

Ray Classification for Accelerated BVH Traversal

J. Hendrich, A. Pospíšil, D. Meister, J. Bittner

Czech Technical University in Prague, Faculty of Electrical Engineering

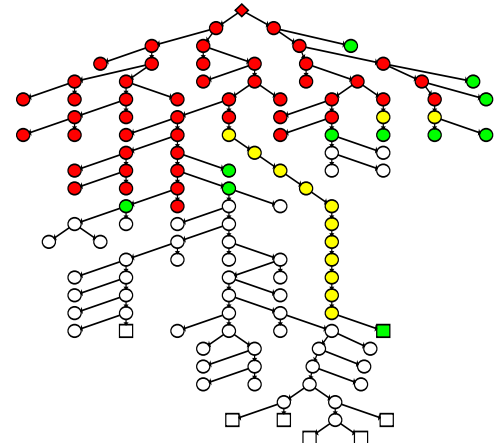


Figure 1: The San Miguel scene and a sample ray corresponding to the central pixel (left), the visualization of tracing the ray using the standard BVH traversal and our method (right). The red nodes are visited only by the standard traversal from the root. The green nodes are entry points of our traversal algorithm. The yellow nodes are visited by neither traversal method and only denote the path to candidate list elements. The white nodes are visited by both types of traversal. In this particular case, our method skips 44% of nodes, both on the path from the BVH root and in lateral branches.

Abstract

For ray tracing based methods, traversing a hierarchical acceleration data structure takes up a substantial portion of the total rendering time. We propose an additional data structure which cuts off large parts of the hierarchical traversal. We use the idea of ray classification combined with the hierarchical scene representation provided by a bounding volume hierarchy. We precompute short arrays of indices to subtrees inside the hierarchy and use them to initiate the traversal for a given ray class. This arrangement is compact enough to be cache-friendly, preventing the method from negating its traversal gains by excessive memory traffic. The method is easy to use with existing renderers which we demonstrate by integrating it to the PBRT renderer. The proposed technique reduces the number of traversal steps by 42% on average, saving around 15% of time of finding ray-scene intersection on average.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Raytracing—I.3.5 [Computer Graphics]: Object Hierarchies—I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—

1. Introduction

Bounding volume hierarchies (BVHs) provide an efficient means of organizing the spatial distribution of a 3D scene. In the context of ray tracing, they are used to determine quickly which parts of the scene are possibly hit by a ray. Many efforts have pushed the performance of BVHs forward [SFD09; Wal12; KA13; GHFB13; GBDA15], which offer fast build/update/query operations, low memory demands, and ease of implementation. Some

techniques are tailored for specific scenarios, such as rendering dynamic scenes [WBS03; LYMT06; WBS07; PL10].

Most efficient BVH methods use the Surface Area Heuristic (SAH) [GS87] during the BVH construction to determine efficient hierarchical object partitioning. This heuristic is however only a static guidance which does not take into account the nature of subsequent ray queries. The question is how to leverage the known ray's properties for faster traversal through a BVH.

Depending on the scene structure, the expected ray distribution, and other possible influences taken into account, the hierarchy is about $\log n$ deep on average, which is also proportional to the number of traversal steps needed to descend from the tree root (representing the whole scene) to the leaves, where the scene geometry is referenced. To find the intersection, the traversal typically visits multiple leaf nodes and thus it branches into several subpaths. There are millions of elements in contemporary scenes, resulting in trees where the distance from the root to the leaves is in the order of several tens of nodes. Combined with millions of rays cast, this takes a considerable amount of time spent by the traversal, so any optimization here has the potential of making a significant impact on the total rendering time.

In this paper, we use the concept of *frustum shafts* as a tool to quickly determine which parts of the scene geometry can be potentially intersected by a given ray. While a BVH organizes scene regions hierarchically into AABBs, shafts are topologically similar to rays, so the volume of a shaft contains those scene parts which are relevant for the corresponding rays. For each frustum shaft we precompute a short list of BVH nodes that are used as entry points for the ray traversal. We describe the necessary criteria for computing these lists so that the ray traversal skips large parts of the root-leaf traversal sequences, thus saving a lot of traversal steps (see Figure 1).

Our method builds on the ray classification idea that has a long history [AK87]. However, until now the technique has been considered impractical for contemporary rendering techniques. We propose a new algorithm which connects ray classification (direct lookup) with contemporary BVHs (hierarchical traversal), demonstrating that an efficient combination is possible.

2. Related Work

In their classical paper, Arvo and Kirk [AK87] proposed the notion of 5D space of rays (three spatial and two directional dimensions) and its subdivision into beams which are assigned only a limited set of candidate objects. A ray is first classified as belonging into one of these beams, followed by intersecting only the relevant candidate objects. Similar ideas gained much attention in other contexts and under different names, where the beams were often referred to as shafts or frusta.

Haines and Wallace [HW94] came up with the idea of shaft culling. They decrease the time spent on determining the mutual visibility of two surfaces in the radiosity method by generating a list of possible occluders between these surfaces. Bala et al. [BDT99] use shaft-culling to (1) accelerate shading: to find occluders that may introduce discontinuities in radiance interpolation; (2) accelerate visibility: exploiting temporal coherence by reprojecting already existing radiance samples, assuming they have not become occluded in the current viewpoint.

