task->index];
numActive_mm = _mm_load_si128((__m128i *) numActive);
for(int i=0; i<task->numRays; i++){
    unsigned short rayId = activeRayStacks[task->index][rayPtr + i].rayid;
    __m128 result4 = BoxIntersect4(rays[rayId].origin4,
                                   rays[rayId].invDir4, rays[rayId].tmin,
                                   rays[rayId].length, node->box4, tHit4);
    length4 = _mm_add_ps(length4, _mm_and_ps(tHit4, result4));
    for(int u=0; u<SIMD_WIDTH; u++)
        activeRayStacks[u][numActive[u]].rayid = rayId;

    numActive_mm = _mm_add_epi32(_mm_and_si128(*((__m128i *)&result4),
          one_int), numActive_mm);
    _mm_store_si128((__m128i *) numActive, numActive_mm);
    gResult = _mm_or_ps(result4, gResult);
}
```

5 Results

All tests were performed on an Intel Core 2 2.4 GHz Q6600 on a *single* core except for the "Better Hardware" section which was run on new Intel CPUs on multiple cores. The reason for choosing a Q6600 for most results as it's more comparable to the 2.4 GHz Opteron used for the results by Boulos et al. [2007]. Unless otherwise specified, a ray group of 2^8 rays is used for MBVH RS. All tests were generated using a distribution ray tracer in the similar spirit to tests performed in [Boulos et al. 2007] to generate incoherent workloads. The distribution tracer uses sudoku tile samples [Boulos et al. 2006] and traces 16 ray paths per pixel and renders depth of field, soft shadows and glossy reflections. Benchmark scenes are shown in Figure 1. On the far left is the BMW scene which features a relatively high polygon car which is fully reflective and sits on a reflective surface. The middle scene is the standard Conference Room scene which is fully reflective and is used to compare performance against other traversal methods. The last scene is the Ferrari scene which is a medium polygon car model sitting on a reflective, subdivided plane with perturbed vertices from a heightmap used to simulate water. No ray attenuation was used in any test and the number of bounces refers to the maximum bounce depth allowed. For all tests, each surface in the scenes were forced to being reflective.

5.1 Overall Scene Performance

Figure 2 shows the performance of each scene in millions of rays traced per second for several glossy bounces. For the BMW and the Ferrari scenes, the performance goes up very slightly or stays level since many bounced rays are moving to parts of the scene with much less geometric complexity. However, since the Conference scene has similar complexity througout the scene, the performance drops slightly as the number of bounces increases due to a drop in the coherency of the rays. This can be attributed to more node fetches which yield a higher memory bandwidth usage and more primitive tests due to a drop in SIMD efficiency. Because of the balanced geometric complexity in the Conference scene, it is a good candidate for a coherency test as done also by Reshetov [2006] and Boulos et al. [2007].

Results for the glossy reflection coherency test is shown in Figure 3. Even at seven bounces, over a million rays per second (rps) per core is achieved. Over the course of seven bounces, the overall ray performance has dropped by 1M. The next section will analyze the architecture-independent data to give better insight to the performance characteristics of the algorithm.

