

Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays*

H. Dammertz¹ and J. Hanika¹ and A. Keller²

¹{holger.dammertz, johannes.hanika}@uni-ulm.de, Ulm University, James-Franck-Ring, 89081 Ulm

²alex@mental.com, mental images GmbH, Fasanenstrasse 81, 10623 Berlin, Germany



Abstract

Photorealistic image synthesis is a computationally demanding task that relies on ray tracing for the evaluation of integrals. Rendering time is dominated by tracing long paths that are very incoherent by construction. We therefore investigate the use of SIMD instructions to accelerate incoherent rays. SIMD is used in the hierarchy construction, the tree traversal and the leaf intersection. This is achieved by increasing the arity of acceleration structures, which also reduces memory requirements. We show that the resulting hierarchies can be built quickly and are smaller than acceleration structures known so far while at the same time outperforming them for incoherent rays. Our new acceleration structure speeds up ray tracing by a factor of 1.6 to 2.0 compared to a highly optimized bounding interval hierarchy implementation, and 1.3 to 1.6 compared to an efficient kd -tree. At the same time, the memory requirements are reduced by 10–50%. Additionally we show how a caching mechanism in conjunction with this memory efficient hierarchy can be used to speed up shadow rays in a global illumination algorithm without increasing the memory footprint. This optimization decreased the number of traversal steps up to 50%.

1 Introduction and Previous Work

Improving ray tracing [Gla89, Shi00] performance has received thorough attention [Hav01, Smi98]. One of the most popular acceleration structures for fast ray tracing is the kd -tree [HB02, SSK07]. Bounding volume hierarchies (BVH) belong to the simplest and most efficient acceleration schemes for ray tracing [GS87, WK06, WBS07, Gei06]. Memory requirements [WK07] and memory latency [Wal07] have been recognized as bottlenecks and fast tree construction has been investigated [WK06, WH06, SSK07]. A lot of work has been spent to exploit SIMD instructions of modern processors for coherent packets of rays [WBWS01, Wal04, RSH05, Ben04, WBS07, Res07].

Algorithms that perform physically correct simulations of light transport [Vea97] tend to shoot rays as wide-spread as possible in order to increase efficiency, e.g. by employing quasi-Monte Carlo methods [KK02]. As a result, most rays that

account for global illumination effects are incoherent. Because of that, major parts of the simulation cannot benefit from tracing packets of rays. This has been recognized in professional rendering products [CFLB06, Sec. 6.4], too. They use SIMD instructions by intersecting multiple objects at a time. Compared to tracing packets this has the major disadvantages that tree traversal is not accelerated and that memory bandwidth is not reduced.

We address the bandwidth problem by reducing the memory footprint of the acceleration data structure. In contrast to reduced-precision methods [Mah05, HMB06], which work best when the overhead of transforming the packed data to world space is amortized over a bundle of rays, we increase the arity of the acceleration hierarchy, which saves memory for nodes and pointers. This is similar to a lightweight BVH [CSE06], but in the context of efficient SIMD processing and without the drawbacks of limited precision and median split tree construction.

We also show how to use the memory layout of the new acceleration structure to contain additional data used to transparently speed up coherent rays. This uses backtracking in a way related to [HB07] but without additional memory requirements and with entry points at every node, not in a sparse way on top of another acceleration structure.

Smits [Smi98] naturally recognizes the possibility of BVHs with higher arity (without regard to parallel processing), but does not indicate how to construct the acceleration structure in this case while still minimizing the cost function [GS87]. Also, the suggested data structure does not support sorted traversal, which we found to be crucial for good performance on current CPU architectures, especially for intersection rays. He also points out the opportunity of speeding up shadow rays using a cache. As this cache only stores one single primitive with the previously found intersection and is not hashed per light source, he states that the method only works well with large triangles and very coherent rays.

2 n -ary Bounding Volume Hierarchy

Bandwidth problems are an issue [Wal07] for efficient streaming processing. We achieve a small memory consumption by flattening the hierarchy and favoring larger leaves which also allow for efficient streaming intersection of the primitives. We use axis-aligned bounding boxes, stored as a minimum and a

*author's preprint of the paper submitted to EGSR 08

maximum corner point, for bounding volumes. A detailed description of the memory layout is given in Section 2.1.

As the SIMD width is $n = 4$ in most current processors, our new BVH implementation is called QBVH (Quad-BVH) throughout this paper, although the concepts are not limited to $n = 4$.

Figure 2 illustrates the new construction of a 4-ary QBVH tree, which is detailed in Section 2.2. Here the same split plane proposals as for the binary case are used but instead of creating nodes after each split the object sets are split again resulting in four disjoint object sets. Now, contrary to the common notion of bounding volume hierarchies, each node does not contain a single bounding volume. Each node directly contains the four bounding volumes of its children. So already the top level node of the tree contains four bounding boxes, which can be processed using SIMD.

Additionally, the four child pointers and the three axes of the proposed split planes are stored for each inner node. These split axes are used for an efficient, sorted traversal similar to binary BVHs as described in Section 2.3.

