Bounding Volume Hierarchies versus Kd-trees on Contemporary Many-Core Architectures

Marek Vinkler*
Faculty of Informatics
Masaryk University

Vlastimil Havran
Faculty of Electrical Engineering
Czech Technical University in Prague

Jiří Bittner
Faculty of Electrical Engineering
Czech Technical University in Prague

Abstract

We present a performance comparison of bounding volume hierarchies and kd-trees for ray tracing on many-core architectures (GPUs). The comparison is focused on rendering times and traversal characteristics on the GPU using data structures that were optimized for maximum performance of tracing rays irrespective of the time needed for their build. We show that for a contemporary GPU architecture (NVIDIA Kepler) bounding volume hierarchies have higher ray tracing performance than kd-trees for simple and moderately complex scenes. Kd-trees, on the other hand, have higher performance for complex scenes, in particular for those with occlusion.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing; I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types; I.3.1 [Computer Graphics]: Hardware architecture—Parallel processing

Keywords: bounding volume hierarchy, kd-tree, performance study, ray tracing

1 Introduction

Solving visibility by ray tracing stands at the core of a number of rendering algorithms, particularly those aiming at computing global illumination. The efficiency of the ray tracing algorithm has significant influence on the total rendering time and thus much research work has been devoted to ray tracing optimization. The main factors influencing the ray tracing performance are the properties and the organization of data structures which spatially index the scene geometry, thus lowering the number of operations needed to find ray-primitive intersections. Over the past decades two acceleration data structures became prominent for this task: bounding volume hierarchies (BVHs) and kd-trees. These two data structures have been compared against each other a few times on various platforms. Due to the development of parallel many-core architectures and optimized algorithms most of these studies are now outdated. In this paper we compare the ray tracing performance of these acceleration data structures irrespective of their build times, therefore targeting offline rendering algorithms and/or static scenes. For those the time of building the data structure is insignificant compared to the rendering time.

*e-mail: xvinkl@fi.muni.cz, author for correspondence

The paper is further structured as follows. Section 2 summarizes the most relevant literature dealing with BVH and kd-tree build and the comparison of the two data structures. Section 3 describes the build algorithms used in our measurements and their termination criteria. The results are presented in Section 4 with their limitations stated in Section 5 and their discussion provided in Section 6. Section 7 concludes the paper and provides some remarks for a future work.

2 Related Work

Below we present shortly the most relevant work related to the data structures tested and evaluated in our paper.

BVHs. The BVHs [Weghorst et al. 1984] have been recently studied in the context of ray traversal efficiency and build algorithms on the GPU. Kopta et al. [2012] focused on fast updates during animation while keeping high traversal efficiency. Bittner et al. [2013] and Karras and Aila [2013] both target improvement of quality of the already built BVHs. The method of Bittner et al. runs on a CPU and produces highest quality BVH, while the method of Karras and Aila runs on a GPU and produces slightly lower quality trees, but in significantly less time. The factors influencing the BVH traversal times on many-core processors were analyzed by Aila et al. [2013] explaining the discrepancy between the surface area heuristic (SAH) cost [Goldsmith and Salmon 1987] and the measured performance. Gu et al. [2013] presented a method for building BVHs using agglomerative clustering that allows for setting a tradeoff between build time and traversal efficiency.

Kd-trees. The kd-trees introduced by Kaplan [1985] have been recently studied in the context of parallelization of the building algorithm on both CPUs [Choi et al. 2010; Roccia et al. 2012] and GPUs [Wu et al. 2011; Roccia et al. 2012]. Choi et al. [2010] focused on accelerating SAH kd-tree build algorithm on multi-core CPUs. To simplify the parallelization they omitted split clipping [Havran and Bittner 2002] from the method, which resulted in lower quality trees. Wu et al. [2011] moved the entire kd-tree build algorithm of Wald and Havran [2006] including split clipping to the GPU, significantly accelerating the algorithm. A hybrid CPU-GPU implementation of the same baseline algorithm was proposed by Roccia et al. [2012], that outperformed the previous approaches.

Performance studies. The efficiency of acceleration data structures for ray tracing was compared in several studies. Havran [2000] formulated hardware independent measures and studied properties of twelve acceleration data structures. He concluded at that time (year 2000), that kd-trees have the highest ray traversal performance on average from all the data structures tested for static scenes on CPUs. Another comparison study by Masso and Lopez [2003] showed similarly to Havran [2000] that BVHs built up by insertion using the algorithm of Goldsmith and Salmon [1987] are significantly worse than top-down built kd-trees. On the GPUs data structures for ray tracing were compared by Zlatuška and Havran [2010]. Their study targeted older GPU hardware (GTX 280 and 8600GT) of year 2007/8 and showed that the

BVHs are the fastest for coherent rays while the kd-trees are the fastest for incoherent rays.

The utility of dynamic data structures proposed for animated scenes were surveyed by Wald et al. [2009]. In the survey the authors suggest that kd-trees are not efficient for dynamic scenes because of their slow rebuild and propose to use uniform grids or BVHs depending on the distribution of primitives in the scene. This survey, however, lacks a direct performance comparison of the discussed methods.

3 Data Structures

We build the two data structures, a binary BVH and a binary kd-tree, fully on the GPU using the algorithm of Vinkler et al. [2013] which is integrated with the framework of Aila and Laine [2009]. We have further extended this framework with a GPU code for building and traversing kd-trees. The kd-tree traversal kernel is organized similarly to the BVH traversal kernel in the while-while layout and with the same triangle intersection test.

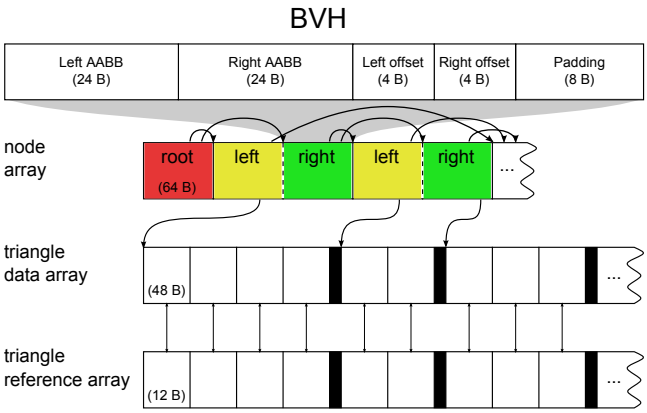

Figure 1: Memory and node layout of BVHs used in our measurements. For BVHs the triangle data array is accessed directly by the traversal algorithm. The leaf is ended with a stop mark (black rectangle in the figure). The BVH node takes 64 Bytes: 24 Bytes for each of the children axis-aligned bounding boxes (AABB), two 4 Byte offsets to the left and right children and a 8 Byte padding.

