

Utrecht University

DEPARTMENT OF INFORMATION AND COMPUTING SCIENCES

MASTER'S THESIS

GAME & MEDIA TECHNOLOGY

Fast Divergent Ray Traversal by Batching Rays in a BVH

Tigran Gasparian

ICA-3705617

Supervisor

Dr. ing. J. (JACCO) BIKKER

Second Examiner

Dr. ir. A.F. (FRANK) VAN DER STAPPEN

5 December 2016

Fast Divergent Ray Traversal by Batching Rays in a BVH

Tigran Gasparian[†1]

[1]Department of Information and Computing Sciences, Utrecht University

Abstract

Ray tracing forms the basis of photorealistic rendering as seen in films and special effects. The process of rendering all the frames of animated films can take thousands of CPU years. Improving the efficiency of the rendering algorithm translates into large savings of time.

In this thesis, we focus on the algorithm that is at the core of all rendering systems, computing the intersection point between a ray and a scene. Our contribution is a novel ray traversal scheme aimed at highly divergent ray distributions. We improve traversal efficiency by batching rays at fixed points in a BVH during traversal. The batched rays benefit from improved cache efficiency and utilization of instruction level parallelism and achieves performance improvements of up to 99% for intersection queries and up to 123% for occlusion queries for ray distributions seen after the first diffuse bounce when compared to a single-ray traversal scheme. Our scheme is orthogonal to recent advances in divergent ray traversal, and for large scenes, substantially improves on state of the art performance.

Keywords: Ray Tracing, Ray Traversal

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Three-Dimensional Graphics and Realism]: Raytracing—Visible line/surface algorithms

1. Introduction

Ray tracing is a technique that forms the basis of photorealistic rendering. It consists of two basic operations: intersection queries and occlusion queries. Given a scene and a ray with an origin, direction and maximum intersection distance, an intersection query determines the nearest intersection point between the ray and a scene that is within the maximum distance from the ray origin, whereas an occlusion query only determines whether there is an intersection of the ray with the scene within the maximum distance.

To accelerate ray/scene intersections, a space partitioning or object partitioning data structure is used. Popular choices are the kD-tree and the Bounding Volume Hierarchy (BVH). In this paper, we focus on BVHs, but our method can also be applied to other acceleration structures.

As CPU speeds have increased at a much higher rate compared to memory access times, many algorithms have become increasingly memory bound. To alleviate this, the CPU uses a hierarchy of caches to hide the latency of retrieving data from memory. To achieve high performance, an algorithm has to optimally utilize these caches. In practice, most scenes and their corresponding BVHs are too large to fit in the CPU cache in its entirety. To achieve good performance, a node from the BVH that is retrieved from memory should be used to test against multiple rays before getting evicted from the cache. Ideally, a node should only be read into the cache once.

Ray tracing algorithms are known to exhibit highly irregular memory access patterns. This is particularly true for divergent ray distributions commonly encountered in path tracing. Under these circumstances, CPU caches fail to effectively hide memory access latency, resulting in poor ray tracing performance.

In this work we propose a batching traversal scheme called RayCrawler. Our scheme operates on a hierarchy of BVHs by splitting an existing BVH into two separate layers, creating a top-level tree and multiple small trees that fit in the L2 cache of modern CPUs. The rays traverse the top-level BVH and are batched at the leaf nodes before traversing the

† gasparianti gran@gmail.com

second layer, amortizing the cost of retrieving the subtree from the second layer from memory over the rays that are batched in the leaf node.

Section 3 gives an overview of the algorithm. Sections 4 and 5 describe the details of the used data structures and the traversal algorithm. Our results are presented in section 6, with the conclusion and suggestions for future work in section 7.

2. Previous work

Considerable progress has been made in improving the efficiency of Whitted-style ray tracing by exploiting the coherence of ray distributions which typically occur in this rendering algorithm. This is achieved by amortizing the cost of fetching data over multiple rays, using pyramid or packet traversal [vdZRJ95] [WSBW01] [BEL*07].

The ray coherence that these algorithms rely on is mostly lost even after a single specular bounce. This led to the development of packet traversal schemes that are still able to exploit ray coherence for somewhat divergent ray distributions [ORM08].

Considering diminishing coherence for divergent ray distributions, several authors suggest using wide BVHs and dropping packet traversal altogether. Vector hardware such as SSE, AVX and AVX-512 benefits from 4-wide, 8-wide and even 16-wide BVH traversal [WBB08] [EG08] [DHK08].

Tsakok proposed a scheme that exploits ray coherence in very incoherent distributions, such as those after the first diffuse bounce in a path tracer [Tsa09]. This algorithm traverses large groups of rays through a 4-wide BVH in a breadth-first manner, intersecting many rays with a single node before evicting the node from the cache. This approach requires all rays to traverse the BVH in the same order, which may resulting in unnecessary nodes being visited.

Dynamic Ray Stream Traversal (DRST) [BAM14] extends this idea by allowing rays to traverse through a 4-wide BVH in different orders. This results in more nodes being culled, but has increased bookkeeping overhead. DRST limits the number of possible traversal orders of the rays from 24 to 8 to keep bookkeeping overhead low. This results in suboptimal ray culling compared to a single ray traversal scheme.

Ordered Ray Stream Traversal (ORST) [FLPE15] builds upon this idea and allows rays to traverse in all possible orders while decreasing bookkeeping costs by utilizing precomputed lookup tables to determine the traversal order of the rays.

This paper proposes a traversal algorithm for incoherent ray distributions based on the RayGrid scheme [Bik12]. This scheme coarsely subdivides the scene using an octree and stores a BVH of the parts of the scene in its leaf nodes. The BVHs stored in the leaf nodes are small enough to fit in the L2 cache. The algorithm traverses rays through the octree, batching them in the leaf nodes. Large batches then traverse the BVHs, amortizing the cost of retrieving the BVH from memory into cache over the batched rays. The scheme achieves modest speedups compared to a single-ray traversal algorithm for secondary rays and proves that a batching scheme can outperform a naive single-ray traversal approach for highly divergent rays. However, the performance gains are limited by the high cost of traversing the octree, which in turn becomes the performance bottleneck for this algorithm.

The comparisons in this work are made with the algorithms implemented in the Embree framework version 2.7.1 [WWB*14]. This framework contains highly optimized ray tracing kernels with many industry applications. The single-ray traversal scheme for 4-wide BVHs serves as the baseline in the comparisons as this algorithm has the best performance for highly divergent ray distributions.

3. Overview

This section gives a short overview of the data structure used throughout the paper and an overview of the traversal algorithm. The goal of our scheme is to improve cache efficiency by batching rays together before traversing parts of the scene, amortizing memory reads over the batched rays. We achieve this by splitting a regular 4-wide BVH in two layers. The rays are batched in the leaf nodes of the top layer before traversing the bottom layer of the data structure. By using the same structure as a regular 4-wide BVH, we can take advantage of existing traversal algorithms in the scheme. Our approach is thus orthogonal to recent approaches that aim to improve traversal efficiency such as DRST and ORST.

Our data structure is constructed by splitting a regular 4-wide BVH in two layers; a top level BVH (Top-BVH) where the leaf nodes point to Leaf-BVHs. Each individual Leaf-BVH is small enough to fit in the L2 cache of modern CPUs. The traversal algorithm starts by first traversing each ray depth-first through the Top-BVH; once the ray reaches a leaf node of the Top-BVH, the ray is batched at the Leaf-BVH that the leaf node is pointing to and the traversal of the ray is suspended. The Top-BVH traversal stack of the ray is stored to resume traversal later on.

Once all rays have traversed the Top-BVH and are distributed among the Leaf-BVHs, a Leaf-BVH is selected to be traversed. All rays batched at the Leaf-BVH traverse this tree. If the query is an occlusion query, the traversal of the ray is terminated if an intersection is found. Otherwise, any intersections found for the rays are stored and the rays are shortened.

Once all rays batched at the Leaf-BVH have been processed, traversal through the Top-BVH is resumed using the stored traversal stacks and the nearest intersection distances

of the rays that traversed the Leaf-BVH. A new Leaf-BVH is selected to be traversed after the rays have been distributed among the Leaf-BVHs. This process continues until no Leaf-BVH with batched rays remains. This process is illustrated in Figure 1.

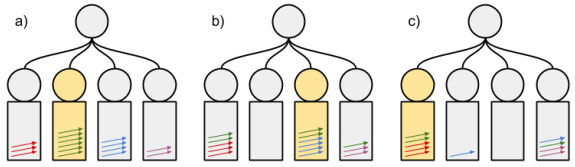


Figure 1: *A high level example of the traversal using a Top-BVH with 5 nodes and 4 Leaf-BVHs. All rays first traverse the Top-BVH and are distributed among the Leaf-BVHs. (a) Then a Leaf-BVH with a high number of batched rays is selected, this is highlighted in yellow. All batched rays traverse the Leaf-BVH. After traversing the Leaf-BVH, the rays again traverse the Top-BVH and can be distributed among the other Leaf-BVHs. This process is repeated until no Leaf-BVH with batched rays remains.*

This system tries to amortize the cost of retrieving a Leaf-BVH from memory by traversing many batched rays through the Leaf-BVH once it has been loaded into cache. A large number of rays needs to be batched at the Leaf-BVH in order to justify the overhead of the batching process itself. This requires a large initial set of rays as the rays are distributed among all Leaf-BVHs. When selecting a Leaf-BVH for traversal, selecting one with a large number of batched rays is preferred.

Traversal of the Top-BVH thus serves to batch rays for Leaf-BVH traversal. While the original rays may intersect several Leaf-BVHs, truncating rays based on intersections with geometry in the Leaf-BVHs may prevent them from being batched in a subsequent Leaf-BVH. Pausing traversal in the Top-BVH after one Leaf-BVH is found, enables us to cull subsequent Leaf-BVHs if an intersection in the first Leaf-BVH is found.

