Dual-Split Trees

Daqi Lin
University of Utah

Konstantin Shkurko
University of Utah

Agatha Mallett
University of Utah

Cem Yuksel
University of Utah

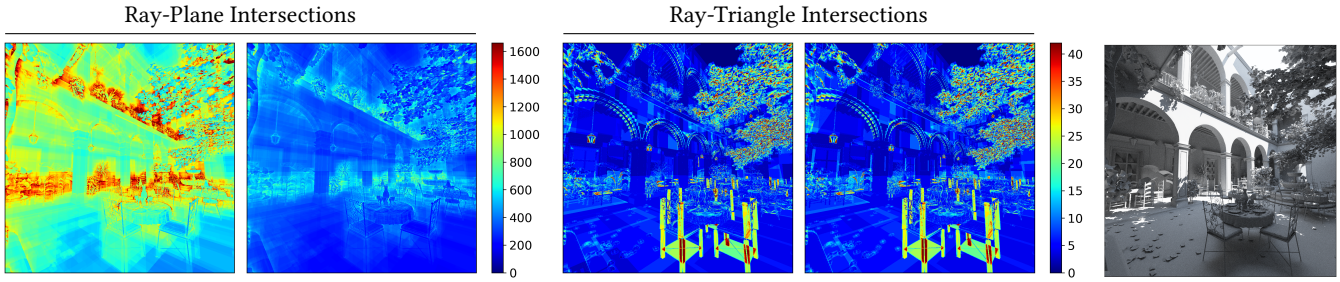


Figure 1: *Comparison of the number of ray-plane and ray-triangle intersection tests generated by the primary rays using identical space partitioning between the BVH and our* ***dual-split tree*** *for the San Miguel scene shown on the right. The results show that our dual-split tree representation substantially reduces the number of ray-plane intersection tests during ray traversal with only a minor increase in the number of ray-triangle intersection tests.*

ABSTRACT

We introduce the dual-split tree, a new tree-based acceleration structure for ray tracing. Each internal node of a dual-split tree uses two axis-aligned planes to either split the parent node into two child nodes or to mark the empty regions of the node. This allows child bounding boxes to overlap when desired. Thus, our dual-split tree is capable of representing space partitioning identical to any given bounding volume hierarchy. Our dual-split tree provides a significant reduction in the required acceleration structure storage by eliminating the redundant bounding planes that are commonplace in bounding volume hierarchies, providing better performance and storage savings than similar previous methods. As a result, we achieve improved rendering performance with dual-split trees, as compared to bounding volume hierarchies with a comparable level of optimization using identical or similar space partitioning.

CCS CONCEPTS

• **Computing methodologies** → **Rendering**; **Ray tracing**;

KEYWORDS

Ray tracing, acceleration structure, space partitioning

ACM Reference Format:
Daqi Lin, Konstantin Shkurko, Agatha Mallett, and Cem Yuksel. 2019. Dual-Split Trees. In *Symposium on Interactive 3D Graphics and Games (I3D '19), May 21–23, 2019, Montreal, QC, Canada.* ACM, New York, NY, USA, 9 pages.
https://doi.org/10.1145/3306131.3317028

I3D '19, May 21–23, 2019, Montreal, QC, Canada

ACM ISBN 978-1-4503-6310-5/19/05…$15.00
https://doi.org/10.1145/3306131.3317028

1 INTRODUCTION

Acceleration structures are crucial for high-performance ray tracing because they significantly reduce the ray traversal costs. Kd-trees were popular until recently, since they were known to provide the best rendering performance in most scenes. Unfortunately, building high-quality kd-trees that can deliver desirable render-time performance has been expensive, thereby undermining their popularity.

More recently, bounding volume hierarchies (BVHs) have become a popular choice for ray tracing acceleration because of the advances in algorithms for fast construction of the high quality BVHs and the good render-time performance they provide. BVHs have the unique property that they permit overlapping bounding volumes for sibling nodes in the tree structure, while other typical acceleration structures (such as kd-trees, octrees, grids, etc.) strictly separate the bounding volumes of sibling nodes. Thus, BVHs do not necessarily spatially separate the scene triangles, regardless of how they are oriented and distributed in a scene.

In this paper we introduce *dual-split trees*, a new acceleration structure for ray tracing. Unlike kd-trees that store a single axis-aligned plane to separate the child nodes of an internal node, our dual-split trees store two axis-aligned planes per node, which can align to either the same or different axes. This allows representing overlapping bounding volumes for sibling nodes, just like BVHs. Therefore, dual-split trees can represent a space partitioning identical to any given BVH. Furthermore, dual-split trees can provide a substantial reduction in storage as compared to a BVH with the identical space partitioning by eliminating the inherent storage redundancies in a typical BVH implementation. This is particularly important for large scenes, where memory accesses can be the bottleneck of ray traversal operations. Compared to other hybrid acceleration structures like the bounding interval hierarchy (BIH) [Wächter and Keller 2006] and H-Tree [Havran et al. 2006], our dual-split trees reduce the storage used and achieve higher traversal performance. In our tests with different scene sizes, we have observed up to 48% reduction in the acceleration structure

storage, as compared to an uncompressed BVH with the identical space partitioning. Dual-split trees can be generated from a given BVH faster than building the source BVH. Even though a dual-split tree contains more nodes than the BVH from which it is generated, the overall storage for dual-split trees is considerably reduced compared to the uncompressed source BVH because dual-split trees avoid storing redundant data and require much less storage per node. Dual-split trees also introduce a similar reduction in the number of ray-plane intersection tests. All of these reductions lead to improvements of the ray traversal performance.

We compare our dual-split trees to uncompressed BVHs and other hybrid acceleration structures representing identical space partitioning. In our experiments, our dual-split trees provide a significant reduction in storage and achieve similar or better performance compared to other acceleration structures. Moreover, we show that additional reductions in storage and computation time can be achieved with our dual-split trees through slight modifications to the space partitioning that eliminates some of the bounding planes.

2 BACKGROUND

A large body of work is dedicated to ray tracing acceleration structures. They reduce the number of ray-object intersection tests and significantly improve ray tracing performance, especially as the geometric complexity of scenes continues to grow [Wald et al. 2009]. Overall, the approaches can be classified based on whether they partition space or objects.

One of the more popular space-partition acceleration structures is a kd-tree [Bentley 1975; Futchs et al. 1988; Havran 2000]. Each node in the binary tree relies on an axis-aligned plane to split its child nodes and thus their geometric primitives. The plane can also separate geometry and empty space. Although kd-trees provide excellent ray traversal performance, they are not the fastest acceleration structures to build and require duplicating references to primitives that span across a splitting plane.

