OPTIMIZING QUERY TIME IN A BOUNDING VOLUME HIERARCHY

Benedikt Bitterli, Simon Kallweit

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

In this paper, we consider the problem of the fast computation of ray-triangle intersections. We target input sets with many triangles and many rays, both on the order of billions. We focus on one particular spatial acceleration structure, called the Bounding Volume Hierarchy (BVH), and examine the effects of memory architecture and vectorization on performance. We introduce a novel ray reordering scheme to improve temporal locality on difficult input sets. Our results have applications in engineering and computer graphics, especially in physically based rendering, where ray-triangle intersections pose a computational bottleneck.

1. INTRODUCTION

Many applications in graphics and engineering rely on the fast computation of intersections of rays with triangles. In movie production, video effects in particular have great use for fast triangle intersection, since ray-triangle intersection usually represents a bottleneck in the generation of realistic imagery.

For our purposes, a ray is a half-infinite line fully defined by its origin and direction. Intersections of a ray with a triangle are defined as the points that are part of both the ray and the triangle. The geometric setup is illustrated in Figure 1.

Brute force intersection of a ray with all triangles is typically infeasible, since the number of triangless can easily exceed hundreds of millions in practice. Instead, hierarchical spatial data structures are employed to quickly identify triangles that are likely to intersect a ray. In the context of spatial data structures, the term *query* is usually used to refer to a ray. This paper focuses on minimizing the *query time*, that is, the time required to answer a ray query. We assume that spatial data structures are queried by many rays once they are built, meaning that expensive preprocessing steps can be performed on the data structure to optimize query time.

In this paper, we will focus on one such data structure, the Bounding Volume Hierarchy (BVH), applied to triangle meshes in the context of physically based rendering. We

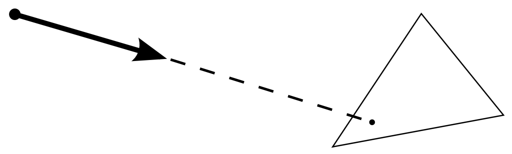

Fig. 1. *A ray, a triangle and the intersection point of the ray with the triangle*

will explore different ways of optimizing for memory architecture and vectorization and examine improvements proposed in literature. We measure performance for different input sets and workloads and compare our results to Intel's state-of-the-art Embree ray tracing framework [**?**]. Our final result is a BVH implementation combining several techniques to provide good performance on difficult input.

Related work. MacDonald et al. [**?**] introduce the surface area heuristic (SAH), which is a cost function that estimates the query cost of a kD tree or BVH. We employ a top-down, greedy optimization strategy of SAH in our framework to create the BVH tree structure. Yoon et al. [**?**] consider the impact of the memory layout of a BVH on the caching behaviour and query performance. Bender et al. [**?**] examine the efficiency of the van Emde Boas layout in the context of cache efficiency of B-trees. Eisenacher et al. [**?**] consider the effect of collecting and sorting incoherent rays to better handle out-of-core scenarios in rendering. We extend their work to in-core rendering and introduce a novel efficient ray ordering scheme to extract hidden coherence.

2. BACKGROUND

Hierarchical spatial data structures in the context of triangle intersection accelerate ray queries by *bounding* and *culling*. Internally, the data structure is represented by a rooted tree. Leaf nodes in the tree contain triangles, whereas internal nodes contain a bounding shape, e.g. a sphere or axis aligned

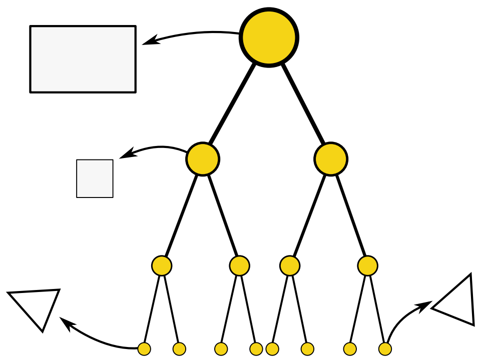

Fig. 2. *Illustration of a simple BVH data structure. Each internal node has two children, as well as a link to a bounding box enclosing them. Leaf nodes store a single link to a triangle*

bounding box. The tree is constrained such that for any internal node, the bounding shape must fully enclose all of its children. Figure 2 illustrates the data structure.

