Pacific Graphics (2014)
J. Keyser, Y. J. Kim, and P. Wonka (Editors)

Quantitative Analysis of Voxel Raytracing Acceleration Structures

M. G. Chajdas[1] and R. Westermann[1]

[1]Technische Universität München, Germany

Abstract

In this work, we provide a comprehensive analysis of GPU acceleration structures for voxel raytracing. We compare the commonly used octrees to BVH and KD trees, which are the standard in GPU triangle raytracing. Evaluating and analyzing of the behavior is complicated, as modern GPUs provide wide vector units with complex cache hierarchies. Even with sophisticated SIMD simulators, it is increasingly hard to model the hardware with sufficient detail to explain the observed performance.
Therefore, instead of relying on SIMD simulation, we use hardware counters to directly measure key metrics like execution coherency on a modern GPU. We provide an extensive analysis comparing different acceleration structures for different raytracing scenarios like primary, diffuse and ambient occlusion rays. For different scenes we show that data structures commonly known for good performance, like KD-trees, are actually not suited for very wide SIMD units. In our work we show that BVH trees are most suitable for GPU raytracing and explain how the acceleration structure affects the execution coherency and ultimately performance, providing crucial information for the future design of GPU acceleration structures.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques —Graphics data structures and data types

1. Introduction

In recent years, voxel raytracing has become increasingly popular. Most of the time, octrees are used due to their simple traversal and easy integration with level-of-detail. In triangle raytracing, BVH trees and KD trees are usually used due to their superior spatial partitioning properties. One aspect which has not been well researched yet is how data structures affect traversal coherency on architectures with very wide vector units as GPUs. So far, coherency has been only simulated or estimated, but not measured on a modern GPU architecture. Recent hardware provides hardware counters, similar to CPUs, which allow for precise measurement of vector unit execution coherency, cache hit rates and bandwidth usage.

In this work, for the first time, we use the hardware counters of a modern GPU to provide comprehensive and accurate measurements of the behavior of different acceleration structures. In particular, we measure and analyze the differences between BVH trees, KD trees and octrees in execution performance of the involved operations as measured by the hardware counters. We do so using multiple large-scale voxel models with different ray-traversal characteristics. In our analysis, we show how coherency, performance and data structure design is related. With the provided information, we hope to further pave the way towards coherency-optimized GPU data structures.

Besides its current popularity, one reason for analyzing voxel raytracing is that the used acceleration structures, unlike those typically used for triangle raytracing, do not have overlaps at the leaf level. In triangle scenes, leaf nodes generally overlap, and handling these overlaps contributes significantly to the raytracing performance.

In voxel raytracing however, most of the execution time is spent in tree traversal. This makes it the ideal vehicle to analyze the acceleration data structure itself.

2. Previous work

Voxel raytracing has been performed on GPUs since many years now [CNLE09,LK11]. So far, all GPU voxel raytracers

DOI: 10.2312/pgs.20141257

use an octree as the underlying acceleration structure due to the simple integration of level-of-detail.

For triangle raytracing, KD trees and BVH trees have been the gold standard for many years. This has carried over to GPU triangle raytracing as well [WMS06, WIK*06, PGSS07, HSHH07, GPSS07]. An early work which focused on a detailed analysis of GPU raytracing is [AL09], in which a SIMD simulator was used to identify the best traversal pattern for a single data structure, which was subsequently validated on actual hardware.

However, even on CPUs, wide vector execution poses problems for efficient raytracing. Packet traversal [BEL*07] was found to be beneficial for primary rays. For incoherent rays, single-ray traversal [WWB*14] may allow for better usage of wide vector units. These coherency problems become more pronounced on 32 & 64-wide vector units as used by current GPUs. So for, a lof of research has focuseod on the traversal, i.e. by re-scheduling of rays as proposed in [AK10, BAM14], but not by investigating how the acceleration structure affects coherency.

