Real-Time Ray Tracing with CUDA

Min Shih[1], Yung-Feng Chiu[1], Ying-Chieh Chen[1], and Chun-Fa Chang[2,*]

[1] National Tsing Hua University, Taiwan
[2] National Taiwan Normal University, Taiwan
{min_shih,yfchiu,louis}@ibr.cs.nthu.edu.tw,
chunfa@ntnu.edu.tw

Abstract. The graphics processors (GPUs) have recently emerged as a low-cost alternative for parallel programming. Since modern GPUs have great computational power as well as high memory bandwidth, running ray tracing on them has been an active field of research in computer graphics in recent years. Furthermore, the introduction of CUDA, a novel GPGPU architecture, has removed several limitations that the traditional GPU-based ray tracing suffered. In this paper, an implementation of high performance CUDA ray tracing is demonstrated. We focus on the performance and show how our design choices in various optimization lead to an implementation that outperforms the previous works. For reasonably complex scenes with simple shading, our implementation achieves the performance of 30 to 43 million traced rays per second. Our implementation also includes the effects of recursive specular reflection and refraction, which were less discussed in previous GPU-based ray tracing works.

Keywords: Ray Tracing, Programmable Graphics Hardware, GPU Computing, CUDA, Multithreaded Architectures.

1 Introduction

Ray tracing algorithm is a widely adopted technique in computer graphics for its high quality in realistic image synthesis. However, ray tracing is also known for its heavy computational requirement. Industrial ray tracing applications are typically off-line rendering systems that often rely on CPU clusters or rendering farm to provide the necessary computational power.

Due to its intrinsic parallelism, the ray tracing algorithm is a good fit for multi-core or multi-processor architectures. In order to speed up ray tracing and to make it practical for interactive applications, researchers have explored many novel parallel architectures [13, 19, 2]. Among them, programmable graphics hardware (GPU) has attracted significant interest for its high throughput. However, implementing ray tracing algorithm on the GPUs suffered from several architectural limitations. Therefore many researchers have focused on developing algorithm tricks that work around those limitations [5, 15, 8, 12, 6].

This paper focuses on real-time ray tracing on the GPU with the Compute Unified Device Architecture (CUDA) [11]. We have the following main contributions: (1) We

[*] Corresponding author.

present a CUDA implementation of ray tracing that is fastest so far to our best knowledge. For simple shading and primary rays only, we achieve 30 to 43 million rays per second on several reasonably complex scenes. An implementation of recursive ray tracing (or Whitted ray tracing [18]) is also demonstrated, which also achieves interactive frame rates. (2) Our work serves as a brief survey on various performance issues in practice. Several implementation decisions taken due to the architectural or algorithmic reasons are discussed in detail in this work.

1.1 Related Work

The study on GPU ray tracing began with the introduction of programmable shaders that are supported by graphics hardware. Purcell et al. [13] first proposed a scheme of GPU ray tracer. They treated GPUs as streaming processors and showed how to map a ray tracer to such a programming model. They used a uniform grid as the spatial partition structure for acceleration due to its simplicity.

It has been shown that the kd-tree built with surface area heuristics (SAH) is the best known acceleration structure for ray tracing of static scenes [7]. Foley et al. [5] overcame the hardness of implementing the stack needed for conventional kd-tree traversal algorithm on the GPU by proposing two stackless kd-tree traversal algorithms, namely *kd-restart* and *kd-backtrack*. Both kd-restart and kd-backtrack outperformed uniform grids on the GPU. Horn et al. [8] extended Foley et al.'s work by an improvement in the kd-restart algorithm. They used a size-restricted *short-stack* in their modified kd-tree traversal algorithm instead of eliminating the stack entirely. In the same year Popov et al. [12] presented another approach to stackless kd-tree traversal algorithm by taking the advantage of the concept of *ropes*, or the links between neighboring leaf nodes. Besides kd-trees, Bounding Volume Hierarchies (BVHs) for GPU ray tracing also have been investigated. Thrane and Simonsen [15] developed a fixed-order BVH traversal algorithm that discarded the use of the stack. Later, Günther et al. [6] presented a shared stack BVH traversal algorithm, which amortized the stack storage over the whole ray packet.