4. Data structures

The hierarchy of BVHs is constructed by splitting a regular 4-wide BVH in two layers. The Top-BVH is built by starting at the root of the original BVH and adding nodes to the Top-BVH until one of the following conditions is met:

- The node contains a child with a size smaller than a certain threshold, related to the size of the L2 cache.
- The node has a depth of 16. The Top-BVH can have a maximum depth of 16 to limit the size of the stack in the Top-BVH traversal. See section 5.1 for details.

The nodes that the leaf nodes of the Top-BVH point to are selected as the root nodes of the Leaf-BVHs. The Leaf-BVHs consist of the root nodes and all their descendants.

Buckets Rays are batched in buckets before traversing the Leaf-BVHs. Every Leaf-BVH has a corresponding linked list of buckets where every bucket can contain a fixed number of rays. The fixed size buckets are obtained from a pre-allocated list of buckets, which prevents run-time memory allocations.

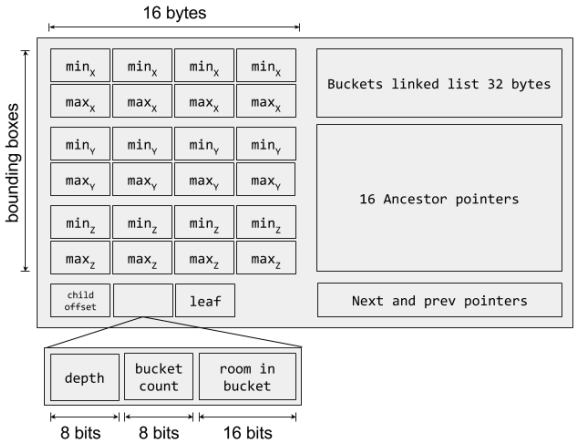


Figure 2: *Top-level node structure. The total size of the node with 16-byte alignment is 224 bytes.*

Top-BVH node structure In order to facilitate an efficient way of pausing and resuming traversal and an efficient way to store the traversal stack, the Top-BVH nodes are augmented with:

- an array of ancestor pointers which is used to quickly navigate up in the tree;
- and the depth of the node in the BVH.

A Top-BVH node also contains additional information to batch rays for the Leaf-BVHs:

- a next and a prev pointer, to act as a node in a linked list;
- a linked list of buckets;
- the number of buckets in the linked list;
- and the room in the last bucket in the linked list.

Figure 2 shows the exact structure of the node.

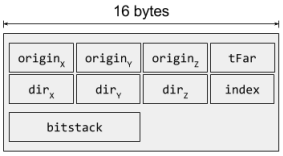


Figure 3: *Top-Level ray structure.*

Top-BVH ray structure In order to facilitate suspending and resuming traversal, the traversal stack is stored in the ray structure as a 64 bit unsigned integer. See section 5 for more information on how the stack is managed. Together with the origin, direction, index and the nearest intersection distance, a ray can be stored in 40 bytes. See Figure 3 for the layout of the ray.

Leaf-BVH node structure Leaf-BVH nodes have the same structure as regular 4-wide BVH nodes and any algorithm that can traverse a 4-wide BVH can be used to traverse the Leaf-BVHs. Our implementation uses Tsakok's Multi-BVH Ray Stream tracing algorithm [Tsa09] and a single ray traversal scheme to traverse the Leaf-BVHs.

5. Traversal

The traversal algorithm consists of two stages. In the first stage, all rays traverse the Top-BVH and get distributed among the Leaf-BVHs. In the second stage, we repeatedly choose a Leaf-BVH with a high number of batched rays. The batched rays traverse the Leaf-BVH and the non-terminated rays traverse the Top-BVH afterwards to get redistributed over the other Leaf-BVHs.

Two distinct traversal algorithms are used to traverse the hierarchy of BVHs. Although any traversal algorithm that works on regular BVHs can be used to traverse the Leaf-BVHs, algorithms with low starting overhead work well here as the Leaf-BVHs are generally small. The traversal algorithm for the Top-BVH needs to be able to pause and resume traversal. Therefore, the traversal algorithm needs to use as little state as possible in order to efficiently store and retrieve this state to resume traversal. This is achieved using a stackless traversal scheme.

5.1. Top-BVH traversal

A stackless traversal scheme is a scheme that uses as little state as possible during traversal. Prior works have investigated stackless traversal schemes for kD-trees [FS05], binary BVHs [Lai10] [HDW*11] and 4-wide BVHs [ÁSK14]. Áfra and Szirmay-Kalos propose a stackless 4-wide BVH traversal scheme using 64 bits to hold the stack by extending the nodes to contain parent and sibling pointers.

This method however does not guarantee an ordered traversal of the tree resulting in more visited nodes. As the Top-BVH is a relatively small tree, it is worth trading in some traversal speed to visit the leaf nodes in the correct order, which may result in more Leaf-BVHs being culled. We propose a different stackless traversal scheme that guarantees an ordered traversal of the nodes, but can intersect the same nodes multiple times. The stack management is presented below followed by the traversal algorithm.

Stack management Because the Top-BVH has a maximum depth of 16, we can encode the stack in a 64 bit unsigned integer. We call this the bitstack.

Every node in the traversal path is represented by four bits in the bitstack. The four bits in the bitstack corresponding to the depth of the current node that is being visited is called the interestMask. The interestMask represents the children of the current node that are being considered to be visited and does not include already visited children.

The root node's children are represented by the four least significant bits in the bitstack. Each child node intersected by the ray (taking into account the current length of the ray) is set to 1. Upon traversal into a child node, the bit corresponding to the child node in the bitstack is set to 0 to prevent the child node from being visited again. The next four bits corresponding to the depth of the child node are set to 1, which initializes traversal at the next level of the tree. This process is illustrated in Figure 4.

Once a ray resumes traversal starting from a leaf node or when there are no children to traverse into, the next node to be visited can be found by determining the most significant bit set in the bitstack. This can be done by a bit scan reverse operation. The most significant bit can be used to determine the depth of the next node (e.g. the four least significant bits represent depth 0, next four bits represent depth 1, etc.). The next node to be visited can be found by using the depth of the current node and the list of ancestor pointers that is stored in the node. See Figure 5 for an example of upward traversal.

Traversal The traversal algorithm is similar to a single-ray depth first BVH traversal scheme with the modification that the traversal stack has been replaced by the bitstack described above. The Top-BVH traversal algorithm is shown in Listing 1.

```
1  def Traverse(ray, n):
2    (hitmask, distances) = Intersect(ray, n)
3    stackMask = 15 << (n.depth * 4)
4    interestMask = (stackMask & ray.bitstack) >> (n.depth * 4)
5    # Clear the bitstack to make setting it easier later.
6    ray.bitstack &= ~stackMask
7    toVisitMask = hitmask & interestMask
8    if toVisitMask == 0:
9      if ray.bitStack == 0:
10       return # Traversal is terminated
11     ancestorDepth = MostSignificantBitIdx(ray.bitstack) >> 2
12     Traverse(ray, n.ancestors[ancestorDepth])
13   else:
14     closestChildIdx = Closest(toVisitMask, distances)
15     toVisitMask -= 1 << closestChildIdx
16     # Update the bitstack
17     ray.bitstack |= toVisitMask << (n.depth * 4)
18     child = n.children[closestChildIdx]
19     if not child.isLeaf():
20       # Set the next four bits to 1
21       ray.bitstack |= 15 << (child.depth * 4)
22       Traverse(ray, child)
23     else:
24       if child.bucketCount > 0:
25         child.lastBucket.add(ray)
26         if child.lastBucket.isFull():
27           child.addBucket(getNewBucket())
```

