Ray Tracing on a GPU with CUDA – Comparative Study of Three Algorithms

Martin Zlatuška
Czech Technical University in Prague
Faculty of Electrical Engineering
Czech Republic
zlatum1{@}fel.cvut.cz

Vlastimil Havran
Czech Technical University in Prague
Faculty of Electrical Engineering
Czech Republic
havran{@}fel.cvut.cz

ABSTRACT

We present a comparative study of ray tracing algorithms implemented on a GPU for three published papers using different spatial data structures evaluated for performance on nine static scenes in walk-through animation. We compare the performance for uniform grids, bounding volume hierarchies (BVHs), and kd-trees evaluated on a GPU for ray casting and Whitted-style ray tracing. We show that performance of ray tracing with BVHs exceeds the performance of ray tracing with kd-trees for coherent rays. Contrary, the ray tracing with kd-trees is faster than that with BVHs for incoherent rays. The performance of ray tracing with uniform grids is slower than both ray tracing with BVHs and kd-trees except for uniformly populated scenes. We show that the performance is highly sensitive to details of implementation on kd-trees.

Keywords: GPU programming, CUDA, performance study, ray tracing, uniform grids, kd-trees, bounding volume hierarchies.

1 INTRODUCTION

While modern graphics cards (GPUs) allow for general computation in a parallel manner, one of the most prominent applications for a GPU is image synthesis. This is thanks to the inherent parallel nature of ray tracing and other global illumination algorithms – the decomposition of images into pixels provides a natural way of creating individual tasks for many parallel processors. Unlike the GPUs a few years ago, modern ones allow us full programmability similar to general CPUs, while the streaming computation model has its own specific issues. This has to be taken into account when adopting the data structures and traversal algorithms for ray tracing on a GPU architecture.

In this paper we compare three formerly published papers that implement ray tracing with spatial data structures on a GPU. These are uniform grids [Pur02], kd-trees [Hor07], and bounding volume hierarchies [Gün07]. While the algorithms were successfully mapped to a GPU, their performance have not been carefully compared on a current programmable GPU architecture as a common implementation framework was not available. In this paper we first present such a comparison study dealing with efficiency of three different data structures for ray tracing on a GPU. We restrict ourselves to a static setting irrespective of the construction time as the data structures are built offline on a CPU for our tests. We show on a kd-tree that even small changes to the implementation of traversal code can lead to the significant change of performance.

This paper is further structured as follows. Section 2 summarizes the previous work of ray tracing on a GPU and performance comparison of data structures for ray tracing. Section 3 describes our choices for implementation. Section 4 shows the results from measurements on two GPUs for a set of scenes. Further it discusses the bottlenecks of a contemporary GPU architecture for ray tracing algorithms. Section 5 concludes the paper with possible prospectives for future work.

2 PREVIOUS WORK

In this section we review chronologically the most significant papers that address mapping of spatial data structures for ray tracing to a GPU. We discuss briefly all three data structures of our interest: uniform grids, kd-trees, and BVHs, while we avoid the discussion of results on other computer architectures except for a GPU as such surveys for CPU implementation have been provided for example in [Wal07].

Uniform grids. The first ray tracing algorithm mapped fully on a GPU has been published by Purcell et al. [Pur02] and uses a uniform grid. Their implementation mapped the computation by means of shaders while their data resided in a texture. In a concurrent work Carr et al. [Car02] present the architecture of a software ray tracer on a GPU with a focus on ray-triangle intersection with predefined BVH hierarchy. The mapping of both mentioned approaches had been influenced by architectural limitations. Recently, Kalojanov and Slusallek [Kal09] presented the algorithm for parallel construction of uniform grids on a GPU.