3.1 BVHs

Construction. We build the BVHs with the surface area heuristic (SAH) used for subdividing inner nodes and for automatic termination of the subdivision. The SAH cost is evaluated in each node for a fixed set of candidate planes (binning). The best splitting plane according to SAH is chosen from 32 splitting plane candidates uniformly distributed inside the bounding box of a node in all three axes (11 planes parallel to the x-axis, 11 planes parallel to the y-axis, and 10 planes parallel to the z-axis). If all 32 candidate planes fail to subdivide the geometric primitives into two non-empty parts, the primitives are subdivided into two equally sized parts. The fixed number of candidate planes is chosen to fit the warp size on the current generation of GPUs, thus allowing for efficient implementation of the build algorithm

A leaf is created when the number of triangles in a node is smaller than or equal to a given threshold ($n_{max} = 4$, triangle count criterion) or when the SAH cost of not subdividing a node is less than the cost of subdividing it (cost criterion). The value $n_{max} = 4$ was chosen from values 2, 4, 8, 16 as it gives the fastest traversal times for shooting incoherent rays over our test scenes (see Table 1). The depth of the hierarchy is not limited since the build is always terminated by the two criteria.

Memory Layout. Figure 1 shows the memory layout of our BVHs, as proposed by Aila and Laine [2009]. The size of each node in the node array is 64 Bytes with children of each node stored in a consecutive chunk of memory. The most significant bit of the child offset determines whether the offset points to an inner node or the first triangle in a leaf. Triangles falling to the same leaf are stored sequentially in memory and followed by a stop mark (0x80000000) that gives the end of the leaf. Triangle's vertex data are stored in the triangle data array and take up $3 \times 12 = 48$ Bytes. Triangle references are stored in a separate triangle reference array with each reference occupying $3 \times 4 = 12$ Bytes. While only 4 Bytes are sufficient for the reference index the data are padded to 12 Bytes so that the same offset can be used to access both the triangle data and triangle reference arrays. The triangle reference array is used to identify the hit triangle's indices which are later used e.g. for shading computation. This layout was implemented in the original framework [Aila and Laine 2009]. The triangle array layout also supports a single triangle to be stored in multiple leaves through duplication of the triangle's data but this is not used in our BVHs.

Traversal. For tracing the rays we use the speculative while-while traversal algorithm of Aila and Laine [2009] which proved to be the fastest one for the BVHs. This algorithm is stacked based and stores the traversal stack in local memory, which is part of the GPU DRAM. The algorithm traces the rays separately of each other (not using packet traversal algorithm) leading to high performance on incoherent rays.

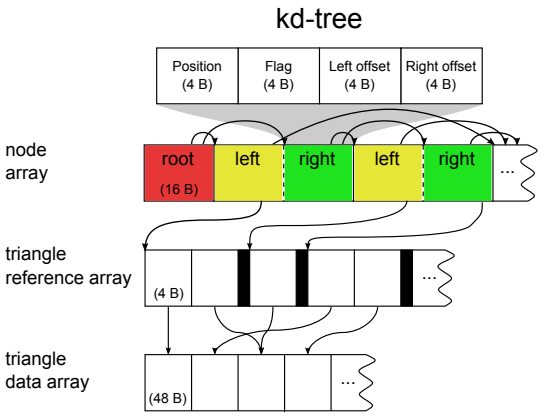

Figure 2: Memory and node layout of kd-trees used in our measurements. For kd-trees the triangle data array is accessed with a dependent memory access through the triangle reference array because the triangle references may be duplicated. The leaf is ended with a stop mark (black rectangle in the figure). The kd-tree node takes 16 Bytes: the position of the splitting plane stored in 4 Byte floating point value, a 4 Byte information for the type of the node and two 4 Byte offsets to the child nodes.

3.2 Kd-trees

Construction. Similarly to BVHs we build SAH kd-trees using 32 splitting plane candidates in each node evenly distributed along the x, y, z extents of its axis-aligned bounding box. Given the axis-aligned bounding box of an interior node, the splitting plane candi-

Scene	BVH								Kd-tree							
	$P_{primary}$ $[Mrays/s]$				$P_{diffuse}$ $[Mrays/s]$				$P_{primary}$ $[Mrays/s]$				$P_{diffuse}$ $[Mrays/s]$			
	2	4	8	16	2	4	8	16	2	4	8	16	2	4	8	16
Sibenik	263.2	263.2	256.4	217.4	70.5	69.3	60.9	46.2	222.2	222.2	217.4	181.8	39.8	39.7	37.1	30.5
Fairy Forest	169.5	169.5	169.5	156.3	69.0	69.4	65.8	54.3	119.0	120.5	113.6	101.0	39.5	38.9	35.6	28.8
Crytek Sponza	163.9	163.9	156.3	140.8	46.8	45.9	41.2	32.7	153.8	151.5	144.9	126.6	37.9	37.6	34.1	26.4
Happy Buddha	238.1	243.9	232.6	200.0	175.4	176.6	161.3	129.4	122.0	123.5	125.0	111.1	86.5	85.5	79.2	64.0
Office House	65.8	65.8	65.8	62.5	19.3	19.4	19.3	17.9	135.1	133.3	128.2	113.6	41.8	41.2	39.2	33.6
Sodahall	312.5	312.5	294.1	263.2	116.6	115.4	107.4	89.1	277.8	263.2	232.6	196.1	86.5	83.5	74.5	58.7
Hairball	57.1	57.5	60.6	60.6	33.5	33.7	35.1	33.9	34.2	34.1	33.0	29.2	23.1	22.7	21.5	17.9
Houses 3x3	181.8	181.8	169.5	147.1	96.3	94.5	86.0	68.7	151.5	153.8	144.9	116.3	81.8	79.7	71.5	55.0
San Miguel	62.1	63.3	63.3	58.1	22.1	22.2	21.1	17.8	59.9	58.8	54.6	46.1	19.9	19.3	17.1	13.6
MPII subset	140.8	138.9	137.0	129.9	56.7	56.6	55.4	50.2	192.3	188.7	175.4	144.9	82.9	80.3	73.2	55.0
Houses 6x5	126.6	129.9	126.6	111.1	64.4	64.6	61.2	50.9	135.1	137.0	125.0	105.3	71.2	68.7	60.7	48.3
Powerplant	62.1	62.1	62.9	62.9	19.4	19.4	19.6	18.9	103.1	101.0	93.5	75.2	37.2	36.2	33.0	25.8
Four SPD [Haines 1987] scenes with triangles used in [Havran 2000]																
mount8	526.3	526.3	476.2	400.0	396.0	381.0	332.0	267.6	153.8	156.3	158.7	140.8	103.9	103.8	98.5	78.8
sombrero4	416.7	384.6	344.8	285.7	327.9	320.0	285.7	233.2	212.8	217.4	232.6	204.1	163.3	162.3	155.9	130.7
teapot40	454.5	454.5	434.8	370.4	285.7	279.7	248.4	201.5	163.9	163.9	161.3	142.9	103.9	103.0	93.3	75.8
tetra8	666.7	666.7	555.6	500.0	606.1	606.1	479.0	416.7	129.9	131.6	123.5	107.5	159.7	159.4	141.3	113.2
Average	137.22	137.69	136.17	126.28	53.85	53.76	51.81	45.23	115.6	115.3	110.2	94.8	50.9	50.0	45.8	36.7