Data structures have only come into attention very recently, in particular, [AKL13] has investigated in detail why BVH trees don't perform as good as expected on GPUs. They introduce two important concepts, the end-point overlap (EPO) and the leaf-count variability (LCV). The EPO describes how many nodes any given point in the scene overlaps. The LCV describes the standard deviation of the number of leaf nodes intersected by a ray. The higher the variance, the more likely it is that the intersection kernel will become more incoherent as some rays will intersect the leaf while the rest will be traversing the tree. In this work, only one data structure is used (a BVH), but many different builders are compared.

One important difference between this work and ours is that we investigate voxel raytracing, which has an EPO of 0 for all data structures. This means that unlike a normal triangle ray-tracer, which has to spend significant amount of times to resolve overlapping leafs, our raytracing kernels spent nearly all of their time in the tree traversal. The impact of leaf intersections on the measurement is significantly reduced, allowing us to focus on the acceleration structure characteristics instead.

3. Implementation & testing methods

For our analysis we implemented an automated test framework. For each data set, we build all acceleration structures and render a fixed set of test cases. In this section, we will describe our test methodology in detail, including our tree building and traversal routines.

The test scenes (see also Figure 1) are constructed by voxelizing triangle input using a standard voxelization algorithm [BLD13]. We use a six-separating voxelization to

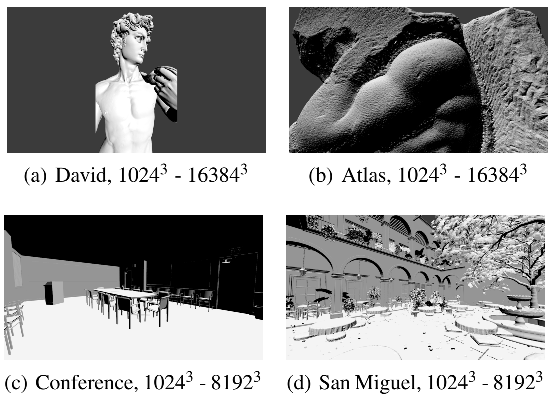
(a) David, 1024^3 - 16384^3 (b) Atlas, 1024^3 - 16384^3 (c) Conference, 1024^3 - 8192^3 (d) San Miguel, 1024^3 - 8192^3

Figure 1: *We have used 4 test scenes, voxelized at different resolutions. Data set sizes vary from 12 MiB (David, 1024^3) up to 6.6 GiB (Atlas, 16384^3)*

minimize the number of generated voxels. To simulate level-of-detail, we have voxelized each test scene at different resolutions. For our target rendering resolution (1280×720), a voxel resolution between 2048^3 and 4096^3 results in the correct level-of-detail, while 1024^3 is undersampled and 8192^3 is oversampled.

We have limited the scene size to 65535^3 voxels, which allows us to use 16-bit bounding boxes for the BVH and 16-bit plane offsets for the KD tree. Voxel positions are also stored as 16-bit integers.

The data structures are generated in an offline process on the CPU and linearized before uploading to the GPU, that is, all pointers are replaced by explicit 32-bit integer offsets. Additionally, the data structure is split into two leaf and interior node buffers, which improves locality during traversal.

We have implemented all kernels using OpenCL [M*09] and integrated them into a common, OpenCL based raytracing framework. We performed all our testing on an AMD FirePro W9100 graphics card with 44 compute units. Each compute unit consists of two parts: A scalar unit and a vector unit. The scalar unit is used to perform branching, while all compute operations are performed on the 64-wide vector unit [AMD12]. For the actual measurement, we use the AMD `GPUPerfAPI` which provides access to the hardware counters. These allow us to measure vector unit utilization, cache hit rates as well as memory usage.

3.1. Test scenarios

Our test framework allows for four different rendering modes (see also Figure 2): Primary rays, shadow rays, soft-shadow rays and ambient occlusion.

