A fast SAH-based construction of Octree

Xin Yang*, Duan-qing Xu, Lei Zhao

College of Computer Science, Zhejiang University

Hangzhou,China
*xinyang@zju.edu.cn

ABSTRACT

Recent GPU ray tracers can already achieve performance competitive to that of their CPU counterparts. Nevertheless, these systems can not yet fully exploit the capabilities of modern GPUs and can only handle medium-sized, static scenes.

We present an octree construction algorithm for the GPU that achieves real-time performance by heavily exploiting the hardware, which has been observed to give superior performance in ray tracing compared to other acceleration structures. We use streaming construction with the surface area heuristic (SAH) that significantly increase the coherence of memory accesses during construction of the octree.

Keywords: **Raytracing, Octree, GPU, SAH**.

I. INTRODUCTION

Ray tracing is a technique for rendering pictures from a three-dimensional model by following the paths of simulated light rays through the scene. One of the most serious problems of ray tracing is that it requires a relatively large amount of computation time. While CPU performance has increased dramatically over the last few years, it is still insufficient for many ray tracing applications. Commodity computer graphics chips are probably today's most powerful computational hardware for the dollar. As a result of continued demand for programmability, modern graphics processing units (GPUs) such as the NVIDIA GeForce 8 Series are designed as programmable processors employing a large number of processor cores [1].

Lately, ray tracing running on GPU have developed to an excellent substitute to CPU-based ray tracers [2, 3]. However, even though optimized for the GPU architecture, these implementations can still not utilize the full power of modern GPUs. GÄźnther et al.[4] point out that, two main problems need to be addressed for gaining maximum performance from the GPU, such as the NVIDIA GeForce 8. First, one needs to keep only a small state per thread to allow for enough active threads to run to keep the GPU busy. The ray tracer of Popov et al. required too many live registers which resulted in a poor GPU utilization of below 33% [3]. Secondly, one needs to assure the coherent execution of threads running in parallel, due to the very wide SIMD architecture of current GPUs (32–48 units execute the same instructions [5]). Execution divergence (i.e. incoherent branching) can limit performance of ray tracing to around 40% of the graphics board's theoretical potential [2].

Because of accustomed acceleration structure only with a relatively small number of nodes at the upper levels, which makes parallelizing over nodes inefficient and leaves the massive parallelism of GPUs underexploited. So, we propose a new spatial partitioning that allows construction of an improved octree (subtrees) by threads independently. This approach takes advantage of the parallelism present by heavily exploiting the hardware for the GPU. Specifically, Kun Zhou et al.[6] firstly implement their parallel kd-tree algorithms in BFS (breadth-first search) order to fully exploit the fine-grained parallelism of modern GPUs at all stages of kd-tree construction. Our algorithm also builds octree nodes

 ISBN 978-80-86943-94-7

in this way, which has been observed to give superior performance in ray tracing compared to other acceleration structures [7], when built using the surface area heuristic (SAH) [8].

Section 2 presents previous work in this area, while Section 3 introduces more formal definitions of our octree construction. Section 4 discusses our system implementation, analyzes the performance of our implementation and compare it with current work. Finally, in Section 5, we offer some thoughts on future work.

II. PREVIOUS WORK

In this section we give a brief overview of prior work on acceleration structures for ray tracing dynamic scenes.