In Section 2.4, the paradigm of streaming processing and low memory consumption is applied to leaf intersection. This is achieved using a small cache storing the triangles in a SIMD-friendly form. This also allows to use more memory efficient, but slower, representations of the triangle data without a major speed impact. In Section 3.3 the vertex-index representation of triangles is compared to the direct 9 floats representation often used in real-time ray tracing. Even though better methods for memory reduction exist [LYM07], they are not as easily integrated into an existing rendering system. Cache-friendly padding of the node structure provides us with some additional memory which can be used to improve performance, we provide a way to exploit this to accelerate shadow rays in Section 4.

2.1 Memory Layout

In real-time rendering systems, the triangle data is often sorted directly during tree construction for more efficient memory access in the leaf intersection. However, this is not an option in production rendering systems, as other parts of the renderer may depend on the original order. For this reason, we choose to sort an additional index array instead.

In the implementation we chose the following data layout for the tree nodes:

```

struct SIMD_BVH_Node {
    float bbox[2*4*3];
    int child[4];
    int axis0, axis1, axis2;
    int fill;
};

```

¹The result is a large BVH node with a size of 128 bytes which perfectly matches the caches of modern computer hardware. The 4 bounding boxes are stored in structure-of-arrays (SoA) layout for direct processing in SIMD registers. Of course the axes could be packed into a single value but for cache alignment reasons the size of 128 bytes is favorable and packing this data would save only 10% of the total memory consumption. The integer `fill` is used to store additional data for optimizations later on. The tree data is kept in a linear array of memory locations, so the child pointer can be stored as integer indices instead of using platform-dependent pointers.

It is important to note that for a leaf no additional `SIMD_BVH_Node` is created. The leaf bounding box is already stored in the parent and the leaf data can be encoded directly into the corresponding `child` integer.

We use the sign of the child index to encode whether a node is a leaf or an inner node.

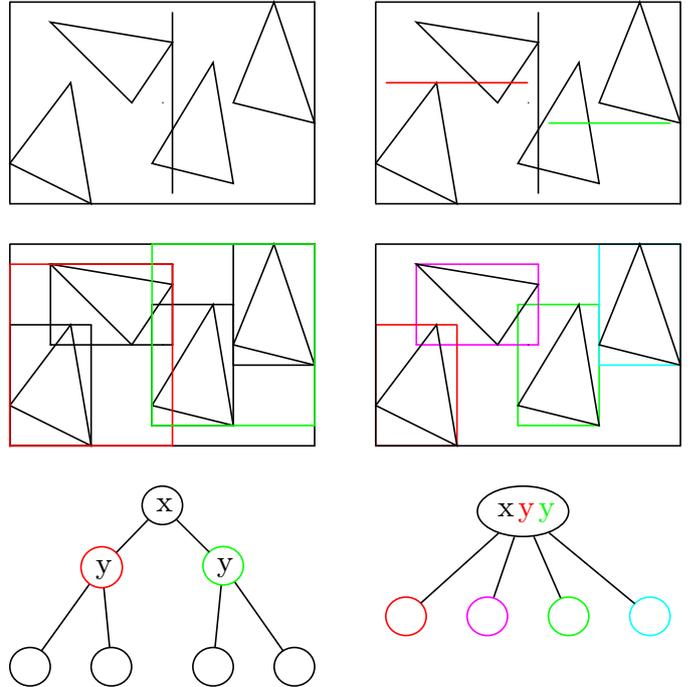


Figure 2: Construction of a binary BVH (left), compared to the construction of a 4-ary BVH (QBVH, right). Four children are created for each node. The top row indicates the split plane candidates common to both approaches.

There exist various encoding possibilities and it depends on the processor architecture and compiler which one is the most efficient. The most memory efficient is to directly encode the leaf information into the remaining 31 bits. Generally we chose 4 bits for the number of triangles in the leaf and the remaining 27 bits as the start index in the triangle array. Since the triangle intersection is done using SIMD, the number is a multiple of 4. So up to 64 triangles per leaf can be stored and up to 2^{27} triangles can be indexed. Empty leaves are encoded in a special value (`INT_MIN`) so the 4 bits can be used for the full 16×4 triangles. Note that when using bounding volume hierarchies the number of triangles per leaf is easily bounded because forcing another split guarantees a reduction in the number of triangles.

If the upper bound of 2^{27} is not acceptable for certain applications, a slightly less memory efficient version is created. The full 31 bits are used to index an additional data structure containing the leaf information. Another memory efficient encoding strategy is to store only the start value of the triangle reference array and to mark the last triangle with a negative index in this array.

2.2 Tree Construction by Flattening Binary Trees

The $n = 2^k$ -ary BVH tree construction can be seen as collapsing a classical binary tree. This is illustrated in Figure 2 for the case of $n = 4$. Each k -th level of the tree is kept and the rest discarded. This results in 2^k bounding volumes per node. Figure 3 illustrates how a QBVH is constructed, for $k = 2$, approximately halving the memory requirements.