Kd-trees. A stack on a GPU with a low level of programmability was studied by Ernst et al. [Ern04] and used for stack-based kd-tree traversal algorithm. Foley and Sugerman [Fol05] presented two algorithms for kd-tree traversal without a stack. Their first algorithm called *kd-restart* is in fact the algorithm published by Kaplan [Kap85]. The second stack-less algorithm called *kd-backtrack* requires the storage of the bounding box and link nodes to its parent for every node of a tree, which significantly increases the memory footprint and hence it decreases performance. Both presented algorithms increase the number of nodes traversed compared to stack-based traversal algorithms. Another paper by Horn et al. [Hor07] addresses the lack of local memory to implement the stack much more efficiently. They propose the use of a push-down and short stack which can avoid most of the restarts of a traversal from the root node. This is possible as ray tracing with the kd-tree traverses only a few leaves on average. In concurrent work Popov et al. [Pop07] suggest to use the augmentation of a data structure by neighbor links among the nodes of a kd-tree. They even exceed the performance of CPU-based ray tracers while they achieve comparable performance as in [Hor07]. Further, Zhou et al. [Zho08] proposed the algorithm for kd-tree construction on a GPU. This method yields the performance of kd-tree construction comparable to CPU-based algorithms for kd-tree construction [She07]. This can be used for dynamic scenes up to 200,000 triangles to yield interactive performance.

Bounding Volume Hierarchies. Bounding volume hierarchies (BVHs) were also successfully implemented on a GPU. Thrane and Simonsen [Thr05] in fact compare kd-trees, uniform grids, and bounding volume hierarchies implemented on a GPU (hardware of year 2005). They conclude the performance of BVHs is low, however higher than the performance of other two data structures when no ray packets are used. Carr et al. [Car06] implemented a variant of BVHs in combination with geometry images. Günther et al. [Gün07] use ray packets and yield interactive performance comparable or exceeding CPU-based implementation, but only for primary and shadow rays. Recently, Lauterbach et al. [Lau09] present an algorithm for fast BVH construction on a GPU, where they report performance comparable to kd-trees [Zho08] only for one scene. Recently, Torres et al. [Tor09] published an algorithm for stack-less BVH traversal, where the use of stack is replaced by ropes connecting the nodes of a BVH in a sibling order. Very recently, Aila and Laine [Ail09] analyze the efficiency of various CUDA kernels for ray tracing with BVH (This paper is not included in our study as our research was completed in January 2009 in [Zla09].).

Comparison. For algorithms on a CPU it is believed that the hierarchical spatial data structures (both kd-trees and BVHs) built up in a top-down fashion yield similar performance. A decade old study by Havran et al. [Hav00] provides thorough performance comparison of twelve data structures implemented on a CPU. More recently Havran [Hav07] discusses the similarities and differences of top-down constructed spatial hierarchies (kd-trees and BVHs) and uniform grids. He argues that while kd-trees and BVHs have very similar properties as they can be mutually emulated in a constant time and space, uniform grids can outperform hierarchical data structures only for uniform distribution of objects in the scene.

To our best knowledge a proper recent experimental comparison of different ray tracers on a modern programmable GPU (year 2008 and 2009) has not been available. We would like fill the gap by our paper for a current GPU architecture (CUDA) of NVidia for a static scene setting (walk-through).

3 ALGORITHM IMPLEMENTATION

We have implemented a standalone compact program that does not need the support by other 3rd party libraries. The program implements a parser for scene format *PLY*, format *BART* [Lex01], and subset of Open Inventor format. While the data structures are built offline on a CPU, the created data structures are transferred to a GPU and used for ray tracing algorithm entirely on the GPU. To study the efficiency of shooting rays using different data structures this methodology is sufficient. The traversal algorithms and shading on the GPU were implemented using NVidia CUDA [PRG08].

The geometry of a scene consisting solely of triangles is represented by a list L_v of vertices and list of materials L_m, where each triangle has a list of three indices to L_v plus an index to the L_m. We tested also the variant where each triangle is represented directly by three vertices, however the memory consumption was increased with the negative impact to the performance. Shading is implemented via simple Phong model and is included in timing. The program can run in two modes - for measurement purposes and with GUI. Since we released the source code to public, we do not discuss many tiny but often relevant implementation details in this paper. Our paper serves as the summary of the Master Thesis [Zla09], where many details are stated, decision choices for that particular solution are discussed, and several unsuccessful attempts to improve the efficiency of algorithm implementations are described.