More recently, bounding volume hierarchies (BVHs) [Rubin and Whitted 1980] have become widely used for ray tracing. They partition objects rather than space, and thus typically do not duplicate references to geometric primitives. Essentially each node stores its bounds and a reference to child nodes contained within. Although a variety of bounding volumes have been explored, the rendering community has settled on axis-aligned bounding boxes, which can be intersected against a ray faster than their alternatives. One issue with the BVH is that node bounds can overlap, forcing the traversal of additional nodes even after a ray intersection is found.

Much research targets improving the BVH, producing many variants. BVH build times can be accelerated significantly by sorting the primitives along a space-filling curve, a highly parallelizable operation [Lauterbach et al. 2009; Pantaleoni and Luebke 2010; Vinkler et al. 2017]. The quality of BVHs built quickly bottom-up can suffer compared to sweep-based top-down builds, so recent methods perform tree rotations [Kensler 2008; Kopta et al. 2012]. Nodes can also be reordered within small neighborhoods (treelets) to improve BVH quality [Karras and Aila 2013; Domingues and Pedrini 2015]. Often the scene geometry is highly irregular and produces lots of bounding volume overlaps, which results in unnecessary intersection tests. A splitting plane can be used to tighter assign the triangles between nodes, duplicating triangle references, while removing bounding box overlaps and the corresponding surface area [Stich et al. 2009]. Aila et. al. evaluate and compare quality of BVHs produced by popular builders [2013].

As scene sizes increase and the gap between compute and memory widens, traversing ever-larger tree structures has become increasingly memory-bound. A wide variety of modern research focuses on compressing scene data and the corresponding BVH. Increasing the arity of the BVH can decrease the structure depth reducing the number of nodes, with an extra benefit of better SIMD utilization during traversal [Dammertz et al. 2008; Wald et al. 2008; Ernst and Greiner 2008]. Storing nearby nodes grouped into small subtrees closer in memory increases the L1 or L2 cache hit rates, thus reducing bandwidth required to render the scene [Aila and Karras 2010]. Many methods tackle compressing BVH data directly [Kim et al. 2010a,b; Bauszat et al. 2010; Vaidyanathan et al. 2016; Ylitie et al. 2017]. Such approaches usually make use of quantized coordinates for the BVH [Mahovsky and Wyvill 2006; Keely 2014] and/or the geometry [Segovia and Ernst 2010] to achieve compression, paying specific attention to memory performance [Liktor and Vaidyanathan 2016].

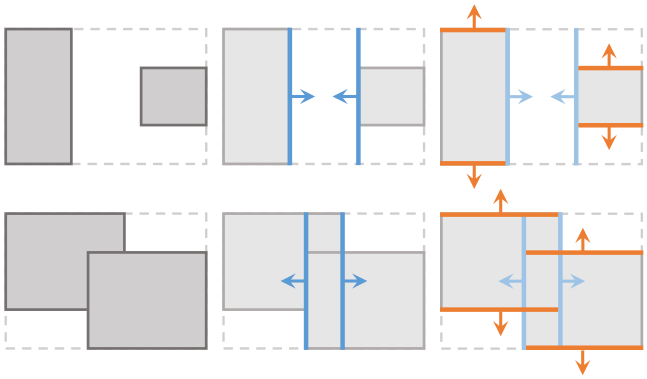
Fabianowski and Dingliana observe that for each axis at least two of the four planes bounding the children of a BVH node are shared with the parent [2009]. Consequently, the *Compact BVH* structure the authors introduce stores only six planes in every node to fully encode the two child bounding boxes, largely reducing the storage required to fully represent a BVH.

Other research explores the middle ground between a kd-tree and a BVH by combining the low traversal cost of a kd-tree and the low construction cost of a BVH. Bounding interval hierarchy [Wächter and Keller 2006] replaces the full axis-aligned bounding boxes for the two child nodes by two parallel clipping planes separating the two children. Havran et al. propose H-Tree [2006], which inserts different types of bounding volumes into a spatial kd-tree [Ooi 1987], helping cull empty space during traversal. Wächter uses two clipping planes for BIH nodes to tighten the bounding volumes [Wächter 2008]. B-KD Trees use two pairs of parallel bounding planes for the two children and accelerate ray tracing via special-purpose hardware [Woop et al. 2006].

While sharing some ideas, our dual-split trees achieve substantial additional reduction in storage and computation by using two planes per node that align to different axes, allowing empty space to be culled by only two planes in many cases. With our compact node representation, in some cases we obtain higher compression rates than the Compact BVH.

3 DUAL-SPLIT TREES

Each node of the dual-split tree stores two axis-aligned planes, which we label by their normal directions. We consider space along the normal direction as empty. There are two types of internal nodes corresponding to the two ways of using the planes: they either split space (*splitting nodes*) or carve out empty space (*carving nodes*). The planes are used to partition objects. However, unlike a classic BVH, we eliminate storing redundant planes which affords both reduced computation and storage.



(a) *BVH Node* **(b)** *splitting node* **(c)** *carving nodes*

Figure 2: *Dual-split tree in 2D: (a) shows two configurations of a BVH node (child bounding boxes are shown in gray); (b) creates a splitting node (split planes depicted in blue); (c) adds carving nodes as children of the splitting node (carving planes depicted in orange). Each arrow indicates the plane normal, with the empty space on the positive side of the plane.*

3.1 Splitting Nodes

The two planes in a splitting node separate the bounding boxes of the node's children into two bounding volumes along a single axis. We refer to these planes as *splitting planes*.

Consider the BVH node with two child nodes shown in Figure 2a. The corresponding separation of child bounding boxes in a dual-split tree would begin with a splitting node, as shown in Figure 2b. The first splitting plane of a splitting node marks the maximum bound of the first child along the splitting axis, and the second splitting plane marks the minimum bound of the second child node.

3.2 Carving Nodes

A carving node, shown in Figure 2c, is used to trim the empty space within the bounding volume. Carving nodes partition the space into two regions: one empty and another non-empty. Each carving node has only a single child node.

Unlike the splitting nodes, the planes of a carving node do not have to be along the same axis. We refer to a carving node with the two planes along the same axis as a *single-axis carving node*. The first plane indicates where the non-empty region begins and the second plane indicates where it ends along the chosen axis (Figure 3a). We refer to a carving node with the two planes each along a different axis as a *dual-axis carving node*. In this case, the empty space around the bounding box is carved along the two separate axes. For a given pair of axes, there are four different carving plane configurations such that each one of the two carving planes can either mark the beginning or the end of the non-empty region (Figure 3b).

In 3D there are three possible combinations for a pair of axes (xy, xz, and yz), each with four different carving plane configurations, resulting in 12 unique types of dual-axis carving nodes. Combined with the three possibilities for single-axis carving nodes, there are 15 different types of carving nodes in a dual-split tree structure.