This view allows to use the same construction principles used for binary trees to construct 2^k -ary trees. In our implementation we used recent results [SSK07, Wal07] to construct good quality trees with fast construction times. We use min-max binning with 8 bins to approximate the surface area heuristic

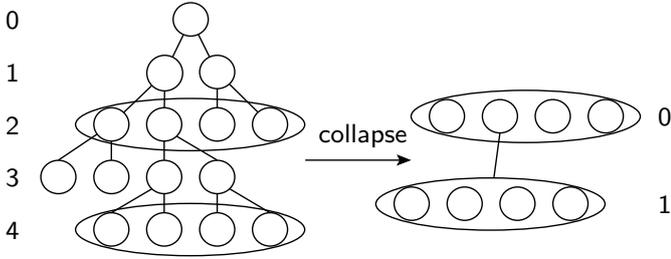


Figure 3: Collapsing a binary tree to a QBVH. Classical build methods are used to create a binary tree, which is then collapsed by leaving out intermediate levels.

(SAH). This implementation is vectorized by the compiler and efficiently uses SIMD instructions. As proposed in [SSK07], only a fraction of the primitives is used for the SAH approximation in higher levels of the tree. Further speedup is achieved by precomputing the bounding boxes of the triangles in a SIMD layout, overwriting the triangle data. This way, the implementation can benefit from SIMD operations during construction without using additional memory. After construction, the triangle data is reloaded from disk. The construction is additionally sped up by allowing more primitives per leaf, resulting in flatter trees. In all measurements, we create a leaf whenever the primitive count drops below 17. Using this implementation, it is possible to get very fast tree construction times that are even comparable to the BIH [WK06].

Figure 3 also illustrates that the resulting tree consumes only a fraction of the memory needed by the original tree, at the expense of larger nodes. But this fits modern streaming SIMD architectures well. A large chunk of aligned data can be loaded and worked on in parallel.

While using four boxes per node would already be sufficient to enable SIMD processing of the above data structure in the manner of [CFLB06], it is important to store along which planes (perpendicular to the canonical axes x , y , or z) the objects were partitioned. The plane indices of the binary hierarchy are stored as three integers `axis0`, `axis1`, and `axis2`, respectively. This allows exploiting the spatial order of the boxes for a more efficient pruning using the sign of the ray direction during tree traversal.

2.3 Tree Traversal

Given a ray, the stack-based traversal algorithm starts by simultaneously intersecting the ray with the four bounding boxes contained in the root `SIMD_BVH_Node` using SIMD instructions. The pointers of the children with a non-empty bounding box intersection are sorted and then pushed on the stack. The routine is repeated by popping the next element as long as there is one.

The ray is prepared prior to traversal by replicating the values for the maximum ray distance (`tfar`), the origin, and reciprocal of the direction across a SIMD register. Additionally the signs of the components of the ray direction are stored as integers to allow easy indexing of the near and far bounding box sides.

For the bounding box intersection we use a branch-free implementation of the slab test [WBMS05]. The 4 boxes are intersected simultaneously with the replicated ray. The return value of this function is a SIMD mask containing all ones when the respective box was intersected and zeroes otherwise.

In order to enable an early pruning of the traversal, the nodes are pushed on the stack according to the order determined by the ray direction. Using the split plane indices stored during

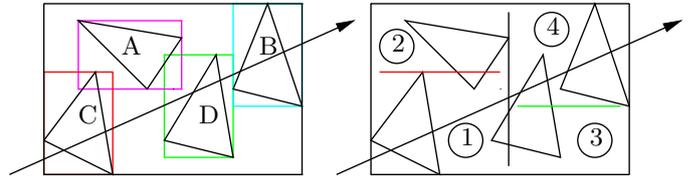


Figure 4: Sorted QBVH traversal. The dimension of the split plane proposals are kept to sort the child nodes using the ray direction.

tree construction, the sign of the respective component of the ray direction vector determines whether or not to swap the two corresponding children, as illustrated in Figure 4. This is similar to the order a kd -tree would define using the same split plane proposals.

Two implementations of this sorting turned out to be good choices. The most elegant variant is based on the SIMD sorting algorithms described in [FAN07], which works branch-free on SIMD registers by using masking and swizzling.

First the four child pointers are loaded into a SIMD register. Second, a SIMD mask is constructed for each of the three axes stored in the BVH node based on the sign of the ray direction of that component. Third this mask is used to select either the original value or a shuffled value resulting in a correctly ordered SIMD register. Finally the contents of the SIMD child register are pushed onto the stack in a scalar way. This is done by pushing a child pointer and then decrementing the stack top pointer by the corresponding entry of the reordered result mask from the bounding box intersection. For this purpose, this mask has to be ordered in the same way as the child pointers to push the correct children onto the stack. The decrement works because the result mask contains all ones in case of an intersection, which equals -1 as an integer, and all zeroes else.