Below we describe the selected details of our implementation for uniform grids, kd-trees, and bounding-volume hierarchies.

3.1 Uniform Grids

The implementation of uniform grids loosely follows the paper [Pur02], with the traversal algorithm described in [Ama87]. The implementation is easier as CUDA is used instead of shaders. To decrease the number of registers we used a constant cache to store the values that do not change such as the direction and origin of the ray and precomputed values for 3D DDA traversal. This allows us to save five registers and get better occupancy [PRG08]. The threads have to be synchronized to compute the intersection with the triangles in the cells. The threads for the rays that do not intersect any cell with triangles are idle.

We tried to optimize the traversal algorithm by sharing the load of rays with many ray-triangle intersections with rays that do not need to compute many ray-triangle intersections. This required the rescheduling of the computation during the visit to the cell. However the resulting algorithm was several times slower than a simple algorithm, where some threads become inactive either when the computation is finished or a ray intersects an empty cell. Further, we also tried to implement packet tracing [Wal06] on a GPU. Although the pilot implementation has a uniform access to the memory and common branching, it resulted in an increased number of cells that were traversed. As a result, for packet of size 8×8 and for packets of size 4×4 the performance was substantially decreased compared to the simple implementation.

3.2 Kd-Trees

Kd-trees were built with surface area heuristics according to the sampling approach described in [She07]. Internally, each node of a kd-tree is represented by 8 Bytes, using the compact representation described in [Wal01].

We have been experimenting with several traversal algorithms and finally we decided to use a short stack traversal [Hor07] with four entries to compromise between number of traversal steps and occupancy. The stack is stored in shared memory. We aim at minimizing the conflicts in the shared memory as the threads for the rays are computed rather independently. We show the performance of two versions of kd-tree traversal code which illustrates the performance of very similar solutions. Initially, we stored three values to the short stack - "mint, maxt, and node address". However, we can decrease the size of stack entry to only two values, as for the farther node traversed the *mint* is equal to *maxt* for the node we just traversed. This changes the occupancy and performance as we show in Section 4. The traversal algorithm referred to as *kdt-3* stores three values to the stack, while the algorithm *kdt-2* stores only two values to the stack.

3.3 Bounding Volume Hierarchies

The BVHs were built in top-down fashion with surface area heuristics using the centroids of bounding boxes for scene triangles, following the paper by Günther et al. [Gün07]. As a BVH does not need to store the *mint* and *maxt* values along the ray, only the node address is saved to the stack. For packet traversal, the stack can be shared by all the threads in a packet, which increases the utilization of the resources. The stack does not need to be shortened to only several entries, which minimizes the number of traversal steps. The stack is similarly to kd-trees stored in shared memory.

The order of traversal among several threads is resolved by a concurrent write to the shared memory, where four memory entries are first initialized to zero. Each thread then writes the preference to one of four entries, value one for one of the four cases: traverse left, traverse right, traverse both, traverse none. The serialization of write operation may occur as threads record their information.

When rays diverge, the traversal continues to the node where most of the rays need to traverse. This is implemented by parallel reduction using auxiliary memory with one entry for each thread. Each thread writes either -1 when a left child should be visited as first, 0 for no preference, and 1 for the right child. The decision which first node should be traversed is then resolved by parallel reduction – the most node wanting to be traversed by most of the rays is visited as first while the other node is stored to the stack. When a thread does not need to visit any node, the node stores simply 0 as a preference. This is different from the algorithm described in [Gün07] and this change increases the performance for secondary rays by up to 20%. The disadvantage of BVH compared to kd-tree is the increased memory space required by the BVH node representation, it is 32 Bytes, which is 4 times higher than for a kd-tree node. However, it is compensated as the number of nodes and object references is strictly limited by the number of objects, so the storage of the whole BVH is typically smaller than the one for a kd-tree.