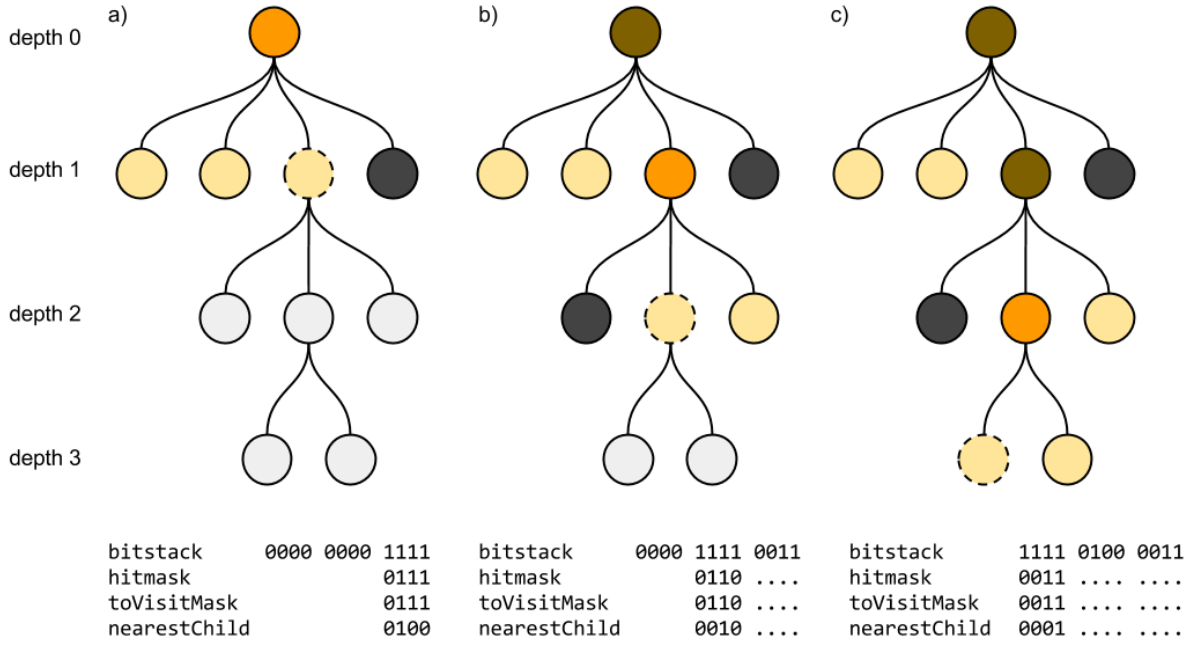


Figure 4: *An example of downward traversal in the Top-BVH. The orange-colored node depicts the current node being visited, dark gray depicts nodes that do not intersect the ray, yellow nodes intersect the ray, the node with dashed outline indicates the nearest active child node and brown nodes indicate already visited nodes. a) The stack is initialized by setting the four least significant bits to 1. We get the hitmask by intersecting the child nodes of the current node. Performing a bitwise AND with the interestMask results in the toVisitMask. We then select the nearest child node that is active in the toVisitMask to traverse into. We set the bit in toVisitMask corresponding to the selected child node to 0 and update the bitstack. b) The four bits corresponding to the depth of the child node are set to 1. The same process is repeated until a leaf node is reached.*

```
28            if child.bucketCount == 2:
29              nodesWithPartialBuckets.remove(child)
30              nodesWithFullBuckets.add(child)
31          else:
32            nodesWithPartialBuckets.add(child)
33            child.addBucket(getNewBucket())
34            child.lastBucket.add(ray)
35        return # Traversal is paused
```

Listing 1: *Top-BVH traversal algorithm*

Upon creation of a ray, the bitstack is initialized by setting the four least significant bits to 1.

The ray is tested against the bounding volumes of the children of node n that is currently being visited, resulting in 4 bits representing which bounding volumes intersect with the ray (hitmask) and the intersection distances if an intersection is found (line 2). The child nodes that are not being considered to be visited are masked out using the interestMask, resulting in the toVisitMask (lines 3-7).

If the toVisitMask is 0 (i.e. there is no child node to be visited) we check if the bitstack is 0. If this is the case, the traversal of the ray terminates (line 9). Otherwise, the index of the most significant bit set on the bitstack is found. This index is used to determine the depth of the node to traverse into next and using the ancestor pointer array in n, the new node to be visited is found (lines 10-11). See Figure 5 for an example.

If the toVisitMask is not 0, the closest child node that is active in the toVisitMask is selected to be visited next (line 14). The bit corresponding to the closest child is set to 0 in the toVisitMask to mark this child as traversed (line 15). The bitstack is updated by setting the four bits corresponding to node n to toVisitMask (line 17).

If the child node that is going to be visited next is not a leaf node, the four bits in the bitstack corresponding to the child node are set to 1 and traversal continues into the child node. (line 17-20). See Figure 4 for an example.

If the child node is a leaf node, the node is checked whether there is already a bucket assigned to it (line 23-24). If there is already a bucket assigned to the node, the ray is added to the bucket and the bucket is checked whether it is full (lines 25-26). If the bucket is full, a new bucket is added to the linked list of the node (line 27) If this is the first

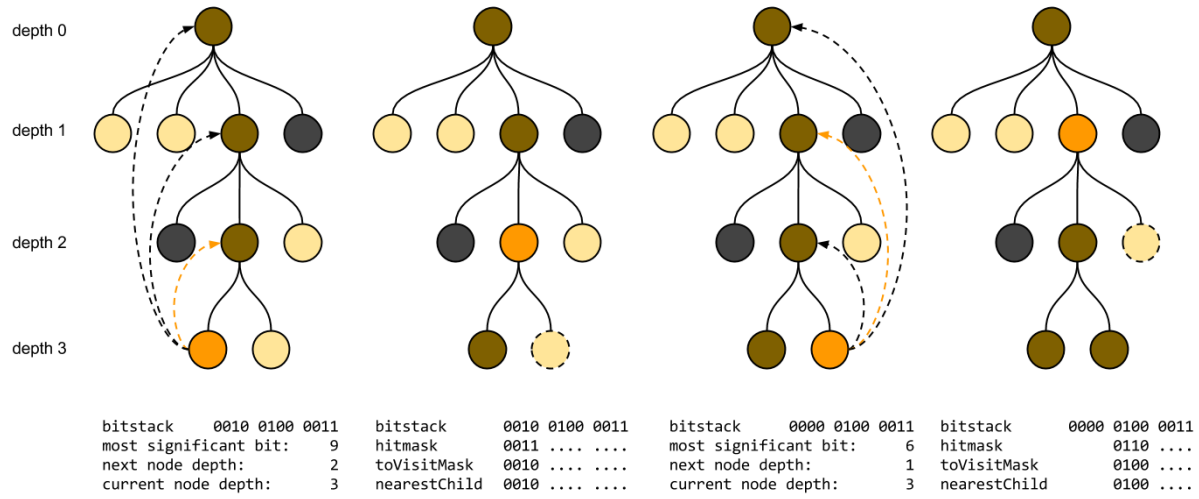


Figure 5: *An example of backtracking in the Top-BVH traversal. The dashed arrows indicate the ancestor pointers of the current node. See Figure 4 for the color coding. Once a ray resumes traversal starting from a leaf node or when there are no children to traverse into, we compute the index of the most significant bit set. We can determine the depth of the next node by dividing this index by 4. Using the depth of the next node and the current node, we select the correct ancestor pointer to move to the next node. Thus we find the first ancestor with unvisited child nodes in O(1) time. This is shown by the orange arrow.*

full bucket in the node, the node is moved from the list of nodes with partially filled buckets to the list of nodes with full buckets (lines 28-30) and the traversal is paused.

If there is no bucket assigned to the node, a new bucket is added to the linked list of the node, the ray is added to the bucket, the node is added to a list of nodes with partially filled buckets (line 32-34) and the traversal is paused.

5.2. Leaf-BVH traversal

Once all rays have traversed the Top-BVH and have been distributed among the Leaf-BVHs, we select a Leaf-BVH that has a full bucket. If no Leaf-BVH with a full bucket can be found, a Leaf-BVH with a partially filled bucket is selected. If all Leaf-BVHs are empty, every ray has found the nearest intersection (if any) and the algorithm terminates.

The rays in the bucket are converted to the required structure for the Leaf-BVH traversal algorithm and traverse the Leaf-BVH. Occlusion queries terminate when an intersection is found; intersection queries record the intersection results and shorten the ray.

The rays that are still active (i.e. occlusion rays that have not intersected and intersection rays that have a non-zero bitstack) resume traversal in the Top-BVH and get distributed among other Leaf-BVHs.

Traversing the Leaf-BVH can be done with any traversal algorithm. Leaf-BVHs are generally small and the number of rays in the buckets is also relatively small; we found that traversal algorithms with a low starting overhead that don't require too many rays in the ray stream tend to perform well.

In our implementation, we use a hybrid of Tsakok's Multi-BVH Ray Stream Tracing algorithm and a single ray traversal scheme. The single ray traversal scheme is chosen when we have less than 12 rays in the bucket. This number is chosen empirically.

6. Results

This section evaluates the performance of RayCrawler compared to a single ray traversal algorithm that will act as our baseline (Single Ray). The single ray traversal implementation is taken from Embree 2.7.1 and adapted to work in our framework. The code is optimized for 4-wide SIMD instructions to support a wide range of CPU architectures.

The tests are run on a system with an Intel Xeon E5-1620v3 processor clocked at 3.5GHz. This CPU has a 32KB L1 instruction cache, a 32KB L1 data cache, a 256KB L2 cache and a shared 10MB L3 cache. The tests are ran on a single core, but the algorithm can be scaled to multiple cores by assigning a different tile of pixels to each core.

The path tracer used in the tests traces rays that bounce up to five times before being terminated. Every bounce spawns a shadow ray towards a light source. Russian roulette path termination has been disabled to ensure a high number of rays at all path depths. All tests are ran using 16 samples per pixel and unless mentioned otherwise, the tile size is set to 512x512 pixels, for a total of 4M paths. We omit the results

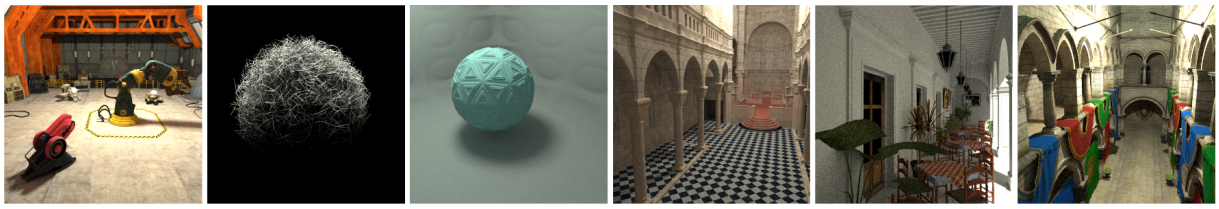


Figure 6: *From left to right, ROBOT LAB (472K triangles), HAIRBALL (2.88M triangles), ICOSPHERE (3.14M triangles), SIBENIK (75K triangles), SAN MIGUEL (7.88M triangles), CRYTEK SPONZA (262K triangles).*

for the primary rays, because these are outside the scope of our algorithm, as primary rays are commonly traversed using much faster packet traversal schemes.

The comparison is based on six test scenes (see Figure 6) where we measured the time spent performing the intersection and occlusion queries at five different path depths, averaged over 3-5 camera positions per scene. The scenes are chosen to show a variety in complexities and to enable comparisons with previous work. HAIRBALL is chosen because it contains a lot of overlapping geometry, which will cause a traversal algorithm to visit many BVH nodes where no intersection can be found. SAN MIGUEL is chosen to evaluate the performance of our scheme on very large scenes and ICOSPHERE is chosen to demonstrate the performance of our algorithm when there is one detailed object in a relatively simple scene.