2 A Brief Introduction to CUDA

The Compute Unified Device Architecture (CUDA) [11] is a new technique for GPGPU (General-Purpose computing on GPUs). By removing several limitations from traditional GPGPU, CUDA provides useful features for implementing ray tracing: (1) a C-like programming language that eliminates the need of mapping the application to graphics API; (2) access on DRAM with general addressing; (3) full support for integer and bitwise operations.

When programming with CUDA, the *device* (GPU) is treated as a coprocessor to the *host* (CPU). A portion of an application that is executed many times on different data independently can be isolated into a function that executed on the GPU as many different *threads*. Such functions are called *kernels*.

When executing the kernel, the threads are organized into a *grid* of equal-sized (the size is assigned when launching the kernel) *blocks*, in which threads can cooperate with some fast shared memory and they can be synchronized by specifying a synchronizing point (block-wide barrier) in the kernel.

The hardware implementation of the device is a set of *multiprocessors*. Each multiprocessor has a Single Instruction Multiple Data (SIMD) architecture; typically it has 8 processors that execute the same instruction on different data (threads). The blocks being processed in a multiprocessor are divided into SIMD groups of threads called *warps* (typically the size is 32, where a basic instruction spends 4 clock cycles for a warp). The warps from all blocks being processed in a multiprocessor are executed in a time-slicing fashion; thus the long latency caused by some operations such as memory access and thread synchronization can be hidden by warp switching. How many blocks each multiprocessor can process in the time-slicing fashion depends on how many registers per thread and how much shared memory per block are required since these resources are shared among all the threads within the block. The ratio of the number of time-sliced warps per multiprocessor to the maximum number limited by the hardware implementation is called multiprocessor *occupancy*. Usually the higher occupancy (good parallelism) stands for the higher performance.

There are different types of memory in CUDA. Their characteristics are listed as Table 1.

Table 1. Memory types in CUDA. Symbol '$' indicates the memory is cached. "MP" stands for Multiprocessor.

Name	Scope	Restriction	Position	Speed	Size
Registers	Per-thread	Read-write	On-chip	Very fast	32KB/MP
Local memory	Per-thread	Read-write	DRAM	Slow	=DRAM size
Shared memory	Per-block	Read-write	On-chip	Very fast	16KB/MP
Global memory	Whole device	Read-write	DRAM	Slow	=DRAM size
Constant memory	Whole device	Read-only	DRAM($)	Fast	64KB
Texture memory	Whole device	Read-only	DRAM($)	Moderate	=DRAM size

3 Implementation Issues for CUDA Ray Tracing

The ray tracing problem can be explained by Figure 1. The view rays are shot from the camera to the scene; the corresponding pixel on the screen is shaded by the information of first intersection between the ray and the scene. Since the number of objects (represented as triangles in our implementation) within the scene can be very large, a spatial index structure is essential for accelerating the process of finding intersection. We choose the kd-tree built with SAH for the reason of performance [7]. For shadowing the hit object, shadow rays are shot from the light sources to the intersection point to determine whether the point is occluded. Whenever a specular or transparent surface has been hit, a reflection or refraction ray is shot in order to simulate the optical effect. If a reflection or refraction ray still hits a specular or transparent surface, the ray is traced recursively until a diffuse surface is hit or the maximum recursion depth is met.

3.1 The Ray Tracing Kernel

The core of our CUDA implementation is the ray tracing kernel, which processes a single ray. The process consists of four parts: ray generation, kd-tree traversal, triangle intersection test, and shading. Figure 2 shows the scheme of the kernel as well as the data structures needed for the kernel.