4 RESULTS

In this section we describe the results for measurement on nine scenes. To provide more variability to testing, we used three scenes of individual objects courtesy of Stanford scene repository, three scenes from BART [Lex01] (camera animated, objects not animated), and three other general interior architectural scenes. The rendered images of all scenes are shown in Figures 3, 4, and 5. These scenes are frequently used to test the performance of ray tracing and global illumination algorithm, the BART scenes [Lex01] scenes were designed for benchmarking of ray tracing.

To decrease the view dependence of results, we created a static walk-through animation for each scene of length 400 frames. All the performance results in this paper were measured on a GPU NVidia GeForce GTX 280 (June 2008), which has compute capability 1.3, 240 multi-threaded processor cores on 600 MHz, and 1 GByte of memory with a bandwidth of 141.7GB/sec. We also measured the results on an older, low-level GPU, an NVidia GeForce 8600GT (April 2007), where we got between 1/10 and 1/6 of the performance for the NVidia GeForce GTX 280.

The static properties of data structures for all nine scenes are shown in Table 1. The average computation time for the animation for a frame is shown in Table 2 for three settings: (1) shooting only primary rays, (2) primary and shadow rays, and (3) Whitted-style recursive ray tracing with two bounces for secondary rays. The occupancy for three scenes is shown in Table 3 for different settings of compilation in dependence on the number of registers where the maximum rendering times are reported. The results demonstrate that both the setting and the use of either three or two values stored to the traversal stack for a kd-tree have remarkable impact on performance. The dependence on the resolution is shown in Figures 1 and 2 for scene `Dragon` and `Robots`. More detailed results and evaluation can be found in [Zla09].

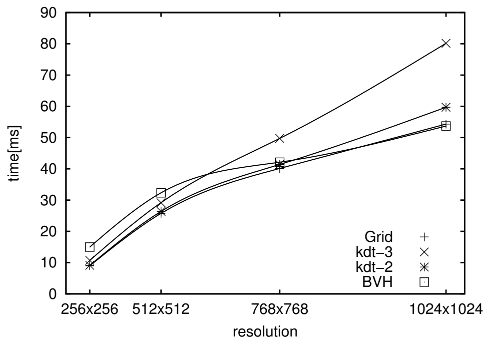

Figure 1: The dependence of computation time[ms] on resolution for scene *Dragon* for different resolutions.

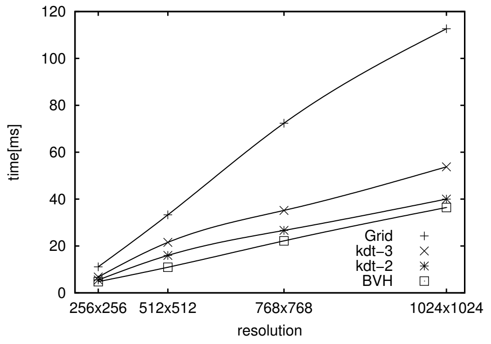

Figure 2: The dependence of computation time[ms] on resolution for scene *Robots* for different resolutions.

Discussion

While an interested reader can draw his/her own conclusion from the numbers in tables, let us provide our interpretation of the measured data. As we tested the performance on two different architectures (G80 and GT200), we can report: the progress of hardware was the most beneficial for the performance increase of a ray tracing with uniform grids. Similarly to the implementation on a CPU, the performance of uniform grids is superior only for uniformly populated scenes (`Bunny`, `Dragon`, and `Buddha`).

The (packet) ray tracing with BVH of incoherent rays is memory bound but is relatively well masked by switching threads. However, BVH has higher performance for (coherent) primary and shadow rays. This is in concordance with the results of Günther et al. [Gün07]. For traversing individual diverging (incoherent) rays such as secondary reflected rays in path tracing, the performance of BVH significantly deteriorates.

For diverging rays the kd-tree with its own short stack for each ray (thread) is a more efficient solution. The small size of each kd-tree node decreases the data traffic between memory and the processor cores. The bottleneck for the kd-tree traversal is a lack of larger local and fast memory for the stack implementation. The increase of local memory should lead to higher performance for upcoming GPU architecture(s).

As the performance of GPU ray tracing is dependent on many details in the implementation, this paper is accompanied by the source code available at: `http://dcgi.felk.cvut.cz/members/havran/rtgpu2009/`. We hope that the released source code can be further utilized in rendering applications and performance studies in future.