In this section we evaluate the effect of three variables on the performance of our scheme: the amount of divergence in the ray distribution, the number of available rays in the system and the number of primitives in the scene.

6.1. Divergence

We use a path tracer to generate ray distributions with varying levels of divergence. After every diffuse bounce, the ray distribution becomes more divergent. Ray traversal algorithms tend to perform better when a ray distribution is coherent and degrade in performance after each diffuse bounce, as the average number of rays that traverse each node decreases and memory access patterns essentially become random.

Figure 7 shows the performance results for two test scenes for various path depths. We see that the traversal speed degrades as the path depth increases. Our scheme shows noticeably less degradation in performance compared to our baseline algorithm as our batching scheme still manages to group rays together even for highly divergent ray distributions. Table 1 shows the raw results for all scenes. We see increases in traversal speed of up to 63% for intersection queries and up to 123% for occlusion queries for path depths after the first bounce compared to the baseline algorithm.

6.2. Available rays

Because our scheme relies on batching rays at Leaf-BVHs to amortize the cost of loading the Leaf-BVHs from memory into the cache over many rays, a high ray count is required to fill the buckets at the leaf nodes. Figure 8 evaluates the performance of our scheme with respect to the number of active rays in the system. We vary the number of active rays in the system by choosing different tile sizes. Recall that we always use 16 samples per pixel. As we would expect, the traversal efficiency of our scheme increases as the number of active rays is increased. Our scheme becomes worthwhile after a tile size of 64x64 pixels, but a tile size of 256x256 or higher is needed to achieve significant performance increases. Choosing the right tile size will depend on the amount of available memory, the number of rays available in the system and the required granularity when parallelizing the algorithm. See Table 2 for the raw results.

6.3. Primitive count

We can see in Table 1 that the relative traversal speed of our scheme is significantly higher for the SAN MIGUEL scene than the other test scenes. This scene also has significantly more triangles than the other test scenes. As a scene becomes more complex, the geometry and its BVH cannot fit in the caches of the CPU anymore and a larger part of the scene becomes solely available in main memory, which leads to high latencies when accessing this data. Because our scheme increases cache efficiency by amortizing the cost of loading a Leaf-BVH into the cache over many rays, we see less degradation in performance than the single ray traversal scheme as the number of triangles in the scene increases.

To evaluate this hypothesis, we simulate higher triangle counts by measuring the traversal speed of our scheme relative to the baseline algorithm for different subdivisions of the ROBOT LAB and CRYTEK SPONZA test scenes. The subdivisions of the scenes are generated using different variations of the subdivision and decimation modifiers in Blender [Ble16]. Figure 9 shows an increase in the relative traversal speed as we increase the number of triangles in the scene. The results become comparable to the performance increases we saw for the SAN MIGUEL scene in Table 1. See Table 5 for the raw results.

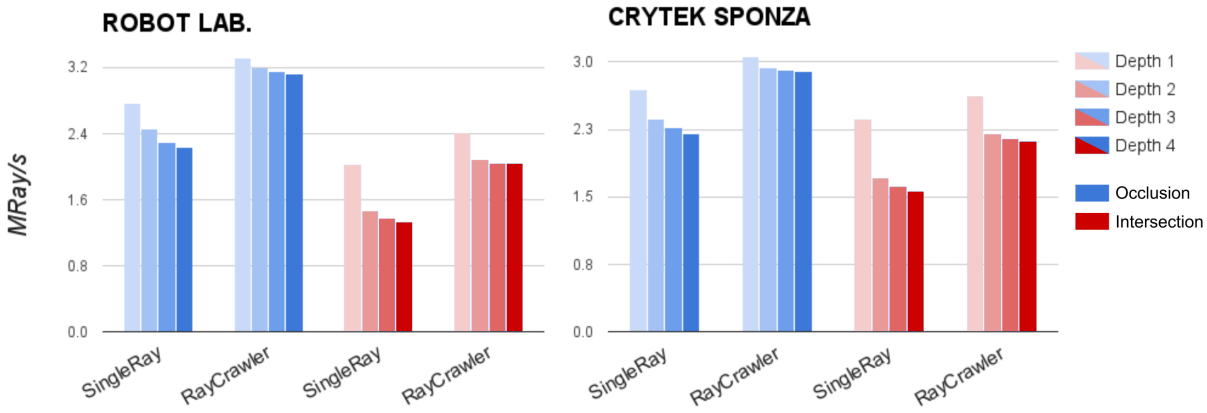


Figure 7: *The traversal speed of our scheme and the single ray traversal algorithm for different path depths. The traversal speed is displayed in millions of rays per seconds (MRay/s).*

Scene	Scheme	1i	2i	3i	4i	0o	1o	2o	3o	4o
ROBOT LAB	Single Ray	2.029	1.465	1.371	1.33	8.076	2.771	2.455	2.288	2.234
	RayCrawler	2.406	2.082	2.045	2.036	4.486	3.332	3.201	3.155	3.128
		+19%	**+42%**	**+49%**	**+53%**	**-44%**	**+20%**	**+30%**	**+38%**	**+40%**
HAIRBALL	Single Ray	1.239	1.28	1.346	1.347	1.303	1.316	1.212	1.198	1.16
	RayCrawler	1.615	1.622	1.496	1.488	2.268	2.261	2.18	2.226	2.201
		+30%	**+27%**	**+11%**	**+10%**	**+74%**	**+72%**	**+80%**	**+86%**	**+90%**
ICOSPHERE	Single Ray	2.553	1.126	0.818	0.788	0.789	2.971	1.861	1.567	1.598
	RayCrawler	2.553	1.565	1.309	1.288	1.293	2.433	2.342	2.139	2.235
		+0%	**+39%**	**+60%**	**+63%**	**+64%**	**-18%**	**+26%**	**+36%**	**+40%**
SIBENIK	Single Ray	2.493	1.906	1.834	1.823	7.587	3.356	2.971	2.843	2.769
	RayCrawler	3.008	2.54	2.463	2.446	5.089	3.813	3.585	3.526	3.488
		+21%	**+33%**	**+34%**	**+34%**	**-33%**	**+14%**	**+21%**	**+24%**	**+26%**
SAN MIGUEL	Single Ray	0.748	0.605	0.622	0.655	2.696	0.675	0.649	0.598	0.595
	RayCrawler	1.191	0.963	0.92	0.908	2.613	1.394	1.387	1.332	1.311
		+59%	**+59%**	**+48%**	**+39%**	**-3%**	**+107%**	**+114%**	**+123%**	**+120%**
CRYTEK SPONZA	Single Ray	2.366	1.712	1.619	1.565	5.991	2.694	2.362	2.265	2.198
	RayCrawler	2.619	2.197	2.144	2.12	3.924	3.057	2.941	2.904	2.896
		+11%	**+28%**	**+32%**	**+36%**	**-35%**	**+13%**	**+25%**	**+28%**	**+32%**

Table 1: *The traversal speed of our scheme compared to the baseline single ray traversal scheme. The results are presented in absolute numbers in MRay/s, the performance difference between the two algorithms is shown in every third row. The columns 1i, 2i, 3i and 4i correspond to intersection queries at different path depths (e.g. 1i for rays spawned after the first bounce, 2i for rays after the second bounce, etc.). 0o, 1o, 2o, 3o and 4o correspond to occlusion queries at different path depths (e.g. 0o for the shadow rays spawned by the primary rays, etc.).*

6.4. Memory usage

Acceleration structure The total number of nodes in our acceleration structure (i.e. the sum of the nodes in the Top-BVH and all Leaf-BVHs) is the same as the number of nodes in the initial BVH that was used to construct our acceleration structure. Although Top-BVH nodes are larger than regular BVH nodes, the Top-BVH doesn't contribute significantly to the overall memory usage of our acceleration structure, as it is typically very small compared to the combined size of all Leaf-BVHs. Thus we ignore the size of this acceleration structure in our memory usage analysis.

Traversal The amount of memory required during traversal is determined by equation 1.

$$n \leq \left\lceil \frac{r}{b} \right\rceil + l \tag{1}$$

Where n is the number of required buckets, r is the number of rays, b is the bucket capacity and l is the number of