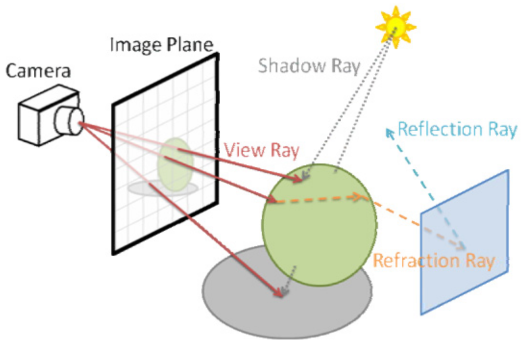


Fig. 1. Ray tracing

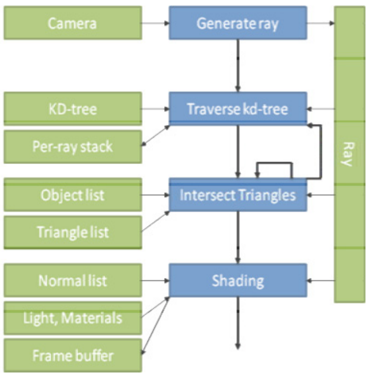


Fig. 2. Ray tracing kernel

3.1.1 Data Organization

Since CUDA has several different kinds of memory with respect to speed, size, and accessibility, in order to achieve a good execution performance, it is important to carefully allocate the data structures, avoiding the long access latency caused by low-speed memory.

The organization of triangles and the kd-tree is shown in Figure 2. The use of object list as a middle-layer between the leaf nodes and the triangles reduces the memory consumption in the case of shared triangles among different leaf nodes. This also increases the cache hit rate. All of the node list, object list, triangle vertex list, and normal list are stored as arrays bound with textures. Though the caching mechanism of textures is originally designed for coherent access patterns rather than random-like patterns such as tree traversal, we have found that it is still beneficial to bind the above structures with textures.

The small, static, and fixed-size structures namely the camera, light, and materials are all kept in constant memory.

Frame buffer is the only data output during the kernel execution. It is written only once per thread when the rendering is done, hence there is no need to cache it in the shared memory.

A ray in the 3-D space is represented by two 3-D vectors indicating the origin point and the direction which the ray points to. Although it seems natural to store the two vectors in local variables, which are referred to registers, such manner causes poor performance; the components of origin and direction vectors are frequently accessed by indexing, i.e. the vector is treated as a 3-element array, which is not supported by registers, and as a result, the compiler uses low-speed local memory to solve the dilemma and eventually worsens the performance. Thus the better solution is to place the vectors in shared memory. Actually this optimization improves the performance significantly.

3.1.2 Triangle Intersection and KD-Tree Traversal

Triangle intersection test and kd-tree traversal are the most time-consuming parts of the whole ray tracing system. Thus the algorithms for fast intersection and traversal have long been investigated and plenty of varieties have been proposed.

We have implemented and examined three kinds of triangle intersection test algorithms:

Möller-Trumbore Test [10]. It is probably the most well-known triangle intersection algorithm. This method is easy to be integrated into any program since it requires only the vertices of the triangle as input data.

Test Projection Test [17]. The projection method takes advantage of a pre-computed acceleration structure and performs well in terms of CPU cycle count.

Plücker Test [3, 14, 1]. Unlike most of intersection algorithms, which usually rely on computing barycentric coordinates, Plücker Test takes advantage of the properties of Plücker coordinates [3, 14]. The method is further optimized by Benthin [1] to amortize the computation over all rays within an origin-shared ray packet.

For kd-tree traversal algorithms, two issues should be discussed: First, whether we should trace the rays one by one separately or in a packet style, and second, whether we should use a stack or to rely on stackless methods as previous research.

Single ray vs. packet. The packet traversal algorithm was developed in order to exploit the computational power of the CPU SIMD [16, 17]. The algorithm bundles the rays together and determines a common traversal order for them, and therefore the SIMD instructions can be applied to operate several rays simultaneously within a thread. However, CUDA execution model differs from the CPU; the CUDA threads are scalar programs and they are forced to be executed in a SIMD manner. In other words, even the single ray algorithm is indeed executed parallelly within a multiprocessor, and no waste of computational power is caused. In contrary, despite the packet traversal reduces off-chip bandwidth and eliminates incoherent branches [9], it brings overhead on synchronizations to force the threads to go along with each other.