5 CONCLUSION AND FUTURE WORK

In this paper that serves as a summary of [Zla09] we have described a performance study comparing ray tracing implemented with CUDA on modern GPU from NVidia. We optimized the implementation for three data structures and traversal algorithms for ray tracing and compared the performance obtained from measurements for nine scenes for shooting primary rays, ray casting with shadow rays, and recursive Whitted-style ray tracing. The performance differed for coherent rays, where the bounding volume hierarchy is the winner, and for incoherent rays, where kd-trees seem to be more efficient on average when implemented using the short-stack as suggested by Horn et al. [Hor07]. However, the performance of ray tracing algorithm on a GPU is sensitive to many implementation details, likely due to the relatively small local cache on GPU architectures.

As future work, the implementation could be extended by several other data structures that can be efficiently mapped to a GPU architecture. The measure-

scene	#	#	grid					kd-tree					BVH			
	triangles	lights	Grid size	#refs	size [MB]	#trav. steps	#int. tests	#leaves [$\times 10^3$]	#refs	size [MB]	#trav. steps	#int. tests	#leaves [$\times 10^3$]	size [MB]	#trav. steps	#int. tests
`Bunny`	69451	1	83×82×64	3.7	8.3	47.4	37.2	154	5.5	9.2	54.0	12.3	23.0	2.84	52.1	8.0
`Dragon`	871414	1	273×193×122	3.1	103.1	117.3	45.5	978	2.3	58.5	68.8	10.4	295.0	35.8	114.1	28.0
`Buddha`	1087716	1	128×312×129	2.8	102.1	100.8	43.1	1265	2.5	76.5	64.3	8.8	389.0	44.7	130.9	30.9
`Robots`	71708	1	128×209×268	19.2	79.9	40.4	66.5	82	6.6	12.7	30.5	8.8	25.0	6.1	30.6	3.5
`Museum`	14380	2	71×43×106	9.5	5.1	60.2	33.6	26	4.5	2.0	19.2	6.7	4.6	0.9	40.1	3.9
`Kitchen`	110559	4	254×128×256	14.1	89.3	154.8	12.7	164	3.9	11.2	6.3	1.0	36.4	4.9	40.0	5.2
`Theatre`	53832	2	172×135×60	23.0	31.4	56.6	37.9	124	8.6	10.9	17.4	5.4	17.7	3.3	33.1	3.7
`Office`	36310	3	93×55×93	6.5	7.9	73.9	71.6	55	6.4	5.1	11.2	4.4	11.1	11.5	30.5	4.9
`Conference R.`	298866	2	387×246×93	10.3	121.3	165.7	31.5	338	8.7	51.6	14.3	4.8	97.9	14.5	34.6	5.9

Table 1: The properties of the test scenes and the spatial data structures built up for them. The general properties include number of triangles and light sources. For each data structure we report the number of leaves/cells. #refs corresponds the average number of references to objects in leaves. The storage for the data structure is given in MBytes. The number of intersection tests and traversal steps are reported for primary and secondary rays, the other results are in [Zla09].