When constructing local illumination environments, Fernandez et al. [FBPG00] determine the blocker list between two objects lazily and iteratively only by sampling the shaft connecting them. This approach may miss some blockers at first, but over time, with more samples taken, it converges to the correct state with the guarantee of maintaining the minimal set of blockers. Schaufler et al.

[SDDS00] construct the shafts not to query the space occlusion between two elements but rather behind an occluder with respect to the viewing point. They attempt to perform a fusion of multiple occluders into one, possibly incorporating occluded empty space as well. Bittner [Bit02] uses shafts to optimize the computation of from-region visibility. For Brière and Poulin [BP96], shafts (here *sections*) are the building blocks of a ray-tree structure for quickly determining what has changed in a scene in order to rerender only the necessary parts of it.

Dmitriev et al. [DHS04] group rays into pyramidal shafts, which are then traversed simultaneously using SIMD instructions. Contrary to the approach of Haines and Wallace, they use kd-trees and boundary/extremal rays instead of BVHs and plane testing. Havran and Bittner [HB00] introduced a technique for efficient traversal of rays within a given shaft: the initial empty subpath common to all the rays can be skipped safely. A simpler version of this technique involves only the leaf nodes traversal. Havran et al. [HBŽ98] offer additional means for traversing a hierarchical spatial structure, which they call the ropes (generalized to rope trees). Ropes introduce other paths within the tree to reach a leaf node, which do not start from the root (i.e., climbing on a rope from a branch to another), successfully exploiting the coherence of rays.

Van der Zwaan et al. [vdZRJ95] construct pyramids for coherent rays with the same origin (e.g., primary rays or shadow rays to area light sources) instead of general shafts; then they classify the nodes of a spatial *bintree* as inside or outside a pyramid using a variant of the Cohen-Sutherland clipping algorithm. Brière and Poulin [BP01] enclose light beams within the shafts. As a side effect, they are able to squeeze down the memory used by the shafts drastically while only slightly affecting the rendering times. Reshetov et al. [RSH05] group the primary and shadow rays into a hierarchy of beams, effectively forming a frustum. This allows to start tracing all participating rays simultaneously from a node deep within a kd-tree representing the whole scene. The depth of the starting node depends on the chosen level within the beam hierarchy – the narrower beam, the deeper the starting node could lie. Following this, Wald et al. [WBS07] apply the frustum traversal to BVHs with the addition of applying it on deformable scenes.

Schröder and Drucker [SD92] build a candidate list for a ray intersection as a union of lists contained in precomputed voxelization of the scene. More recently, Keul et al. [KML16] precompute visibility in a scene using shaft culling, which is then stored in a line space structure. This data structure then terminates the traversal of a ray through a regular *n-tree* once it is clear that the ray cannot hit the currently traversed subnode content. Recently, Müller et al. [MGN17] used a 5D tree structure to guide the Monte Carlo process into highly contributing pathspace regions. They use a binary tree spatially partitioning the 3D scene, which references 2D directional partitioning quadrees from its leaf nodes.

Our method takes advantage of the classical ideas of ray classification [AK87; MGN17], shaft culling [HW94; RSH05], and scene structuring using BVHs [GS87; WBS07; GBDA15]. This combination offers substantial performance gain in ray traversal while providing control over the memory consumption as well as seamless integration of the method into existing ray tracing implementations.

3. Efficient BVH Traversal

In this section, we discuss in detail how the standard traversal of a BVH progresses down the tree, and identify its weak spots. Then we briefly introduce the idea of culling the scene geometry with shafts, which is able to cut the traversal costs substantially.

3.1. Standard Traversal

Having a spatial index built above the geometry of a scene, we need to traverse it to determine the nearest intersection for primary or secondary rays, or any intersection for shadow rays, which test visibility between a surface and a light source. This is usually done by descending the tree. We start with pushing the root node id onto the stack; the topmost stack entry is then popped repeatedly to be processed further. If it represents a leaf node, the referenced geometry is checked for an intersection; for inner nodes, the bounding boxes of its children are checked for an intersection. All the children of an inner node which have been hit by the ray (if any) are pushed back onto the stack, usually after depth-sorting them along the ray for efficiency. The running closest intersection with geometry is maintained for primary and secondary rays throughout the traversal, effectively pruning the search tree even more.

In scenes with large and complex geometry, the path length from the root to the geometry elements amounts in tens of traversal steps. The actual traversal is even more complex as it visits many lateral branches, which in most cases do not advance the results of the search. This is because the hierarchical index is a general spatial structure with no information about the properties of the rays being traced in the scene; it does not take into account the restricted part of the scene only relevant to a given ray.

3.2. Algorithm Overview

Our method assumes there is already a BVH built which organizes the scene geometry. We construct a set of convex *frustum shafts*, which extend from a scene voxel in a (narrow) interval of directions. The voxels are elements of a 3D regular grid and the direction intervals are constructed on a regular 2D grid subdividing the space of directions. For each shaft, we identify the parts of the BVH which are intersected by its volume and represent these subtrees as a short array of indices to them (a *candidate list*). The sorted list associated with a shaft thus represents a combination of the hierarchical nature of the BVH and the directionality of the shaft, as shown in Figure 2.

The tracing of a ray starts with identifying the shaft which contains the ray. The traversal algorithm is then seeded with the node ids in the associated candidate list. With these ids pushed onto stack, the traversal proceeds as usual. As the list contains only a very limited subset of the scene geometry, the traversal process does not have to take unnecessary steps through the usual initial parts. Most importantly, it does not have to roam through many lateral subtrees, thus saving significant effort both in terms of traversal steps and memory traffic.