Stack vs. stackless. While traversing a tree-based structure usually needs a stack, it was difficult to implement a per-ray stack on the GPU due to architectural restrictions. The stackless traversal algorithms were developed to overcome this problem. However such restrictions are relaxed when programming with CUDA, which supports general DRAM addressing so that every thread can own its private part of memory, and thus it is possible to allocate a stack for each thread. The stackless manner may still have benefits because it avoids the long latency of access in non-cached DRAM. Nevertheless, the use of a stack keeps the rendering kernel simple, which is favorable for CUDA. Another option on CUDA may be brought up is to implement a stack using shared memory, but the size of shared memory is too small to even feed a warp of threads with enough storage space.

We have implemented and evaluated both the single ray and the packet traversal algorithms. For the single ray style, a stack traversal algorithm and three different stackless algorithms were implemented for comparison:

KD-Restart [5]. The kd-restart procedure eliminates the use of the stack by returning back to the root node whenever a stack-pop should have happened (that is, a leaf node has been just tested) and then "re-starts" the traversal step from the root node.

Short-Stack [8]. This method extends kd-restart by using a size-restricted *short-stack* instead of eliminating the stack entirely. The short-stack behaves just as an ordinary stack except a stack underflow occurs, in which the procedure goes back to the standard kd-restart manner.

KD-Tree Traversal with Ropes [12]. The ropes approach removed the use of the stack by the means of taking the advantage of the concept of *ropes*, or the links between neighbor cells. Whenever a leaf node has been finished processing, the traversal continues by determining from which face the ray exits the node and following the rope of this face to the adjacent node.

3.2 Shadow Rays and Secondary Rays

3.2.1 Shadow Rays

The shadow ray uses the same traversal code with the primary ray. A remarkable issue here is whether to process shadows rays in a one-pass or a two-pass manner.

One-pass. The shadowing processing is combined with primary ray as a single kernel. This style benefits from avoiding the information exchange between two kernels and reduces the overhead invited by kernel invocation. However, the drawback is that the kernel would become bigger and more complex, leading to the increase of register usage.

Two-pass. The shadowing task is separated from the primary ray as another kernel, and the shadow kernel is invoked after the primary kernel. The advantage is that the simplicity as well as the low register usage of the kernel can be kept, while the cost is brought by the necessity of communication between the two kernels. The information passing can be done with the use of a global buffer.

3.2.2 Secondary Rays

Since the reflection and refraction rays may be traced many times for accurate light transport simulation, it is better to implement them in independent kernels. Because the scene materials are not always perfectly reflective or transparent, the final shading result should combine the effects from primary and secondary rays. By adding a weight value to each ray, the final result of shading can be computed in an accumulative manner. To implement the recursive ray tracing algorithm that simulates the shading effects caused by multi-bounce reflection/refraction, the primary kernel and the secondary kernels form a *kernel tree* (instead of a ray tree in ordinary CPU ray tracing); the whole rendering process corresponds to invoking all these kernels in an appropriate order. The information passing between the kernels is done by the use of global buffers. The most memory-saving manner to process the kernel tree is to invoke the secondary kernels in a depth-first order, in which the necessary number of buffers for worst case is equal to the maximum number of bounces plus one.

4 Experiments and Results

We evaluated our implementation with an NVIDIA GeForce 8800 GTS graphics card running with an Intel 2.13 GHz Core 2 Duo. The CUDA program was built and executed with NVIDIA's CUDA development tools version 1.1 for Windows XP (including the CUDA toolkit 1.1, CUDA SDK 1.1, and 169.21 driver). The rendered image was displayed with OpenGL; the performance results were measured with the whole system executing including displaying. For testing several reasonably complex scenes are used: BUNNY, SPONZA, KITCHEN, and DRAGON. The scenes are shown in Figure 3. Table 2 lists the statistical data for the kd-trees built for the scenes. All scenes were rendered at 1024×1024 resolution.