Figure 2: *performance in million of rays per second for benchmark scenes with varying bounces*

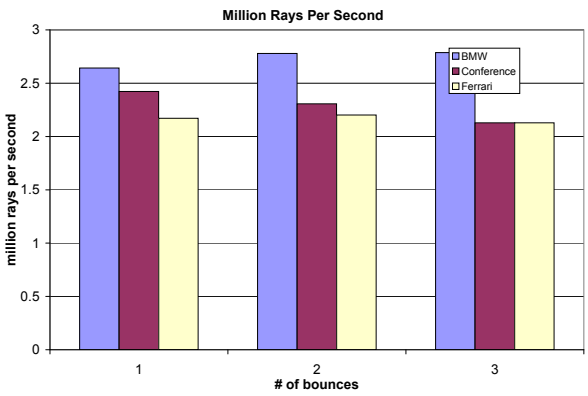
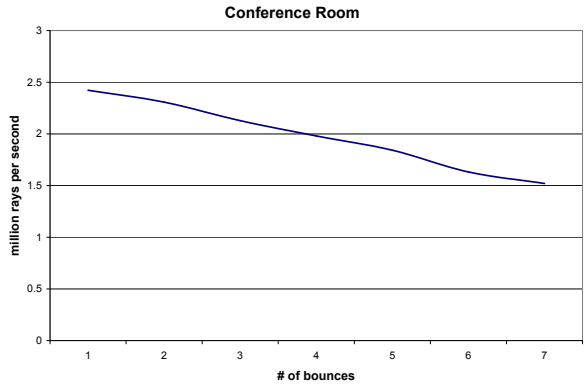


Figure 3: *performance in million of rays per second for Conference with varying bounces*

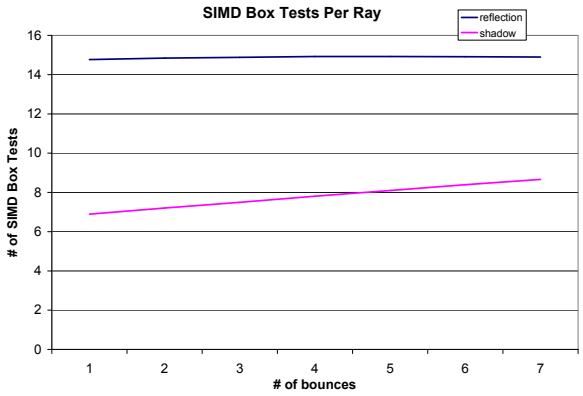


5.2 Performance Counters

An important performance metric is the average number of SIMD box tests performed per ray. Figure 4 shows this metric for the Conference scene for varying number of glossy reflection bounces for both reflection and shadow rays. Like traditional MBVH, as the rays become more incoherent, the number of box tests stays relatively unchanged since each ray is traced independently against multiple boxes. For primary rays, the number of box tests for MBVH RS is an order of magnitude more than a packet based approach but less for shadow and reflection rays as shown in [Boulos et al. 2007] where incoherency plays a more significant factor.

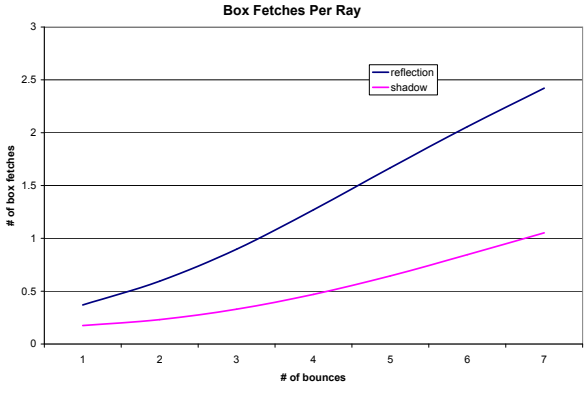
Another important metric to measure memory bandwidth is the number of box fetches per ray as shown in Figure 5. Unlike MBVH,

Figure 4: *Number of box tests per ray for Conference with varying glossy bounces*



which fetches a new box for each new box test, the cost of each box fetched is amortized over the number of active rays within the current node. This metric is highly dependent on coherency as more box fetches are performed for rays that do not follow the same path through the scene. Even at seven bounces, the number of box fetches per ray stays below three for both reflection and shadow rays which is much less than that of an MBVH traversal which can reach up to 15 in this test system as a box fetch must be done per box test.