The light transport in the scene is usually distributed very unevenly and there are many shafts which do not contain a significant number of rays. The construction can be therefore guided by the

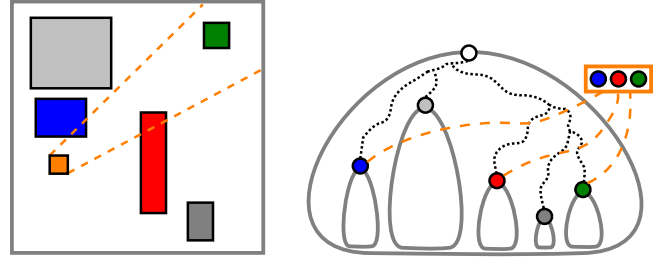


Figure 2: A simple scene with one selected frustum shaft (orange box and dashed lines) and several intersected bounding volumes (left) and corresponding situation in the associated candidate list (orange array of node ids) referring to base BVH (right). The shaft contains only a small subset of the scene geometry, which can be represented by a few nodes inside the BVH. The traversal of nodes above them (dotted lines) and in many parallel branches not intersecting the shaft (grey subtrees) can be skipped altogether.

result of an optional preprocessing step, in which we sample the ray distribution in the scene. Only those shafts which hold the most rays will have their candidate lists built in the end.

4. Shaft Culling in BVH Traversal

Here we describe and discuss the proposed method in detail, starting with the underlying data structures and their setup followed by their usage during the ray tracing phase.

4.1. Collection of Shafts

We subdivide the scene AABB into a regular 3D grid, where every single voxel forms a base for individual frustum shafts and groups together the origins of contained rays (see Figure 3). For each voxel, we subdivide the ray directions into the six major intervals of directions corresponding to standard cube mapping. Each face of the directional cube map is further subdivided into a regular 2D grid.

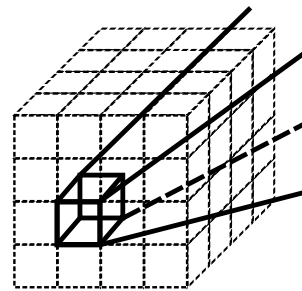


Figure 3: Spatial subdivision using a regular grid with a sample frustum shaft originating at the selected voxel. In most scenes, the grid resolution differs among the three axes in order to form cube-like voxels.

There are different ways to set up the spatial and directional resolutions. It turns out a single spatial resolution in all three axes does not often fit well, especially in scenes with largely non-cubic proportions; so we allow all the three axes to be divided into a different

number of equal intervals. On the other hand, for most scenes there are no substantially dominant directions of rays (or at least they are not known in advance), so we use uniform resolution for directional subdivision.

To easily determine the spatial grid resolution, a convenient option is to derive it automatically from the allowed memory budget for the shaft collection (given either in absolute amount or relative to the memory occupied by the scene geometry). We force the voxels to be as close to cube shape as possible, which aligns well with the SAH preference for cube-like nodes. These soft-constraints are limiting the useful values for resolution from both sides: the shafts should not be too wide, in which case they do not cut off much of the scene geometry and thus do not save much traversal costs. They should not be too thin either as this induces high memory consumption, caused by both increased redundancy of content among adjacent shafts and the vast number of shaft structures themselves.

The collection consists of two parts: the list area, which is a concatenation of all candidate lists, and the shaft lookup table, which maps each shaft's id to the corresponding offset of the associated candidate list in the list area. The node ids are just 4-byte indices to the main BVH node array. An example of a BVH with a shaft collection is shown in Figure 4.

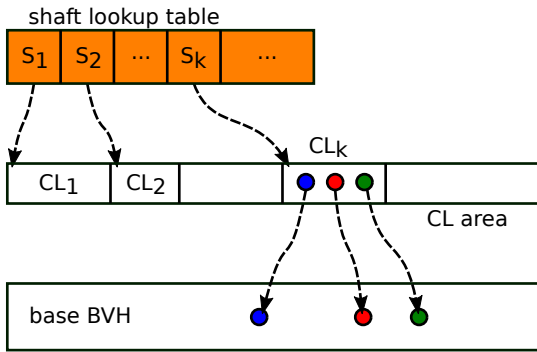


Figure 4: Schematic layout of a BVH and a shaft collection. The shaft lookup table maps shaft ids to the offset of their associated candidate lists in the candidate list area. The lists contain direct indices to the nodes of the base BVH.

The shaft id is computed as a combination of the shaft's voxel $x/y/z$ coordinates and the u/v directional coordinates. This id then serves as a key to the shaft lookup table.

4.2. Shaft Geometry

A shaft's geometrical definition is fully described by its 3D grid base voxel together with culling planes enclosing the subinterval of directions, very much like Arvo and Kirk [AK87] proposed. The voxel is surrounded by the culling planes, which leave the frustum shaft open on its other end.

At most seven culling planes are used to define the shaft: four of them enclose the directional range of the shaft, and up to three planes contain the outer faces of the shaft's voxel. The shaft geometry is utilized only during the candidate list construction, so we do not store it with the data structure past that step.