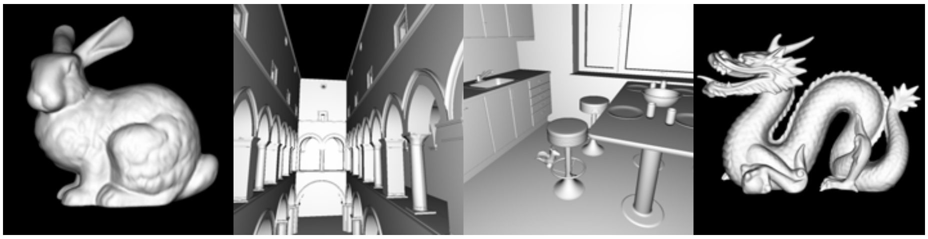


Fig. 3. Test scenes. From left to right: BUNNY, SPONZA, KITCHEN, and DRAGON.

Table 2. KD-Tree properties for test scenes. "#neleaves" stands for non-empty leaves. "#links" means the average number of references to triangles per non-empty leaf.

Scene	#triangles	#nodes	#neleaves	#links	Size
BUNNY	69,451	379,713	120,511	2.72	11MB
SPONZA	66,454	292,195	124,059	2.63	10MB
KITCHEN	103,343	362,585	146,253	2.63	14MB
DRAGON	871,414	1,687,237	607,852	2.59	103MB

Table 3. Execution time in milliseconds for pure triangle intersection test on 1000 triangles. "Reg. usage" row indicates the number of registers the kernels consumes.

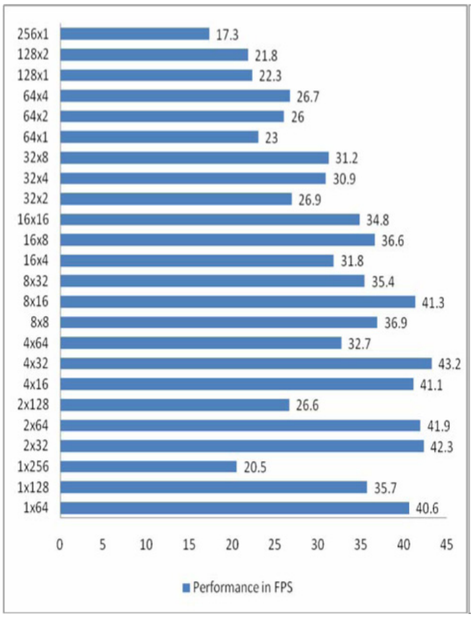
Scene	MT [10]	Projection [17]	Plücker [3, 14, 1]
BUNNY	515.3	439.8	566.3
SPONZA	515.3	431.5	578.7
Reg. usage	12	18	22

4.1 Block Size

The block size set for launching kernel affects the actual execution manner for the threads in the multiprocessors. We have tested many combinations of block sizes as shown in Figure 4. The results showed that the good combinations were 2×32 and 4×32, which might be explained by concept of *ray coherence* [4]. The coherence within a warp is significant since a warp is a SIMD group of threads, which means that if the threads of a warp are incoherent, the execution path of the warp is likely to diverge, and thus the utilization of SIMD is lessened. The coherence within a multiprocessor is also important because the threads being executed in a multiprocessor share the texture cache, which is persistently used for loading the kd-tree and triangle data. Usually the rays that consist in a nearly squared beam have good coherence. The block size of 2×32 seems not so coherent within a warp (the block is divided into warps in a y-major style; 2×32 causes 2×16 warps), but the coherence within a multiprocessor is moderate (14×32). Comparing to 2×32, the 4×32 block loses some multiprocessor occupancy, but this size takes the advantage of the higher warp coherence (4×8 warps). In conclusion, the key point is to keep three things: high occupancy, high coherence within a warp, and high coherence within a multiprocessor. We have made some experiments to verify the idea; we tried the sizes of 14×32 and 22×20, and they perform well just as we had thought.

4.2 Triangle Intersection

Table 3 lists the absolute performance of pure triangle intersection test for different algorithms. The testing kernel tested the first 1000 triangles of the scene in a for-loop. The results showed that the projection test [17] obviously outperformed. Since the computation of projection test consists of operations on scalar variables rather than vectors, the kernel requires fewer registers.