Table 1: The dependence of the rendering performance of the BVH and the kd-tree on the maximum number of triangles in a leaf (n_{max}). $P_{primary}$ is the ray tracing performance for primary rays in Mrays/s, and $P_{diffuse}$ is the ray tracing performance in Mrays/s for shooting 8 diffuse sample rays per primary ray. The average performance is computed from the average traversal time in milliseconds (different from average of performances in Mrays/s).

dates are distributed the same way as for the BVH. Unlike for the BVHs, when all triangles fall into one child of a node, an empty leaf is created for the other child.

The kd-trees are built with split clipping [Havran and Bittner 2002] to get high quality trees. With split clipping enabled a triangle is only considered to fall into a child if it intersects its bounding box. In the original algorithm the bounding boxes of all triangles and their fragments are maintained, and shrunk upon intersections by splitting planes. These auxiliary bounding boxes are then used for more precise triangle-child intersection tests. This technique is difficult to implement on GPUs as it requires frequent dynamic memory allocation for the newly created auxiliary bounding boxes. We implement split clipping by directly computing the intersection of each triangle with the bounding boxes of the children of the currently subdivided node. The new formulation that repeatedly computes the intersections between triangles and bounding boxes does not require to keep the auxiliary boxes, but is more computationally intensive. The proposed solution fits well the GPU architecture. We use the algorithm of Akenine-Möller [2005] for the intersection of an axis-aligned box and a triangle.

We use the termination criteria of Havran and Bittner [2002] with modified constants to achieve fast ray traversal algorithm on the GPU. A leaf is created when (1) the number of triangles in a node is smaller than or equal to $n_{max} = 2$ (triangle count criterion) or (2) the depth of the node is higher than the maximum allowed depth, $d_{max} = k_1.log_2 N + k_2$ with $k_1 = 1.2$ and $k_2 = 2.0$ (node depth criterion), or (3) the node has failed to pass the cost criterion several times. The cost criterion specifies failure by terms of the cost of a subdivided node and an unsubdivided one: $C_{new}/C > r_q^{min}$ with $r_q^{min} = 0.9$. The maximum number of failures of the cost criterion along the path from the root to the current node is set to $F_{max} = K_{fail}^1 + K_{fail}^2.d_{max}$ with $K_{fail}^1 = 1.0$ and $K_{fail}^2 = 0.26$.

Scene/r_q^{min}	0.7	0.8	0.9	1.0	1.1	1.2
	$P_{primary}$ $[Mrays/s]$					
Crytek Sponza	114.94	153.85	153.85	153.85	156.25	153.85
San Miguel	51.81	59.17	59.52	59.52	59.52	59.52
Scene	$P_{diffuse}$ $[Mrays/s]$					
Crytek Sponza	21.41	35.97	37.93	38.22	38.22	38.20
San Miguel	16.39	19.48	19.62	19.56	19.48	19.50

Table 2: The effect of changing r_q^{min} on the rendering performance of the kd-tree on two complex test scenes.

The value of n_{max} was chosen from values 2, 4, 8, 16 as it gives the fastest ray traversal times over our test scenes (see Table 1). The value of k_1 was selected from the range $\langle 1.0, 1.2 \rangle$ with step 0.1 and k_2 was selected from the range $\langle 2.0, 8.0 \rangle$ with step 1.0. The value of r_q^{min} was selected from the range $\langle 0.7, 1.2 \rangle$ with step 0.1 for two scenes to get the highest performance as shown in Table 2. Similarly, K_{fail}^2 was selected from the range $\langle 0.15, 0.35 \rangle$ with step 0.01 to provide overall the highest performance for ray traversal algorithm. The constants k_1, k_2, K_{fail}^1 and K_{fail}^2 are used for tuning the build process based on the scene complexity.

Memory Layout. Figure 2 shows the memory layout of our kd-tree, which is inspired by the BVH layout, but it is more memory efficient when triangles are split into multiple references. The children of an inner node are again stored in a consecutive chunk of memory with each node taking 16 Bytes. Similar to the BVH the most significant bit of the child offset differentiates between an inner node offset and a triangle offset. A triangle offset with the largest value is reserved for empty leaves so that no extra memory is required for them. The flag variable in the node structure holds information about the axis of the split plane. The 8 Byte node lay-

out of Wald et al. [2001] cannot be used as the kd-tree is being built in parallel; this would require to rewrite the whole kd-tree at the end to the new memory layout. The triangle offsets point to the triangle reference array instead of the triangle data array to prevent duplication of triangle data. Only triangle references are duplicated and each reference occupies just 4 Bytes to save space. The triangle references falling into a leaf are again stored in consecutive memory and the leaf is terminated with a stop mark as in the BVH layout.