	primary rays time[ms]				primary and shadow rays time[ms]				primary, shadow, secondary rays time[ms]			
	grid	kdt-3	kdt-2	BVH	grid	kdt-3	kdt-2	BVH	grid	kdt-3	kdt-2	BVH
`Bunny`	16.0	41.6	31.5	13.8	27.4	61.7	52.1	27.0	—	—	—-	—-
	116%	*301%*	*228%*	*100%*	*53%*	*118%*	*100%*	*52%*	—	—	—	—
`Dragon`	40.2	55.9	42.3	39.0	73.3	86.0	73.4	80.0	—	—	—-	—-
	103%	*143%*	*108%*	*100%*	*100%*	*117%*	*100%*	*109%*	—	—	—	—
`Buddha`	34.6	45.6	34.2	36.8	69.1	73.5	62.9	81.8	—	—	—-	—-
	94%	*124%*	*93%*	*100%*	*110%*	*117%*	*100%*	*130%*	—	—	—	—
`Robots`	27.3	20.5	16.2	25.9	53.8	35.6	30.1	50.0	89.0	43.8	38.7	64.3
	105%	*79%*	*63%*	*100%*	*179%*	*118%*	*100%*	*166%*	*230%*	*113%*	*100%*	*166%*
`Museum`	25.0	46.2	35.7	20.0	68.3	86.2	73.4	53.2	168.5	184.1	162.4	163.7
	125%	*231%*	*179%*	*100%*	*93%*	*117%*	*100%*	*72%*	*104%*	*113%*	*100%*	*101%*
`Kitchen`	41.6	40.5	31.9	29.3	209.6	130.0	110.8	138.8	442.8	244.4	214.3	403.9
	142%	*138%*	*109%*	*100%*	*189%*	*117%*	*100%*	*125%*	*207%*	*114%*	*100%*	*188%*
`Theatre`	43.1	42.3	33.1	34.3	119.7	87.3	74.3	93.6	379.7	201.6	177.5	292.1
	126%	*123%*	*97%*	*100%*	*161%*	*117%*	*100%*	*126%*	*214%*	*114%*	*100%*	*165%*
`Office`	52.9	44.1	34.2	22.7	218.6	116.1	101.5	87.9	224.0	120.1	107.7	94.2
	233%	*194%*	*151%*	*100%*	*215%*	*114%*	*100%*	*87%*	*208%*	*112%*	*100%*	*87%*
`Conference`	83.2	83.7	66.2	28.9	228.2	153.0	132.7	85.2	292.8	—	—	114.1
`Room`	*288%*	*290%*	*229%*	*100%*	*172%*	*115%*	*100%*	*64%*	*(257%)*	—	—	*(100%)*
Average[%]	*148%*	*181%*	*140%*	*100%*	*141%*	*117%*	*100%*	*104%*	*193%*	*113%*	*100%*	*142%*

Table 2: Average computation time for a frame [ms] for three settings rendered in resolution 1024×1024 for rendering 400 frames animations: (1) primary rays only (2) primary and shadow rays (ray casting) (3) primary, shadow, and secondary rays for recursion depth two (one primary ray per pixel). For individual objects (`Bunny`, `Dragon`, and `Buddha`) the setting (3) is meaningless. There was not enough memory for scene `Conference Room` to compute the recursive ray tracing with kd-trees. Timing includes also shading by Phong model. kdt-3/kdt-2 stands for storing 3 or 2 values to the stack during traversal.

ments and observations can provide interesting feedback to architects of graphics hardware in future.

ACKNOWLEDGMENTS

This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research program MSM 6840770014 and LC-06008 (Center for Computer Graphics) and the Aktion Kontakt OE/CZ grant no. 2009/6.

REFERENCES

[Ail09] Aila, T., and Laine, S.. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High-Performance Graphics 2009*, pages 145–150, New York, NY, USA, 2009. ACM.

[Ama87] Amanatides, J., and Woo, A. A fast voxel traversal algorithm for ray tracing. In G. Marechal, editor, *Eurographics '87*, pages 3–10. North-Holland, August 1987.

	# reg.	occupancy [%]	Robots		Kitchen		Museum	
			time[ms]	speedup [%]	time[ms]	speedup [%]	time[ms]	speedup [%]
grid	59	25	430.7		907.7		266.2	
	32	50	350.2	19	704.8	23	220.5	18
kdt-3	56	25	110.7		429.5		240.5	
	32	25	129.0	-18	496.2	-15	278.2	-15
kdt-2	56	25	110.7		429.5		240.5	
	40	37.5	96.0	13	372.2	13	211.4	12
BVH	53	25	151.0		1064.8		274.7	
	32	50	129.9	25	762.4	29	214.4	22

Table 3: GPU occupancy and timing for NVidia GeForce GTX 280 for three BART scenes for ray tracing with primary, secondary, and shadow rays in resolution 1024×1024.