2.1 Ray Tracing On Parallel Architectures.

There has been a significant interest in studying ray tracing on parallel architectures. Ray tracing on GPUs has stimulated much interest recently. [9] implemented ray-triangle intersection on the GPU. [10] designed the first ray tracer that runs entirely on the GPU, employing a uniform grid for acceleration. [11] introduced two stackless kd-tree traversal algorithms, which outperform the uniform grid approach.[12] implemented a limited GPU ray tracer for dynamic geometry based on bounding-volume hierarchies and geometry images. None of the above GPU ray tracers outperforms a well-optimized CPU ray tracer. Recently, two techniques [2, 3] achieved better performance than CPU ray tracers. Both techniques use stackless kd-tree traversal and packet tracing. Unfortunately these two techniques work for static scenes only. For dynamic scenes, most existing methods are CPU-based (e.g., [13, 14]). Kun Zhou et al.[6] first implemented their parallel kd-tree algorithms in BFS(breadth-first search) order to fully exploit the fine-grained parallelism of modern GPUs at all stages of kd-tree construction, achieves real-time performance. Shevtsov et al. [15] presented a shared memory architecture with many CPU-like cores, including recent multi-core CPUs. The algorithm first partitions the space into several balanced sub-regions and then builds a sub-tree for each sub-region in parallel and in DFS (depth-first search) order. The algorithm cannot be mapped well to GPU architecture because modern GPUs require 103 ~ 104 threads for optimal performance [5], orders of magnitude greater than the possible thread number on multi-core CPUs (e.g., four threads tested in the paper).

2.2 Acceleration Structures

The relative performance of different acceleration structures has been widely studied. Havran [16] compares a large number of acceleration structures across a variety of scenes and determines that the kd-tree is the best general-purpose acceleration structure for CPU raytracers. It would seem natural, therefore, to try to use a kd-tree to accelerate GPU ray tracing. Construction of high quality KD-tree is bandwidth hungry and computationally expensive task. Attempts to reduce time spent on kd-tree construction were performed using hybrid data structure combining kd-tree with bounding volumes [17]. Similar combining of BVH and spatial partitioning for increasing overall performance was made in [18]. However these approaches still lack parallel implementation and optimized traversal like MLRT [19] and thus demonstrate modest overall performance.

Our algorithm relies on a modified work stealing approach to ensure an high performance width-first octree computation. We speed-up and optimize the construction of (SAH based) octrees. It ensures all processing units progress simultaneity to the bottom of the octree, enabling to stop the algorithm at anytime while ensuring a balanced Octree exploration (it avoids to waste processing power in a deep exploration of an octree branch, while an other process stays several depth level behind). We build our octree with all stages running in parallel with minimal synchronization overhead allowing to exploit as many threads as available to achieve fast octree re-build from scratch every frame. So our method is an excellent strategy for interactive ray tracing of dynamic scenes, which does not require any prior information about vertices motion.

III. CONSTRUCTION OF IMPROVED OCTREE

Using high-quality acceleration structures is essential for achieving interactive ray tracing performance. In this section, we describe how to build our improved Octrees for ray tracing.

3.1 Conventional Octree Construction

Typically an octree [20] is a hierarchical data structure showing how objects are distributed in the object space, which has been mainly used in image processing or solid modeling areas, it was first used for ray tracing by Glassner [21]. Conventional octree

construction divides a three-dimensional space for each axis using the spatial median, obtains eight subspaces, which can be represented by an octree. The root node of an octree represents the entire object space. If the entire space contains more objects than a given limit, the space is divided into eight sub-spaces represented by eight children nodes. A subspace thus created is defined as a voxel. These voxels are further divided into eight voxels, and this process is repeated until the voxels satisfy the given criteria. In general, the criteria used to determine whether or not the given octree should be divided further depend upon the number of objects intersecting with a voxel and the maximum depth of an octree allowed[22][23].

However, the octree contains cell boundaries that are static and their location is independent of the objects. This independence makes the more intersections. The kd-tree, on the other hand, places the boundaries around the objects, especially if the empty space is cut off, thus it can result in much higher intersection probability. Furthermore, conventional octree construction uses uniform voxels to get spatial partition, which leads to higher depth of the tree, and generates much empty nodes which waste much memory.

3.2 Improved Octree Construction

The kd-tree is the best general-purpose acceleration structure, which uses the SAH [8] estimates the ray tracing performance of a given acceleration structure. But modern GPU architecture contains multiple physical multi-processors and requires tens of thousands of threads to make the best use of these processors [5], while accustomed acceleration structure only with a relatively small number of nodes at the upper levels, which makes parallelizing over nodes inefficient and leaves the massive parallelism of GPUs underexploited. Therefore, we propose an improved octree to exploit the hardware to the largest possible degree.