Traversal. The ray traversal algorithm is similar to the one for the BVH, but its speculative variant is not used. We have experimented with this optimization and found out it slows down the traversal algorithm when used for kd-trees. We suspect this is caused by the more scattered memory access of kd-trees leading to increased memory bandwidth and latency. Also this optimization mitigates the benefit of the early termination and increases code complexity. According to the classification used in [Hapala and Havran 2011] we are using a recursive stack based traversal algorithm with the near/far sorting of child nodes based on ray direction (see Algorithm 1). The traversal stack keeps just two values per entry (node address and exit distance) instead of the three often used (node address, entry distance, and exit distance).

Algorithm 1: Traversal code for the kd-tree, *tmin* is the entry distance of a ray in the current node and *tmax* is the exit distance in the current node.

```
1  Traverse() begin
2      Intersect ray with scene AABB, compute tmin and tmax;
3      while (not hit and tmax > tmin) do
4          while (node is inner node and tmax > tmin) do
5              Fetch Kdtree node;
6              t ← (split − orig) / dir;
7              Choose near/far based on ray direction;
8              if (t ≥ tmax) then
9                  node ← near;
10             else if (t ≤ tmin) then
11                 node ← far;
12             else
13                 node ← near;
14                 push(far, tmax);
15                 tmax ← t;
16         while (node is leaf) do
17             for (each triangle in node) do
18                 intersect triangle, store distance in t;
19                 tmax ← t;
20             if (not hit) then
21                 tmin ← tmax;
22                 pop(node, tmax);
23             else
24                 break;
```

4 Results

We evaluated the algorithms on a PC with Intel Core i7-2600, 16 GB of RAM and NVIDIA GeForce GTX 680 running 64-bit Windows 7.

Both data structures were tested on sixteen scenes with varying number of primitives and depth complexity. Four of the scenes were taken from the Standard Procedural Database (SPD) [Haines 1987] to allow comparison on the triangular scenes with the old performance study [Havran 2000]. The measured data were averaged over four viewpoints for the non-SPD scenes, while for the SPD scenes a single viewpoint given by the SPD proposal was used. The images of rendered scenes are shown in Figure 3. The measurement of ray tracing performance was run five times for each viewpoint, with the maximum performance reported.

The summary results from measurements with BVHs and kd-trees on our test scenes are shown in Table 3. Since the kd-tree is a space subdivision data structure, its build produces more nodes and triangle references (pointers to the same triangle data) than the build of the BVH. The number of references cannot be estimated in advance. In our measurements up to 14.6× more nodes and 11.8× more references were created for kd-trees than for BVHs for the Hairball scene. This increases both the build times and the memory footprint of the acceleration data structure.

Multiple reference to triangles due to splitting in kd-trees can both increase or decrease the ray tracing performance. The decrease of the ray tracing performance happens because more memory accesses are required, and those are more scattered in the address space. On the other hand, the capability to decrease the triangle bounds and the early termination of ray traversal algorithm upon the first hit allows the kd-tree to achieve lower numbers of ray-triangle intersection tests (N_{it}) and traversal steps (N_{ts}). The reduction in the number of algorithmic steps can become significant for complex scenes with high occlusion. For example in the Office House scene the kd-tree performs only 17% ray-triangle intersection tests (N_{it}) and 30% traversal steps (N_{ts}) compared to the BVH and this reduction leads to a twofold speedup. For some other scenes even significant reduction in the number of operations need not translate to faster traversal algorithm, such as for the San Miguel scene. When the number of algorithmic steps (both traversal steps and intersection tests) is roughly similar (Happy Buddha and teapot40) for both BVHs and kd-trees, then the ray tracing with kd-trees is about twice slower than for the BVHs.

Taking into account the scene complexities, the BVH is usually more efficient on small to medium sized scenes and less efficient on large scenes as shown in the graph in Figure 4. The Office House scene is an exception to this rule (small to medium sized scenes) because it is formed by many large triangles causing significant overlaps of BVH nodes, so the ray tracing with kd-tree is about twice faster than with BVH.

Even if the build times for the data structures on a GPU are not the subject of our study, we mention them briefly. We have found out that for top-down build algorithms the times are by factor of 10 to 20 times higher for kd-trees than for BVHs. This is due to the need for the dynamic memory allocation required for kd-trees that is rather slow on a GPU. When we do not use the split clipping, the build times for kd-trees are then decreased by about 20%. The major bottleneck is the dynamic memory allocation needed for multiple triangle references in kd-trees and the increased memory traffic caused by the new triangle references.

We have tested four SPD scenes [Haines 1987] consisting purely of triangles (as our software framework supports only such scenes) that were also used in Havran's Ph.D. thesis [2000]. Comparing the results from our kd-tree implementation to those in Havran's Ph.D. thesis, measured for the same geometry data and the same viewpoints, we can see that similar trees are built (number of nodes, triangle references, and traversal characteristics). The main difference in the build is that we only use 32 uniformly distributed candidate splitting planes instead of all the planes coinciding with the bounding boxes of triangles (maximum 6.N candidate splitting planes for N triangles for all three axes). Using only a few planes was reported

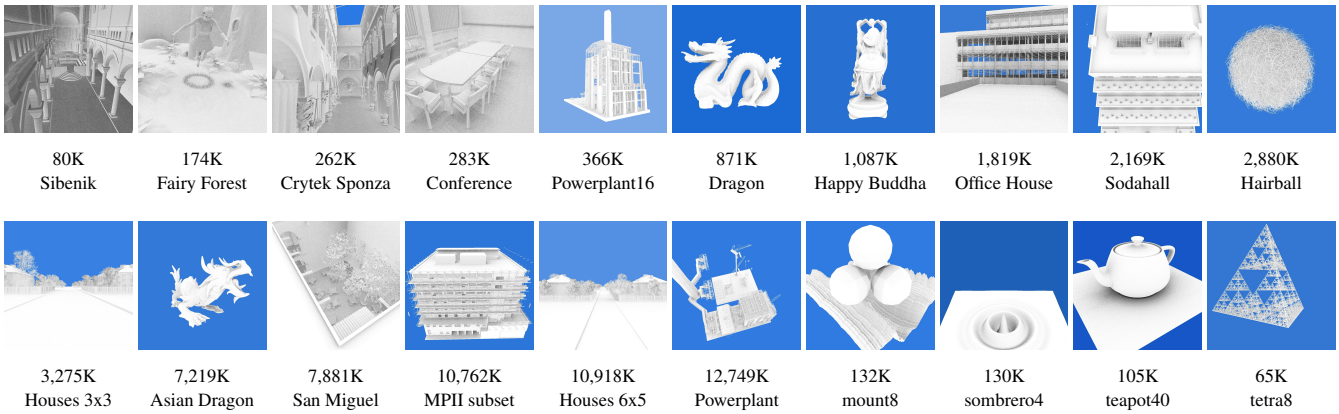

Figure 3: Rendered images of our twenty test scenes in resolution 1024×1024 pixels. Eight diffuse samples per primary ray were used for rendering. The MPII subset consists of layers #1,4,7,8,9,10,12,13 of the whole model [Havran et al. 2009].