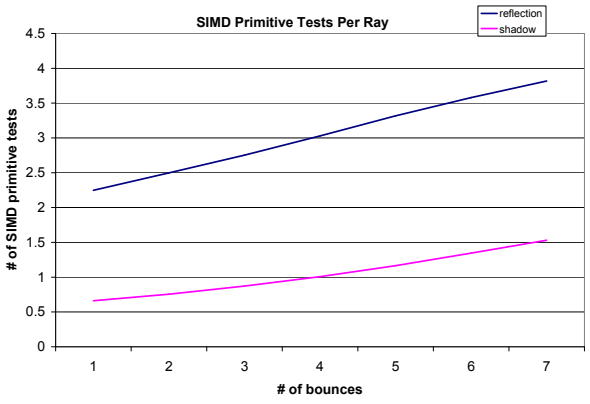
Figure 5: *Number of box fetches per ray for Conference with varying glossy bounces*



The final metric used for profiling is the average number of SIMD primitive tests per ray as shown in Figure 6. For primary rays, MBVH RS performs very similar to packets since each active ray in both methods must be tested against each primitive. This can be seen on the graph as around two primitive tests per ray are per-

formed much like packets for primary rays [Boulos et al. 2007]. As coherency drops at higher bounces, MBVH RS performs less primitive tests per ray versus packets since inactive rays are filtered out of the group at each traversal step. This metric is also dependent on coherency as SIMD primitive tests become less efficient if less full SIMD ray packets cannot be found within the leaf nodes. This problem is made worse for a larger SIMD width as more active rays are needed within the leaf nodes to take full advantage of the SIMD units. This problem is further analyzed in the following section on SIMD efficiency.

Figure 6: *Number of primitive tests per ray for Conference with varying glossy bounces*



5.3 SIMD Efficiency

Figure 7 shows the SIMD efficiency for primitive tests on a 4-wide architecture for the 3 bounce Conference test. For this architecture, the SIMD units are being highly utilized as enough active rays are found within the leaf nodes to build full SIMD ray packets. As the ray group size increases, the efficiency slightly increases. However, for a 16-wide architecture like Larrabee [Seiler et al. 2008], the efficiency becomes much lower as there is not enough coherency within the rays as shown in Figure 8.

At 2^{12} group size, the efficiency is moderately high at around 60% for the Conference scene but still not comparable to 4-wide SIMD. To bring the efficiency higher, the SAH tree cost function can be changed to allow for less node splits and a shallower tree. By doing this, larger leaf nodes would be generated with more primitives and thus offloading traversal work to the primitive intersection tests. Another method to increase efficiency would be to revisit multi-polygons again as done in traditional MBVH. Finally, the most obvious solution to increase efficiency would be to increase the ray group size so that there are more rays traced together and a higher probability for coherent sub groups. However, the disadvantage to this approach is that a larger memory requirement is needed to store ray IDs yielding a decrease in cache coherency when dealing with a large number of ray data.

5.4 Better Hardware

The distribution tracer was also run on a 4 socket 2.66 GHz Intel Dunnington machine which is a 24 core machine. The machine

Figure 7: *SIMD efficiency on a 4-wide architecture for primitive tests*

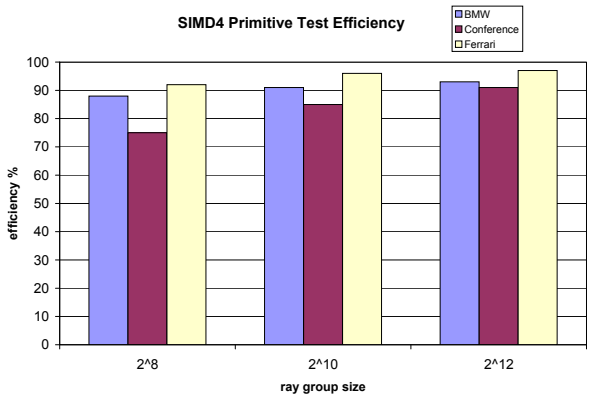
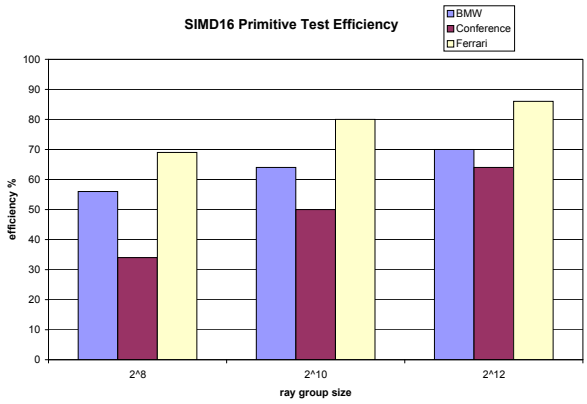


Figure 8: *SIMD efficiency on a 16-wide architecture for primitive tests*



yielded an extra 4% increase in overall performance per core than the Q6600 results. At a 512×512 resolution, each scene ran at around 4 frames per second for a single glossy bounce with all effects turned on at 16 ray paths per pixel. Table 1 shows the full results for each scene for varying glossy bounces with soft shadows, depth of field and glossy reflections. The number of rays per frame is shown the last row of the table.