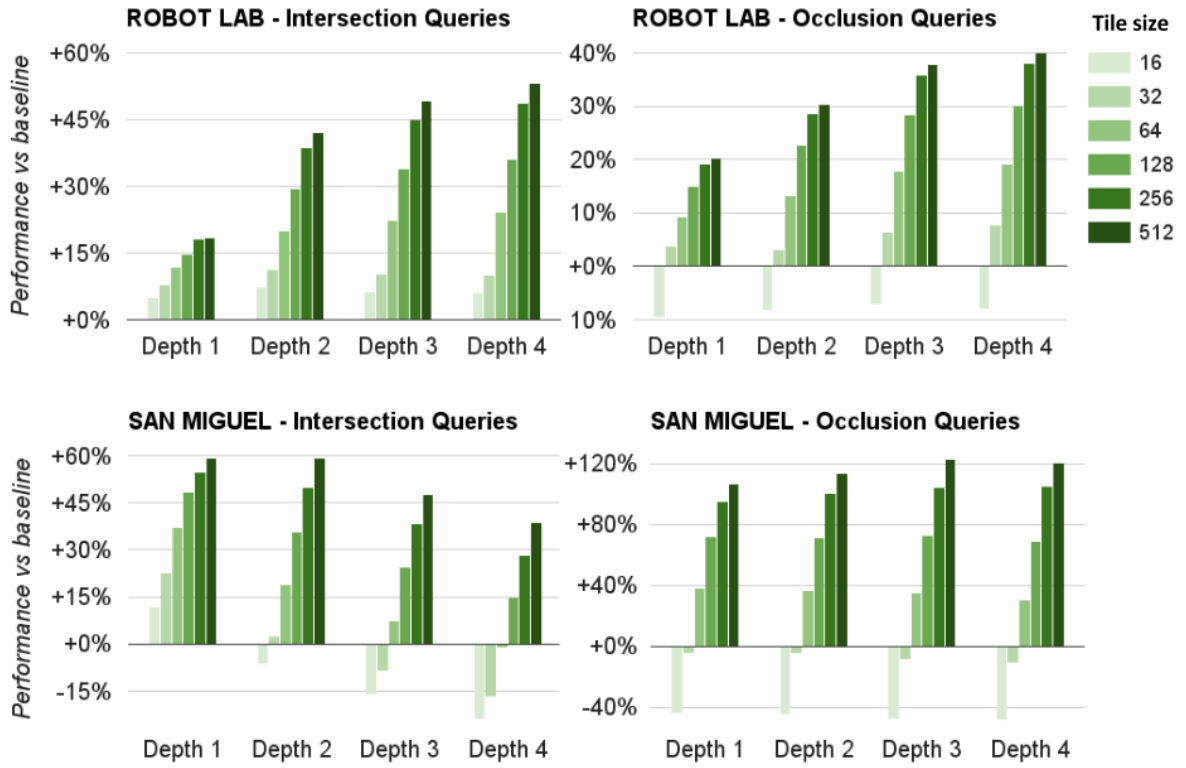


Figure 8: *The traversal performance of our scheme relative to our baseline algorithm for the tile sizes 16x16, 32x32, 64x64, 128x128, 256x256 and 512x512. The number of samples per pixel is always set to 16.*

Scene	Scheme	1i	2i	3i	4i	0o	1o	2o	3o	4o
ROBOT LAB	Single Ray	2.03	1.46	1.37	1.33	8.08	2.77	2.46	2.29	2.23
	RayCrawler (16x16)	+5%	+7%	+6%	+6%	-43%	-10%	-8%	-7%	-8%
	RayCrawler (32x32)	+8%	+11%	+10%	+10%	-42%	+4%	+3%	+6%	+8%
	RayCrawler (64x64)	+12%	+20%	+22%	+24%	-43%	+9%	+13%	+18%	+19%
	RayCrawler (128x128)	+15%	+30%	+34%	+36%	-44%	+15%	+23%	+28%	+30%
	RayCrawler (256x256)	+18%	+39%	+45%	+49%	-44%	+19%	+29%	+36%	+38%
	RayCrawler (512x512)	+19%	+42%	+49%	+53%	-44%	+20%	+30%	+38%	+40%
SAN MIGUEL	Single Ray	0.75	0.6	0.62	0.65	2.7	0.68	0.65	0.6	0.6
	RayCrawler (16x16)	+12%	-6%	-16%	-24%	-10%	-44%	-45%	-48%	-48%
	RayCrawler (32x32)	+23%	+3%	-9%	-17%	-4%	-5%	-5%	-9%	-11%
	RayCrawler (64x64)	+37%	+19%	+7%	-1%	-2%	+38%	+37%	+35%	+31%
	RayCrawler (128x128)	+48%	+36%	+24%	+15%	-2%	+72%	+71%	+73%	+69%
	RayCrawler (256x256)	+55%	+50%	+38%	+28%	-3%	+95%	+101%	+105%	+105%
	RayCrawler (512x512)	+59%	+59%	+48%	+39%	-3%	+107%	+114%	+123%	+120%

Table 2: *The traversal speed of our scheme for varying tile sizes. The results are shown relative to the performance of our baseline single ray traversal scheme. See Table 1 for an explanation of the columns.*

Leaf-BVHs. We have fixed the bucket capacity to 128 rays in our tests. With a ray size of 40 bytes, this results in a size of 5120 bytes per bucket.

The first part of the equation, $\lceil \frac{r}{b} \rceil$, is only dependent of the tile size while the second part is only dependent of the scene. Table 3 shows the required amount of memory for varying tile sizes and Table 4 shows the number of Leaf-BVHs and required memory for all scenes. We see that the number of

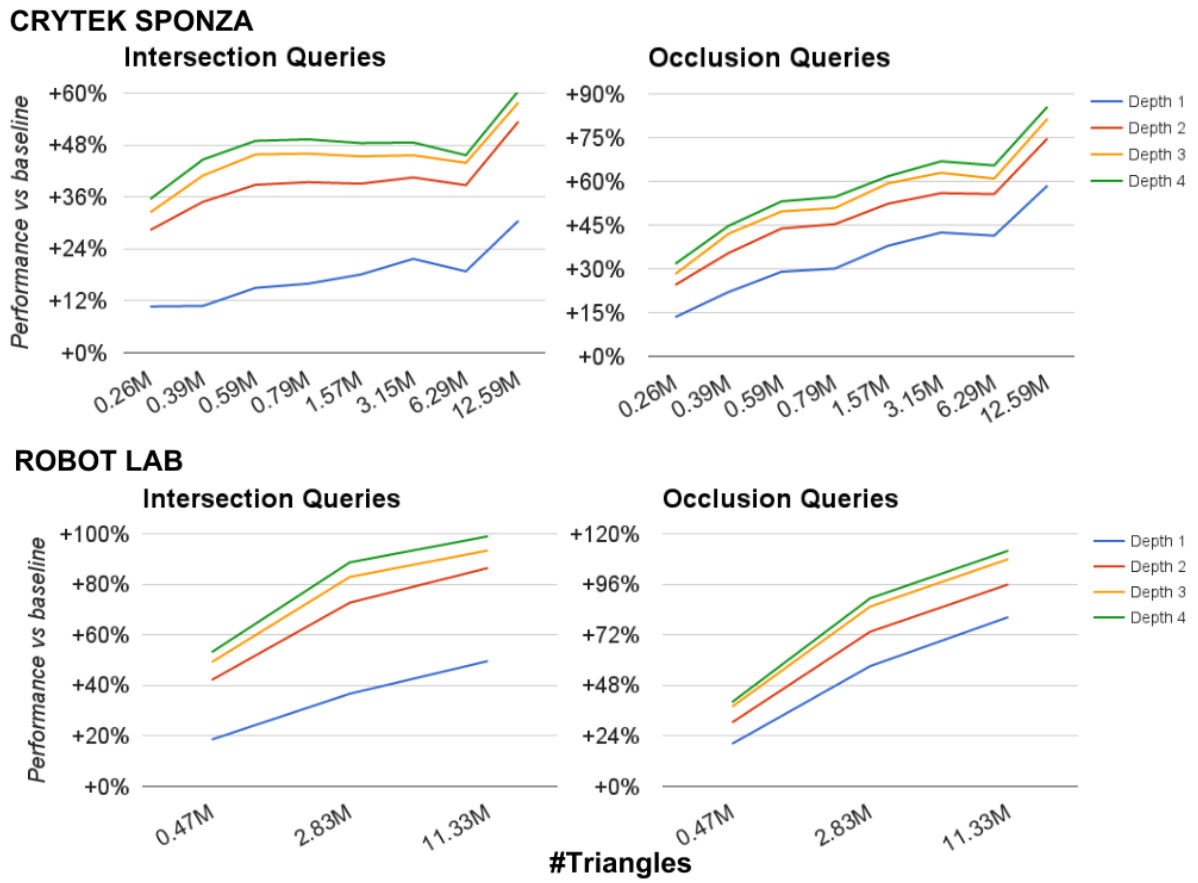


Figure 9: *The relative traversal speed of our scheme compared to the baseline for different path depths as the number of triangles in the scene is increased.*

Leaf-BVHs in the scene only becomes significant with very large scenes.

Tile size	#buckets	Memory (MB)
16	32	0.156
32	128	0.625
64	512	2.5
128	2048	10
256	8192	40
512	32768	160

Table 3: *The number of buckets and amount of memory required for varying tile sizes. Recall that the number of samples per pixel is fixed to 16.*

Scene	#Leaf-BVHs	Memory (KB)
SIBENIK	4	20
ICOSPHERE	10	50
CRYTEK SPONZA	13	65
ROBOT LAB	16	80
HAIRBALL	46	230
SAN MIGUEL	49	245

Table 4: *The number of Leaf-BVHs for all test scenes. One (partially filled) bucket is reserved for every Leaf-BVH.*

7. Conclusion

We presented an efficient ray traversal scheme for highly divergent ray distributions that uses batching at fixed points in the BVH during traversal. We significantly improved the traversal performance for secondary rays compared to the baseline single-ray traversal scheme by up to 99% for intersection queries and 123% for occlusion queries. We showed that a batching scheme can be a viable approach to improve performance for divergent ray traversal. Our scheme is orthogonal to recent advances in divergent ray traversal algo-

rithms and can be combined with these techniques to achieve even higher traversal speeds.

The requirement of a high ray count, as seen in Section 6.2, makes our scheme less suitable for real-time ray tracing applications, where progressive refinement is desired, as these applications typically have a low number of available rays in the system. This makes our scheme more suitable for offline rendering tasks where the number of samples per pixel is typically very high and a single frame can take several hours to render.

For future work, our scheme can be improved in several areas. Leaf-BVH traversal may be improved by using Ordered Ray Stream Traversal instead of our current hybrid approach. Switching from a 4-wide BVH to a BVH2 for the Top-BVH may yield two benefits; splitting a binary tree in two layers will be easier and will result in more optimal splits as we only have to consider two children instead of four. Traversing the Top-BVH will also become simpler as we will not have to worry about the order of the child nodes in the traversal stack. A more thorough investigation on splitting the initial BVH into two layers may also result in a more optimal data structure, as the optimal splitting strategy is scene dependent.