4.3. Candidate List

The shaft's candidate list provides complete coverage of the scene geometry enclosed by the shaft. Its elements are ids of the nodes of the original BVH which intersect the shaft. In Algorithm 1, we show in pseudocode the construction of a candidate list.

```

1: stack.push(bvh.root)
2: while !stack.empty() do
3:   node ← stack.top()
4:   stack.pop()
5:   if shaft.overlaps(node) then
6:     if node.isInner() then
7:       p ← shaft.getHitProbability(node, sampleRays)
8:       if p ≥ minHitProbability then
9:         if shaft.centerRay.dir[node.splitAxis] > 0 then
10:          stack.push(node.child)
11:          stack.push(node.child+1)
12:        else
13:          stack.push(node.child+1)
14:          stack.push(node.child)
15:       else
16:         node ← shaft.cullNodeSubtree(node)
17:         if node.valid() then
18:           candidateList.append(node)
19:       else
20:         candidateList.append(node) // leaf

```

Algorithm 1: Construct shaft candidate list for a binary BVH.

A crucial component of our method is the *getHitProbability()* function. This function tests the intersection of a small number (20) of rays within the shaft with the node's AABB to estimate the probability of traversing this node in the rendering phase. These rays are generated randomly using a uniform distribution within the given shaft and they are cast using the standard BVH to establish the intersections with the scene. The hit probability estimation uses these rays to check if the intersections with the node's AABB lie before the intersections of the sample rays with the scene. The estimation thus also considers occlusion, which prevents creating candidate list entries in occluded areas of the scene. The hit probability serves as a measure which tells us whether the node is worth opening and being replaced with its children. If the node passes this check, its children are pushed back onto the stack with respect to the orientation of the shaft's center ray, i.e., depth-sorted to increase the probability of early ray termination later in the rendering phase.

For nodes which do not pass the check of minimum hit probability, in *cullNodeSubtree()* we try to find a single descendant node which still contains all relevant geometry. This is done by opening the current node and checking the number of its children intersected by the shaft; if we find a single descendant with this property, it replaces the ancestor node in the candidate list. Occasionally, all the descendant nodes are culled, effectively preventing this entire subtree from being part of the candidate list.

In some cases, the algorithm produces more nodes than we allow as the maximum size of candidate lists. To address this issue, we progressively increase the *minHitProbability* threshold and call

the algorithm again, which tends to generate fewer nodes into the candidate list.

It can be shown that to probabilistically reduce the number of traversal steps, the minimum hit probability threshold has to be set to $\frac{k-1}{k}$ for k -ary BVH. If this threshold was lower, the algorithm would descend deeper in the tree and produce longer candidate lists. As a result, we could potentially end up traversing even more nodes than with the standard traversal. Conversely, if the threshold was larger, we would not utilize the potential of the method, staying too close to the BVH root.

Having a candidate list and the BVH subtrees referenced from the nodes in the list, we effectively form a set of sub-BVHs of substantially lower height than the base BVH (see Figure 1). These smaller BVHs can be interpreted as presorted view-dependent subsets of scene geometry with respect to a group of coherent rays within a shaft.

4.4. Memory Optimizations

In a brute-force implementation, the upper bound on build time complexity of a shaft collection is $O(s^3 \cdot d^2 \cdot k)$ (s representing a single spatial resolution in all three axes, d a directional resolution in both coordinates, and k the maximum length of candidate list); here we regard the BVH culling traversal as constant. The space complexity is then also at most $O(s^3 \cdot d^2 \cdot k)$ since the storage needed to keep a shaft's candidate list is less than or equal to k .

We can use an optional reduction scheme, which limits the computational costs very efficiently. The idea relies on ray sampling when only the most ray-occupied shafts are allowed to build their candidate lists. The shaft occupancy criterion can be given by a fraction of rays or shafts we want to use for accelerated traversal; it operates on shaft statistics data gathered in the sampling phase and sorted from the most used shafts to the less important ones. The ray sampling is a rendering phase executed in a lower image resolution and/or with fewer samples per pixel. The light transport or any potential information gathered by these sample rays can be utilized during rendering to amortize the cost of sampling.

A cheaper alternative to ray sampling is to construct the candidate lists only for shafts originating in voxels occupied by scene geometry. This approach handles all secondary rays and also shadow rays cast from surfaces towards light sources. However, it excludes the primary rays from processing (unless the camera is located in a nonempty voxel).

Often the candidate lists of adjacent shafts (those with nearby base voxels and similar ray directions) are identical. Therefore, during the construction we index the candidate lists in a hashmap which gives us a quick answer to whether there already is an identical candidate list in the collection. If so, we do not store the new copy again, but rather let the respective shaft refer to the original instance. This very cheap check saves between 10-80% of memory in the candidate list area in our scenes.

4.5. Traversal

The traversal of a ray starts by classifying the ray and looking up the corresponding candidate list. If we successfully found the list,

all its nodes become the new entry points of the traversal, being pushed onto the traversal stack instead of the BVH root. Otherwise, we continue in the usual manner by following the BVH root.

This is the only modification of a standard traversal kernel which needs to be made: except for the different start-up, we traverse the BVH in the usual fashion. Thus, we remove large parts of the traversal path. It might be necessary to increase the traversal stack capacity, as the stack has to be able to hold the entire candidate list plus some extra space for the traversal of the individual subtrees referenced from the list.