Given this setup, we can efficiently answer a ray query using recursive traversal. The procedure is as follows: Given a node of the tree, we first test whether the ray intersects the *bounding* shape of the node. If this is not the case, we can immediately discard (*cull*) the entire subtree rooted at the node. If the ray does intersect the node, we recursively repeat this procedure on its child nodes. When we reach a leaf node, we perform the ray-triangle intersection.

On average, such a data structure will perform on the order of $O(logN)$ intersections of the ray with triangles in the tree, where N is the number of triangles. Note however that an asymptotic bound better than $O(N)$ cannot be guaranteed due to the spatial structure of the problem - it is always possible to produce a degenerate set of triangles such that a worst-case ray will perform intersection tests with all of the triangles in the tree. This is not usually a problem in practice, however.

Coherent vs. Incoherent rays. In graphics, there is typically a distinction between *coherent* and *incoherent* ray distributions. Coherent rays are rays that share a common origin and point in similar directions, as shown in Figure 3(i). These rays arise in Whitted-style ray tracers [**?**]. Coherent ray queries tend to perform better in tree structures since it is likely that two successive ray queries will traverse a similar path through the tree, meaning that there is good temporal locality between queries. Coherent work loads also allow for efficient vectorization by bundling multiple rays into ray packets and traversing them simultaneously [**?**].

Incoherent ray distributions arise in distributed ray tracing, e.g. Monte Carlo path tracing, which is employed in physically based rendering algorithms to achieve realistic images; an example distribution is shown in Figure 3(ii).

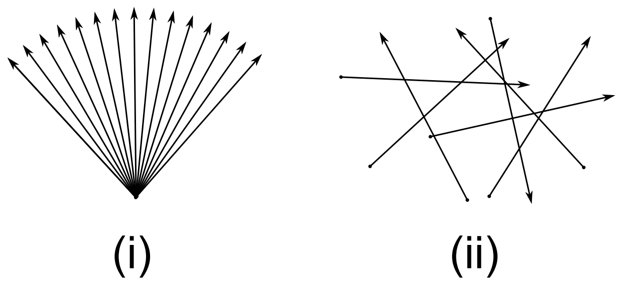

Fig. 3. *Coherent (i) and incoherent (ii) input sets*

In incoherent work loads, successive ray queries generally do not share similar origin or direction. This can make it challenging to optimize for memory architecture or vectorization, since there is poor temporal locality or shared computation between successive rays.

Cost Analysis. The performance of BVHs strongly depends on the triangle distribution and the ray queries. For our cost function, we choose to measure the number of bounding shape intersections as well as the number of triangle intersections. Since we know the number of FLOPS required for each operation as well as the memory size of bounding shapes and triangles, we can approximately estimate the efficiency of the implementation due to bandwidth and computation.

3. PROPOSED METHOD

In this section, we will describe our optimization process and the resulting implementations. Initially, we started by naively implementing the BVH data structure and validating it against unaccelerated intersection code. We then implemented successively more optimized versions based on observation, measurement and literature. Ultimately we produced 11 different BVH implementations and introduced a novel strategy to extracting hidden ray coherence. In the following subsections, we will only present a subset of the implemented versions for brevity.

3.1. Baseline implementation

For our baseline measurements, we implemented a naive BVH that does not employ low-level optimizations. In this implementation, each node consists of a bounding box, two pointers to its two children and a pointer to a triangle. For internal/leaf nodes, the triangle/child pointer is set to null. Each node is allocated separately on the heap.