For secondary effects, we use a two-pass rendering to enable precise measurements. First, the scene is traced and the hit-points are recorded. Second, a new kernel is started

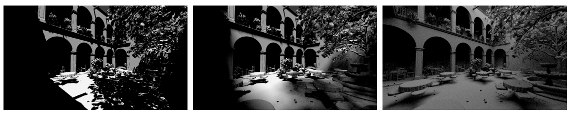

Figure 2: *Besides primary rays, we used shadow, soft-shadow and ambient occlusion rays to cover a wide range of ray types. For soft-shadows and ambient occlusion rays, 36 rays per pixel where used. The ambient occlusion ray length is $1/3$ of the scene size.*

which reads the hit-points and traces the shadow, soft-shadow and ambient occlusion rays.

The hardware counters and execution time is only measured for the second pass. Soft-shadows and ambient-occlusion are computed similarly, by constructing a basis at the hit point and tracing rays in a hemisphere around the light direction or the surface normal. For soft-shadows and ambient occlusion, random directions are required. For ambient-occlusion, we scale the ray length with the scene size such that the ray-length is $1/3$ of the total scene size. We used 36 rays for both soft-shadows and ambient occlusion raytracing.

Every option is turned on/off using preprocessor definitions. Loop counters like the number of soft shadow or ambient occlusion rays are also provided as compile-time constants to allow for loop unrolling. We have also inspected the generated ISA to ensure that no surprising inlining has been performed for one of the traversal algorithms.

All traversal routines are based on a short-stack with restart [HSHH07] if the stack is empty. With a short-stack, we can provide a fair comparison between all data structures and also make the measurements more relevant for practical use, as BVH trees have to be traced with auxiliary storage. In our testing, we have found a short-stack with 4-8 entries to be most beneficial.

Our traversal algorithms differ from normal triangle raytracing in an important way: Leaf nodes and voxels never overlap. This implies that the traversal can stop once a hit has been found as long as the tree is traversed strictly in front-to-back order. Compared to a triangle raytracer, this reduces the time spent in ray/leaf traversal drastically. Due to this property, our work primarily measures the traversal performance of the acceleration structure and is nearly independent of the actual ray/primitive intersection speed.

The voxels themselves are stored with a per-voxel position for all of our trees. We have also used the same leaf-size (8) for all builders.

3.2. BVH

We use a standard BVH tree builder which subdivides at the midpoint along the longest axis. Compared to an SAH based builder, the middle split builder creates a more balanced tree while providing the same raytracing performance. The BVH tree is built top-down by identifying the longest axis first, and then splitting in the middle. Leaf nodes are created once less 8 or less voxels remain.

During the linearization, the tree is packed by moving the bounding boxes up to their direct parents. Every interior node consists of the two bounding boxes of its children and the split axis. This enables an efficient traversal and allows us to store the leaf nodes without bounding boxes.

The traversal routine starts by computing which of the two children of an interior node is hit. If both are hit, the traversal continues into the closer one and the further away node is pushed onto a short traversal stack. Once a leaf node has been reached, the traversal stack is popped; if the stack is empty, a full restart is performed. Unlike for triangle raytracing, where overlapping nodes have to be resolved using auxiliary storage, we can perform a restart once the stack becomes empty.

3.3. KD

The KD tree is built using an SAH builder. This turned out to be crucial to get a high-quality KD tree which renders efficiently. For example, a middle-split KD tree results in roughly 10x slower raytracing performance. We use a binned SAH builder with 32 bins [WH06]. For each bin, the cost is estimated as the size of the sub-volume times the number of voxels. As we treat each voxel as a small cube, the surface area is directly proportional to the number of voxels. Leaf nodes are created once less 8 or less voxels remain. We use a standard KD tree which stores the split position and axis at each interior node.

The traversal routine computes the intersection between the ray and the current node's splitting plane. If an intersection has been found, the traversal continues into the closer node and the further away node, together with the current ray maximum, is pushed onto the stack. Just like the BVH and octree traversal, once a leaf node is reached the stack is popped; once the stack is empty, a restart is performed.