(a) *single-axis carving node* **(b)** *dual-axis carving node*

Figure 3: *Two carving node types shown in 2D with the node bounding box shown by a dashed line. The carving planes are shown in orange, with arrows indicating their normal. The carving planes bound the bounding box of a child node (gray).*

Algorithm 1 Pseudocode for dual-split tree traversal.

```
procedure DualSplitTreeTraversal
    if miss global bounding box then
        return miss
    N ← root node
    while true do
        if N is leaf then
            test triangles and record the closest hit
        if N is splitting node then
            if intersect either of N's children then
                N ← nearest intersected child
                if exists, push far child onto Stack
                continue
        if N is carving node then
            if intersect N's child then
                N ← child node
                continue
        repeat
            if Stack is empty then
                return closest hit or miss
            else
                N ← pop Stack
        until newly popped t_min < closest hit
```

3.3 Tree Structure

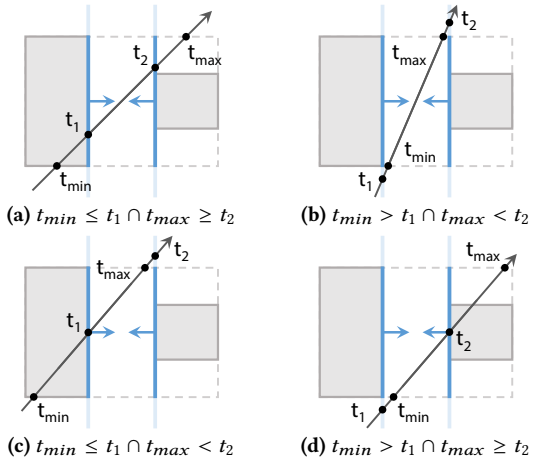
A dual-split tree contains a collection of splitting nodes and carving nodes. Each splitting node divides the scene data into two partitions, which may be overlapping or separated by empty space. Carving nodes are placed after splitting nodes to mark the empty regions of each partition.

Carving nodes can also act as leaf nodes. A carving node that is a leaf would simply point to a set of scene primitives rather than a child node. The carving planes are treated in the same way, regardless of whether a carving node is a leaf.

A dual-split tree can also contain leaf nodes that are not carving nodes. Such leaf nodes would need no splitting or carving planes.

3.4 Traversal

The traversal of dual-split trees is similar to kd-trees, shown as pseudocode in Algorithm 1. Each ray keeps track of the valid depth range $[t_{min}, t_{max}]$ along its direction. As the ray traverses through each node, it checks intersections with split and carved bounding volumes. Each traversal step modifies the current valid depth range corresponding to the bounds of the nearest child, while pushing the range for the far child and its index onto the traversal stack. When a node is popped from the traversal stack, it is culled using the current



(a) $t_{min} \leq t_1 \cap t_{max} \geq t_2$ **(b)** $t_{min} > t_1 \cap t_{max} < t_2$

(c) $t_{min} \leq t_1 \cap t_{max} < t_2$ **(d)** $t_{min} > t_1 \cap t_{max} \geq t_2$

Figure 4: *Four cases in splitting node traversal. Subcaptions specify the conditions when a ray (black arrows) intersects (a) both children, (b) neither child, (c) the closer child, and (d) the farther child. When a ray intersects the empty space (b), traversal stops.*

closest hit distance. Unlike kd-tree traversal which terminates as soon as a hit is found, dual-split tree traversal continues until the traversal stack is empty, because node bounding boxes can overlap.

A single ray-node intersection generates two hit distances, one for each plane. Similar to intersecting against an axis-aligned bounding box, one must properly consider plane orientations relative to the ray direction. Let t_1 and t_2 be the ray distances marking the intersections with the two planes.

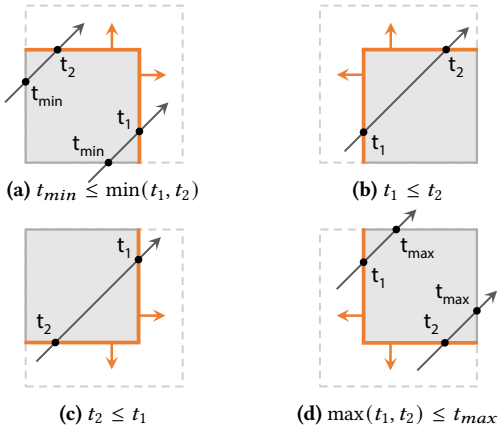
For splitting nodes, we order t_1 and t_2 such that t_1 corresponds to the plane with its normal direction having the same sign as the ray direction, and t_2 corresponds to the intersection with the other plane. The ray intersects the closer child node when $t_{min} \leq t_1$ and intersects the other child node when $t_{max} \geq t_2$. Note that the ray would miss both child nodes if it is passing through the empty space in between the splitting planes, as shown in Figure 4.

The intersection with a carving node first trims the ray's valid depth range to the non-empty volume defined by the carving planes. Then, the values t_{min} and t_{max} are compared such that if $t_{min} > t_{max}$, the ray traverses the empty space, thus missing the child of the carving node. The ray intersects a single-axis carving node only when both $t_{min} \leq t_1$ and $t_{max} \geq t_2$.

Intersecting a dual-axis carving node resembles testing against a 2D axis-aligned bounding box defined by the two carving planes. Because the four possible configurations share different bounds with the parent node, each needs to be considered separately as shown in Figure 5.

4 IMPLEMENTATION DETAILS

Our dual-split tree implementation stores the acceleration structure linearly in memory with nodes in depth-first order. Sibling nodes are stored next to each other. The geometry is stored as an array of vertices and normals. Triangles are stored as an array of vertex indices. Leaf nodes reference into a global list of triangle indices.



(a) $t_{min} \leq \min(t_1, t_2)$ **(b)** $t_1 \leq t_2$

(c) $t_2 \leq t_1$ **(d)** $\max(t_1, t_2) \leq t_{max}$

Figure 5: *Four configurations of carving planes (orange) in a dual-axis carving node. Subcaptions specify the conditions when a ray (black arrows) intersects the child within the node. These tests are similar to intersecting against a 2D axis-aligned bounding box.*

4.1 Node Data Layout

In our implementation of dual-split trees, each node begins with a 6-bit header that determines the node type followed by a 26-bit integer offset. Leaf nodes with no carving planes are packed into 4 bytes. Other nodes that store two splitting planes require an additional 8 bytes to store the two plane locations as single-precision floating-point numbers. Thus, we use variable node sizes in our implementation.