[Car02] Carr, N.A., Hall, J.D., and Hart, J.C. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[Car06] Carr, N.A., Hoberock, J., Crane, K., and Hart, J.C. Fast GPU ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of Graphics Interface 2006*, pages 203–209, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[Ern04] Ernst, M., Vogelgsang, C., and Greiner, G. Stack implementation on programmable graphics hardware. In *Vision Modeling and Visualization 2004*, pages 255–262, 2004.

[Fol05] Foley, T., and Sugerman, J. KD-tree acceleration structures for a GPU raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM.

[Gün07] Günther, J., Popov, S., Seidel, H.-P., and Slusallek, P. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118, September 2007.

[Hav00] Havran, V., Přikryl, J., and Purgathofer, W. Statistical Comparison of Ray-Shooting Efficiency Schemes. Technical Report TR-186-2-00-14, Institute of Computer Graphics, Vienna University of Technology, Favoritenstrasse 9/186, A-1040 Vienna, Austria, May 2000.

[Hav07] Havran, V. About the Relation between Spatial Subdivisions and Object Hierarchies Used in Ray Tracing. In *23rd Spring Conference on Computer Graphics (SCCG 2007)*, pages 55–60, Budmerice, Slovakia, May 2007.

[Hor07] Horn, D.R., Sugerman, J., Houston, M., and Hanrahan, P. Interactive k-D Tree GPU Raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM.

[Kal09] Kalojanov, J. and Slusallek, P. A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 23–28, New York, NY, USA, 2009. ACM.

[Kap85] Kaplan, M.R. The uses of spatial coherence in ray tracing. In *ACM SIGGRAPH '85 Course Notes 11*, July 1985.

[Lau09] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, April 2009. (Proceedings of Eurographics 2007).

[Lex01] Lext, J., Assarsson, U., and Möller, T. A Benchmark for Animated Ray Tracing. *IEEE Comput. Graph. Appl.*, 21(2):22–31, 2001.

[Pop07] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).

[PRG08] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide, 2008. Version 2.1.

[Pur02] Purcell, T.J., Buck,I., Mark, W.-R., and Hanrahan, P. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, New York, NY, USA, 2002. ACM.

[She07] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum*, 26(3):395–404, September 2007. (Proceedings of Eurographics).

[Thr05] Thrane, N., and Simonsen, L.O. A comparison of acceleration structures for GPU assisted ray trac-

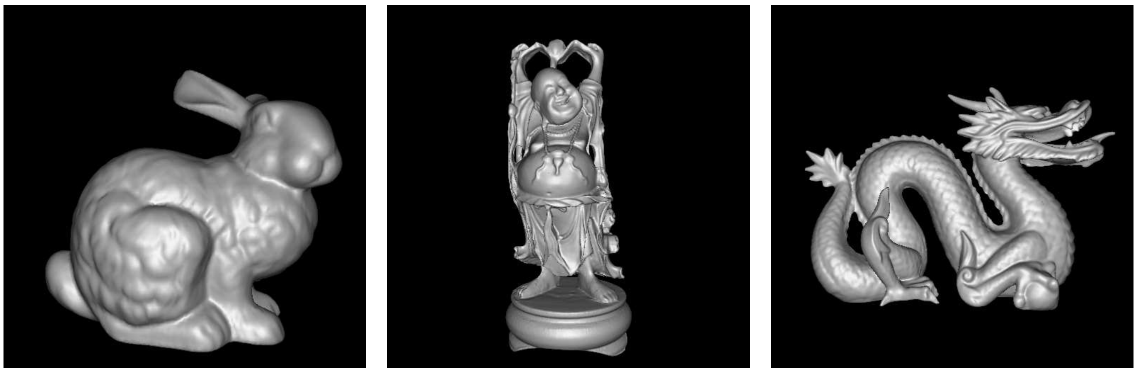
Figure 3: Stanford scenes: Bunny, Buddha, Dragon.