Although the naive BVH already results in several magnitudes of speedup compared to not using an acceleration structure at all, it has obvious issues: Allocating each node separately incurs a large overhead on the heap as well as memory segmentation. Also, the node structure is much

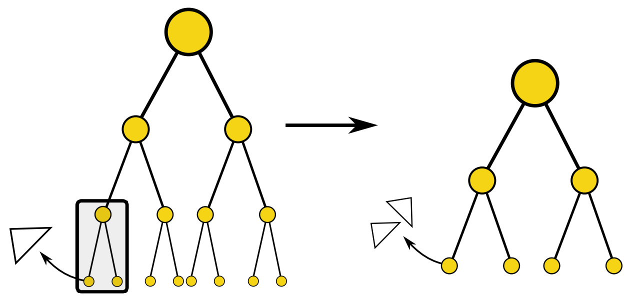

Fig. 4. *Tree modification for vectorization of ray-triangle intersection*

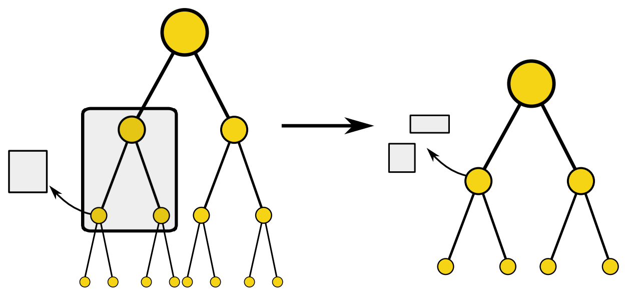

Fig. 5. *Tree modification for vectorization of ray-bounding box intersection*

larger than it needs to be, which has implications on memory consumption and bandwidth.

3.2. Optimizing bandwidth

Our first optimization efforts concentrated on reducing memory segmentation and the size of the data. Rather than allocating each node individually on the heap, we instead store the tree nodes in a large, contiguous 1D array. This already improves performance due to less memory segmentation, but also allows for more optimization. For example, we can now use relative addressing, allowing us to replace the two 8 byte child pointers with two 4 byte indices. Also, since we have control over how the nodes are laid out in the array, we can enforce that the two children of an internal node are always stored at two successive slots in the array. This way, only the relative address of the first child needs to be stored, since the second child is always in the consecutive slot. Additionally, since the triangles are also stored in a flat array, we can also convert the triangle pointer to a 4 byte index.

We note that a node utilizes either the child pointer or the triangle pointer, but never both, meaning that we can merge both the child and triangle pointer into one 4 byte field and use a single bit to determine whether the node is a leaf node or an internal node.

This reduces the size of the node structure from 40 bytes to 28 bytes, a significant reduction.

3.3. Exploiting SIMD

The default binary tree structure does not allow for easy application of SIMD instructions. This is because leaf nodes only contain a single triangle, and single triangle-ray intersections can only be poorly vectorized. To overcome this issue, we introduce a tree modification that allows for better vectorization, illustrated in Figure 4. In a first step, subtrees containing four or less leaf nodes are identified. In a second step, these subtrees are collapsed into a single leaf node containing up to four triangles. The triangle data is then stored in four wide SIMD vectors, allowing for efficient vectorization by intersecting a ray with up to four triangles in parallel.

Vectorizing triangle intersections already leads to a considerable increase in performance. However, there is still more SIMD potential, in particular during the ray-bounding box intersections. Since only a single bounding box is stored in internal nodes, direct vectorization only leads to a marginal increase in performance. However, we can apply the same tree modification rules as for the leaf nodes, as illustrated in Figure 5: First we identify subtrees with one parent and two children, then we collapse each subtree into a single node. Internal nodes now contain two bounding boxes, which can be efficiently vectorized. Due to the large number of box intersections during traversal, vectorization leads to an impressive increase in performance.

3.4. Reducing TLB Pressure

One issue which was glossed over in Section 3.2 is the order in which the tree nodes are stored in the 1D array. In practice, nodes are commonly arranged according to their depth-first or breadth-first ordering. Since BVHs can get quite large, child nodes can end up being stored far away in memory relative to their parents. In practice, this could lead to frequent TLB misses when traversing through the tree, as jumps across pages can occur at each traversal step. To reduce the likelihood of this occurring, we implemented a special tree layout from literature, the van Emde Boas ordering[**?**], which is a cache-oblivious layout designed to keep certain subtrees close together in memory. The ordering is recursively defined as follows:

The van Emde Boas ordering of a single node is the node itself.

The van Emde Boas ordering of a tree T of depth d_T is the van Emde Boas ordering of the top subtree ending at depth $\lceil \frac{d_T}{2} \rceil$ followed by the van Emde Boas ordering of all child subtrees rooted at depth $\lceil \frac{d_T}{2} \rceil + 1$.