The integer offset within the internal nodes points to the node's first child as a relative index offset from the node's position. The integer offset within each leaf node stores the index of the first triangle. Leaf nodes avoid storing the triangle count explicitly; instead, they rely on how triangle indices are stored in the global triangle index list. We use the sign bit of the triangle index to specify that a triangle is the last one belonging to a particular leaf node. Therefore, ray-triangle intersection starts at a given index list location and iterates until it reaches (and processes) a triangle specified by a negative index.

The 6-bit node header is shown in Figure 6. Node type dictates how the remaining bits are interpreted. Splitting nodes use two bits

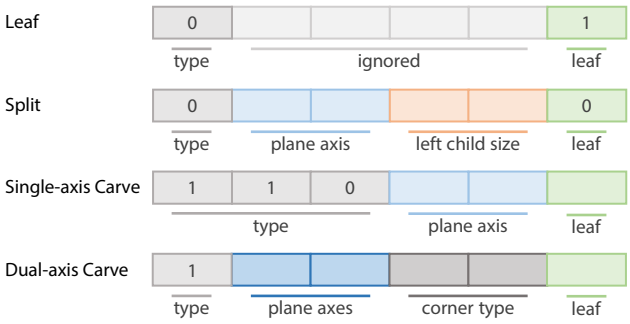


Figure 6: *Node headers in our implementation of the dual-split tree consist of six bits allocated differently depending on the node type. Empty spaces store data as described in Section 4.1.*

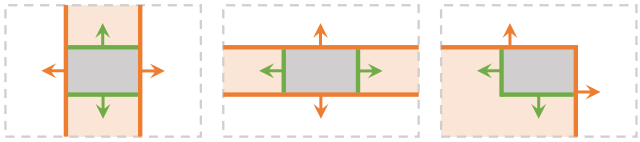


Figure 7: *Examples of carving empty space around a node (gray box) in 2D using two levels of carving nodes. The first level carving planes (orange lines) leave some empty space (orange) which gets carved by the second carving level (green lines). One can use either single-axis carving nodes (left, center) or dual-axis carving nodes (right).*

to store the axis used for splitting and another two bits to store the size of the first child in words, which can be either 1 or 3 (meaning 4 or 12 bytes). This simplifies computing the address for the second child using the integer offset.

Single-axis carving nodes use two bits to store the axis used for carving planes. Dual-axis carving nodes use two bits to store which axes are used for carving, encoded in the following manner for convenience: `00` for xy, `01` for xz, and `11` for yz. The two bits, labeled *corner type* in Figure 6, denote which of the four possible configurations is used, all illustrated in Figure 5.

Carving nodes can also act as leaves based on the *leaf* bit. If it is set, the index offset is interpreted as an index into the global triangle array. Leaf carving nodes still store and use the carving planes during traversal.

4.2 Construction from BVH

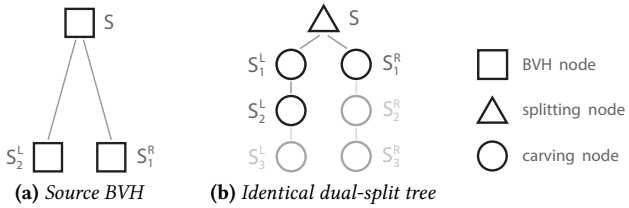
For fair comparison between a BVH or other hybrid acceleration structures and our dual-split tree, we construct the latter directly from a given BVH. The builder constructs a collection of dual-split tree nodes from each BVH node in a top-down manner. The builder relies on the surface area heuristic (SAH) [MacDonald and Booth 1990] to choose the configuration of splitting and carving nodes.

Given a BVH node with two child nodes, our builder first creates a splitting node. The splitting axis can be chosen as any one of the three possibilities. After splitting, carving nodes are placed as the child nodes of the splitting node as needed. Figure 7 shows examples of different carving node configurations in 2D that can be used to trim the same empty space. As shown in Figure 8, up to three carving nodes may need to be placed one after another to achieve the same space partitioning as the child nodes of the BVH node. Yet, three carving nodes following a splitting node is rare in practice, since most BVH nodes share some planes with their parents. In most cases, zero or one carving node is sufficient to represent the space partitioning identical to the BVH.

Let $k \in \{x, y, z\}$ represent the splitting axis for a splitting node and $C_k = C_k^L + C_k^R$ be the SAH cost of selecting axis k, where C_k^L and C_k^R are the costs of the two child nodes. C_k^L can be computed, including up to $\ell \leq 3$ carving nodes underneath it, such that

$$C_k^L = \sum_{j=1}^{\ell} \frac{S_{j-1}^L}{S} C_j + \frac{S_\ell^L}{S} C_I T^L \,, \tag{1}$$

where S and S_0^L are the surface areas of the parent splitting node and its immediate left child node; S_1^L, S_2^L, and S_3^L are the surface areas of up to three carving nodes; C_1, C_2, and C_3 are their traversal costs; C_I is the triangle intersection cost, and T^L is the number of



(a) *Source BVH* **(b)** *Identical dual-split tree*

Figure 8: *BVH vs. dual-split tree: (a) three nodes of a BVH and (b) the corresponding dual-split tree using up to three carving nodes below each splitting node. Note that in most cases one carving node is sufficient in practice (see Table 3) and if one child of the splitting node has three carving nodes, the other child will have no carving nodes.*

triangles within the left child node. C_k^R is computed similarly. In our implementation we use $C_j = \sigma\, C_I$, where single-axis carving nodes set $\sigma = 0.3$, and dual-axis carving nodes set $\sigma = 0.5$ (see Figure 7). To select an optimal configuration of the nested carving nodes, we compare the SAH costs of all possible configurations. The builder generates the nested carving nodes corresponding to the configuration with the lowest SAH cost.

4.2.1 Identical bounds. Dual-split tree can match the BVH partitioning exactly by generating up to three carving node levels for every BVH node. In this case, the SAH costs (Equation 1) use the bounding boxes S, S_ℓ^L and S_ℓ^R that are provided directly by the BVH for each node. Although this approach provides a faithful representation of the BVH, it can generate more nodes than required. We refer to this dual-split tree construction as *identical*.

4.2.2 Approximate bounds. In some cases, carving small amounts could generate an excessive number of carving nodes which would increase the traversal and storage costs. It is more beneficial to delay the creation of a carving node until deeper in the tree, so that a single carving node can carve empty space accumulated over several BVH levels. We enable this optimization by letting SAH dictate when to avoid the generation of a carving node. The builder keeps track of node bounding boxes rather than using what BVH provides directly.

5 RESULTS

We evaluate the performance of our dual-split tree and compare it to a traditional uncompressed binary BVH, a BIH, an H-Tree, and a Compact BVH. We do not compare with B-KD tree, as it is designed for special hardware, and cannot be converted directly from the BVH to use the same spatial partitioning. We evaluate the acceleration structure storage and the traversal performance measured by render times and ray intersection counts.