Scene	$N_{tris}[-]$	$N_G/N_{tris}[-]$			N_{ref}/N_{tris} $[-]$		*Memory* $[MB]$		$N_{it}[-](primary)$			$N_{ts}[-](primary)$			$P_{primary}$ $[Mrays/s]$			$P_{diffuse}$ $[Mrays/s]$		
		BVH	KDT	ratio	BVH	KDT	BVH	KDT	BVH	KDT	ratio	BVH	KDT	ratio	BVH	KDT	ratio	BVH	KDT	ratio
Sibenik	80K	0.32	0.87	2.73	1.0	3.4	1.87	2.11	8.6	8.1	0.94	64.4	36.8	0.57	263.2	222.2	0.84	69.5	40.3	0.58
Fairy Forest	174K	0.31	0.91	2.91	1.0	3.5	3.99	4.75	12.4	11.2	0.90	64.2	51.0	0.79	169.5	120.5	0.71	69.4	39.7	0.57
Crytek Sponza	262K	0.30	1.36	4.49	1.0	4.2	5.84	9.72	12.4	7.6	0.61	95.7	49.0	0.51	163.9	153.8	0.94	45.8	37.9	0.83
Happy Buddha	1,087K	0.33	2.32	7.02	1.0	5.0	26.11	59.44	3.9	4.0	1.03	35.0	29.6	0.85	243.9	122.0	0.50	176.6	86.5	0.49
Office House	1,819K	0.27	0.65	2.38	1.0	3.3	35.98	39.65	58.8	9.8	0.17	142.7	42.6	0.30	65.8	131.6	2.00	19.3	42.0	2.17
Sodahall	2,169K	0.30	1.03	3.43	1.0	3.8	48.09	66.12	5.6	4.5	0.80	64.1	41.3	0.64	312.5	277.8	0.89	115.4	87.0	0.75
Hairball	2,880K	0.30	4.42	14.63	1.0	11.8	64.13	324.95	24.8	16.7	0.67	101.3	77.3	0.76	57.5	34.4	0.60	33.6	23.0	0.68
Houses 3x3	3,275K	0.33	1.84	5.66	1.0	4.6	77.54	150.69	5.9	4.4	0.75	45.1	31.6	0.70	181.8	153.8	0.85	94.1	81.8	0.87
San Miguel	7,881K	0.31	1.19	3.82	1.0	3.8	180.70	259.07	16.5	6.6	0.40	135.9	77.5	0.57	63.3	59.5	0.94	22.0	19.6	0.89
MPII subset	10,762K	0.28	1.28	4.56	1.0	4.5	225.78	397.01	9.3	3.4	0.37	83.3	34.8	0.42	138.9	212.8	1.53	56.8	91.2	1.61
Houses 6x5	10,918K	0.32	2.60	8.00	1.0	5.4	257.99	660.80	8.2	4.5	0.55	55.3	35.3	0.64	128.2	147.1	1.15	64.6	75.2	1.16
Powerplant	12,749K	0.26	0.95	3.70	1.0	3.9	247.76	378.16	46.3	8.5	0.18	114.9	49.7	0.43	62.1	112.4	1.81	19.4	40.3	2.08
Four SPD [Haines 1987] scenes with triangles used in [Havran 2000]																				
mount8	132K	0.28	1.75	6.33	1.0	4.4	2.72	5.72	5.2	5.9	1.13	30.8	33.9	1.10	384.6	156.3	0.41	248.4	105.7	0.43
sombrero4	130K	0.29	1.78	6.24	1.0	4.5	2.76	5.81	2.6	3.5	1.35	18.3	16.5	0.90	526.3	212.8	0.40	386.5	163.6	0.42
teapot40	105K	0.30	2.12	7.03	1.0	5.2	2.34	5.52	3.3	3.4	1.03	26.6	26.7	1.00	370.4	166.7	0.45	224.7	106.2	0.47
tetra8	65K	0.25	2.34	9.36	1.0	5.5	1.25	3.73	1.7	3.5	2.06	16.7	23.1	1.38	588.2	137.0	0.23	540.5	163.3	0.30
Average	-	0.30	1.71	5.77	1.0	4.8	74.05	148.33	14.1	6.6	0.81	68.4	41.0	0.72	136.8	117.7	0.89	53.3	51.6	0.91

Table 3: Comparison of various statistics of BVHs and kd-trees (KDT) on 16 scenes including ratios of some characteristics. N_{tris} is the number of scene triangles, N_G/N_{tris} is the number of built nodes divided by the number of scene triangles, N_{ref}/N_{tris} is the number of references to triangles divided by the number of scene triangles (1.0 for BVHs because there is no triangle splitting), *Memory* is the summed memory consumed by the nodes and triangle references of the data structure in MBytes, N_{it} is the number of ray-triangle intersection tests per ray, N_{ts} is the number of traversal steps per ray, $P_{primary}$ is the ray tracing performance for primary rays in Mrays/s, and $P_{diffuse}$ is the ray tracing performance for 8 diffuse samples in Mrays/s. The average performance is computed from the average traversal time in milliseconds (different from average of performances in Mrays/s). The value of the performance ratio column indicates the speedup of the kd-tree over the BVH (value greater than 1.0 corresponds to the kd-tree being faster than the BVH).

by several authors to only slightly decrease the quality of the data structure. For kd-trees Hurley et al. [2002] showed that there is little benefit for having more than 32 split plane candidates per axis and Hunt et al. [2006] used 8 uniform and 8 adaptive planes per axis. For BVHs Wald [2007] uses 7 candidate planes per axis.