Surprisingly, an implementation using cascaded branches based on the ray direction signs and split axes is equally fast on an Intel Core 2 processor in the general case and even slightly faster when only primary rays are traced. This may be due to the sophisticated branch prediction of this particular processor. The implementation is also much simpler than the SIMD sorting. The first decision based on `axis0` chooses which two of the four children should be intersected first. In Figure 4, this corresponds to the decision whether to visit (A, C) or (B, D) first. The second and third decision sort each of the child pairs again, resulting in (A, C) or (C, A) and (B, D) or (D, B) .

As the bounding boxes are already intersected while processing the parent, a late early-out is performed: if a triangle intersection is found, pruning of bounding boxes which are farther away can only be done one step later, during processing of the next node. Additionally storing the distance of the bounding box intersections with the child pointer on the stack solves this. While on current Intel processors and the Cell processor the traversal is faster without this “optimization”, it may pay off for wider SIMD operations as announced for example for the Larrabee processor.

In addition we like to note that our experiments report less bounding box intersections as compared to sorting the bounding boxes by their actual intersection distances with the ray. Table 1 gives a comparison of these two methods.

2.4 Triangle Intersection, SIMD Caching and Vertex-Index Data Structure

Similar to [Res07] we can even get away with much flatter hierarchies at the cost of leaves that contain more triangles. This



| scene | axes-based | distance-based |
|---------------|------------|----------------|
| power plant | 799.6 | 810.9 |
| sponza atrium | 339.2 | 347.6 |
| conference | 219.1 | 224.2 |

Table 1: Number of bounding box intersections (in millions) using axes-based sorting vs. distance to box entry point, measured using path tracing (512×512 with 16 samples per pixel). The images correspond to the scene and camera position, in this order.

further improves traversal speed and also enables the efficient use of streaming SIMD instructions to process multiple objects at once [CFLB06]. Compared to [Res07], we use single rays and avoid performance losses caused by masking.

If memory were of no concern, the triangle data in the leaves could be precomputed and stored in a SIMD-friendly layout. Since most of the time this is not practical, the original triangle layout is left untouched. For high performance ray tracing, Wald [Wal04] uses 48 bytes per triangle, Wächter [WK06] 36 bytes. The more compact vertex-index layout stores all vertices in an array and only three indices per triangle referencing these vertices. This is a widely used layout in 3D modeling applications. Usually this introduces performance penalties due to an indirection in memory access.

Our implementation supports the 36 bytes layout as well as the vertex-index form. We efficiently intersect four primitives at once using a branch-free SIMD version of the test described by Möller and Trumbore [MT97]. For this test, the triangle data has to be collected and swizzled to an SoA SIMD layout. This already works surprisingly fast, but to maximize performance, we use a small direct-mapped cache for each rendering thread to store the precomputed layout. This caching mechanism also allows the use of vertex-index representation for the triangle data without significant performance loss (see Table 6).

3 Results

We integrated our new BVH into two different rendering systems. The first one contains highly optimized implementations of a BIH [WK06] and a memory-bounded k d-tree and uses different unbiased rendering algorithms. This allows for a direct comparison of rendering performance on the same hardware and using the same rendering algorithm. The second system is based on instant radiosity [Kel97] and is used for the entry point caching comparison in Section 4. All measurements were performed using a single thread on an Intel Core 2 CPU at 2.33GHz. Note that all measurements were done in a full rendering system where sampling, shading and texture look-up are a significant part of the rendering time and are not sped up by faster ray tracing.

Section 3.1 and 3.2 use the first rendering system to compare the QBVH to the other acceleration structures. This system only supports the 36 byte triangle layout. Two versions of the QBVH are compared. The first one does not employ the SIMD triangle caching from Section 2.4 and is called ncQBVH. It uses leaves of 8 triangles. The second version uses a SIMD cache size

of 128 entries with a leaf size of 16 triangles.

In Section 3.3 the second renderer is used to evaluate the speed impact of using vertex-index based triangle representation.

3.1 Rendering Performance

To estimate the performance in a usual rendering setup we used three scenes of different complexity and left all parameters at their default settings. Figure 5 shows the scenes used. The conference room consists of 1M triangles, the interior scene is a complete house with 900K triangles and the power plant model has 12M triangles.



Figure 5: The three scenes (conference, interior, power plant) used for comparison of the QBVH against BIH and a k d-tree. The conference room and interior were rendered with a bi-directional path tracer using 8 bi-directional samples per pixel. The power plant was rendered with a path tracer and 32 paths per pixel.

Table 2 shows the measured rendering performance from invocation of the renderer to the final image (total time to image: TTI) for each acceleration structure (ACC). Additionally the time for constructing the acceleration structure (ACT), the memory consumed by the acceleration structure (MEM), and the time spent in the ray tracing kernel (RTT) is given. The QBVH uses very little memory while even outperforming the tree construction times of the BIH. Also note that the ray tracing kernels were all included in the same rendering system for a precise comparison. This made some optimizations in the tree construction impossible—optimizations which would allow building the QBVH for the plant in around 14 seconds. For detailed statistics about the tree quality and ray tracing behavior of the different kernels, see Table 3.