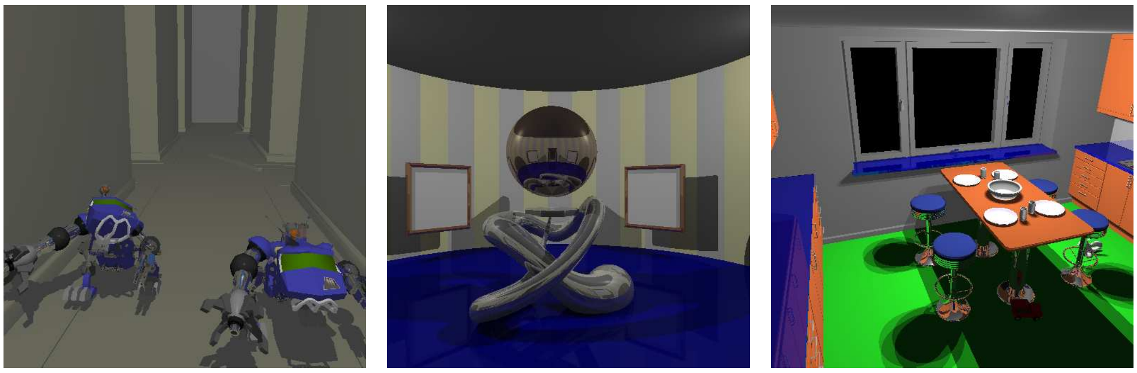
Figure 4: BART scenes: Robots, Museum, Kitchen.

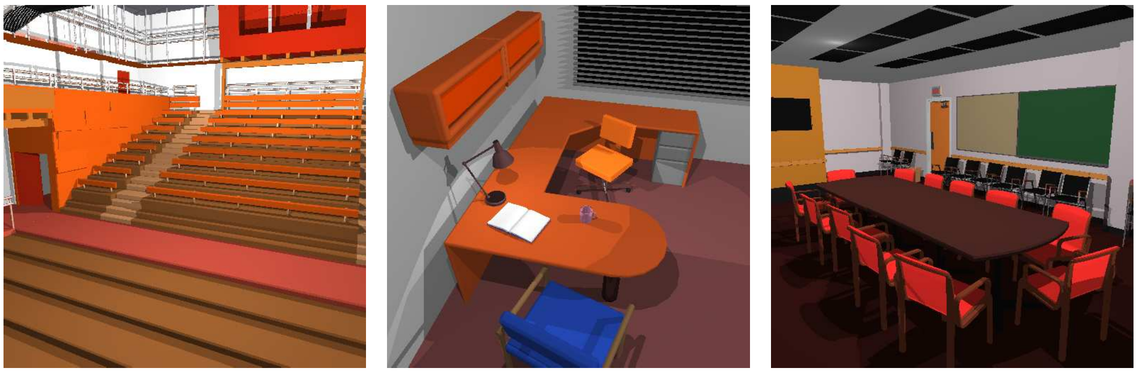
Figure 5: MGF scenes: Theatre, Office, Conference Room.

ing. M.Sc. Thesis, University of Aarhus, Denmark, 2005.

[Tor09] Torres, R., Martin, P.J., and Gavilanes, A. Ray Casting using a Roped BVH with CUDA. In *25th Spring Conference on Computer Graphics (SCCG 2009)*, pages 107–114, Budmerice, Slovakia, April 2009.

[Wal01] Wald, I., Slusallek, P., Benthin, C., and Wagner, M. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).

[Wal06] Wald, I., Ize, T., Kensler, A., Knoll, A., and Parker, S.G. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, pages 485–493, 2006. (Proceedings of ACM SIGGRAPH 2006).

[Wal07] Wald, I., Mark, W.R., Günther, J., Boulos, S., and Ize, T. Warren Hunt, Steven G Parker, and Peter Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*, 2007.

[Zho08] Zhou, K., Hou, Q., Wang, R., and Guo, B. Real-time KD-tree construction on graphics hardware. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, pages 1–11, New York, NY, USA, 2008. ACM.

[Zla09] Zlatuška, M. Ray Tracing Algorithms on Modern GPUs. M.Sc. Thesis, Czech Technical University in Prague, Jan 2009. http://dcgi.felk.cvut.cz/members/havran/rtgpu2009/.