Informally, this guarantees that any subtree is likely to be stored in a contiguous memory segment; in other words, the traversal algorithm is likely to work in a locally contiguous memory segment for many traversal steps before making a large jump through memory.

Although this should increase performance in theory, in practice, no performance improvement can be observed. It appears that TLB misses do not play a significant role in the traversal performance.

3.5. Exploiting hidden coherence

One of the observations we made during performance measurements was that coherent input rays were consistently faster than incoherent input rays. This makes sense, since successive ray queries in coherent input sets are more likely to traverse similar nodes of the tree, leading to good temporal locality. Unfortunately, incoherent rays are far more frequent and more important in practice than coherent rays, meaning that exploiting temporal locality is not an easy task.

After further investigation and measurements, we found that, while it is true that successive rays in the incoherent input sets may traverse very different parts of the tree, it is not true that there are *no* two rays in the incoherent input rays that visit the same nodes during traversal. In fact, due to the large size of the input, it is very likely that there are groups of rays that visit the same nodes during traversal; however, they do not usually occur close together in the input set, meaning that they are traced far apart temporally.

Our intuition was that, if there was a way to find these groups of rays that traverse similar nodes and trace them successively, we should be able to achieve much improved temporal locality and similar performance to the coherent input sets. We term this property *hidden coherence*, since the incoherent input sets contain coherent subsets that are not immediately apparent.

To achieve this, we first have to define a grouping heuristic to extract this hidden coherence. It is easy to see that for two rays to traverse similar nodes in the tree, they must originate at points in space close together and point in very similar directions. Therefore, we define our grouping heuristic based on the spatial and angular similarity of rays.

Determining the similarity of rays and grouping them in a global manner turned out to be a complex problem, since it is not easy to find an efficient, effective and unique ordering. Ultimately we came up with a hierarchical reordering scheme that recursively splits space and directions into subregions. Figure 6(i) illustrates this idea for the ray origins: In this 2D example, we first split space into four subregions of equal size. Rays originating in the same subregion are considered to be more similar to each other than to rays of other subregions, and thus each subregion forms a group. This is obviously a very coarse heuristic, and thus we split the subregions again and again in a recursive manner until each subregion only contains a single ray. This is illustrated in Figure 6(ii). At each level of the recursive split, we consider rays in the same subregion to be more similar to each

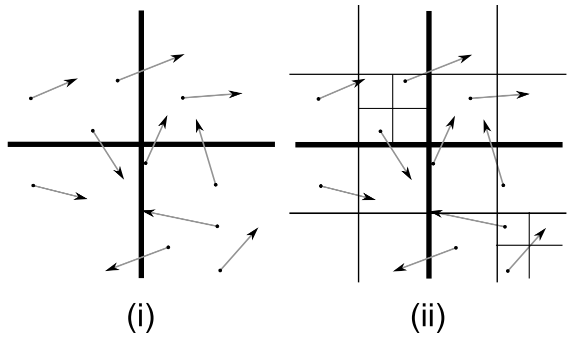

Fig. 6. *Recursive spatial subdivision for a group of 2D rays*

other than to rays of other subregions. This gives rise to a natural, recursive definition of the query order of the rays: The query order of a single ray is the ray itself. The query order of a region split into four subregion is the query order of the rays in the top-left subregion, followed by the query order of the rays in the top-right subregion and so forth. This means that if we pick any subregion at any level of the recursion, the rays originating within that subregion will be traced successively.

As it turns out, we can perform a very similar grouping heuristic for the directions. This is because we can interpret the unit length direction vectors as points on a sphere, and we can apply the same hierarchical subdivision as we did for the ray origins. In practice, we interleave steps of spatial subdivision with steps of directional subdivision, to give equal weight to origin and direction.

Although this ordering scheme is very effective at extracting hidden coherence, its straightforward implementation as a hierarchical reordering algorithm is unfortunately very slow, and the benefits of improved temporal locality is outweighed by the expensive reordering step. To remedy this issue, we instead restated our reordering scheme in terms of a sorting problem on integer keys, which can be implemented efficiently.