The binary BVH is built using the Intel Embree API v.2.16.5 [Wald et al. 2014] without splits and with a maximum of 8 triangles per leaf node. We store the uncompressed BVH linearly in memory where each parent stores the bounding boxes of its children and siblings are stored next to each other. We use varying node sizes, such that the internal BVH nodes are 52 bytes, and leaf nodes are 4 bytes (since their bounding boxes are stored by their parent nodes). The Compact BVH is derived from the binary BVH and stores each node

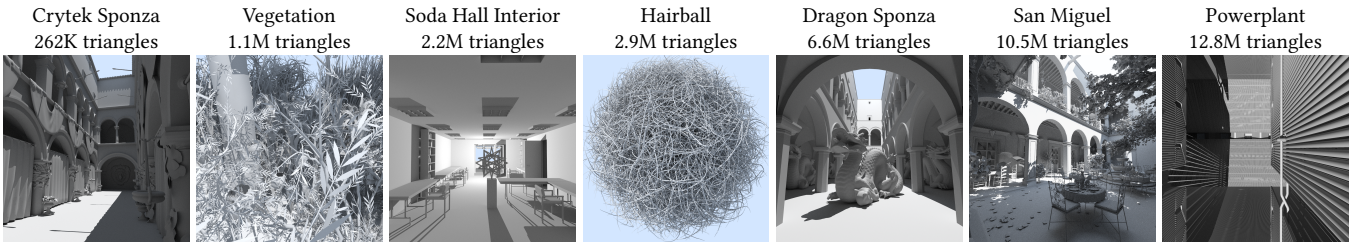


Figure 9: *Scenes used for all performance tests and comparisons.*

Table 1: *Comparison of the performance of several acceleration structures.*

		BVH	Dual-Split Trees (identical)	Dual-Split Trees (similar)	BIH	H-Tree	Compact BVH
Accel. Structure Storage (MB)	Crytek Sponza	7.04	4.53 (64%)	**3.59** (51%)	6.55 (93%)	6.13 (87%)	4.02 (57%)
	Vegetation	22.7	15.7 (69%)	13.8 (61%)	23.9 (105%)	21.8 (96%)	**13.0** (57%)
	Soda Hall Interior	51.9	28.7 (55%)	**25.6** (49%)	37.2 (72%)	36.3 (70%)	29.7 (57%)
	Hairball	43.8	28.7 (66%)	**22.9** (52%)	41.5 (95%)	37.7 (86%)	25.0 (57%)
	Dragon Sponza	182	112 (61%)	**98.2** (54%)	148 (81%)	147 (81%)	104 (57%)
	San Miguel	266	174 (66%)	**145** (55%)	259 (97%)	242 (91%)	152 (57%)
	Powerplant	316	164 (52%)	**147** (47%)	213 (67%)	207 (65%)	180 (57%)
Avg. Frame Time (sec)	Crytek Sponza	2.65	**2.04** (77%)	**2.04** (77%)	2.36 (89%)	2.35 (88%)	2.83 (107%)
	Vegetation	2.58	**2.35** (91%)	**2.34** (91%)	2.71 (105%)	2.66 (103%)	2.94 (114%)
	Soda Hall Interior	3.21	**2.64** (82%)	2.65 (83%)	2.85 (89%)	2.76 (86%)	3.31 (103%)
	Hairball	1.70	1.57 (92%)	**1.53** (90%)	1.78 (104%)	1.72 (101%)	1.93 (113%)
	Dragon Sponza	2.31	**1.92** (83%)	1.95 (84%)	2.06 (89%)	2.20 (95%)	2.41 (104%)
	San Miguel	3.62	3.21 (89%)	**3.20** (88%)	3.71 (102%)	3.62 (100%)	4.03 (111%)
	Powerplant	7.21	**6.21** (86%)	**6.19** (86%)	7.21 (100%)	7.17 (99%)	7.93 (110%)
Avg. Triangle Tests / Ray	Crytek Sponza	6.13	**5.68** (93%)	6.11 (100%)	**5.70** (93%)	6.25 (102%)	6.13 (100%)
	Vegetation	23.3	**23.1** (99%)	23.9 (103%)	**23.1** (99%)	**23.1** (99%)	23.3 (100%)
	Soda Hall Interior	**11.2**	12.8 (114%)	13.6 (121%)	12.8 (114%)	11.3 (101%)	**11.2** (100%)
	Hairball	**28.7**	28.9 (101%)	29.8 (104%)	28.9 (101%)	28.9 (101%)	**28.7** (100%)
	Dragon Sponza	**5.57**	5.82 (104%)	6.11 (110%)	5.90 (106%)	6.09 (109%)	**5.57** (100%)
	San Miguel	**10.8**	11.0 (102%)	12.1 (112%)	11.1 (103%)	11.0 (102%)	**10.8** (100%)
	Powerplant	57.0	58.1 (102%)	58.8 (103%)	58.3 (102%)	57.7 (101%)	**56.9** (100%)
Avg. Plane Tests / Ray	Crytek Sponza	610	231 (38%)	**219** (36%)	300 (49%)	279 (46%)	324 (53%)
	Vegetation	774	337 (44%)	**327** (42%)	444 (57%)	450 (58%)	413 (53%)
	Soda Hall Interior	611	223 (37%)	**219** (36%)	271 (44%)	252 (41%)	327 (53%)
	Hairball	620	273 (44%)	**260** (42%)	356 (57%)	362 (58%)	327 (53%)
	Dragon Sponza	447	160 (36%)	**156** (35%)	195 (44%)	214 (48%)	237 (53%)
	San Miguel	688	301 (44%)	**292** (42%)	391 (57%)	382 (56%)	365 (53%)
	Powerplant	1336	543 (41%)	**529** (40%)	697 (52%)	686 (51%)	707 (53%)
Avg. Nodes Traversed / Ray	Crytek Sponza	**57.0**	113 (199%)	109 (192%)	150 (263%)	107 (188%)	**57.0** (100%)
	Vegetation	**74.9**	168 (224%)	166 (221%)	225 (300%)	160 (214%)	**74.9** (100%)
	Soda Hall Interior	**58.2**	111 (192%)	110 (190%)	137 (235%)	101 (174%)	**58.2** (100%)
	Hairball	**61.4**	136 (221%)	134 (218%)	182 (296%)	128 (208%)	**61.4** (100%)
	Dragon Sponza	**41.9**	78.5 (187%)	77.8 (186%)	97.1 (232%)	79.8 (190%)	**41.9** (100%)
	San Miguel	**64.8**	149 (230%)	147 (227%)	304 (197%)	217 (141%)	**64.8** (100%)
	Powerplant	**129**	275 (213%)	271 (210%)	357 (277%)	262 (203%)	**129** (100%)

Percentages are relative to the (uncompressed) BVH. Lower values are better. Values shown in ***red*** *are the smallest.*

using 32 bytes. Our implementation closely follows the implementation details provided by Fabianowski and Dingliana [2009]. For BIH and H-Tree, we use an optimized node representation (shown in the supplemental material) that is comparable to ours, instead of directly using the structures in the original publications [Wächter and Keller 2006; Havran et al. 2006].