Given our kd-tree is only slightly lower quality than the one of Havran [2000] we can compare the ray traversal performances. The measurements for primary rays show that only the development of the hardware accounts for an average performance speedup of 1070× for tracing rays in a span of 15 years. Interestingly, this practically matches the progress predicted by Moore's law (doubling of computation performance every 18 months): speedup of $2^{10} = 1024\times$ for 15 years (single core Pentium II in 1997 to massively parallel GPU architecture Kepler GK104 in 2012). This is most likely thanks to ray tracing being efficiently parallelizable and cannot be generalized to other algorithms, or the CPU to GPU comparison [Lee 2010].

Our setup for testing the data structures for ray tracing significantly differs from the one of Zlatuška and Havran [2010] in several aspect. We measure our results on the NVIDIA Kepler architecture introduced in 2012, that is two generations newer than the NVIDIA Tesla architecture used in the paper by Zlatuška and Havran [2010] (year 2007/8, GeForce 6 Series and GeForce 200 Series). In particular, hardware caches were introduced in the newer hardware

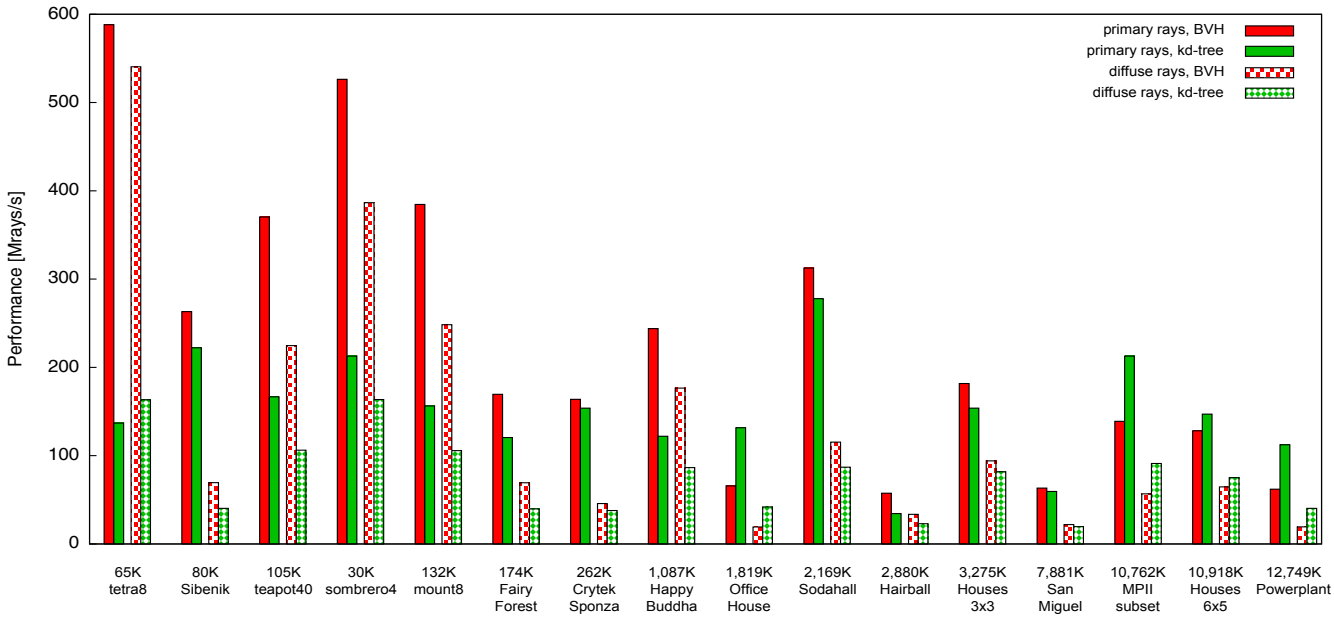

Figure 4: Performance of BVHs and kd-trees on all of our test scenes for both primary (solid color) and diffuse rays (color pattern). The scenes are sorted in ascending order with the number of triangles increasing to the right. For very large scenes the kd-tree performs faster than the BVH.

making stack based ray traversal methods more efficient. We use the single ray traversal algorithm [Aila and Laine 2009] for BVHs while the study [Zlatuška and Havran 2010] used a packet traversal one [Günther et al. 2007]. The kd-tree ray traversal algorithms differ as well: modified algorithm of Aila and Laine [2009] versus the algorithm of Horn et al. [2007] based on short stack with infrequent restarts of ray traversal from the root node. In particular, the packet ray traversal algorithm used in the the older study caused the BVH to be faster only on coherent rays while it was significantly slower for the incoherent ones. The algorithms compared in our study are much more similar, showing the fundamental differences between the two acceleration data structures. Moreover, only scenes up to 1M primitives (Happy Buddha) were used in the original paper, while we compare scenes up to 12M primitives (Powerplant) where the kd-tree is more beneficial than the BVH as shown in Table 3. Last but not least, the older study used data structures built up on a CPU and then ray tracing of these data structures on a GPU. However, we are able to build up both data structures on a GPU directly, using relatively similar algorithm.

5 Limitations

First, our study is limited to scenes composed of triangles only. Second, we have used and studied stack-based algorithms that are reported to be faster on the GPU than the stackless ones. Third, we have decided not to include the idea of BVH spatial splits (SBVH) [Stich et al. 2009; Popov et al. 2009] in our comparison as this method represents a hybrid concept between the BVH and the kd-tree. The ratio of the number of splits using spatial subdivision to the number of splits using the object subdivision and the impact of spatial splits on the ray tracing performance are scene dependent, which makes the analysis even more difficult. Moreover, the parallel build algorithm of SBVHs has not been proposed as it likely faces more challenges than the construction of kd-trees, especially with increased size of the data structure and temporary storage which would likely not fit into the main memory on current GPUs for large scenes. Note that the parallel build algorithm of Karras and Aila [Karras and Aila 2013] implements an algorithm in the spirit of [Ernst and Greiner 2007] and not of the SBVH as described in [Stich et al. 2009; Popov et al. 2009]. The comparison will be the subject of future work when SBVH build algorithm is implemented on a GPU.

6 Discussion

We have implemented the data structures and the ray traversal algorithms to utilize the features of the GPU architecture, which has a different data processing workflow compared to the CPU (cache-based versus stream-based model). We will discuss these issues below and document them by the numbers from measurements.