The next test was done on 2 socket 3.2 GHz Intel Core i7 with 4 cores each (8 cores in total) with the same rendering setup as in Table 1. With hyperthreading, each core was able to perform at 4.0M rps for the Conference scene at 4 bounces with the overall frame rates for each scene shown in Table 2.

Even with 24 cores on the Dunnington, the distribution renderer

Table 1: *Frame rate for a* 512×512*, at 16 ray paths per pixel, rendering using a distribution tracer for varying bounces on a 24 core Intel Dunnington*

	1	2	3	4
BMW	5.3	3.3	2.4	1.8
Conference	4.8	2.7	1.8	1.3
Ferrari	4.3	2.6	1.8	1.3
num rays	12.6M	21M	29M	38M

Table 2: *Frame rate for a* 512×512*, at 16 ray paths per pixel, rendering using a distribution tracer for varying bounces on a 8 core Intel Core i7 with hyperthreading*

	1	2	3	4
BMW	3.4	2.1	1.5	1.1
Conference	3.1	1.8	1.2	0.9
Ferrari	2.7	1.7	1.2	0.9
num rays	12.6M	21M	29M	38M

performance is far from "real-time" for game applications due to sheer number of rays required to reach moderate convergence for fuzzy effects. The hope is that this algorithm will be able to scale well and provide better performance for upcoming CPU architectures supporting a wider SIMD and scatter/gather instructions with many cores.

For offline production renderers, this gives a significant boost in performance for ray tracing. However, to use this algorithm, the software architecture must changed to allow the batching of rays before traversal into large groups. Like packets, the traversal of these groups of rays will benefit from coherence. However, unlike packets, MBVH RS uses larger groups of rays (streams) to extract hidden coherence due to the filtering during traversal. This is helpful in the design of a renderer since a few incoherent rays will not negatively impact an otherwise coherent group of rays as is the case when using packets.

6 Conclusion

A new technique called "Multi-BVH Ray Stream Tracing" was introduced which allows the fast traversal of many partially coherent and incoherent rays through the fast filtering of inactive rays using a 4-way BVH structure. The technique does not rely on the use of ray packets which allows for a simpler software implementation into a production renderer where shaders are independent of traversal. The algorithm has also been designed to scale for CPU architectures with wider SIMD. MBVH RS extracts hidden coherency from groups of rays which is used to reduce memory bandwidth due to box fetches and to increase SIMD efficiency during primitive tests. However, when no coherency exists, MBVH RS keeps a high SIMD utilization for box tests.

Results show high performance for the Conference scene with multiple glossy bounces. Important traversal metrics were analyzed for incoherent rays which show lower number of primitive and box tests over packets and lower number of box fetches over MBVH. Different ray group sizes were also investigated and how SIMD primitive test efficiency was affected. For larger group sizes, a higher intersection efficiency was obtained but the overall performance suffered due to less cache coherency. An optimal ray group size of 2^8 was found and used for all tests.

The test renderer was also run on a 24 core Intel Dunnington computer which yielded increase performance per core and close to linear speed up with increasing core count. Overall frame rate of the distribution ray tracer was not fast enough for interactive game applications due to the sheer number of rays required for convergence of fuzzy effects. However, for incoherent rays (4 bounce), MBVH RS provides 4.0M rps per core on current generation Intel Core i7s on moderately complex scenes which is useful today for offline production renderers. Attached to this paper are sample videos of the BMW and Conference scenes running on the Dunnington computer with 1 level of reflection to give the reader a better look at the interactive distribution tracer used for testing.