5. Results

We implemented the proposed method in standard C++11 with thread parallelism integrated into the open-source PBRT v3 renderer [PJH16]. We performed a series of tests on ten scenes, comparing the ratio of traversed steps, the ray tracing performance, the build times, and the memory footprint of our method, having the standard BVH traversal as reference. The measurements were performed on a PC equipped with Intel Xeon E3-1245 with 8 cores @ 3.5 GHz and 16 GB RAM, both the shaft collection build and ray tracing phases used 8 working threads. The source code of the algorithm can be downloaded from the project website: <http://dcgi.felk.cvut.cz/projects/rc+bvh>.

The BVHs were constructed using Binning SAH method [Wal07] with 32 bins and 4 triangles per leaf; we used them as static bases for generating different configurations of shaft collections. For direct comparison with PBRT, we built binary BVHs; we also measured the performance on quaternary hierarchies, for which we had to slightly adjust the renderer's traversal code. We chose 31 nodes as the maximum size of candidate lists. This value proved to be a good balance to describe the shaft content with enough detail yet not overly memory demanding.

5.1. Ray Tracing Performance

To evaluate the performance of ray tracing aided by shafts, we adjusted the BVH traversal code in the PBRT renderer to execute our lookup and stack population code. The evaluation uses the default rendering setup in PBRT, i.e., path tracing with max. 5 bounces and Russian roulette, accelerated by binary BVHs. This setup incorporates a good amount of incoherent load for most scenes we tested. Each scene was traced from three different camera locations and orientations using 1920×1080 image resolution with 8 samples per pixel. We tested three different variants of the method:

- **Complete** – Candidate lists of all shafts were built.
- **OccupiedVoxels** – Candidate lists were built only for voxels occupied by scene geometry.
- **ViewDependent** – Ray sampling phase was executed to determine the shafts most used by rays. The candidate lists were computed for 10% of the most populated shafts.

For all variants, we used three different spatial resolutions (100k, 200k, 500k voxels), combined with two different directional resolutions (2 and 4).

A comprehensive overview of the results for the *OccupiedVoxels* variant and resolution of $200k \times 4$ is summarized in Table 1. In

the ray tracing phase, the number of traversed steps ranges between 37% and 81% related to the standard traversal with rendering times measured between 81% and 97%. The major reason for this difference between traversed steps and rendering times ratios is that the rendering process also includes geometry intersection and shading on top of the ray traversal itself. These rendering phases are not targeted by our method and remain roughly constant.

The results evaluated on quaternary BVHs exhibit on average 5% lower savings of traversal steps than binary BVHs. Considering there are three times fewer interior nodes in the quaternary BVH than in the binary variant, the achieved savings are actually higher than we initially expected. PBRT does not implement SIMD traversal, therefore our rendering times for quaternary BVHs are generally slightly higher than those for binary BVHs. Using the SIMD traversal would speed up both the reference quaternary BVH method as well as the method accelerated by shafts.

A graphical overview of the speedups for the *FirstHit* and *AnyHit* routines as well as the total rendering speedup is shown in Figure 5. In most cases, the complete build yields the best speedups, at the expense of larger memory consumption and longer build time.

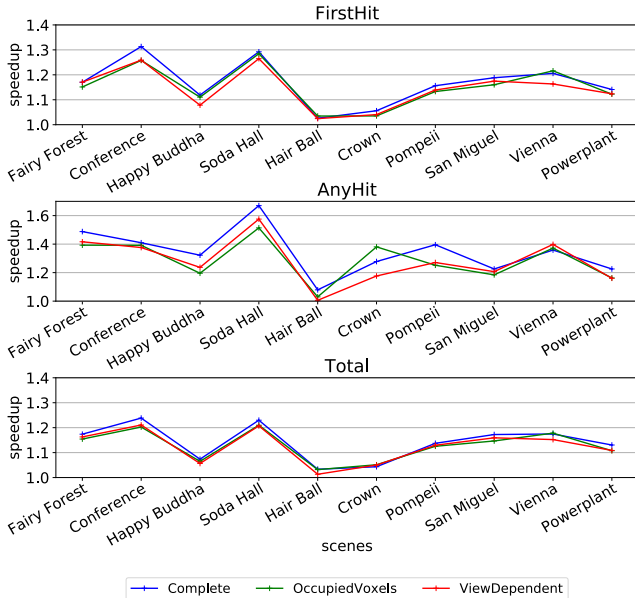


Figure 5: The performance of all variants of our method on different scenes for *FirstHit* and *AnyHit* routines and the total rendering time. The speedups are calculated relatively to the performance of the standard traversal without shafts using a binary BVH.

The relation of the speedups averaged over all tested scenes to the average relative memory consumption of all evaluated variants is shown in Figure 6. The flat shape of the graphs suggests it does not make much sense increasing the memory budget beyond using approx. the same amount as for the geometry. The speedup for shadow rays is significantly higher than for the other types of rays. The method saves about 15% of time of finding ray-scene intersections, dominated by the higher ratio of *FirstHit* calls. The *OccupiedVoxels* variant saves substantial memory, but suffers from not handling the primary rays. The *ViewDependent* variant leads to

lowest memory consumption, but due to some uncovered rays the speedups are also slightly lower than for the *Complete* variant.