We have analyzed our ray traversal kernels on primary rays using the NVIDIA Nsight version 2.2 [NVidia 2012]. Below we use for the discussion the Happy Buddha scene, for which the number of intersection tests and traversal steps is similar for both the BVH and the kd-tree, so that the other measured characteristics can be meaningfully compared. The number of performed floating point operations, is clearly in favor of the kd-tree (235 million operations for BVH versus 81 million operations for kd-tree). This is expected since the cost of a single traversal step is higher for the BVH and the ray traversal algorithm is carried out speculatively. However, GPUs have high performance in floating point operations and the difference in performed operations does not form a bottleneck. Further, the percentage of divergent branching (percentage of branches where different threads of a warp took different paths, and the warp's execution must be serialized, to the total number of branches) is worse for the kd-tree which does not use the speculative traversal (14.3% for BVH with speculative ray traversal algorithm versus 27.1% for kd-tree). We have also profiled a modified BVH traversal kernel without speculative traversal and verified that

it is slower than the speculative version and its percentage of divergent branching is 18.3%. While Aila and Laine [2009] report that the traversal kernel is compute limited for the BVH, no such study has been conducted for kd-trees. Our analysis below shows that the kd-tree ray traversal algorithm is memory limited (bandwidth and latency), explaining the results of Table 3.

There are also significant differences between the memory access patterns of both data structures. For the BVH the two children bounding boxes are fetched with coherent memory access, while for the kd-tree several accesses are needed to fetch a similar amount of geometrical information. Reading several nodes for a kd-tree instead of just one for the BVH, that represent roughly the same geometric information, increases the amount of transferred data from memory (390 MBytes for BVH versus 874 MBytes for kd-tree) because of the unused data in each memory fetch. Although this can be mitigated by an improved memory layout, the two children 64 Byte node of the BVH proposed in [Aila and Laine 2009] would be likely difficult to conquer. Further, the latency for accessing the same amount of geometry is higher for kd-trees as well due to the dependent memory fetches to GPU DRAM. The memory latency has been recently identified as a bottleneck of the ray traversal performance for the BVHs as well [Guthe 2014], but in our opinion it has even higher impact on the results for the kd-tree.

The traversal kernel for kd-trees requires to save two items on the traversal stack (node address and traversed distance along a ray, in total 8 Bytes), while the BVH ray traversal algorithm saves only the node address (4 Bytes). Moreover, when different threads in a warp write into different stack indices the memory writes are not coalesced and the data transfer further increases. For the scene Happy Buddha this accounts for an increased size of data stored to local memory (43 MBytes for BVH versus 180 MBytes for kd-tree). Given the traversal stack is located in slow local memory (physically in GPU DRAM) this increases the memory traffic of the already memory limited code and hence it is not suited for the limited size of local cache of contemporary GPU architecture. The amount of data read from local memory is similar for both data structures (72 MBytes for BVH versus 84 MBytes for kd-tree) but the L1 cache hit ratio of the read accesses is very different (86% for BVH versus 48% for kd-tree). The amount of utilized read data is probably similar because early ray termination of the kd-tree traversal algorithm may leave some stack items unread. The different cache hit ratio is possibly caused by different threads in the same warp reading different stack indices from global memory, which causes incoherent accesses to GPU DRAM. We can only speculate that faster and larger local memory and/or cache needed by stack based traversal algorithms could be beneficial for performance in upcoming GPU architectures for both BVHs and kd-trees.

We also performed profiling on the Powerplant scene where the kd-tree performs significantly less intersection tests and traversal steps than the BVH. The profiled statistics mostly keep the same relations as for the Happy Buddha scene with one notable difference. The amount of data transferred from GPU memory to on-chip memory (cores) is much smaller for the kd-tree due to much fewer nodes and triangles being accessed during the ray traversal algorithm (3650 MBytes for BVH versus 1860 MBytes for kd-tree). A lower percentage of divergent branches and a lower data traffic for the traversal stack cannot outweigh the significant difference in the amount of requested memory, explaining the lower performance of the BVH.

7 Conclusion and Future Work

In this paper we have compared the performance of kd-trees and BVHs on today's GPUs by NVIDIA with Kepler core (year 2012/13). The number of traversal steps and intersection tests for ray tracing is lower for kd-trees than for BVHs built for the same scenes, while the ray tracing performance is typically higher for BVHs than for kd-trees. In particular, the BVH is faster on small to medium sized scenes. This is really important for rendering on devices with low compute power, such as mobile devices, that are capable of rendering only small scenes. On the other hand, for larger scenes with high-occlusion kd-trees outperform BVHs as the traversal overhead of BVHs is significant for spatial regions that overlap.

Our measurements and conclusions hold for a simple memory layout model of the trees representing BVHs and kd-trees, i.e., each node is allocated in the memory irrespective of the memory addresses of other nodes and some simple memory allocator is used. The future work could address the comparison of both data structures with memory layout optimized for decreased memory traffic of contemporary GPUs, where nodes with high spatial locality are organized into memory chunks such as in [Szécsi 2003]. Also SBVH [Stich et al. 2009; Popov et al. 2009] with the building algorithm running on a GPU should be added to the comparison.

Acknowledgements

We would like to thank Marko Dabrovic for Sibenik model, DAZ3D (www.daz3d.com) for Fairy Forest, Frank Meinl at Crytek for the Crytek Sponza model, Carlo H. Séquin for Sodahall model, Samuli Laine and Tero Karras for Hairball model, Guillermo Llaguno for the San Miguel model, the UNC for the Powerplant model, and Stanford repository for Happy Buddha. MPII model was supported by the project of Havran et al. [2009].

We would also like to thank Tero Karras, Timo Aila, and Samuli Laine for releasing their GPU ray tracing framework. This research was supported by the Czech Science Foundation under research programs P202/12/2413 (Opalis) and the Grant Agency of the Czech Technical University in Prague, grant No. SGS13/214/OHK3/3T/13.

References

AILA, T., AND LAINE, S. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of HPG 2009*, ACM SIGGRAPH/Eurographics, New Orleans, Louisiana, 145–149.

AILA, T., KARRAS, T., AND LAINE, S. 2013. On Quality Metrics of Bounding Volume Hierarchies. In *Proceedings of HPG 2013*, ACM SIGGRAPH/Eurographics, New York, NY, USA, 101–107.