Conventional SAH kd-tree evaluates the SAH costs for all splitting plane candidates, then pick the optimal candidate with the lowest cost and split the node into two child nodes. Unlike this method, we extensively work on all three dimensions at once during SAH evaluation, and build eight sub-nodes at one time. Furthermore, we record the optimal candidate with the lowest cost at X dimension, Y dimension and Z dimension. Then, the three candidates divide a node into eight sub-nodes, as shown in Figure 1.

For each potential partition we need to compute Eq. (1), hence we need to know the primitive counts and the surface areas of children. To compute these counts efficiently, Wald et al. [13, 24] proposed to sort the primitives. However, a much more efficient method was recently published, which avoids sorting and which additionally features memory friendly access patterns [25, 26]. For our octree builder, we adapt the binning method of [25], which was originally proposed for building kd-trees. We iterate over the primitives on all three dimensions at one time to bin

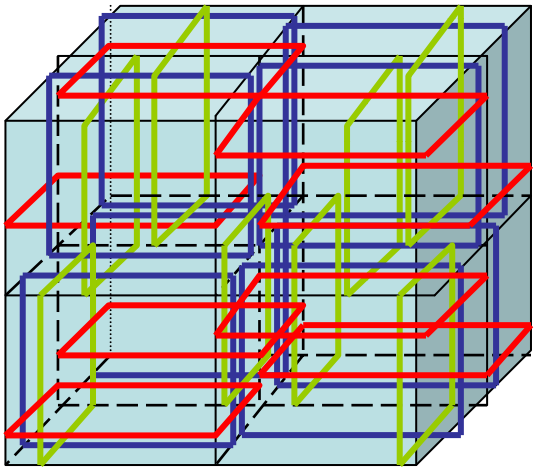

Figure 1: improved octree construction with SAH

them by means of their theory, and by doing so, to accumulate their count and extend in several bins. The gathered information in the bins is then used to reconstruct the primitive counts and the surface areas on both sides of each border between bins, and thus to compute the SAH cost function at each border plane. We can use SIMD operations to compute the SAH cost function on the three dimensions together that exploit the parallel performance. As Popov et al. [25], we minimize memory bandwidth by performing the binning in all three dimensions during the split of the parent node.

$$C_P = K_T + \frac{K_I}{SA(N)}[n_l SA(N_l) + n_r SA(N_r)], \qquad (1)$$

where n_l and n_r are the number of contained primitives in the respective child nodes. We take that partition that has minimal local cost C_P, or terminate if creating a leaf, which has cost $K_I \cdot n$, is cheaper, with $n = n_l + n_r$ being the number of primitives in the current node.

We give some details of our implementation concerning efficiency and robustness. Our octree is completely built on the GPU. We store an AABB and a counter in a bin. The primitives are represented by the centroid and the extent of their AABBs. For each primitive we compute the indices of the bins of all three dimensions from its centroid in SIMD. Then, the counters of all three bins are incremented, and their

AABBs are enlarged with the primitive's AABB using SIMD min/max operations. After all the calculations of a node are executed at three dimensions, we should record the counters and the surface areas, which will be used in the calculation of its children. The cost function needs to be sampled along all three dimensions.

The value of the cost function at a sampling position depends solely on the count of AABBs that intersect the left and right sub-volumes for the latter location. The algorithm collects this data in two phases. In the first phase, the algorithm sweeps the array of AABBs and stores at each sample the count of AABBs that end between this and the previous sample. The algorithm also stores the number of AABBs that start between the sample and the next one. In the second phase, the algorithm uses the collected information to incrementally reconstruct the AABBs counts at the sample locations using the formula

$$n_l^{i+1} = n_l^i + S^i,$$
$$n_r^{i+1} = n_r^i - E^i,$$

where $n^i_{l|r}$ describes the number of AABBs to the left/right of the sample i, and the number of AABBs that start and end between samples i and i+1 is denoted by Si and Ei, respectively. The algorithm evaluates the cost function at the sample locations and than distributes the AABBs to the left and right subtrees as described above.