We generate a single, 32 bit integer key for each ray that incorporates our ordering scheme, which is illustrated in Figure 7. At each subdivision step, we label the new subregions with a two-bit string that describes which of the four new subregions the ray origin lands in. The concatenation of those two-bit strings, starting from the largest to the smallest subregion, then results in an integer sorting key. If we sort the rays by their keys in increasing order, then rays that belong to the same subregion at any level will land at successive slots in the sorted array. We can easily incorporate directions as well, by interleaving bit strings from directional subdivision with bit strings from the spatial subdivision.

In practice, we can generate the integer keys extremely efficiently by exploiting the internal representation of the

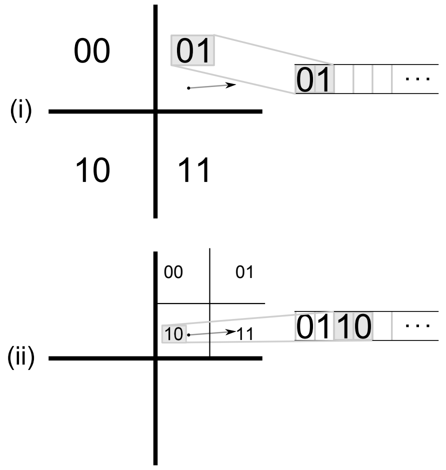

Fig. 7. *Recursive sort key generation of a single ray for similarity ordering*

IEEE 754 single precision floating point; see code for details. To efficiently sort the rays, we employ a six digit radix sort with five passes.

Note that the description of our algorithm focused on a 2D domain for simplicity. However, it easily extends to 3D by using eight subregions instead of four and three-bit strings instead of two-bit strings.

4. EXPERIMENTAL RESULTS

To measure the performance of our algorithms, we use a combination of hardware performance counters and code instrumentation. The following section describes our experiments in detail and shows the relevant results.

Experimental setup. We ran all experiments on a recent Intel-based platform shown in Table 1. All of our code is compiled with the GCC 4.8.1 compiler, using standard optimization flags (e.g. `-O3`). All experiments are executed in a single-threaded environment to simplify the interpretability of the results. As the BVH data structure is inherently *read-only*, running queries in parallel is trivial and commonly done in a rendering application.

Benchmarking framework. We have implemented a benchmarking framework specifically targeted to measure the performance of the implemented algorithms. For each combination of an algorithm with a test set, the framework runs two passes. In the first pass, *rdtsc* and hardware performance counters [**?**] are used to measure the runtime and additional performance indicators. For these measurements to be accurate, the framework runs the same test multiple times and generates statistics on the measured values (mean and variance). In the second pass, the test is repeated with enabled code instrumentation to gather additional indicators. Note that we used C++ templating to compile each algorithm twice, with and without code instrumentation calls, to preclude any overhead in the runtime measurement.

CPU	Intel Core i7-4770K 3.50 GHz
Microarch.	Haswell
Frequency	3.50 GHz
Cores	8
RAM	16 GB
L1 Cache	32 kB, 8-way
L2 Cache	256 kB, 8-way
L3 Cache	8 MB, 16-way
Cache Lines	64 Bytes
OS	Ubuntu Linux 64-bit (3.11.0 Kernel)

Table 1. Details of the platform used for the experiments

Test sets. We use freely available 3d meshes, *Sponza* (66447 triangles) and *SanMiguel* (7838629 triangles), and created two sets of rays (coherent/incoherent) for each scene using a Monte Carlo Path Tracer. This results in a total of 4 test sets: *Sponza-Coherent*, *Sponza-Incoherent*, *SanMiguel-Coherent* and *SanMiguel-Incoherent*.

Performance counters. Generating valid data from performance counters has proved to be extremely difficult in our application. Our measurements on many different indicators such as cache hit ratios, memory traffic, TLB misses and others have mostly shown too much variance to be useful for any kind of explanation on efficiency. This effectively left us with nothing else than the *runtime* for comparing the performance of our algorithms.

Code instrumentation. Using code instrumentation, we were able to get some insight into our algorithms. Among other values, we compute the size of the BVH in memory and the number of box and triangle intersections.