Table 4 gives an idea of how effective the use of SIMD instructions is, by comparing packet traversal to the QBVH in the setting for which it has not been designed: casting coherent primary rays only.

3.2 Memory Bound Performance

Memory consumption is an important part of any rendering system. All three acceleration structures allow to a-priori bound the memory used [WK07]. Table 5 shows the performance of each acceleration structure with bounded memory. The same scenes and rendering settings as in the previous section are used. The k d-tree has serious problems with tight memory bounds (note that a build of the conference room with 5 MB was impossible to render in acceptable time) while especially the QBVH reacts robustly to limiting the memory.

3.3 Vertex-Index Triangle Representation

The second renderer supports switching between the flat 36 byte triangle representation and a more memory efficient index-based representation. The indexed representation is more time consuming for ray tracing because an additional indirection has to be performed prior to triangle intersection. This overhead is

| Scene | ACC | inner nodes | leaves | tris/leaf | tris/ray | inner/ray | leaves/ray |
|--------------------|-----------|-------------|---------|-----------|----------|-----------|------------|
| Conf. (1M) | <i>kd</i> | 371018 | 384349 | 13.2 | 16.2 | 40.7 | 6.36 |
| | BIH | 1066634 | 557709 | 1.8 | 7.2 | 71.2 | 3.78 |
| | QBVH | 44704 | 133323 | 7.4 | 20.4 | 11.8 | 2.29 |
| Interior (0.9M) | <i>kd</i> | 429294 | 450133 | 9.0 | 20.6 | 50.0 | 4.89 |
| | BIH | 931393 | 510266 | 1.8 | 8.4 | 101.1 | 4.58 |
| | QBVH | 41569 | 123144 | 7.4 | 21.7 | 18.1 | 2.48 |
| Plant (12M) | <i>kd</i> | 4379594 | 4801842 | 14.9 | 29.0 | 59.1 | 4.28 |
| | BIH | 10923900 | 6717548 | 1.9 | 10.5 | 141.4 | 5.01 |
| | QBVH | 548908 | 1603848 | 7.9 | 32.6 | 24.5 | 2.93 |

Table 3: Detailed statistics for different acceleration kernels (ACC). This table shows, for the respective kernel and scene, in this order: the total number of inner nodes, the total number of non-empty leaf nodes, the average number of triangles per leaf node, the average number of intersected triangles per ray, the average number of traversed inner nodes per ray, and the average number of intersected leaf nodes per ray. The measurements have been made under the same conditions as in Table 2.

| Scene | MEM | ACC | ACT | TTI | RTT |
|--------------------|--------|-----------|-------|--------|--------|
| Conf. (1M) | 32 MB | <i>kd</i> | 2.1s | 113.7s | 80.5s |
| | 23 MB | BIH | 1.4s | 139.3s | 104.7s |
| | 24 MB | ncQBVH | 1.5s | 95.7s | 63.6s |
| | 16 MB | QBVH | 1.2s | 85.0s | 53.8s |
| Interior (0.9M) | 27 MB | <i>kd</i> | 1.8s | 107.3s | 77.7s |
| | 21 MB | BIH | 1.4s | 131.1s | 101.4s |
| | 22 MB | ncQBVH | 1.5s | 85.5s | 56.3s |
| | 14 MB | QBVH | 1.1s | 81.1s | 52.0s |
| Plant (12M) | 392 MB | <i>kd</i> | 30.0s | 118.5s | 97.3s |
| | 206 MB | BIH | 21.5s | 143.6s | 112.4s |
| | 310 MB | ncQBVH | 23.9s | 104.1s | 78.0s |
| | 185 MB | QBVH | 19.1s | 94.7s | 56.1s |

Table 2: Performance comparison of BIH, *kd*-tree, non-cached QBVH (ncQBVH) and cached QBVH. The scenes are shown in Figure 5. The number in brackets are the number of triangles in millions. MEM is the memory needed by the acceleration structure, ACC the type of acceleration structure used. ACT is the tree construction time, TTI the total time to image and RTT the pure ray tracing time.

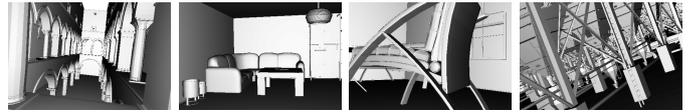
reduced when the leaf caching based QBVH implementation is used.

The same scenes as in the previous sections are used for this comparison (see Figure 1). Each rendering was performed at a resolution of 800×600 pixels with 16 passes. Table 6 shows the rendering times and the memory used by the triangle data of the scenes. The resulting QBVH trees are the same as in Section 3.1 and both versions use the SIMD triangle layout caching.

4 Accelerating Shadow Rays for Point Light Based Global Illumination

Visibility rays (or: shadow rays) are an essential part in many rendering algorithms with deterministic connections of points, e.g. in bi-directional path tracing or instant radiosity. They are merely used to determine the mutual visibility of two given points. Any intersection of an interval on a ray with an object is sufficient to determine occlusion. Therefore, three optimizations are to quit the traversal on the first intersection found, to omit the sorting of the nodes pushed on the stack, and not to compute intersection distances, which can simplify object intersection routines [Smi98].