Furthermore, we would like to investigate adapting our scheme for higher SIMD widths. This will however largely depend on the chosen traversal algorithms for the Top- and Leaf-BVHs. It may also be worth investigating the feasibility of our scheme on a GPU.

Given the high performance gains for complex scenes, it may make sense to add an additional level in the BVH hierarchy for extremely large scenes where the individual subtrees no longer fit in the L2 cache.

8. Acknowledgements

Many thanks to the authors of our test scenes Frank Meinl and Marko Dabrovic for CRYTEK SPONZA, Samuli Laine and Tero Karras for HAIRBALL, Guillermo M. Leal Llaguno for SAN MIGUEL, Unity Technologies for ROBOT LAB and Marko Dabrovic for SIBENIK. Also thanks to Valentin Fuetterling for the discussions on possible approaches to integrate ORST into our scheme.

References

[ÁSK14] ÁFRA A. T., SZIRMAY-KALOS L.: Stackless Multi-BVH traversal for CPU, MIC and GPU ray tracing. *Computer Graphics Forum 33*, 1 (2014), 129–140. doi:10.1111/cgf.12259. 4

[BAM14] BARRINGER R., AKENINE-MÖLLER T.: Dynamic ray stream traversal. *ACM Transactions on Graphics (TOG) 33*, 4 (2014), 151. 2

[BEL*07] BOULOS S., EDWARDS D., LACEWELL J. D., KNISS J., KAUTZ J., SHIRLEY P., WALD I.: Packet-based whitted and distribution ray tracing. In *Proceedings of Graphics Interface 2007* (2007), ACM, pp. 177–184. 2

[Bik12] BIKKER J.: Improving data locality for efficient in-core path tracing. In *Computer Graphics Forum* (2012), vol. 31, Wiley Online Library, pp. 1936–1947. 2

[Ble16] BLENDER ONLINE COMMUNITY: *Blender - a 3D modelling and rendering package*. Blender Foundation, Blender Institute, Amsterdam, 2016. URL: http://www.blender.org. 7

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. In *Computer Graphics Forum* (2008), vol. 27, Wiley Online Library, pp. 1225–1233. 2

[EG08] ERNST M., GREINER G.: Multi bounding volume hierarchies. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (2008), IEEE, pp. 35–40. 2

[FLPE15] FUETTERLING V., LOJEWSKI C., PFREUNDT F.-J., EBERT A.: Efficient ray tracing kernels for modern cpu architectures. *Journal of Computer Graphics Techniques Vol 4*, 4 (2015). 2

[FS05] FOLEY T., SUGERMAN J.: Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2005), ACM, pp. 15–22. 4

[HDW*11] HAPALA M., DAVIDOVIČ T., WALD I., HAVRAN V., SLUSALLEK P.: Efficient stack-less bvh traversal for ray tracing. In *Proceedings of the 27th Spring Conference on Computer Graphics* (2011), ACM, pp. 7–12. 4

[Lai10] LAINE S.: Restart trail for stackless bvh traversal. In *Proceedings of the Conference on High Performance Graphics* (2010), Eurographics Association, pp. 107–111. 4

[ORM08] OVERBECK R., RAMAMOORTHI R., MARK W. R.: Large ray packets for real-time whitted ray tracing. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (2008), IEEE, pp. 41–48. 2

[Tsa09] TSAKOK J. A.: Faster incoherent rays: Multi-bvh ray stream tracing. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), ACM, pp. 151–158. 2, 4

[vdZRJ95] VAN DER ZWAAN M., REINHARD E., JANSEN F. W.: Pyramid clipping for efficient ray traversal. In *Rendering TechniquesâĂŹ 95*. Springer, 1995, pp. 1–10. 2

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets-efficient simd single-ray traversal using multi-branching bvhs. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (2008), IEEE, pp. 49–57. 2

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive rendering with coherent ray tracing. In *Computer graphics forum* (2001), vol. 20, Wiley Online Library, pp. 153–165. 2

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: a kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (TOG) 33*, 4 (2014), 143. 2

Appendices

```
1  void RayCrawler::Traverse(ToplevelNode* node, RayInstance&
       ray)
2  {
3    const unsigned __int64 mask = (unsigned __int64( 1 ) << ((
       node->bitPos & 63) + 4)) - 1;
4    unsigned __int64 stack = ray.stack & mask; // slightly
       faster to cache this
5    while (node)
6    {
7      // get traversal bits at current depth
8      const uint bitPos = node->bitPos & 63;
9      const uint bits = (stack >> bitPos) & 15;
10     // find nearest intersection of the ray and the child
         boxes
11     __m128 tmin4, result4;
12     { // scope limiting
13       const __m128 rdx4 = _mm_set_ps1( ray.rdx ), ox4 =
           _mm_set_ps1( ray.ox );
14       const __m128 rdy4 = _mm_set_ps1( ray.rdy ), oy4 =
           _mm_set_ps1( ray.oy );
15       const __m128 rdz4 = _mm_set_ps1( ray.rdz ), oz4 =
           _mm_set_ps1( ray.oz );
16       const __m128 t4 = _mm_set_ps1( ray.t );
17       // remove the bits for now so we can easily add them
           back later
18       stack -= bits << bitPos;
19       // Compute intersection
20       const __m128 t1x = _mm_mul_ps( _mm_sub_ps( node->
           bmin4x, ox4 ), rdx4 );
21       const __m128 t2x = _mm_mul_ps( _mm_sub_ps( node->
           bmax4x, ox4 ), rdx4 );
22       const __m128 t1y = _mm_mul_ps( _mm_sub_ps( node->
           bmin4y, oy4 ), rdy4 );
23       const __m128 t2y = _mm_mul_ps( _mm_sub_ps( node->
           bmax4y, oy4 ), rdy4 );
24       const __m128 tmin4x = _mm_min_ps( t1x, t2x ), txf =
           _mm_max_ps( t1x, t2x );
25       const __m128 t1z = _mm_mul_ps( _mm_sub_ps( node->
           bmin4z, oz4 ), rdz4 );
26       const __m128 tmin1 = _mm_max_ps( eps4, tmin4x );
27       const __m128 tmax1 = _mm_min_ps( t4, txf );
28       const __m128 t2z = _mm_mul_ps( _mm_sub_ps( node->
           bmax4z, oz4 ), rdz4 );
29       const __m128 tyn = _mm_min_ps( t1y, t2y ), tyf =
           _mm_max_ps( t1y, t2y );
30       const __m128 tmin2 = _mm_max_ps( tmin1, tyn ), tmax2 =
            _mm_min_ps( tmax1, tyf );
31       const __m128 tzn = _mm_min_ps( t1z, t2z ), tzf =
           _mm_max_ps( t1z, t2z );
32       tmin4 = _mm_max_ps( tmin2, tzn );
33       result4 = _mm_cmple_ps( tmin4, _mm_min_ps( tmax2, tzf
           ) ); // always valid?
34     }
35     // determine valid childs & write back findings
36     const uint newBits = bits & _mm_movemask_ps( result4 );
37     if (newBits)
38     {
39       // find nearest active child for this ray
40       uint childIdx;
41       { // scope limiting
42         const __m128 v1 = _mm_blendv_ps( inf4, tmin4,
             bitMask[newBits] );
43         const __m128 v2 = _mm_and_ps( v1, idxmask4 );
44         const __m128 v3 = _mm_or_ps( v2, idxadd4 );
45         const float smallest = hor_min( v3 );
46         childIdx = (reinterpret_cast<const unsigned int&>(
             smallest)) & 3;
47       }
48       // this ray will now visit the nearest child
49       ToplevelNode* child = &pool[node->data + childIdx];
50       stack += (15 << (bitPos + 4)) + ((newBits - (1 <<
             childIdx)) << bitPos);
51       if (likely( !child->treelet )) { node = child;
           continue; }
52       // ray reached treelet level; add to ray container for
           postponed batch processing
53       if (likely( child->room >> 16 ))
54       {
55         // early in; we know we have room
56         RayBucket* bucket = (RayBucket*)child->buckets.
             PeekLast(); // guaranteed to have room
57         const uint nextIdx = bucket->rays++;
58         ray.stack = stack;
59         child->room -= 65536;
60         if (unlikely( !(child->room >> 16) ))
61         {
62           const uint idx = ((child->bucketCount >> 8) & 255)
               - 1;
63           if (idx < 2)
64           {
65             // move this node from 'nodesWithPartialBucket'
                 to 'nodesWithFullBucket'
66             child->Detach();
67             nodesWithBucket[idx].Add( child );
68           }
69         }
70         bucket->ray[nextIdx] = ray;
71       }
72       else
73       {
74         RayBucket* newBucket = (RayBucket*)emptyBuckets.
             PopFirst();
75         newBucket->rays = 1;
76         newBucket->node = child;
77         if (child->buckets.Empty()) // no buckets at all;
             make room
78         {
79           child->bucketCount = (child->bitPos & 0xff00ff) +
               256;
80           nodesWithPartialBucket.Add( child );
81         }
82         else child->bucketCount += 256; // we have buckets,
             but no partial filled buckets; make room
83         child->buckets.AddAtEnd( newBucket );
84         ray.stack = stack;
85         newBucket->ray[0] = ray;
86         child->room = ((BUCKETSIZE - 1) << 16) + (child->
             bitPos & 65535);
87       }
88       return;
89     }
90     // this ray has no children left to visit; find the
         first ancestor that has work left
91     { // scope limiting
92       unsigned long index;
93       const __int64 mask = (unsigned __int64( 1 ) << bitPos)
             - 1;
94       if (!(stack & mask)) break;
95       _BitScanReverse64( &index, stack & mask );
96       node = &pool[node->ancestor[index >> 2]];
97     }
98   }
99 }
```