Algorithm 1: **Improved Octree Construction**

Procedure BIN(AABB, sample)

D_{xyz} ← samples of min point of AABB

U_{xyz} ← samples of max point of AABB

For all dim ∈ {x,y,z} do

Increase sample.obj_S[U[dim]]

Increase sample.obj_E[D[dim]]

End for

End procedure

Function Found(head,dim)

Head[dim].Left[0]←0

Head[dim].Right[0]←NumObj

For each sample point at the dim

Head[dim].Left[i]←Head[dim].Left[i-1]+Head[dim].obj_S[i]

Head[dim].Right[i]←Head[dim].Right[i-1]+Head[dim].obj_E[i]

End for

Evaluates the cost function at the sample locations at each dim

Return the best found split

End function

For each input triangle t **in parallel**

Compute AABB for triangle t, add into AABBset

End for

Head←0

BIN(all aabb ∈AABBset,0)

For all dim ∈ {x,y,z} **do In parallel**

If NumObj<threshold **then**

Run conventional kd-tree routine for subtree

End if

Found(head,dim)

End for

Calculate the counters and the surface areas

Repeat

IV. IMPLEMENTATION AND RESULTS

In this section, we describe our implementation and the performance comparison. The described algorithm has been tested on an Intel Xeon 3.7GHz CPU with an NVIDIA GeForce 8800 ULTRA (768MB) graphics card.

4.1 Implementation

We implemented the above octree builder using NVIDIA's CUDA framework [5]. Previous GPU programming systems limit the size and complexity of GPU code due to their underlying graphics API based implementations. CUDA supports kernels with much larger code sizes with a new hardware interface and instruction caching. The GeForce 8800 allows for general addressing of memory via a unified processor model, which enables CUDA to perform unrestricted scatter-gather operations.

The GeForce 8800 consists of 16 streaming multiprocessors (SMs), each containing eight streaming processors (SPs), or processor cores, running at 1.35GHz. Each core executes a single thread's instruction in SIMD (single-instruction, multiple-data) fashion, with the instruction unit broadcasting the current instruction to the cores. Each core has one 32-bit, single-precision floating-point, multiply-add arithmetic unit that can also perform 32-bit integer arithmetic.

GÄźnther [4] point out that the number of bins is a crucial parameter controlling the construction speed and accuracy. The more bins there are, the more accurate is the sampling of the SAH cost function, but the more work has to be done during calculation of the SAH function from the binned data (the binning steps are independent from the number of the bins). Additionally, binning becomes inefficient if the number of bins is close to the number of to-be-binned primitives. Therefore the number of bins k per dimension should be adaptively chosen linearly depending on number of primitives n and bin-ratio r: k = n/r and clamp it to [kmin, kmax]. Just as what GÄźnther said on [4], we experimented with different parameter sets representing a trade-off between speed and accuracy. The default settings are kmax = 128, kmin = 8, and r = 6. The fast settings are kmax = 32, kmin = 4, and r = 16.

We need to specify the number of thread blocks and threads per block for the parallel primitives and the code fragments marked by in parallel. In our current implementation, we use 256 threads for each block. The block number is computed by dividing the total number of parallel threads by the number of threads per block. During octree construction, we store all data in linear device memory allocated via CUDA. For structures with many fields such as nodes and triangles, we use structure of arrays (SoA) instead of array of structures (AoS) for optimal GPU cache performance. As the level of the tree increases, the data that needs to be processed per thread in order to construct the next child node decreases, so the numbers of the threads in a block executed in parallel at one time can be increased significantly. Furthermore, because of the small sets of data, the amount of SAH calculation is decreased significantly, and shorten the time for dividing. Besides, our octree construction has lower levels, which makes ray tracing traversal faster. Considering that conventional kd-tree can be more efficient on small sets of data, so if the algorithm encounters a partition with size bellow some threshold, it switches to a conventional kd-tree construction at the point.