We evaluate two versions of the proposed dual-split tree: *identical* which follows the BVH bounds precisely (Section 4.2.1), and *similar*

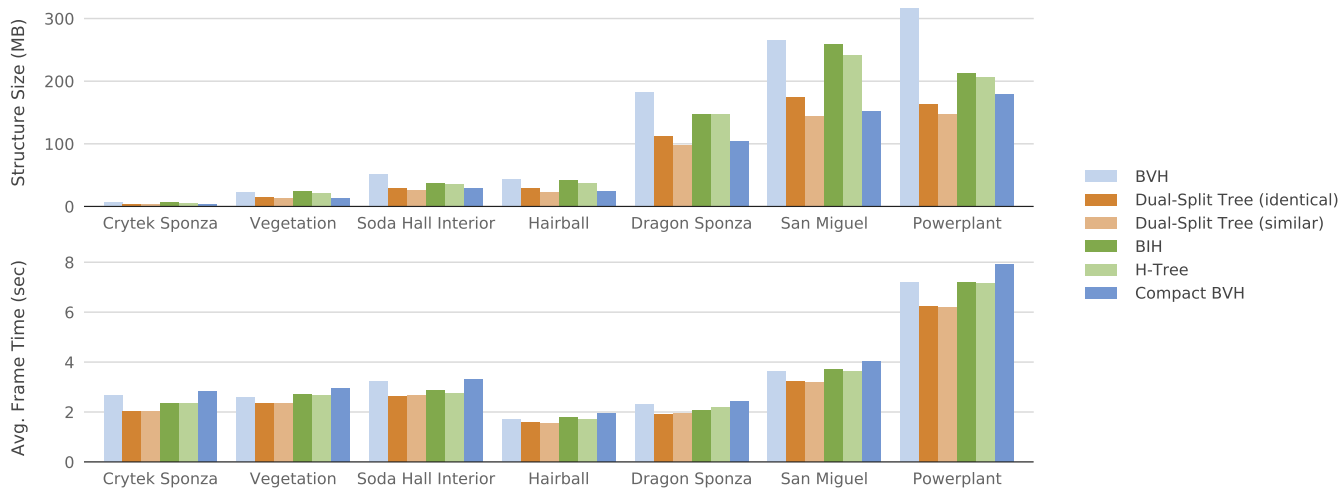


Figure 10: *Performance comparisons of several acceleration structures for different scenes: (top) acceleration structure storage and (bottom) the resulting frame render times, provided in Table 1.*

which approximates the bounds by removing some carving nodes (Section 4.2.2). Both versions are built by converting the binary BVH obtained from Embree. This conversion is much simpler than a full acceleration structure build and takes about a half of the BVH build time in our tests. For the tested scenes, Embree builds the source BVHs in 47 to 1,603 ms, while our unoptimized parallel converter generates dual-split trees in 18 to 913 ms. Both BIH and H-Tree are also converted from the same BVH which guarantees identical space partitioning. For H-Tree, we use the SAH scheme that Havran et al. provide [2006]. For BIH, we use the SAH scheme similar to ours since the authors rely on a non-greedy fast builder [Wächter and Keller 2006].

To evaluate the ray tracing performance of the acceleration structures, we rely on path tracing [Kajiya 1986] with up to five diffuse bounces without Russian Roulette. This workload generates rays that are highly incoherent. We render a variety of scenes, shown in Figure 9, at the image resolution of 1024×1024 pixels with 1 sample per pixel. We report the results as an average over 32 frames. Our results are generated using a system with an Intel i7-5930K CPU with 6 cores (12 threads) running at 3.5GHz base frequency. We provide the source code for the traversal kernels in the supplemental materials. The rest of the rendering code is common to all tested acceleration structure methods. Our traversal implementation does not use ray packets.

5.1 Acceleration Structure Storage

First, we compare the storage of the acceleration structures, shown in Table 1 and the top row of Figure 10. Overall, the identical dual-split tree uses 31 – 48% less storage than the uncompressed BVH, while the similar dual-split tree uses 39 – 53% less. The Compact BVH uniformly reduces the storage by 43% compared to BVH, while both BIH and H-Tree save less than 20% on average. Dual-split trees use less space for scenes with more regular geometry (Soda Hall Interior, Dragon Sponza, and Powerplant), because scenes with more irregular geometry (Vegetation and Hairball) have more overlapping bounding volumes. The same behavior also holds for BIH and H-Tree. While the identical dual-split tree saves less storage than the Compact BVH for scenes with more irregular geometries, the similar dual-split tree saves more storage than the Compact BVH for all tested scenes except for Vegetation.

We show the node counts and the percentages of the dual-split tree node types in Table 2. The similar dual-split tree uses fewer carving nodes than the identical dual-split tree. Table 3 shows a reduction in the number of carving node levels after a single splitting node. Since scenes with more irregular geometry include multiple levels of carving nodes, removing some of the carving nodes saves a significant amount of storage. In addition, for most tested scenes, the similar dual-split tree also generates more single-axis carving nodes because their traversal cost is lower.

Note that the dual-split trees contain more carving nodes for scenes where geometry is more irregular. As shown in Table 1, for scenes with irregular (regular) geometry, the dual-split trees require more (less) than twice as many nodes traversed per ray compared to the uncompressed BVH. Dual-split trees require more ray-plane intersection tests for the scenes with more irregular geometry. The intersection counts provide an insight of why our dual-split tree results in better improvement in performance for scenes with more regular geometry.

5.2 Frame Render Time

On average, both dual-split tree versions render frames about 17% faster than BVH flavors for our test scenes, as shown in Table 1 and the bottom row of Figure 10. More specifically, the identical dual-split tree is 8.4 – 30.0% (17.0% on average) faster, and the similar dual-split tree is 10.4 – 30.1% (17.3% on average) faster. Although the similar dual-split tree that has fewer carving nodes reduces storage, it provides a small reduction in render time in our tests compared to the identical version. In comparison, BIH and H-Tree are on average only 3.7% and 4.5% faster than the uncompressed BVH.