Listing 2: *C++ code of the Top-BVH traversal algorithm*

```
1  template<bool shadow> void Node4::TraceEmbree(const vec3& O,
       const vec3& D, float& t, float& u, float& v, uint&
       triIdx ) const
2  {
3    Node4* pool4 = Scene::mbvh->pool4;
4    // stack state
5    __declspec( align( 64 ) ) StackItem stack[64], *stackPtr =
         stack + 1;
6    stack[0].ptr = (uint)(this - Scene::mbvh->pool4);
7    stack[0].dist = std::numeric_limits<uint>::min();
```

```
// load the ray into SIMD registers
const __m128 ray_rdir = rcp_safe( _mm_set_ps( D.x, D.y, D.z, D.z ) );
const __m128 org_x = _mm_set_ps1( O.x ), org_y = _mm_set_ps1( O.y ), org_z = _mm_set_ps1( O.z );
const __m128 rdir_x = _mm_set_ps1( ((float*)&ray_rdir)[3] );
const __m128 rdir_y = _mm_set_ps1( ((float*)&ray_rdir)[2] );
const __m128 rdir_z = _mm_set_ps1( ((float*)&ray_rdir)[1] );
const __m128 ray_near = _mm_setzero_ps();
const __m128 ray_far = _mm_set_ps1( t );
// offsets to select the side that becomes the lower or upper bound
const int nearX = ((float*)&ray_rdir)[3] >= 0.0f ? 0 : 16;
const int nearY = ((float*)&ray_rdir)[2] >= 0.0f ? 32 : 48;
const int nearZ = ((float*)&ray_rdir)[1] >= 0.0f ? 64 : 80;
// pop loop
while (true) pop:
{
  // pop next node
  if (stackPtr == stack) break;
  stackPtr--;
  uint cur = stackPtr->ptr;
  // if popped node is too far, pop next one
  if (unlikely(*(float*)&stackPtr->dist > t)) continue;
  // downtraversal loop
  while (true)
  {
    __m128 tNear;
    // stop if we found a leaf node
    if (unlikely((cur & (1 << 31)) != 0)) break;
    const Node4* ptr = &pool4[cur];
    const int farX = nearX ^ 16, farY = nearY ^ 16, farZ = nearZ ^ 16;
    const __m128 tNearX = _mm_mul_ps( _mm_sub_ps( _mm_load_ps( (const float*)((const char*)&ptr->bmin4x + nearX) ), org_x), rdir_x );
    const __m128 tNearY = _mm_mul_ps( _mm_sub_ps( _mm_load_ps( (const float*)((const char*)&ptr->bmin4x + nearY) ), org_y), rdir_y );
    const __m128 tNearZ = _mm_mul_ps( _mm_sub_ps( _mm_load_ps( (const float*)((const char*)&ptr->bmin4x + nearZ) ), org_z), rdir_z );
    const __m128 tFarX  = _mm_mul_ps( _mm_sub_ps( _mm_load_ps( (const float*)((const char*)&ptr->bmin4x + farX) ), org_x), rdir_x );
    const __m128 tFarY  = _mm_mul_ps( _mm_sub_ps( _mm_load_ps( (const float*)((const char*)&ptr->bmin4x + farY) ), org_y), rdir_y );
    const __m128 tFarZ  = _mm_mul_ps( _mm_sub_ps( _mm_load_ps( (const float*)((const char*)&ptr->bmin4x + farZ) ), org_z), rdir_z );
    tNear = _mm_max_ps( _mm_max_ps( tNearX, tNearY ), _mm_max_ps( tNearZ, ray_near ) );
    const __m128 tFar = _mm_min_ps( _mm_min_ps( tFarX, tFarY ), _mm_min_ps( tFarZ, ray_far ) );
    const __m128 vmask = _mm_cmple_ps( tNear, tFar );
    const int mask = _mm_movemask_ps( vmask );
    // if no child is hit, pop next node
    switch (mask)
    {
    case 0:
      goto pop;
    case 1:
      cur = ptr->data[0];
      continue;
    case 2:
      cur = ptr->data[1];
      continue;
    case 3:
      {
        const uint c0 = ptr->data[0], d0 = ((uint*)&tNear)[0], c1 = ptr->data[1], d1 = ((uint*)&tNear)[1];
        if (shadow || d0 < d1) { stackPtr->ptr = c1, stackPtr->dist = d1, stackPtr++, cur = c0; }
        else { stackPtr->ptr = c0, stackPtr->dist = d0, stackPtr++, cur = c1; }
        continue;
      }
    case 4:
      cur = ptr->data[2];
      continue;
    case 5:
      {
        const uint c0 = ptr->data[0], d0 = ((uint*)&tNear)[0], c1 = ptr->data[2], d1 = ((uint*)&tNear)[2];
        if (shadow || d0 < d1) { stackPtr->ptr = c1, stackPtr->dist = d1, stackPtr++, cur = c0; }
        else { stackPtr->ptr = c0, stackPtr->dist = d0, stackPtr++, cur = c1; }
        continue;
      }
    case 6:
      {
        const uint c0 = ptr->data[1], d0 = ((uint*)&tNear)[1], c1 = ptr->data[2], d1 = ((uint*)&tNear)[2];
        if (shadow || d0 < d1) { stackPtr->ptr = c1, stackPtr->dist = d1, stackPtr++, cur = c0; }
        else { stackPtr->ptr = c0, stackPtr->dist = d0, stackPtr++, cur = c1; }
        continue;
      }
    case 7:
      {
        // Push all nodes on the stack
        stackPtr[0].ptr = ptr->data[0]; stackPtr[0].dist = ((uint*)&tNear)[0];
        stackPtr[1].ptr = ptr->data[1]; stackPtr[1].dist = ((uint*)&tNear)[1];
        stackPtr[2].ptr = ptr->data[2]; stackPtr[2].dist = ((uint*)&tNear)[2];
        stackPtr += 2;
        // Sort them
        if (!shadow) sort(stackPtr[0], stackPtr[-1], stackPtr[-2]);
        // Pop top element
        cur = stackPtr->ptr;
        continue;
      }
    case 8:
      cur = ptr->data[3];
      continue;
    case 9:
      {
        const uint c0 = ptr->data[0], d0 = ((uint*)&tNear)[0], c1 = ptr->data[3], d1 = ((uint*)&tNear)[3];
        if (shadow || d0 < d1) { stackPtr->ptr = c1, stackPtr->dist = d1, stackPtr++, cur = c0; }
        else { stackPtr->ptr = c0, stackPtr->dist = d0, stackPtr++, cur = c1; }
        continue;
      }
    case 10:
      {
        const uint c0 = ptr->data[1], d0 = ((uint*)&tNear)[1], c1 = ptr->data[3], d1 = ((uint*)&tNear)[3];
        if (shadow || d0 < d1) { stackPtr->ptr = c1, stackPtr->dist = d1, stackPtr++, cur = c0; }
        else { stackPtr->ptr = c0, stackPtr->dist = d0, stackPtr++, cur = c1; }
        continue;
      }
    case 11:
```

```
113         {
114           // Push all nodes on the stack
115           stackPtr[0].ptr = ptr->data[0]; stackPtr[0].dist =
                  ((uint*)&tNear)[0];
116           stackPtr[1].ptr = ptr->data[1]; stackPtr[1].dist =
                  ((uint*)&tNear)[1];
117           stackPtr[2].ptr = ptr->data[3]; stackPtr[2].dist =
                  ((uint*)&tNear)[3];
118           stackPtr += 2;
119           // Sort them
120           if (!shadow) sort(stackPtr[0], stackPtr[-1],
                  stackPtr[-2]);
121           // Pop top element
122           cur = stackPtr->ptr;
123           continue;
124         }
125       case 12:
126         {
127           const uint c0 = ptr->data[2], d0 = ((uint*)&tNear)
                  [2], c1 = ptr->data[3], d1 = ((uint*)&tNear)
                  [3];
128           if (shadow || d0 < d1) { stackPtr->ptr = c1,
                  stackPtr->dist = d1, stackPtr++, cur = c0; }
129           else { stackPtr->ptr = c0, stackPtr->dist = d0,
                  stackPtr++, cur = c1; }
130           continue;
131         }
132       case 13:
133         {
134           // Push all nodes on the stack
135           stackPtr[0].ptr = ptr->data[0]; stackPtr[0].dist =
                  ((uint*)&tNear)[0];
136           stackPtr[1].ptr = ptr->data[2]; stackPtr[1].dist =
                  ((uint*)&tNear)[2];
137           stackPtr[2].ptr = ptr->data[3]; stackPtr[2].dist =
                  ((uint*)&tNear)[3];
138           stackPtr += 2;
139           // Sort them
140           if (!shadow) sort(stackPtr[0], stackPtr[-1],
                  stackPtr[-2]);
141           // Pop top element
142           cur = stackPtr->ptr;
143           continue;
144         }
145       case 14:
146         {
147           // Push all nodes on the stack
148           stackPtr[0].ptr = ptr->data[1]; stackPtr[0].dist =
                  ((uint*)&tNear)[1];
149           stackPtr[1].ptr = ptr->data[2]; stackPtr[1].dist =
                  ((uint*)&tNear)[2];
150           stackPtr[2].ptr = ptr->data[3]; stackPtr[2].dist =
                  ((uint*)&tNear)[3];
151           stackPtr += 2;
152           // Sort them
153           if(!shadow) sort(stackPtr[0], stackPtr[-1],
                  stackPtr[-2]);
154           // Pop top element
155           cur = stackPtr->ptr;
156           continue;
157         }
158       case 15:
159         {
160           // Push all nodes on the stack
161           stackPtr[0].ptr = ptr->data[0]; stackPtr[0].dist =
                  ((uint*)&tNear)[0];
162           stackPtr[1].ptr = ptr->data[1]; stackPtr[1].dist =
                  ((uint*)&tNear)[1];
163           stackPtr[2].ptr = ptr->data[2]; stackPtr[2].dist =
                  ((uint*)&tNear)[2];
164           stackPtr[3].ptr = ptr->data[3]; stackPtr[3].dist =
                  ((uint*)&tNear)[3];
165           stackPtr += 3;
166           // Sort them
167           if(!shadow) sort(stackPtr[0], stackPtr[-1],
                  stackPtr[-2], stackPtr[-3]);
168           // Pop top element
169           cur = stackPtr->ptr;
170           continue;
171         }
172       #ifdef _MSC_VER
173       default:
174           __assume(0); // see https://msdn.microsoft.com/
                  en-us/library/1b3fsfxw(VS.80).aspx
175       #endif
176       }
177     }
178     // this is a leaf node
179     if (shadow)
180     {
181       if (Intersect((cur & 0x7fffffff) >> 6, cur & 63, O, D,
              t))
182       {
183         triIdx = 1;
184         return;
185       }
186     }
187     else
188     {
189       Intersect((cur & 0x7fffffff) >> 6, cur & 63, O, D, t,
              u, v, triIdx);
190     }
191   }
192 }
```