Another important general optimization is to start the traversal deeper in the tree instead from the root node. If a suitable entry point is known, this can save major parts of the process



| Kernel | Sponza | Interior 1 | Interior 2 | Plant |
|---------|--------|------------|------------|-------|
| 2x2 BIH | 16.3 | 16.6 | 10.9 | 3.6 |
| BIH | 6.4 | 6.6 | 4.1 | 1.4 |
| QBVH | 14.1 | 11.9 | 9.03 | 2.4 |

Table 4: Comparison of frames per second for 2×2 SSE packets in a BIH vs mono ray BIH vs. mono ray QBVH on an Intel Core 2 Duo (480 \times 320). The images depict the camera positions, in the same order as in the table. In the packet version, flat shading and ray creation is also done using SIMD. Most of the performance benefit of the packets is due to coherent memory access, which is already lost for Interior 2, where QBVH and packet traversal perform about the same. So the QBVH is able to make effective use of SIMD instructions, even without exploiting high coherence for memory access.

of traversing down to the first leaf.

Reshetov et al. [RSH05] use an entry point search for large bundles of rays in *kd*-trees. In our approach we use a backtracking method that allows every node of the tree to be used as an entry point and will give correct results. For the backtracking, we store the parent with each node. As the `SIMD_BVH_Node` still provides some room to store additional information, it is possible to keep the index of the parent node without additional memory requirements.

This method proved especially suitable for shadow rays. It is possible to take advantage of implicit coherence of shadow rays and start tracing at the last node where an intersection was found.

Figure 6 shows the idea in the setting of a point light source based global illumination renderer. Consecutive visibility queries result in occlusion by different but spatially nearby triangles. This fact can be exploited by using the node of the last occlusion as the entry point for the next shadow ray. The tree is then searched for intersecting geometry using a backtracking algorithm: the regular traversal stack is initialized with the children of the entry point node and intersected the usual way. If no occlusion has been found so far, the entry point node is the root node, so there is no special code for this case. Next, if the stack is empty, all children of the parent of the entry point

| Conference | | | | | | | | | | |
|------------|-----------|------|--------|------|-----------|------|--------|------|--|--|
| MEM | 16 MB | | | | 5 MB | | | | | |
| | <i>kd</i> | BIH | ncQBVH | QBVH | <i>kd</i> | BIH | ncQBVH | QBVH | | |
| ACC | | | | | | | | | | |
| ACT | 1.3s | 1.2s | 1.2s | 1.2s | — | 0.7s | 0.7s | 0.7s | | |
| TTI | 119s | 138s | 99s | 85s | — | 220s | 224s | 120s | | |
| RTT | 89s | 106s | 86s | 54s | — | 189s | 190s | 91s | | |

| Interior | | | | | | | | | | |
|----------|-----------|------|--------|------|-----------|------|--------|------|--|--|
| MEM | 16 MB | | | | 8 MB | | | | | |
| | <i>kd</i> | BIH | ncQBVH | QBVH | <i>kd</i> | BIH | ncQBVH | QBVH | | |
| ACT | 1.3s | 1.2s | 1.1s | 1.1s | 0.8s | 0.9s | 0.9s | 0.9s | | |
| TTI | 113s | 130s | 91s | 81s | 250s | 139s | 118s | 88s | | |
| RTT | 84s | 101s | 63s | 56s | 219s | 110s | 90s | 59s | | |

| Plant | | | | | | | | | | |
|-------|-----------|------|--------|------|-----------|------|--------|------|--|--|
| MEM | 185 MB | | | | 100 MB | | | | | |
| | <i>kd</i> | BIH | ncQBVH | QBVH | <i>kd</i> | BIH | ncQBVH | QBVH | | |
| ACT | 21s | 18s | 19s | 19s | 14s | 15s | 16s | 16s | | |
| TTI | 129s | 139s | 105s | 95s | 369s | 145s | 117s | 95s | | |
| RTT | 85s | 98s | 78s | 56s | 330s | 107s | 90s | 68s | | |

Table 5: The performance of the different acceleration structures when the available memory is a-priori bounded (MEM). ACC is the type of acceleration structure used. The timings are given for the acceleration structure construction time (ACT), the total time to image (TTI) and the time spend in the ray tracing core (RTT).

| Scene | 36 TTI | 36 MEM | VI TTI | VI MEM |
|------------|--------|--------|--------|--------|
| Interior | 107s | 32 MB | 108s | 16 MB |
| Plant | 191s | 438 MB | 197s | 213 MB |
| Conference | 62s | 43 MB | 66s | 18 MB |

Table 6: This table compares the impact of using the flat triangle layout with 36 bytes per triangle (36) to a vertex-index (VI) based representation. TTI is the total time to image and MEM is the memory consumption of the triangle data.

node are pushed to the stack, except the one that has just been processed. For the next iteration, the entry point node is set to its parent. The algorithm terminates when the parent of the root node is about to be processed. So the regular traversal is only extended by one variable and one additional loop. If a leaf node is found as an occluder, its parent node is cached for this point light source.