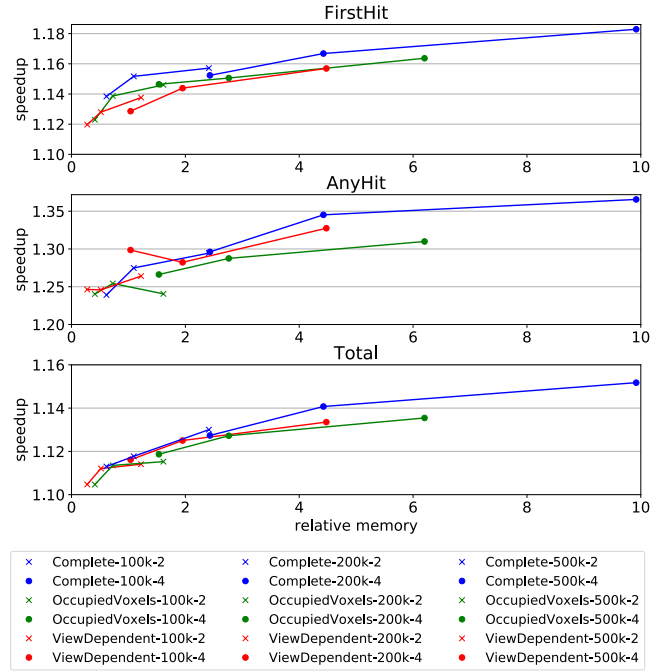


Figure 6: Average speedups in relation to the memory demands of our method itself, related to the size needed for storing the geometry of the scenes. To better show the influence of increasing the spatial resolution, the measurements with the same resolution of the directional subdivision for a given method are connected.

To gain more insight into how many elements there are in the candidate lists, we gathered statistics for two scenes (Conference and San Miguel) which largely differ in number of geometry primitives and whether they represent interior of exterior. It turns out that despite these differences, the distributions are very similar, resembling the binomial distribution (see Figure 7). In the Conference scene, most candidate lists are able to contain all nodes in the first pass of Algorithm 1, the number of the most populated CLs approaching zero. On the other hand, in San Miguel with much higher number of primitives, we have to iterate the algorithm with increased threshold in order to squeeze the candidate lists into the allowed size, leaving many candidate lists with high node counts.

To evaluate the behavior with increased ratio of incoherent rays, we ran an additional test with a closed interior scene (Conference), higher max. bounces (10), and Russian roulette disabled. The speedups remained practically unchanged.

5.2. Construction Overhead

For high-quality rendering with many samples per pixel and/or many camera frames, the shaft construction times are easily compensated during rendering. This can be seen from most results shown in Table 1, although we used a relatively low number of samples per pixel (8). Still, reduction of the construction overhead may be of concern. The *ViewDependent* variant uses only the most









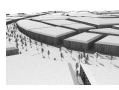
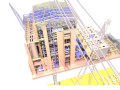
																				
Fairy Forest #triangles 174k				Conference #triangles 331k				Happy Buddha #triangles 1087k				Soda Hall #triangles 2169k				Hair Ball #triangles 2850k				
	build	build	traversals	render	build	build	traversals	render	build	build	traversals	render	build	build	traversals	render	build	build	traversals	render
	size	time	per ray	time	size	time	per ray	time	size	time	per ray	time	size	time	per ray	time	size	time	per ray	time
	[MB]	[s]	[- / %]	[s / %]	[MB]	[s]	[- / %]	[s / %]	[MB]	[s]	[- / %]	[s / %]	[MB]	[s]	[- / %]	[s / %]	[MB]	[s]	[- / %]	[s / %]
Binary BVH																				
no shafts	11	0.2	44.8 / 100%	522 / 100%	22	0.4	45.7 / 100%	1006 / 100%	71	1.3	23.5 / 100%	90 / 100%	139	2.3	62.0 / 100%	739 / 100%	190	3.7	75.3 / 100%	358 / 100%
with shafts	122	19.9	21.3 / 47%	451 / 86%	127	17.6	16.9 / 37%	837 / 83%	223	23.4	16.0 / 68%	83 / 93%	325	29.0	27.2 / 44%	596 / 81%	851	161.5	57.7 / 77%	345 / 96%
Quaternary BVH																				
no shafts	10	0.2	43.3 / 100%	551 / 100%	20	0.4	41.0 / 100%	1049 / 100%	65	1.3	23.1 / 100%	93 / 100%	128	2.4	59.0 / 100%	800 / 100%	173	3.8	71.5 / 100%	360 / 100%
with shafts	125	21.2	22.2 / 51%	469 / 85%	130	17.2	17.5 / 43%	869 / 83%	236	30.6	16.8 / 73%	87 / 93%	335	42.6	32.8 / 56%	680 / 85%	963	412.2	58.1 / 81%	350 / 97%
																				
Crown #triangles 4868k				Pompeii #triangles 5632k				San Miguel #triangles 7842k				Vienna #triangles 8637k				Power Plant #triangles 12759k				
	build	build	traversals	render	build	build	traversals	render	build	build	traversals	render	build	build	traversals	render	build	build	traversals	render
	size	time	per ray	time	size	time	per ray	time	size	time	per ray	time	size	time	per ray	time	size	time	per ray	time
	[MB]	[s]	[- / %]	[s / %]	[MB]	[s]	[- / %]	[s / %]	[MB]	[s]	[- / %]	[s / %]	[MB]	[s]	[- / %]	[s / %]	[MB]	[s]	[- / %]	[s / %]
Binary BVH																				
no shafts	314	7.1	48.0 / 100%	219 / 100%	366	6.9	79.7 / 100%	503 / 100%	514	10.1	124.0 / 100%	1898 / 100%	581	10.6	59.9 / 100%	372 / 100%	809	15.3	76.8 / 100%	399 / 100%
with shafts	406	13.5	36.2 / 76%	208 / 95%	790	76.4	50.0 / 63%	445 / 89%	664	40.5	70.8 / 57%	1640 / 86%	786	46.5	31.6 / 53%	311 / 84%	911	23.9	49.9 / 65%	364 / 91%
Quaternary BVH																				
no shafts	288	7.2	46.7 / 100%	229 / 100%	335	6.9	72.7 / 100%	518 / 100%	470	10.2	120.5 / 100%	1974 / 100%	528	10.8	57.3 / 100%	391 / 100%	744	15.4	71.7 / 100%	426 / 100%
with shafts	385	16.1	37.2 / 80%	220 / 96%	809	94.4	50.4 / 69%	472 / 91%	635	48.6	75.9 / 63%	1715 / 87%	757	51.4	33.1 / 58%	329 / 84%	849	27.2	51.0 / 71%	391 / 92%