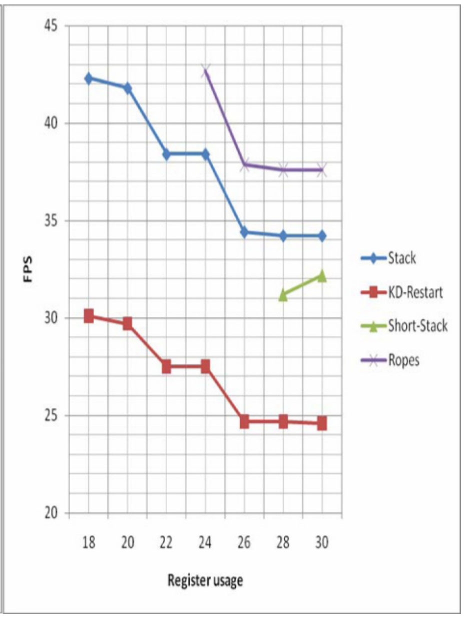


Fig. 4. Performance for different block sizes

Fig. 5. Performance to register usage. The frame rate descends as occupancy does.

Table 4. Comparison of kd-tree traversal algorithms. Performance numbers are given in frames per second. All scenes were rendered with primary rays only and simple shading.

Scene	Stack single ray	packet	Stackless kd-restart [5]	short-stack [8]	ropes [12]
BUNNY	43.2	15.8	30.9	31.7	38.4
SPONZA	40.5	16.5	28.6	29.5	34.4
KITCHEN	39.6	-	28.9	28.2	33.4
DRAGON	30.4	-	21.4	21.8	26.4
Reg. usage	18	20	17	28	24
Occupancy	50%	50%	50%	33%	33%

4.3 KD-Tree Traversal

Table 4 shows the comparison between the single ray traversal and the packet traversal as well as the comparison between the stack traversal and the stackless algorithms. It is obvious that packet approach was not suitable for CUDA at all. Exchanging information between the threads within a packet spent too much processing time. The results also showed that the conventional kd-tree traversal algorithm which used a stack was better than all stackless methods in our implementation. Trying to explain the results, another experiment was taken: We controlled the register usage of each kernel by adding redundant variables to observe the occupancy effects. The results as shown in Figure 5 suggested that the kd-restart algorithm [5] and the short-stack algorithm [8] were defeated by stack traversal even with the same register usage. It seems that the long latency invited by accessing local memory, where the stack were kept, was successfully hidden by multithreading. On the other hand, kd-tree traversal with ropes [12] was faster than the stack approach as long as the register usage was the same since it benefited from the pre-computing. The limited performance of the ropes approach was clearly caused by the low occupancy.

4.4 Shadow Rays and Secondary Rays

Figure 6 shows the rendering results for shadow and secondary rays. Table 5 lists the comparison between the one-pass and the two-pass approach for shadow rendering

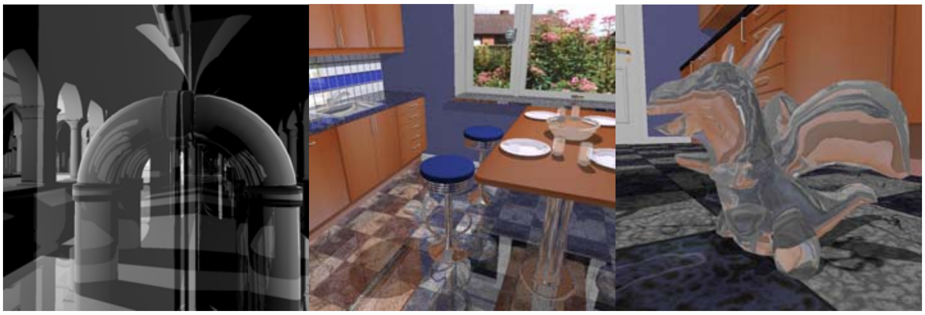


Fig. 6. Rendering results for shadow and secondary rays. From left to right: 3-bounce reflection/refraction in SPONZA, 4-bounce reflection/refraction in KITCHEN, and 4-bounce refraction only in KITCHEN.