Table 2: *Distribution of nodes based on their type.*

	BVH	Dual-Split Tree (identical)				Dual-Split Tree (similar)			
	Total	Total	Split	Carve	Leaf	Total	Split	Carve	Leaf
Crytek Sponza	263,533	408,081	32%	63%	4.5%	362,405	36%	41%	22%
Vegetation	851,475	1,401,624	30%	67%	2.9%	1,314,868	32%	53%	15%
Soda Hall Interior	1,945,211	2,706,022	36%	53%	11%	2,571,141	38%	42%	21%
Hairball	1,640,939	2,575,607	32%	64%	3.7%	2,372,584	35%	41%	25%
Dragon Sponza	6,797,063	10,255,159	33%	60%	7%	9,853,031	34%	46%	20%
San Miguel	9,957,577	15,661,806	32%	64%	4%	14,445,716	34%	45%	20%
Powerplant	11,825,435	15,275,412	39%	52%	9%	14,855,701	40%	40%	20%

In our test scenes, the Compact BVH renders the frames 3.3 – 12.4% slower than the uncompressed BVH. Profiling shows that the performance drop is multifactorial–the prefetcher, caches, and branches all seem to have a harder time, and the compiler fails to optimize the branches in the Compact BVH traversal as well as the uncompressed BVH traversal. Although this does not agree with the results given by Fabianowski and Dingliana [2009], we speculate that the difference can be attributed to the differences in hardware (CPU vs GPU). In-particular, we speculate that the compressed nature of nodes makes hardware prefetching difficult, and CPUs have more trouble hiding the latency this introduces, owing to their memory architectures being optimized for latency over bandwidth.

5.3 Traversal Costs per Ray

While dual-split trees can maintain the space partitioning provided by the BVH, the tree structure is different. We evaluate the effects of this difference by considering the ray traversal in detail. Overall, the dual-split trees generate more nodes than the BVH (50% on average), although each dual-split tree node uses much less storage (12 or 4 bytes per dual-split tree node compared to 52 or 4 bytes per uncompressed BVH node). On average, the rays traverse 2× as many dual-split tree nodes as BVH nodes. However, because each node of a dual-split tree stores only two planes, the total number of ray-plane intersections is around 40% that of the uncompressed BVH, 51% of the BIH, and 51% of the H-Tree.

On the other hand, the identical dual-split tree does not perform exactly the same number of ray-triangle intersection tests per ray as compared to the BVH. This is because the traversal order is similar to a kd-tree: when using early ray termination, the child bounding box intersected first is not guaranteed to be traversed first, which could cause more triangle intersections compared to a BVH. As a result, this can lead to a minor increase in the number of ray-triangle intersection tests in some scenes (2% on average). This increase is more prominent with similar dual-split trees (7% on average).

6 DISCUSSION AND FUTURE WORK

It is important to note that our dual-split tree is not simply a compression scheme for a BVH. Although the dual-split tree can represent space partitioning identical to a given BVH but with less storage, the dual-split tree can also represent space partitioning that would not be possible with a BVH. Our tests use space partitioning identical (or similar) to a BVH to provide fair comparisons.

Our proposed dual-split trees significantly reduce the number of ray-plane tests required during traversal, but at the cost of visiting more nodes. On GPUs where memory latency is hidden by keeping many threads in flight, rather than relying on caches, this might not immediately translate to better performance. Increasing the number of dependent memory accesses might be detrimental to performance whereas pure compute can generally be well hidden. In fact, the state of the art in GPU traversal of wide BVHs [Ylitie et al. 2017] is to fetch fewer and larger nodes. We leave the modifications to increase the width of the dual-split trees for future work.

The proposed dual-split tree presents fertile ground for future research. Although the results in this paper build the dual-split tree from a BVH to provide direct comparisons, one can imagine a dedicated builder that can help improve the space savings and traversal performance further. Also, exploring methods to update the dual-split tree dynamically for animated scenes would be an interesting direction for future work.

Many high-performance ray tracing implementations today use BVH flavors with more than two children per node. These variants are known to provide performance improvements, especially when combined with vectorization, SIMD processing, and ray packets. While our dual-split tree can also represent space partitioning identical to such BVH variants, exploring modifications of the dual-split tree to achieve the same benefits of the non-binary BVH flavors would be an interesting future research direction.

One difficulty with the dual-split tree is that the node traversal kernel is significantly more complicated than a simple BVH traversal kernel. Therefore, both compiler and hand optimizations of the dual-split tree traversal kernel may pose additional challenges. Indeed, the significant reductions in storage and the number of ray-plane intersections with our dual-split tree correspond to a relatively small improvement in render time. We believe that the additional complexity of the dual-split tree node traversal might be one of the factors limiting the performance gain in our implementation. Additionally, different types of nodes result in code branching during traversal, further affecting performance. However, the dual-split tree may be an attractive option for the dedicated ray tracing hardware which can include specialized decoding units for any acceleration structure.

7 CONCLUSION

We have introduced a dual-split tree, an acceleration structure that is capable of representing the spatial partitioning identical to any BVH with substantial space savings and improved ray traversal performance, which outperforms previous hybrid acceleration

Table 3: *Distribution of carving node types for the dual-split tree flavors reported as a percentage of the total number of carving nodes.*

	Dual-Split Tree (identical)						**Dual-Split Tree (similar)**					
	Carving Node Levels				Single-	Dual-	Carving Node Levels				Single-	Dual-
	None	1	2	3	Axis	Axis	None	1	2	3	Axis	Axis
Crytek Sponza	14%	73%	12%	0.10%	65%	35%	49%	44%	6.1%	0.07%	72%	28%
Vegetation	9.8%	71%	19%	0.43%	64%	36%	33%	53%	14%	0.28%	66%	34%
Soda Hall Interior	31%	65%	4.6%	0.10%	75%	25%	49%	48%	3.6%	0.10%	77%	23%
Hairball	15%	70%	15%	0.29%	75%	25%	52%	37%	11%	0.17%	77%	23%
Dragon Sponza	16%	78%	6.1%	0.003%	71%	29%	39%	56%	5.3%	0.004%	70%	30%
San Miguel	13%	74%	13%	0.36%	62%	38%	43%	48%	8.6%	0.3%	68%	32%
Powerplant	35%	62%	2.9%	0.06%	76%	24%	53%	45%	2.6%	0.05%	79%	21%

Carving Node Levels *shows the number of carving node levels used consecutively after a splitting node.* None *means no carving nodes between two splitting nodes. The* Single-Axis *and the* Dual-Axis *columns report the percentages of the single-axis and the dual-axis carving nodes. Note that the similar dual-split tree removes a considerable number of carving nodes: the corresponding* None *column has a much larger percentage.*

structures. By using a traversal order similar to a kd-tree but adding one more splitting plane and additional carving planes which can align to different axes, our dual-split tree significantly reduces the number of ray-plane intersection tests during ray traversal through a BVH-like spatial partitioning. We have also presented a simple method for converting any BVH into the dual-split tree with identical or similar space partitioning. We expect that future research on this new data structure can reveal additional improvements beyond the performance and space-saving benefits we report in this paper.