4.2 Performance Comparison

In this section we present the results of our experimental work. The quality of the trees is assessed in two ways. Firstly, we compute the construction time. Secondly, we evaluate the practical effect of tree quality on render time by using the constructed trees in a ray tracer. The results are presented as a table. To evaluate the performance of our construction algorithm, we compared it to the conventional construction algorithm by measuring the time needed to build a SAH kd-tree using a variety of scenes, ranging from simple to reasonably complex, namely BUNNY, FAIRYFOREST, and CONFERENCE. The scenes and the viewpoints for the tests can be seen on Figure 2. Table 1 summarizes the comparison results for several publicly available scenes as shown in Fig.2. As shown, our octree construction algorithm is 8 ~ 12 times faster for these scenes. Although our technique is capable of constructing high quality octrees in real-time, it has its limitations. For small scenes with less than 5K triangles, CUDA's API overhead becomes a major bottleneck. In this case, it is more efficient to switch to a complete CPU method.

Our octree construction algorithm also scales well with the number of GPU processors. The running time contains a scalable portion and a small non-scalable portion due to the overhead of CUDA API and driver. Theoretically, the running time is linear with respect to the reciprocal of the number of processors. As shown in Fig.3, we ran the algorithm on a GeForce 8800 ULTRA graphics card with 16, 32, 48, 64, 80, 96, 112, and 128 processors respectively.As shown in the table, our algorithm always offer better rendering performance. For dynamic scenes, our ray tracer can build an octree from scratch through our fast and efficient construction method without prior knowledge about geometry motion, which is a competitive alternative to a state-of-the-art ray tracer. Note that here we do not claim that our GPU ray tracer is faster than all CPU ray tracers. Indeed, implementing the fastest CPU ray tracer is like chasing a moving target because various optimizations could be used for higher performance and some optimizations are hardware dependent, and better performance can be achieved by adding more CPU cores.

Figure 2: The scenes used for testing, from left to right: 1)"BUNNY" 2)"CONFERENCE" 3)"FAIRYFOREST"

Scene and #triangles	Highly optimized kd-trees		Our routine	
	constr. time	FPS.	constr. time	FPS

 ISBN 978-80-86943-94-7

“BUNNY ”,69 K	0.62s	4.9	0.08s	9.3
“CONFEREN CE ”,282K	1.41s	2.7	0.11s	5.27
“FAIRYFORE ST ”,180K	1.15s	2.5	0.15s	4.17

Table 1: Construction time and FPS performance for 1024x1024 resolution， including shading with shadow. We include the time needed to read back the result from CUDA and draw them through OpenGL in our results.

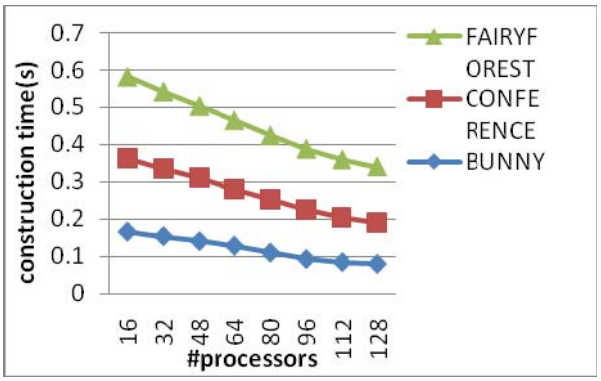

Figure 3: The tree construction time decreases quickly with the increase in the number of GPU processors

V.CONCLUSIONS AND FUTURE WORK

In this paper, we present an octree construction algorithm for the GPU that achieves real-time performance by heavily exploiting the hardware. This technique has four important features. Firstly, it builds octrees in real-time by exploiting the fine-grained parallelism on the GPU. Secondly, As the level of the tree increases, the data that needs to be processed per thread in order to construct the next child node decreases, so the count of SAH calculation is decreased significantly, and shorten the time for dividing. Thirdly, our octree construction has lower levels, which makes ray tracing traversal faster. Fourthly, because of using streaming construction, our octree consume less memory.

In future we plan investigation in the following directions. We plan to incorporate packets [13] into the GPU ray tracer for further performance enhancements. We also intend to amend the algorithm to add more secondary rays to render. Thirdly, we consider offloading some of the octree construction steps to a CPU. Vectorization using SIMD instructions will increase performance of the construction algorithm.