3.4. Octree

The octree is built by sorting the voxels along their morton-order index and recursive subdivision from top. We use a standard 2^3 octree. Leaf nodes may be created at high levels of the tree of the region is sparse; as we store per-voxel positions, there is no need to build the tree down to the leaf level. The octree does not encode any positions, as these are stored implicitly in the tree.

The traversal computes which interior planes are intersected by the current ray and traverses into the closest one. In this case, the current node and the current ray minimum/maximum is pushed onto the stack. Pushing the current node instead of computing all intersected children and pushing them allows for better usage of the short stack and simplifies the traversal routine. Similar to the BVH tree, the traversal stack is popped once a leaf node is reached; if the stack is empty, a restart is performed. This traversal is similar to the one used in [LK11], which is also used by the most recent octree rendering work [KSA13]. Unlike [KSA13], our octree builder does not merge identical paths through the tree. However, this only improves locality and thus memory efficiency but not traversal efficiency, as the execution effiency is not affected by merged paths.

The octree does not use level-of-detail during traversal. While simple level-of-detail is a major feature of octrees, we didn't integrate it into our traversal to allow for a fair comparison of the data structures. Instead, as mentioned above, we have voxelized the scene at different levels-of-detail.

4. Results & Analysis

We have tested four different scenes: `Atlas`, `David`, `Conference` and `San Miguel`. Atlas is a 3D scan with a lot of surface detail as well as noise. Unlike the other scenes, Atlas exhibits many tiny holes in the surface, which complicates traversal. David is a very clean 3D scan without any holes, allowing nearly all rays to traverse once to the leaf level and terminate.

San Miguel and Conference are in-door scenes. In both cases, the camera is placed within the boundaries of the data set. Unlike for Atlas and David, where traversal starts from the outside, the rays have to traverse down to a very low tree level and continue traversing near leaf nodes, putting increased pressure on restart and stack performance.

As we can see in Table 4, the BVH has on average the highest coherency across all test scenes and is also the fastest traversal algorithm 4. The coherency of the BVH tree is even high for very large trees, unlike the KD and octree. It also degrades less for incoherent rays, maintaining at least 30% efficiency even for the largest data sets, or, on the 64-wide vector unit, a usage of approximately 19 active lanes.

We can also see that primary ray coherency is up to twice as high as for secondary rays. This result is similar to [AL09], however, in their work, a 32-wide architecture was used.

Besides the coherency, we also measured the actual execution time to understand how the coherency translated into ray-tracing performance. As we can see in Table 4, the coherency is closely related to the performance, indicating that all of our traversal algorithms are limited by execution speed and not by memory bandwidth. For very low coherency values, the execution time increases non-linearly – we assume that we can see the effects of few, very long running rays in this case.

Coherency is also related to the tree depth. Very deep trees are likely to exhibit lower coherency, as long execution chains can be created. We have measured the average depth at which leafs nodes are present. Compared to the BVH, the KD tree is between 38% to 92% deeper. The BVH is 120% to 170% deeper than the octree. Notice that the first six levels of the KD tree, corresponding to 13%-20% of the total tree depth, are the global bounds of the scene and thus highly coherent. We cannot conclude that a higher tree depth has significant impact on the coherency, as the KD tree exhibits similar coherency to the octree, despite much deeper trees. For instance, in the `Conference` scene, the KD tree (depth 35, variance 7.4) reaches 40% coherency for primary rays, compared to 46% for the octree (depth 12, variance 0.02). For comparison, the BVH achieves 66% coherency (tree depth 26, variance 3.4.)

We have also measured the cache usage to identify the impact of large nodes in the acceleration structure. The BVH tree has the biggest interior nodes, containing 33 byte of data (padded to 36 byte); the KD interior nodes are 11 bytes (padded to 12) and the octree nodes are 32 bytes (no additional padding.) As expected, the small node sizes lead to a higher cache efficiency for the KD tree, yet it does not translate into better performance. Due to its small nodes, the KD tree exhibits the best cache behavior. In this context, we also measured whether the memory subsystem was a bottleneck for traversal, that is, whether the execution stalled due to memory accesses. For all tested scenes and data structure, the execution was never blocked on memory accesses; indicating that the traversal performance was limited by the computations and not the memory.