Results. Figure 8 and 9 show the speedups achieved with the different BVH implementations compared to the naive implementation. Note that not all of our implementations are shown; measurements showed that most of our optimizations targeting data size and scalar code did not significantly improve performance, and these code variations are omitted from the graph for brevity.

Triangle SIMD. Code instrumentation shows that optimizing triangle intersections using SIMD increases the effective number of computed intersections roughly by a factor of 2. This is due to less efficient culling, as leaf nodes now contain 4 triangles instead of one. Still, using SIMD for triangle intersections increases performance in all test cases.

Full SIMD. Code instrumentation shows that the number of box intersections is roughly 15-20 times higher than the number of triangle intersections. This clearly indicates the potential for optimizing box intersections with SIMD.

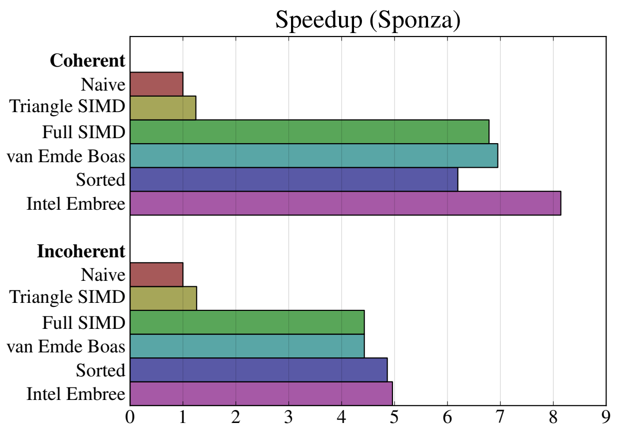

Fig. 8. Comparison of total speedup for SIMD optimized algorithms using the *Sponza* test sets

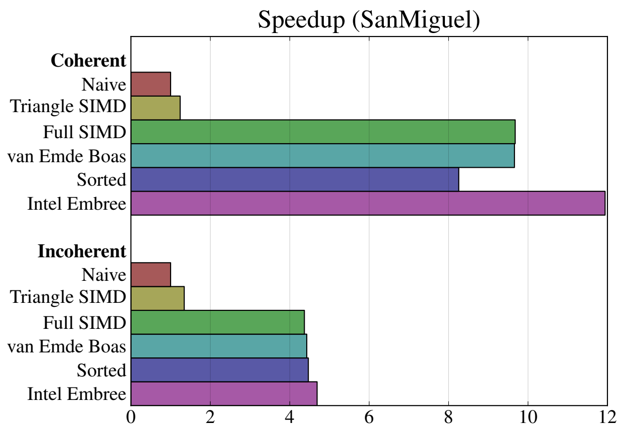

Fig. 9. Comparison of total speedup for SIMD optimized algorithms using the *SanMiguel* test sets

Indeed, we measure a dramatic increase in performance when using SIMD for both box and triangle intersections in all test cases.

van Emde Boas. Minimizing the number of TLB misses should theoretically lead to a performance increase. However, our experiments show that there is no significant speedup. Unfortunately, measuring the number of TLB misses using performance counters did not result in further clarification, as the measurements were not usable due to excessive variance.

Sorted. Extracting hidden coherency by pre-sorting the query rays leads to a measurable performance increase in the two incoherent test cases. For the coherent test cases, sorting naturally decreases performance, as the sorting step cannot be amortized by the improved query time.

5. CONCLUSIONS

We have studied various ways for optimizing the query time in a BVH and implemented a series of increasingly sophisticated algorithms. With our benchmarking framework we have shown the gradual performance increases obtained with each optimization step. We have reached similar performance as a state-of-the-art implementation from Intel. Using a novel approach for ray reordering, we have further increased performance when using incoherent rays. With some additional work, we believe our implementation could even beat the current state-of-the-art in some specific scenarios.

The countless hours we spent measuring performance with help of hardware performance counters did not prove to be very fruitful. We hoped to get evidence on how our algorithms improve bandwidth usage but unfortunately were not able to produce significant results. Using code instrumentation however, we were able to get a few additional insights and explain some of our benchmarking results.