ACKNOWLEDGMENTS

This work was supported in part by NSF Grant #1409129. Scenes: Crytek Sponza (F. Meinl, M. Dabrovic), Vegetation and Hairball (S. Laine), Dragon Sponza (M. Dabrovic, Stanford U., C. Yuksel), San Miguel (G. L. Laguno), and Powerplant (U. of North Carolina).

REFERENCES

Timo Aila and Tero Karras. 2010. Architecture Considerations for Tracing Incoherent Rays. In *High-Performance Graphics (HPG '10)*. 113–122.

Timo Aila, Tero Karras, and Samuli Laine. 2013. On Quality Metrics of Bounding Volume Hierarchies. In *High-Performance Graphics (HPG '13)*. 101–107.

Pablo Bauszat, Martin Eisemann, and Marcus A Magnor. 2010. The Minimal Bounding Volume Hierarchy. In *Vision, Modeling, and Visualization*. 227–234.

Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517.

Holger Dammertz, Johannes Hanika, and Alexander Keller. 2008. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. In *Proceedings of the Nineteenth Eurographics Conference on Rendering (EGSR '08)*. 1225–1233.

Leonardo R. Domingues and Helio Pedrini. 2015. Bounding Volume Hierarchy Optimization Through Agglomerative Treelet Restructuring. In *High-Performance Graphics (HPG '15)*. 13–20.

M. Ernst and G. Greiner. 2008. Multi bounding volume hierarchies. In *2008 IEEE Symposium on Interactive Ray Tracing*. 35–40.

Bartosz Fabianowski and John Dingliana. 2009. Compact BVH storage for ray tracing and photon mapping. In *Proc. of Eurographics Ireland Workshop*. 1–8.

H. Futchs, Z. M. Kedem, and B. F. Naylor. 1988. Tutorial: Computer Graphics; Image Synthesis. Chapter On Visible Surface Generation by a Priori Tree Structures, 39–48.

Vlastimil Havran. 2000. *Heuristic Ray Shooting Algorithms.* Ph.D. Thesis. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel. 2006. On the fast construction of spatial hierarchies for ray tracing. In *IEEE Symposium on Interactive Ray Tracing*. IEEE, 71–80.

James T. Kajiya. 1986. The Rendering Equation. In *Proceedings of the Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '86)*. 143–150.

Tero Karras and Timo Aila. 2013. Fast Parallel Construction of High-quality Bounding Volume Hierarchies. In *High-Performance Graphics (HPG '13)*. 89–99.

S. Keely. 2014. Reduced Precision for Hardware Ray Tracing in GPUs. In *High-Performance Graphics (HPG '14)*. 29–40.

A. Kensler. 2008. Tree rotations for improving bounding volume hierarchies. In *2008 IEEE Symposium on Interactive Ray Tracing*. 73–76.

Tae-Joon Kim, Yongyoung Byun, Yongjin Kim, Bochang Moon, Seungyong Lee, and Sung-Eui Yoon. 2010a. HCCMeshes: Hierarchical-Culling oriented Compact Meshes. *Computer Graphics Forum* 29, 2 (2010), 299–308.

Tae-Joon Kim, Bochang Moon, Duksu Kim, and Sung-Eui Yoon. 2010b. RACBVHs: Random-accessible compressed bounding volume hierarchies. *IEEE Transactions on Visualization and Computer Graphics* 16, 2 (2010), 273–286.

Daniel Kopta, Thiago Ize, Josef Spjut, Erik Brunvand, Al Davis, and Andrew Kensler. 2012. Fast, Effective BVH Updates for Animated Scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '12)*. 197–204.

C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum* 28, 2 (2009), 375–384.

Gábor Liktor and Karthikeyan Vaidyanathan. 2016. Bandwidth-efficient BVH layout for incremental hardware traversal. In *High-Performance Graphics (HPG '16)*. 51–61.

J David MacDonald and Kellogg S Booth. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer* 6, 3 (1990), 153–166.

J. Mahovsky and B. Wyvill. 2006. Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing. *Computer Graphics Forum* 25, 2 (2006), 173–182.

Beng C Ooi. 1987. Spatial kd-tree: A data structure for geographic database. In *Datenbanksysteme in Büro, Technik und Wissenschaft*. Springer, 247–258.

J. Pantaleoni and D. Luebke. 2010. HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry. In *High-Performance Graphics*. 87–95.

Steven M Rubin and Turner Whitted. 1980. A 3-dimensional representation for fast rendering of complex scenes. In *ACM SIGGRAPH Comp. Graphics*, Vol. 14. 110–116.

Benjamin Segovia and Manfred Ernst. 2010. Memory Efficient Ray Tracing with Hierarchical Mesh Quantization. In *Graphics Interface 2010 (GI '10)*. 153–160.

Martin Stich, Heiko Friedrich, and Andreas Dietrich. 2009. Spatial Splits in Bounding Volume Hierarchies. In *High-Performance Graphics (HPG '09)*. 7–13.

Karthikeyan Vaidyanathan, Tomas Akenine-Möller, and Marco Salvi. 2016. Watertight ray traversal with reduced precision.. In *High-Performance Graphics (HPG '16)*. 33–40.

Marek Vinkler, Jiri Bittner, and Vlastimil Havran. 2017. Extended Morton Codes for High-Performance Bounding Volume Hierarchy Construction. In *High-Performance Graphics (HPG '17)*. 9:1–9:8.

Carsten Wächter. 2008. *Quasi-Monte Carlo light transport simulation by efficient ray tracing.* Ph.D. Dissertation. Universität Ulm.

Carsten Wächter and Alexander Keller. 2006. Instant ray tracing: The bounding interval hierarchy. *Rendering Techniques* 2006 (2006), 139–149.

I. Wald, C. Benthin, and S. Boulos. 2008. Getting rid of packets - Efficient SIMD single-ray traversal using multi-branching BVHs. In *2008 IEEE Symposium on Interactive Ray Tracing*. 49–57.

Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. 2009. State of the Art in Ray Tracing Animated Scenes. *Computer Graphics Forum* 28, 6 (2009), 1691–1722.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph.* 33, 4 (July 2014), 143:1–143:8.

Sven Woop, Gerd Marmitt, and Philipp Slusallek. 2006. B-KD trees for hardware accelerated ray tracing of dynamic scenes. In *Graphics Hardware*. 67–77.

Henri Ylitie, Tero Karras, and Samuli Laine. 2017. Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs. In *High-Performance Graphics (HPG '17)*. 4:1–4:13.