Table 1: Performance comparison of the tested method for the OccupiedVoxels variant with spatial resolution of 200k and direction resolution of 4. The build times, apart from BVHs build itself, refer to the view independent construction of candidate lists; for static scenes they can be amortized over many rendered frames. The render times comprise ray generation, ray traversal, intersection evaluation, and shading.

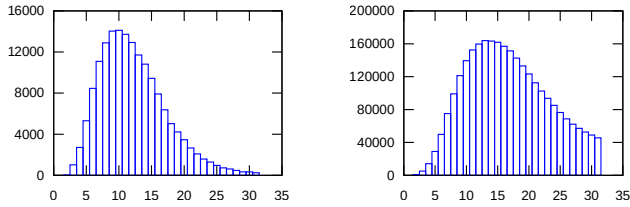


Figure 7: Distribution of candidate list lengths in the Conference scene (left) and in the San Miguel scene (right), both having maximum length equal to 31 indices. In both cases, the most frequent number of elements in candidate lists lies between 10 and 15.

used shafts and it provides about three times faster construction at the expense of a slightly more complicated implementation (note that the memory footprint drops accordingly, too). In a production renderer, it would also be possible to further optimize the construction using lower level optimizations, such as SIMD instructions.

6. Conclusion and Future Work

We described a simple and flexible algorithm for accelerating BVH traversal. The method builds an additional structure on top of an existing BVH and is able to cut off unnecessary parts of traversal based on spatial and directional classification of individual rays. Groups of similar rays are enclosed in frustum shafts, each of which

contains only a limited subset of BVH nodes and scene geometry. This subset is represented by a candidate list, thus forming a forest of sub-BVHs of substantially lower traversal costs than the base BVH.

Although previously deemed impractical, we showed that the ray classification can support contemporary BVHs traversal successfully, and we highlighted the important technical details that make it work. On the PBRT renderer, we showed that the method can be plugged easily into an existing framework. Our experiments showed that it saves a large portion of traversal steps, about 42% on average, and a significant amount of rendering time.

We provided a basic view on how the algorithm behaves with quaternary hierarchies, which shows similar results as with the native binary variants. We plan to conduct a more in-depth analysis of the behavior of the method with wide BVHs and optimized traversal kernels. In our implementation, we construct the candidate lists with secondary rays in mind. We plan to study shafts specialized for any-hit usage, which would take into account the different objective of tracing shadow rays.

Our method uses ray indexing that can be used in combination with ray reordering techniques to improve cache usage [Bik12]. Another interesting topic for future work is the combination of our method with the path guiding approach [MGN17] that also uses a subdivision of ray space that could be easily shared by the two algorithms.

Acknowledgments

This research was supported by the Czech Science Foundation under project GA18-20374S, by the Grant Agency of the Czech Technical University in Prague, grant No. SGS19/179/OHK3/3T/13, by the Research Center for Informatics No. CZ.02.1.01/0.0/0.0/16_019/0000765, and by the TACC through Intel Graphics and Visualization Institutes of XeLLENCE (Intel GVI). Jakub Hendrich would like to thank Paul Navrátil for fruitful discussions on the topic during his stay at TACC.