ACKNOWLEDGEMENTS

The first author would like to thank the anonymous reviewers for their insight and helpful comments， and Wen-qiao Zhu for his enthusiastic discussion during the early stage of this work. Ren C. has provided his help in implementation, and experimentation. This research work has been partially supported by National Key Technology R&D Program in the 11th Five year Plan of China(2007BAH11B05).

REFERENCES

[1] J. Owens. Streaming architectures and technology trends. GPU Gems 2, pages 457–470, March 2005.

[2] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-D Tree GPU Raytracing. In I3D ’07: Proceedings of the 2007 symposium on Interactive 3D graphics and games (2007), ACM Press, pp. 167–174.

[3]POPOV S., GĂźnther J., SEIDEL H.-P., SLUSALLEK P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. Computer Graphics Forum 26, 3 (Sept. 2007). (Proceedings of Eurographics)

[4] GĂźnther et al. Realtime Ray Tracing on GPU with BVH-based Packet Traversal the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007

[5] NVIDIA: The CUDA Homepage.

http://developer.nvidia.com/cuda. 1, 2, 3

[6] Kun Zhou et al. Real-Time KD-Tree Construction on Graphics Hardware, SIGGRAPH Asia 2008

[7] V. Havran. Heuristic Ray Shooting Algorithms. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[8] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. In Graphics Interface Proceedings 1989, pages 152–163.

[9] CARR, N. A., HALL, J. D., AND HART, J. C. 2002. The ray engine. In Proceedings of Graphics Hardware, 37–46.

[10] PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. ACM Trans. Gr. 21, 3, 703–712.

[11] FOLEY, T., AND SUGERMAN, J. 2005. Kd-tree acceleration structures for a GPU raytracer. In Graphics Hardware'05.

[12] CARR, N. A., HOBEROCK, J., CRANE, K., AND HART, J. C. 2006. Fast GPU ray tracing of dynamic meshes using geometry images. In Proceedings of Graphics Interface, 203–209.

[13] WALD I., HAVRAN V.: On building fast kd-trees for Ray Tracing, and on doing that in O(N log N). In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing (Sept.2006), pp. 61–70. 4

[14] YOON, S.-E., CURTIS, S., AND MANOCHA, D. 2007. Ray tracing dynamic scenes using selective restructuring. In Eurographics Symposium on Rendering.

[15] SHEVTSOV, M., SOUPIKOV, A., AND KAPUSTIN, A. 2007. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In Eurographics'07, 395–404.

[16] HAVRAN V.: Heuristic Ray Shooting Algorithms. Ph.D. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. 1, 2

[17] HAVRAN V., HERZOG W., SEIDEL H.-P.: On the Fast Construction of Spatial Hierarchies for Ray Tracing. In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing (2006), IEEE Computer Society, pp. 71–80.

[18] WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. In Proceedings of the Eurographics Symposium on Rendering (2006)

[19] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. ACM Trans. Graph. 24, 3 (2005)

[20] Kyu-Young Whang, et al. Octree-R: An Adaptive Octree for Efficient Ray Tracing IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS, VOL. 1, NO. 4, DECEMBER 1995

[21] AS. Glassner, "Space subdivision for fast ray tracing," IEEE Computer Graphics and Applications, vol. 4, no. 10, pp. 15-22, Oct. 1984.

[22] GOLDSMITH J., SALMON J.: Automatic Creation of Object Hierarchies for Ray Tracing. IEEE Computer Graphics and Applications 7, 5 (May 1987), 14–20. 3

[23] MACDONALD J. D., BOOTH K. S.: Heuristics for Ray Tracing using Space Subdivision. In Graphics Interface Proceedings 1989 (June 1989), A.K. Peters, Ltd, pp. 152–163. 1, 3

[24] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. ACM Transactions on Graphics 26, 1 (Jan. 2007), 6. 1, 3, 4, 5, 6

[25] POPOV S., G¨U NTHER J., SEIDEL H.-P., SLUSALLEK P.: Experiences with Streaming Construction of SAH KD-Trees. In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing (Sept. 2006), pp. 89–94. 1, 3, 4, 6

[26] HUNT W., STOLL G., MARK W.: Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing (Sept. 2006), pp. 81–88. 4, 6