Listing 3: *C++ code of the baseline SingleRay traversal algorithm*

Scene	#Triangles	Scheme	1i	2i	3i	4i	0o	1o	2o	3o	4o
CRYTEK SPONZA	0.262M	Single Ray	2.366	1.712	1.619	1.565	5.991	2.694	2.362	2.265	2.198
		RayCrawler	2.619	2.197	2.144	2.12	3.924	3.057	2.941	2.904	2.896
			+11%	**+28%**	**+32%**	**+36%**	**-35%**	**+13%**	**+25%**	**+28%**	**+32%**
	0.393M	Single Ray	2.063	1.466	1.374	1.33	5.654	2.401	2.064	1.952	1.897
		RayCrawler	2.286	1.977	1.937	1.923	3.659	2.931	2.795	2.773	2.745
			+11%	**+35%**	**+41%**	**+45%**	**-35%**	**+22%**	**+35%**	**+42%**	**+45%**
	0.589M	Single Ray	1.908	1.347	1.254	1.215	5.42	2.198	1.878	1.787	1.728
		RayCrawler	2.193	1.87	1.829	1.81	3.636	2.835	2.701	2.674	2.646
			+15%	**+39%**	**+46%**	**+49%**	**-33%**	**+29%**	**+44%**	**+50%**	**+53%**
	0.786M	Single Ray	1.832	1.289	1.2	1.163	5.314	2.11	1.796	1.707	1.649
		RayCrawler	2.124	1.796	1.751	1.737	3.595	2.744	2.609	2.574	2.549
			+16%	**+39%**	**+46%**	**+49%**	**-32%**	**+30%**	**+45%**	**+51%**	**+55%**
	1.573M	Single Ray	1.649	1.144	1.062	1.029	4.884	1.816	1.552	1.462	1.42
		RayCrawler	1.947	1.591	1.544	1.527	3.469	2.504	2.364	2.329	2.296
			+18%	**+39%**	**+45%**	**+48%**	**-29%**	**+38%**	**+52%**	**+59%**	**+62%**
	3.146M	Single Ray	1.424	0.977	0.908	0.877	4.51	1.575	1.34	1.263	1.216
		RayCrawler	1.732	1.373	1.322	1.303	3.307	2.244	2.09	2.058	2.03
			+22%	**+40%**	**+46%**	**+49%**	**-27%**	**+42%**	**+56%**	**+63%**	**+67%**
	6.293M	Single Ray	1.327	0.899	0.832	0.807	4.236	1.399	1.188	1.13	1.087
		RayCrawler	1.576	1.247	1.197	1.175	2.952	1.977	1.849	1.819	1.799
			+19%	**+39%**	**+44%**	**+46%**	**-30%**	**+41%**	**+56%**	**+61%**	**+65%**
	12.585M	Single Ray	1.066	0.723	0.671	0.652	3.611	1.114	0.946	0.9	0.871
		RayCrawler	1.391	1.109	1.059	1.047	2.702	1.765	1.651	1.633	1.615
			+30%	**+53%**	**+58%**	**+61%**	**-25%**	**+59%**	**+75%**	**+81%**	**+86%**
ROBOT LAB	0.472M	Single Ray	2.029	1.465	1.371	1.33	8.076	2.771	2.455	2.288	2.234
		RayCrawler	2.406	2.082	2.045	2.036	4.486	3.332	3.201	3.155	3.128
			+19%	**+42%**	**+49%**	**+53%**	**-44%**	**+20%**	**+30%**	**+38%**	**+40%**
	2.833M	Single Ray	1.317	0.9	0.832	0.803	6.137	1.736	1.524	1.396	1.362
		RayCrawler	1.8	1.554	1.523	1.516	3.818	2.728	2.643	2.589	2.579
			+37%	**+73%**	**+83%**	**+89%**	**-38%**	**+57%**	**+73%**	**+85%**	**+89%**
	11.334M	Single Ray	0.923	0.628	0.589	0.569	4.826	1.247	1.105	1.02	0.997
		RayCrawler	1.382	1.17	1.14	1.132	3.282	2.25	2.166	2.121	2.112
			+50%	**+86%**	**+93%**	**+99%**	**-32%**	**+80%**	**+96%**	**+108%**	**+112%**

Table 5: *Evaluation of the performance of our scheme compared to the baseline algorithm for different subdivisions of the CRYTEK SPONZA and ROBOT LAB scenes. See Table 1 for an explanation of the columns.*

	Single Ray	RayCrawler, 64x64	RayCrawler, 128x128	RayCrawler, 256x256	RayCrawler, 512x512
intersection 0	4.103	4.183	4.090	4.102	4.081
intersection 1	2.029	2.272	2.330	2.398	2.406
intersection 2	1.465	1.758	1.898	2.033	2.082
intersection 3	1.371	1.675	1.836	1.986	2.045
intersection 4	1.330	1.650	1.810	1.976	2.036
occlusion 0	8.076	4.630	4.501	4.494	4.486
occlusion 1	2.771	3.030	3.186	3.304	3.332
occlusion 2	2.455	2.778	3.010	3.155	3.201
occlusion 3	2.288	2.699	2.939	3.106	3.155
occlusion 4	2.234	2.662	2.908	3.081	3.128

Table 6: *Results in MRay/s for the ROBOT LAB scene for all tile sizes and depths.*

	Single Ray	RayCrawler, 64x64	RayCrawler, 128x128	RayCrawler, 256x256	RayCrawler, 512x512
intersection 0	2.055	2.081	2.084	2.088	2.067
intersection 1	0.748	1.027	1.110	1.158	1.191
intersection 2	0.605	0.720	0.821	0.906	0.963
intersection 3	0.622	0.668	0.774	0.862	0.920
intersection 4	0.655	0.646	0.752	0.841	0.908
occlusion 0	2.696	2.629	2.639	2.615	2.613
occlusion 1	0.675	0.932	1.159	1.318	1.394
occlusion 2	0.649	0.889	1.109	1.302	1.387
occlusion 3	0.598	0.807	1.035	1.224	1.332
occlusion 4	0.595	0.779	1.005	1.222	1.311

Table 7: *Results in MRay/s for the SAN MIGUEL scene for all tile sizes and depths.*

	Single Ray	RayCrawler, 128x128	RayCrawler, 256x256	RayCrawler, 512x512
intersection 0	2.408	2.424	2.418	2.412
intersection 1	1.239	1.601	1.618	1.615
intersection 2	1.280	1.662	1.631	1.622
intersection 3	1.346	1.534	1.496	1.496
intersection 4	1.347	1.519	1.486	1.488
occlusion 0	1.303	2.213	2.224	2.268
occlusion 1	1.316	2.262	2.305	2.261
occlusion 2	1.212	2.223	2.224	2.180
occlusion 3	1.198	2.242	2.212	2.226
occlusion 4	1.160	2.149	2.191	2.201

Table 8: *Results in MRay/s for the HAIRBALL scene for all tile sizes and depths.*

	Single Ray	RayCrawler, 128x128	RayCrawler, 256x256	RayCrawler, 512x512
intersection 0	4.529	4.579	4.572	4.484
intersection 1	2.493	3.047	3.038	3.008
intersection 2	1.906	2.497	2.532	2.540
intersection 3	1.834	2.400	2.445	2.463
intersection 4	1.823	2.379	2.426	2.446
occlusion 0	7.587	5.143	5.106	5.089
occlusion 1	3.356	3.811	3.831	3.813
occlusion 2	2.971	3.544	3.590	3.585
occlusion 3	2.843	3.468	3.522	3.526
occlusion 4	2.769	3.425	3.485	3.488

Table 9: *Results in MRay/s for the SIBENIK scene for all tile sizes and depths.*

	Single Ray	RayCrawler, 128x128	RayCrawler, 256x256	RayCrawler, 512x512
intersection 0	3.497	3.438	3.449	3.510
intersection 1	2.227	2.623	2.651	2.510
intersection 2	1.622	2.109	2.186	2.139
intersection 3	1.529	2.020	2.118	2.089
intersection 4	1.491	1.983	2.096	2.074
occlusion 0	6.010	3.971	3.961	3.824
occlusion 1	2.683	3.008	3.061	3.107
occlusion 2	2.328	2.833	2.931	2.969
occlusion 3	2.233	2.784	2.899	2.945
occlusion 4	2.166	2.770	2.880	2.915

Table 10: *Results in MRay/s for the CRYTEK SPONZA scene for all tile sizes and depths.*