References

- [AK87] ARVO, JAMES and KIRK, DAVID. “Fast Ray Tracing by Ray Classification”. *Computer Graphics (SIGGRAPH '87 Proc.)* 21.4 (July 1987), 55–64 2, 4.
- [BDT99] BALA, KAVITA, DORSEY, JULIE, and TELLER, SETH. “Radiance interpolants for accelerated bounded-error ray tracing”. *ACM Trans. Graph* 18.3 (1999), 213–256 2.
- [Bik12] BIKKER, J. “Improving Data Locality for Efficient In-Core Path Tracing”. *Comput. Graph. Forum* 31.6 (2012), 1936–1947 7.
- [Bit02] BITTNER, Jiří. “Hierarchical techniques for visibility computations”. *Prague: Department of Computer Science and Engineering, Czech Technical University* (2002) 2.
- [BP01] BRIÈRE, NORMAND and POULIN, PIERRE. “Adaptive Representation of Specular Light”. *Computer Graphics Forum* 20.2 (2001), 149–162 2.
- [BP96] BRIÈRE, NORMAND and POULIN, PIERRE. “Hierarchical View-Dependent Structures for Interactive Scene Manipulation”. *SIGGRAPH 96 Conference Proceedings*. ACM SIGGRAPH. Aug. 1996, 83–90 2.
- [DHS04] DMITRIEV, KIRILL, HAVRAN, VLASTIMIL, and SEIDEL, HANS-PETER. *Faster Ray Tracing with SIMD Shaft Culling*. Research Report. Max-Planck-Institut für Informatik, Dec. 2004 2.
- [FBPG00] FERNANDEZ, SEBASTIAN, BALA, KAVITA, PICCOLOTTO, MORENO A., and GREENBERG, DONALD P. *Interactive Direct Lighting in Dynamic Scenes*. Technical report PCG-00-2. Program of Computer Graphics, Cornell University, Jan. 2000 2.
- [GBDA15] GANESTAM, PER, BARRINGER, RASMUS, DOGGETT, MICHAEL, and AKENINE-MÖLLER, TOMAS. “Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees”. *Journal of Computer Graphics Techniques (JCGT)* 4.3 (2015), 23–42 1, 2.
- [GHF13] GU, YAN, HE, YONG, FATAHALIAN, KAYVON, and BLELLOCH, GUY E. “Efficient BVH Construction via Approximate Agglomerative Clustering”. *Proceedings of High Perform. Graphics*. 2013, 81–88 1.
- [GS87] GOLDSMITH, JEFFREY and SALMON, JOHN. “Automatic Creation of Object Hierarchies for Ray Tracing”. *IEEE Computer Graphics and Applications* 7.5 (1987), 14–20 1, 2.
- [HB00] HAVRAN, VLASTIMIL and BITTNER, Jiří. “LCTS: Ray Shooting using Longest Common Traversal Sequences”. *Comput. Graph. Forum* 19.3 (2000), 59–70 2.
- [HBŽ98] HAVRAN, VLASTIMIL, BITTNER, Jiří, and ŽÁRA, Jiří. *Ray Tracing with Rope Trees*. en. 1998 2.
- [HW94] HAINES, ERIC A. and WALLACE, JOHN R. “Shaft Culling for Efficient Ray-Traced Radiosity”. *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)*. 1994 2.
- [KA13] KARRAS, TERO and AILA, TIMO. “Fast Parallel Construction of High-Quality Bounding Volume Hierarchies”. *Proceedings of High Performance Graphics*. 2013, 89–100 1.
- [KML16] KEUL, KEVIN, MÜLLER, STEFAN, and LEMKE, PAUL. “Accelerating spatial data structures in ray tracing through precomputed line space visibility”. *Computer Science Research Notes* (2016). ISSN: 2464-4617 2.
- [LYMT06] LAUTERBACH, CHRISTIAN, YOON, SUNG-EUI, MANOCHA, DINESH, and TUFT, DAVID. “RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs”. *IEEE Symposium on Interactive Ray Tracing*. Sept. 2006, 39–46 1.
- [MGN17] MÜLLER, THOMAS, GROSS, MARKUS, and NOVÁK, JAN. “Practical path guiding for efficient light-transport simulation”. *Computer Graphics Forum*. Vol. 36. 4. Wiley Online Library. 2017, 91–100 2, 7.
- [PJH16] PHARR, MATT, JAKOB, WENZEL, and HUMPHREYS, GREG. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2016. URL: <https://pbrt.org/5>.
- [PL10] PANTALEONI, JACOPO and LUEBKE, DAVID. “HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry”. *Proceedings of High Performance Graphics*. 2010, 87–95 1.
- [RSH05] RESHETOV, ALEXANDER, SOUPIKOV, ALEXEI, and HURLEY, JIM. “Multi-Level Ray Tracing Algorithm”. *ACM Transactions on Graphics* 24.3 (2005), 1176–1185 2.
- [SD92] SCHRÖDER, PETER and DRUCKER, STEVEN M. “A data parallel algorithm for raytracing of heterogeneous databases”. *Proceedings of Computer Graphics Interface* (May 1992), 167–175 2.
- [SDS00] SCHAUFER, GERNOT, DORSEY, JULIE, DECORET, XAVIER, and SILLION, FRANÇOIS X. “Conservative Volumetric Visibility With Occluder Fusion”. *Proceedings of the Computer Graphics Conference 2000 (SIGGRAPH-00)*. July 2000, 229–238 2.
- [SFD09] STICH, MARTIN, FRIEDRICH, HEIKO, and DIETRICH, ANDREAS. “Spatial splits in bounding volume hierarchies”. *Proceedings of High Performance Graphics*. New Orleans, Louisiana, 2009, 7–13 1.
- [vdZJR95] Van der ZWAAN, MAURICE, REINHARD, ERIK, and JANSEN, FREDERIK W. “Pyramid Clipping for Efficient Ray Traversal”. *Rendering Techniques '95*. 1995, 1–10 2.
- [Wal07] WALD, INGO. “On fast construction of SAH-based bounding volume hierarchies”. *2007 IEEE Symposium on Interactive Ray Tracing*. IEEE. 2007, 33–40 5.
- [Wal12] WALD, INGO. “Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture”. *IEEE Transactions on Visualization and Computer Graphics* 18.1 (2012), 47–57 1.
- [WBS03] WALD, INGO, BENTHIN, CARSTEN, and SLUSALLEK, PHILIPP. “Distributed Interactive Ray Tracing of Dynamic Scenes”. *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. 2003, 77–86 1.
- [WBS07] WALD, INGO, BOULOS, SOLOMON, and SHIRLEY, PETER. “Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies”. *ACM Trans. Graph.* 26.1 (Jan. 2007) 1, 2.