4.1. Analysis

Our analysis shows that the data structure has a significant impact on the traversal coherency. Interestingly, the data structure with the most complex spatial partitioning performs best from both a coherency and a performance point of view. This can be explained by investigating how precise the spatial partitioning is. In general, a single step in the BVH traversal performs 12 plane intersections (6 for each child); a single step in the octree performs 3 plane intersections and the KD tree only plane intersection.

We can get a better intuition for this problem by considering how the selectivity of an acceleration structure affects execution. If the selectivity is very low, the higher the chance that the execution graph will split up. We can see this for the KD tree. If the selectivity is very high, it is unlikely that the execution will split up, and many paths will terminate quickly, as can be seen for the BVH. The octree is in-between – initially, it would seem that its behavior should be equal to a KD tree, but the fact that the ray is shortened by potentially up to three planes per step significantly reduces the chance of an intersection. The key observation is that the coherency does not depend on the node intersection test, as long as the test can be formulated in a branch-free manner, but on the selectivity of the data structure.

Tree		BVH				KD				Octree			
Scene	Res	Prm	Shd	Soft	AO	Prm	Shd	Soft	AO	Prm	Shd	Soft	AO
Atlas	1024	59.7	52.6	47.4	33.5	40.7	30.9	28.0	16.2	52.4	40.4	37.5	22.9
	2048	50.5	42.9	40.5	32.2	32.2	22.5	21.3	14.5	44.5	31.3	29.9	20.5
	4096	44.3	37.0	36.0	31.5	27.1	17.4	17.0	13.4	39.4	24.7	24.2	19.0
	8192	41.0	33.5	33.0	30.9	24.5	14.4	14.2	12.5	35.8	20.4	20.2	17.8
	16384	39.2	31.3	30.9	30.4	23.1	12.4	12.4	11.9	33.4	17.5	17.5	16.8
David	1024	57.9	58.5	58.2	35.6	41.4	39.2	39.9	18.4	53.3	49.2	49.5	25.2
	2048	50.0	50.3	50.6	34.6	33.5	29.4	30.2	16.5	46.8	39.7	40.4	23.0
	4096	45.0	45.1	45.6	33.8	28.7	23.2	23.9	15.0	41.9	32.6	33.4	21.1
	8192	42.1	42.2	42.7	33.2	26.0	19.5	20.1	13.9	38.4	27.7	28.4	19.6
	16384	40.3	40.5	40.9	32.7	24.5	17.1	17.6	13.1	35.9	24.2	24.9	18.5
Conference	1024	70.0	56.9	55.5	35.8	43.7	39.3	40.3	19.1	48.8	44.0	43.6	21.8
	2048	67.8	50.3	49.9	35.9	43.6	34.3	35.4	17.9	47.3	38.6	38.4	20.9
	4096	66.6	49.9	49.6	36.2	41.9	31.1	32.1	17.0	46.7	36.0	35.5	20.7
	8192	66.2	49.6	49.2	36.4	40.5	28.8	29.6	16.3	46.0	33.7	33.0	20.3
San Miguel	1024	65.2	52.7	43.8	27.2	46.1	34.0	28.1	14.1	51.3	40.4	34.9	18.7
	2048	56.5	43.7	37.4	26.3	36.0	24.6	21.0	12.1	42.4	31.9	27.9	16.9
	4096	50.2	37.4	33.9	26.1	29.4	18.7	16.8	10.7	36.7	26.0	23.4	15.6
	8192	46.8	34.9	32.7	26.5	26.4	15.6	14.5	10.0	33.3	22.0	20.5	14.8
Average		53.3	45.0	43.2	32.2	33.9	25.1	24.6	14.6	43.0	32.2	31.3	19.7