What remains is finding a suitable entry point for an occlusion query. The described algorithm above is correct for any entry point, but an acceleration can only be expected when the hit is close to a previous occlusion. The rendering system uses occlusion queries given by two points, where the first one is the path’s hit point and the second one the light source position. In our implementation we use a simple unbounded (i.e. hashed) voxel grid and the light position as hash. For each position the last found occlusion entry point is recorded. If no occlusion occurred, the root node is used. In the statistics we used a hash table with 4096 entries and divided the scene into $(10cm)^3$ voxels (assuming the scenes are modeled in the correct scale).

This optimization can be transparently incorporated into any existing rendering system which spawns shadow rays with two points to be checked for visibility.

5 Results for Accelerating Shadow Rays

All benchmarks were done using a single thread on a Core 2 processor. Extending this to a multi-threaded version is straight-

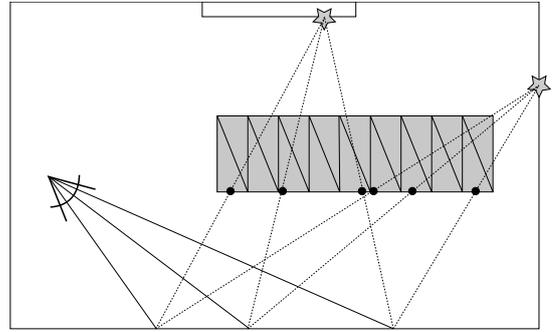


Figure 6: Spatial coherence of occluders when using point light based global illumination. The entry point cache is able to quickly find occlusions, since the visibility rays to the same point light source (depicted with a star) exhibit some coherence. The big leaves of the QBVH make a hit even in the same leaf more likely. The performance gain is achieved in a way which is completely transparent to the underlying renderer, no additional complexity is introduced to existing systems (as would be the case for ray packets).

forward for the application of photorealistic rendering, where the fast tree construction times vanish compared to the time required for ray tracing. The resolution for the benchmarks are 800×600 pixels and for the point light based approach 16 rendering passes were used with an average of 22 point light sources per pass. Primary rays were created per scanline, additional speedup can be expected when using Hilbert curves. We compare the normal implementation of the QBVH (using only the standard shadow optimizations) with the one using the entry point caching. The statistics contain the total number of bounding box intersections performed for the final image. The speedup is of course larger in scenes with a larger amount of occlusion. This explains why the interior scene only shows a marginal speed up. The results are shown in Table 7.

| Scene | N#BBox | NTTI | C#BBox | CTTI |
|--------------|--------|------|--------|------|
| Conference | 16.8 | 73s | 11.7 | 62s |
| Conference 2 | 20.7 | 92s | 9.9 | 60s |
| Interior | 17.3 | 109s | 15.4 | 107s |
| Plant | 44.8 | 297s | 25.0 | 191s |

Table 7: Comparison of the number of box intersections and total time to image image between the normal QBVH version (N#BBox, NTTI) and the entry point cache version (C#BBox, CTTI). The images at the beginning of this paper show the test scenes used. For the measurement they were rendered only at a resolution of 800×600 pixels with 16 passes and an average of 22 point light sources per pass. Conference 2 is the same conference scene but with the camera placed partly under the table.

The caching method could also be used for other acceleration structures. But compared to acceleration structures like the *kd*-tree, the BIH or binary BVHs which use many, very densely packed, small nodes, this method works especially well for the QBVH. The additional memory requirement of one more parent index per node would result in 50% more memory for standard 8 bytes *kd*-tree nodes.

The approach can also be used for non-shadow rays. Then

the procedure cannot benefit from the early-out of shadow rays, i.e. processing must always continue up to the root node. Still it can be beneficial, if the rays expose some kind of coherence, for example primary rays from the eye or coherent reflections. This allows the ray tracing core to transparently exploit implicit coherence by caching previous intersection entry points without changing the interface and without performance loss when incoherent rays are used.

6 Conclusion and Further Work

We efficiently applied SIMD instructions for accelerating the tracing of single rays and kept the memory footprint of the algorithms as low as possible. The algorithm is simple to use in a wide range of applications and still allows for taking advantage of modern computer architectures. We showed that the performance of our algorithm is better than current high performance single ray tracing algorithms and the memory requirements are reduced due to the flat hierarchies and caching we use.

Contrary to tracing ray packets to reduce memory bandwidth, higher arity trees achieve the reduction of memory bandwidth by just using less memory. This in turn increases cache coherence and thus effectively reduces the sum of latencies.

It is obvious how to generalize the approach to higher arity. A multiple of 4 perfectly supports current processor's streaming SIMD extensions (Altivec, SSE). In contrast to tracing ray packets, the implementation on novel architectures like the CELL and GPUs with CTM or CUDA is simplified; the 16-way SIMD path of Intel's Larrabee processor is an obvious candidate for our algorithm and the large node size suits modern cache architectures and memory layouts very well.