AKENINE-MÖLLER, T. 2005. Fast 3D Triangle-box Overlap Testing. In *ACM SIGGRAPH 2005 Courses*, ACM, New York, NY, USA, SIGGRAPH '05.

BITTNER, J., HAPALA, M., AND HAVRAN, V. 2013. Fast Insertion-Based Optimization of Bounding Volume Hierarchies. *Computer Graphics Forum 32*, 1, 85–100.

CHOI, B., KOMURAVELLI, R., LU, V., SUNG, H., BOCCHINO, R. L., ADVE, S. V., AND HART, J. C. 2010. Parallel SAH k-D

Tree Construction. In *Proceedings of HPG 2010*, ACM SIGGRAPH/Eurographics, Aire-la-Ville, Switzerland, 77–86.

ERNST, M., AND GREINER, G. 2007. Early Split Clipping for Bounding Volume Hierarchies. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, IEEE Computer Society, Washington, DC, USA, 73–78.

GOLDSMITH, J., AND SALMON, J. 1987. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications 7*, 5 (May), 14–20.

GU, Y., HE, Y., FATAHALIAN, K., AND BLELLOCH, G. 2013. Efficient BVH Construction via Approximate Agglomerative Clustering. In *Proceedings of HPG 2013*, ACM SIGGRAPH/Eurographics, New York, NY, USA, 81–88.

GÜNTHER, J., POPOV, S., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, IEEE Computer Society, Washington, DC, USA, 113–118.

GUTHE, M. 2014. Latency Considerations of Depth-first GPU Ray Tracing. In *Eurographics (Short Papers)*, Eurographics Association, Strasbourg, France, E. Galin and M. Wand, Eds., 53–56.

HAINES, E. A. 1987. A Proposal for Standard Graphics Environments. *IEEE Computer Graphics and Applications 7*, 11 (Nov), 3–5.

HAPALA, M., AND HAVRAN, V. 2011. Review: Kd-tree Traversal Algorithms for Ray Tracing. *Computer Graphics Forum 30*, 1, 199–213.

HAVRAN, V., AND BITTNER, J. 2002. On Improving KD-Trees for Ray Shooting. *Journal of WSCG 10*, 1 (Feb), 209–216.

HAVRAN, V., ZAJAC, J., DRAHOKOUPIL, J., AND SEIDEL, H.-P. 2009. MPII Building Model as Data for Your Research. Res.rep. MPI-I-2009-4-004, MPI Informatik, Dec.

HAVRAN, V. 2000. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive K-d Tree GPU Raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '07, 167–174.

HUNT, W., MARK, W., AND STOLL, G. 2006. Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2006*, IEEE Computer Society, Washington, DC, USA, 81–88.

HURLEY, J., KAPUSTIN, A., RESHETOV, A., AND SOUPIKOV, A. 2002. Fast Ray Tracing for Modern General Purpose CPU. In *Proceedings of Graphicon*, 8.

KAPLAN, M. 1985. Space-Tracing: A Constant Time Ray-Tracer. In *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes*, Addison Wesley, 149–158.

KARRAS, T., AND AILA, T. 2013. Fast Parallel Construction of High-Quality Bounding Volume Hierarchies. In *Proceedings of HPG 2013*, ACM SIGGRAPH/Eurographics, New York, NY, USA, 89–99.

KOPTA, D., IZE, T., SPJUT, J., BRUNVAND, E., DAVIS, A., AND KENSLER, A. 2012. Fast, Effective BVH Updates for Animated Scenes. In *Proceedings of the I3D conference*, ACM, New York, NY, USA, 197–204.

LEE, V. E. A. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture*, ACM, New York, NY, USA, ISCA '10, 451–460.

MASSO, J. P. M., AND LOPEZ, P. G. 2003. Automatic Hybrid Hierarchy Creation: a Cost-model Based Approach. *Computer Graphics Forum 22*, 1, 5–13.

NVIDIA, 2012. NVIDIA Nsight ver 2.2, NVIDIA developer zone, `https://developer.nvidia.com/`.

POPOV, S., GEORGIEV, I., DIMOV, R., AND SLUSALLEK, P. 2009. Object Partitioning Considered Harmful: Space Subdivision for BVHs. In *Proceedings of HPG 2009*, ACM SIGGRAPH/Eurographics, New York, NY, USA, 15–22.

ROCCIA, J.-P., PAULIN, M., AND COUSTET, C. 2012. Hybrid CPU/GPU KD-Tree Construction for Versatile Ray Tracing. In *Eurographics (Short Papers)*, Eurographics Association, C. Andújar and E. Puppo, Eds., 13–16.

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial Splits in Bounding Volume Hierarchies. In *Proceedings of HPG 2009*, ACM SIGGRAPH/Eurographics, New York, NY, USA, 7–13.

SZÉCSI, L. 2003. *An Effective Implementation of the k-D Tree, book Graphics Programming Methods*. Charles River Media, Inc., Rockland, MA, USA, 315–326.

VINKLER, M., BITTNER, J., HAVRAN, V., AND HAPALA, M. 2013. Massively Parallel Hierarchical Scene Processing with Applications in Rendering. *Computer Graphics Forum 32*, 8, 13–25.

WALD, I., AND HAVRAN, V. 2006. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2006*, IEEE Computer Society, Washington, DC, USA, 61–69.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3, 153–164.

WALD, I., MARK, W. R., GÜNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. 2009. State of the Art in Ray Tracing Animated Scenes. *Computer Graphics Forum 28*, 6, 1691–1722.

WALD, I. 2007. On fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, IEEE Computer Society, Washington, DC, USA, 33–40.

WEGHORST, H., HOOPER, G., AND GREENBERG, D. P. 1984. Improved Computational Methods for Ray Tracing. *ACM Transactions on Graphics 3*, 1 (Jan.), 52–69.

WU, Z., ZHAO, F., AND LIU, X. 2011. SAH KD-tree Construction on GPU. In *Proceedings of HPG 2011*, ACM SIGGRAPH/Eurographics, New York, NY, USA, 71–78.

ZLATUŠKA, M., AND HAVRAN, V. 2010. Ray Tracing on a GPU with CUDA – Comparative Study of Three Algorithms. In *Proceedings of WSCG'2010, communication papers*, 69–76.