Table 1: *Measured coherency accross various test scenes and ray types.* `Prm` *are primary rays,* `Shd` *shadow rays,* `Soft` *are soft-shadows and* `AO` *are ambient occlusion rays. For soft-shadows and ambient occlusion,* $36\times$ *as many rays have been casted as for the primary and shadow rays tests.*

Tree		BVH				KD				Octree			
Scene	Res	Prm	Shd	Soft	AO	Prm	Shd	Soft	AO	Prm	Shd	Soft	AO
Atlas	1024	3.0	4.1	168.9	190.8	4.4	6.1	219.8	271.3	3.7	4.3	201.9	216.3
	2048	4.0	5.9	233.1	244.7	5.6	10.1	336.3	378.1	5.0	6.6	300.4	291.0
	4096	5.3	7.9	315.7	295.6	7.4	14.7	501.3	490.6	6.2	9.7	434.3	369.6
	8192	6.5	9.9	399.2	348.3	9.0	20.8	712.0	620.9	7.8	13.6	601.4	457.0
	16384	7.5	12.0	480.7	447.8	10.4	28.0	949.9	873.6	9.0	18.2	795.2	551.8
David	1024	2.3	1.1	43.7	108.6	3.1	1.7	55.0	144.8	3.4	1.4	61.4	132.0
	2048	3.2	1.7	63.4	140.0	4.3	2.9	91.4	200.7	4.4	2.0	89.3	172.2
	4096	4.0	2.3	88.1	172.0	5.6	4.5	143.1	268.6	5.4	3.0	128.1	218.7
	8192	4.9	3.1	115.6	203.5	7.2	6.5	205.5	342.7	6.7	4.2	177.2	271.8
	16384	5.7	3.9	142.3	236.7	8.2	8.8	282.8	427.0	7.8	5.8	236.0	328.4
Conference	1024	3.7	2.3	96.0	331.2	2.8	2.0	69.3	274.6	5.5	2.9	138.0	397.2
	2048	4.3	2.9	125.0	364.1	3.4	2.6	90.8	322.5	6.2	4.0	185.6	489.5
	4096	4.9	3.4	143.1	393.1	3.9	3.1	110.0	366.5	6.5	4.8	227.1	545.9
	8192	5.5	3.8	158.3	423.3	4.4	3.9	132.9	411.4	6.9	6.0	274.5	602.0
San Miguel	1024	4.5	4.5	209.5	516.5	5.0	5.0	199.5	499.0	4.7	3.8	199.1	370.8
	2048	6.1	6.4	292.6	638.6	7.1	8.3	321.9	714.4	6.7	6.1	314.1	532.4
	4096	7.8	8.5	378.0	736.1	9.1	12.5	469.1	929.6	9.1	8.9	448.4	714.3
	8192	9.3	10.9	443.1	866.9	11.2	17.0	621.6	1129.8	11.4	12.5	611.6	900.0
Average		5.1	5.3	216.5	369.9	6.2	8.8	306.2	481.5	6.5	6.5	301.3	420.0

Table 2: *Measured execution time in milliseconds. The captions are the same as in Table 4.*

Another important observation is that even the most incoherent acceleration structure never dips below 12% coherency in the worst case – that is, 8 units of the 64-wide vector unit can be always filled. This strongly suggests that for CPUs, which only support 4 & 8-wide SIMD, efficient re-packing of rays would allow to reach near perfect utilization of this comparatively narrow vector units.

5. Conclusion & future work

We have provided an extensive, quantitative measurement of several popular acceleration structures in different render scenarios. Our focus has been on voxel raytracing, which is more dependent on the actual acceleration structure than triangle raytracing, where a significant amount of time is spent in leaf nodes.

Under these conditions, we observed that BVH trees, which are typically not used for voxel raytracing, provide a compelling alternative to octrees. We have also identified coherency as the core reason behind the performance differences. In the future, we hope to be able to optimize the BVH and KD tree builders to incorporate coherency-improving measures instead of solely focusing on SAH costs.

In this work, we used a GPU with a very wide vector unit. As the tests are written in OpenCL, we hope to expand our test framework on other architectures with different vector widths.