Experiments showed that forcing the same split plane axis for all three splits does not introduce severe performance penalties and the simplified sorting step in the traversal routine can even pay off for a SIMD width of four in some scenes. With the availability of wider SIMD units, this approach has to be further investigated.

Additionally an implementation of this acceleration structure as special purpose hardware (for example in an FPGA) looks very promising.

The tree construction could also be improved. While using the standard binary tree construction for building n -ary BVHs is a simple and practical approach, faster build times could be achieved if an equally good heuristic could be found for multiple split planes at once. The entry point cache could be extended by a better hashing mechanism to accelerate bi-directional light transport algorithms equally well as point light source based ones.

7 Acknowledgments

The authors would like to thank mental images GmbH for support and funding of this research. Credits go also to *toxie* for providing his implementation of state-of-the art ray tracing kernels and *necro* for the global illumination framework to enable fair comparisons on the same machine.

References

[Ben04] BENTHIN C.: *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, 2004.

[CFLB06] CHRISTENSEN P., FONG J., LAUR D. M., BATALI D.: Ray tracing for the movie 'Cars'. In *Proc.*

2006 IEEE Symposium on Interactive Ray Tracing (2006), pp. 73–78.

- [CSE06] CLINE D., STEELE K., EGBERT P.: Lightweight bounding volumes for ray tracing. *Journal of Graphics Tools* 11, 4 (2006), 61–71.
- [FAN07] FURTAK T., AMARAL J. N., NIEWIADOMSKI R.: Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In *SPAA '07: Proc. of the 19th annual ACM Symposium on Parallel Algorithms and Architectures* (2007), pp. 348–357.
- [Gei06] GEIMER M.: *Interaktives Ray Tracing*. PhD thesis, Koblenz-Landau University, Germany, 2006.
- [Gla89] GLASSNER A.: *An Introduction to Ray Tracing*. Academic Press, 1989.
- [GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics & Applications* 7, 5 (1987), 14–20.
- [Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, 2001.
- [HB02] HAVRAN V., BITTNER J.: On improving kd-trees for ray shooting. *Journal of WSCG* 10, 1 (2002), 209–216.
- [HB07] HAVRAN V., BITTNER J.: Ray tracing with sparse boxes. In *Spring Conference on Computer Graphics (SCCG 2007)* (2007), pp. 49–54.
- [HMB06] HUBO E., MERTENS T., BEKAERT P.: The quantized kd-tree: compression of huge point sampled models. *ACM SIGGRAPH 2006 Sketches* (2006), 179.
- [Kel97] KELLER A.: Instant radiosity. *ACM Transactions on Graphics (Proc. SIGGRAPH 1997)* (1997), 49–56.
- [KK02] KOLLIG T., KELLER A.: Efficient bidirectional path tracing by randomized quasi-Monte Carlo integration. In *Proc. Monte Carlo and Quasi-Monte Carlo Methods 2000*. Springer, 2002, pp. 290–305.
- [LYM07] LAUTERBACH C., YOON S.-E., MANOCHA D.: Ray-strips: A compact mesh representation for interactive ray tracing. In *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 19–26.
- [Mah05] MAHOVSKY J.: *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary, 2005.
- [MT97] MÖLLER T., TRUMBORE B.: Fast, minimum storage ray/triangle intersection. *Journal of Graphics Tools* 2, 1 (1997), 21–28.
- [Res07] RESHETOV A.: Faster ray packets - triangle intersection through vertex culling. In *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 105–112.
- [RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM Transactions on Graphics (Proc. SIGGRAPH 2005)* (2005), 1176–1185.

- [Shi00] SHIRLEY P.: *Realistic Ray Tracing*. AK Peters, Ltd., 2000.
- [Smi98] SMITS B.: Efficiency issues for ray tracing. *Journal of Graphics Tools* 3, 2 (1998), 1–14.
- [SSK07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN A.: Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In *Computer Graphics Forum (Proc. Eurographics 2007)* (2007), pp. 395–404.
- [Vea97] VEACH E.: *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1997.
- [Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.
- [Wal07] WALD I.: On fast construction of SAH based bounding volume hierarchies. In *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 33–40.
- [WBMS05] WILLIAMS A., BARRUS S., MORLEY R. K., SHIRLEY P.: An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools* 10, 1 (2005), 49–54.
- [WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics* 26, 1 (2007).
- [WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proc. Eurographics 2001)* (2001), pp. 153–164.
- [WH06] WALD I., HAVRAN V.: On building fast kd-trees for ray tracing, and on doing that in $O(N \log N)$. In *Proc. 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 18–20.
- [WK06] WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006 (Proc. 17th Eurographics Symposium on Rendering)* (2006), pp. 139–149.
- [WK07] WÄCHTER C., KELLER A.: Terminating spatial partition hierarchies by a priori bounding memory. In *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 41–46.