Future work involves keeping traversal and primitive test efficiency high for a wider SIMD while also taking advantage of scatter/gather operations for writing out ray IDs to the active ray stacks and building temporary ray packets for primitive tests. Hopefully through these new architecture features, more cores and a faster primary visibility solution, a real time renderer can be built which captures many high fidelity effects requiring lots of incoherent rays for complex scenes with many different materials.

References

AMANATIDES, J. 1984. Ray tracing with cones. *SIGGRAPH Comput. Graph. 18*, 3, 129–135.

ARVO, J., AND KIRK, D. 1989. A survey of ray tracing acceleration techniques. *An introduction to ray tracing*, 201–262.

BENTHIN, C. 2006. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Computer Graphics Group, Saarland University.

BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., SHIRLEY, P., AND WALD, I. 2006. Interactive Distribution Ray Tracing. *Technical Report, SCI Institute, University of Utah, No UUSCI-2006-022*.

BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., SHIRLEY, P., AND WALD, I. 2007. Packet-based whitted and distribution ray tracing. In *GI '07: Proceedings of Graphics Interface 2007*, ACM, New York, NY, USA, 177–184.

BOULOS, S., WALD, I., AND BENTHIN, C. 2008. Adaptive Ray Packet Reordering. *IEEE/Eurographics Symposium on Interactive Ray Tracing 08*.

CHRISTENSEN, P. H., FONG, J., LAUR, D. M., AND BATALI, D. 2006. Ray Tracing for the Movie Cars. *IEEE Symposium on Interactive Ray Tracing 06*.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. *SIGGRAPH Comput. Graph. 18*, 3, 137–145.

DAMMERTZ, H., HANIKA, J., AND KELLER, A. 2008. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Rendering Techniques 2008, Proceedings of the Eurographics Symposium on Rendering, 2008*.

ERNST, M., AND GREINER, G. 2008. Multi Bounding Volume Hierarchies. *IEEE/Eurographics Symposium on Interactive Ray Tracing 08*.

GEORGIEV, I., AND SLUSALLEK, P. 2008. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *IEEE/Eurographics Symposium on Interactive Ray Tracing 2008*.

GRIBBLE, C. P., AND RAMANI, K. 2008. Coherent Ray Tracing via Stream Filtering. *IEEE/Eurographics Symposium on Interactive Ray Tracing 08*.

JENSEN, H. 2001. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, Wellesley, MA, USA.

KAJIYA, J. T. 1986. The rendering equation. *SIGGRAPH Comput. Graph. 20*, 4, 143–150.

KAY, T. L., AND KAJIYA, J. T. 1986. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 269–278.

MANSSON, E., MUNKBERG, J., AND AKENINE-MOLLER, T. 2007. Deep coherent ray tracing. *IEEE/Eurographics Symposium on Interactive Ray Tracing 07*.

OVERBECK, R., RAMAMOORTHI, R., AND MARK, W. R. 2008. Large Ray Packets for Real-time Whitted Ray Tracing. In *IEEE/Eurographics Symposium on Interactive Ray Tracing 2008*.

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

RESHETOV, A. 2006. Omnidirectional ray tracing traversal algorithm for kd-trees. *IEEE Symposium on Interactive Ray Tracing 06*, 57–60.

RESHETOV, A. 2007. Faster ray packets - triangle intersection through vertex culling. In *SIGGRAPH '07: ACM SIGGRAPH 2007 posters*, ACM, New York, NY, USA, 171.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, ACM, New York, NY, USA, 1–15.

VEACH, E., AND GUIBAS, L. J. 1997. Metropolis light transport. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 65–76.

WALD, I., BENTHIN, C., WAGNER, M., AND SLUSALLEK, P. 2001. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, Blackwell Publishers, Oxford, A. Chalmers and T.-M. Rhyne, Eds., vol. 20, 153–164. available at `http://graphics.cs.uni-sb.de/~wald/Publications`.

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

WALD, I., 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. *IEEE/Eurographics Symposium on Interactive Ray Tracing 08*.

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

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