Finally, our results show that the incorrect level-of-detail has a significant impact on the resulting ray coherency. So far, only level-of-detail techniques for voxel octree rendering have been presented. We believe that a level-of-detail approach for BVH trees would be highly beneficial.

Acknowledgements

We would like to thank AMD for the FirePro W9100 donation.

References

[AK10] Aila T., Karras T.: Architecture considerations for tracing incoherent rays. In *Proc. High-Performance Graphics 2010* (2010), pp. 113–122. 2

[AKL13] Aila T., Karras T., Laine S.: On quality metrics of bounding volume hierarchies. In *Proc. High-Performance Graphics* (2013). 2

[AL09] Aila T., Laine S.: Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009* (2009), pp. 145–149. 2, 4

[AMD12] Amd southern islands/sea islands acceleration programming guide. URL: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/10/si_programming_guide_v2.pdf. 2

[BAM14] Barringer R., Akenine-Möller T.: Dynamic ray stream traversal. to appear. 2

[BEL*07] Boulos S., Edwards D., Lacewell J. D., Kniss J., Kautz J., Shirley P., Wald I.: Packet-based whitted and distribution ray tracing. In *Proceedings of Graphics Interface 2007* (New York, NY, USA, 2007), GI '07, ACM, pp. 177–184. URL: http://doi.acm.org/10.1145/1268517.1268547, doi:10.1145/1268517.1268547. 2

[BLD13] Baert J., Lagae A., Dutré P.: Out-of-core construction of sparse voxel octrees. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 27–32. URL: http://doi.acm.org/10.1145/2492045.2492048, doi:10.1145/2492045.2492048. 2

[CNLE09] Crassin C., Neyret F., Lefebvre S., Eisemann E.: Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)* (Boston, MA, Etats-Unis, feb 2009), ACM, ACM Press. URL: http://maverick.inria.fr/Publications/2009/CNLE09. 1

[GPSS07] Günther J., Popov S., Seidel H.-P., Slusallek P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007* (Sept. 2007), pp. 113–118. http://www.mpi-inf.mpg.de/~guenther/BVHonGPU/index.html. 2

[HSHH07] Horn D. R., Sugerman J., Houston M., Hanrahan P.: Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2007), I3D '07, ACM, pp. 167–174. URL: http://doi.acm.org/10.1145/1230100.1230129, doi:10.1145/1230100.1230129. 2, 3

[KSA13] Kämpe V., Sintorn E., Assarsson U.: High resolution sparse voxel dags. *ACM Trans. Graph. 32*, 4 (July 2013), 101:1–101:13. URL: http://doi.acm.org/10.1145/2461912.2462024, doi:10.1145/2461912.2462024. 4

[LK11] Laine S., Karras T.: Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics 17* (2011), 1048–1059. doi:http://doi.ieeecomputersociety.org/10.1109/TVCG.2010.240. 1, 4

[M*09] Munshi A., et al.: The opencl specification. *Khronos OpenCL Working Group 1* (2009), 11–15. 2

[PGSS07] Popov S., Günther J., Seidel H.-P., Slusallek P.: Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (Sept. 2007), 415–424. http://www.mpi-inf.mpg.de/~guenther/StacklessGPURT/index.html. 2

[WH06] Wald I., Havran V.: On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING* (2006), pp. 61–70. 3

[WIK*06] Wald I., Ize T., Kensler A., Knoll A., Parker S. G.: Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics* (2006), 485–493. http://doi.acm.org/10.1145/1141911.1141913. 2

[WMS06] Woop S., Marmitt G., Slusallek P.: B-KD Trees for hardware accelerated ray tracing of dynamic scenes. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (New York, NY, USA, 2006), ACM, pp. 67–77. http://doi.acm.org/10.1145/1283900.1283912. 2

[WWB*14] Wald I., Woop S., Benthin C., Johnson G. S., Ernst M.: Embree: A kernel framework for efficient cpu ray tracing. to appear. 2