Table 5. Shadow and secondary rays. Performance numbers are given in frames per second.

Scene	Shadow one-pass	two-pass	Multi-bounce reflection/refraction 1-bounce	2-bounce	3-bounce
BUNNY	16.6	18.4	-	-	-
SPONZA	21.0	23.9	11.3	7.2	5.0
KITCHEN	18.1	20.1	9.1	5.9	3.9
Reg. usage	22	19/18	-	-	-
Occupancy	33%	50%	-	-	-

and the performance for multi-bounce reflection/refraction. The results suggested that two-pass approach for shadow rendering was better for its low register usage.

5 Conclusions and Future Work

In this paper we presented real-time GPU ray tracing with CUDA. The comparison shown in Table 6 shows that our work outperform previous works in terms of absolute performance in primary rays. The implementation of full-featured recursive ray tracing was also able to run at an interactive speed. We discussed the implementation decisions as well as their effects on performance. The data was carefully organized to fit the CUDA memory model. Architectural issues such as block size decision and the multiprocessor occupancy have shown significant influence on the performance. Our experiment results suggested that the single ray traversal using a stack with the projection test was the best combination of kd-tree traversal and triangle intersection algorithms. It was also shown that a two-pass approach outperformed one-pass approach for shadow rendering due to occupancy.

As for future work, we would like to extend the work to include the comparison with the BVH. The use of BVHs also means that it can be considered to do ray tracing with deformable scenes. Another possible extension of this work is to support even higher rendering quality through the implementation of distributed ray tracing or path tracing.

Table 6. Comparison of related work and our work

	Method	M rays/s	Scene	Platform
CPU	KD-tree, stack, frustum culling	10-20	Sponza (66K)	Intel Core 2 Duo
Foley et al. [5]	KD-tree, kd-restart	1.43	Bunny (69K)	ATI X800 XT PE (2004)
Horn et al. [8]	KD-tree, short-stack	15.2	Conference (174K)	ATI X1900 XTX (2006)
Popov et al. [12]	KD-tree, ropes	12.7/16.7	Bunny (69K) /Conference (174K)	NVIDIA 8800 GTX (2006)
Günther et al. [6]	BVH, shared stack	19	Conference(174K)	NVIDIA 8800 GTX (2006)
Our work	KD-tree, stack	30-43	Various (66K-871K)	NVIDIA 8800 GTS (2006)

Acknowledgements. This work is supported in part by by National Science Council (Taiwan) under grant NSC 97-2220-E-003-001.

References

1. Benthin, C.: Realtime Ray Tracing on Current CPU Architectures. PhD thesis, Saarland University (2006)
2. Benthin, C., Wald, I., Scherbaum, M., Friedrich, H.: Ray Tracing on the CELL Processor. In: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing (2006)
3. Erickson, J.: Plücker Coordinates. Ray Tracing News (1997), `http://tog.acm.org/resources/RTNews/html/rtnv10n3.html#art11`
4. Foley, J., van Dam, A., Feiner, S.K., Hughes, J.F.: Computer Graphics – Principles and Practice, 2nd edn. Addison Wesley, Reading (1997)
5. Foley, T., Sugerman, J.: Kd-tree acceleration structures for a gpu raytracer. In: HWWS 2005: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pp. 15–22. ACM Press, New York (2005)
6. Günther, J., Popov, S., Seidel, H.-P., Slusallek, P.: Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In: RT 2007: IEEE Symposium on Interactive Ray Tracing, pp. 113–118 (2007)
7. Havran, V.: Heuristic Ray Shooting Algorithms. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague (2001)
8. Horn, D.R., Sugerman, J., Houston, M., Hanrahan, P.: Interactive k-d tree GPU raytracing. In: I3D 2007: Proceedings of the, symposium on Interactive 3D graphics and games, pp. 167–174. ACM Press, New York (2007)
9. Houston, M.: Performance analysis and architecture insights. In: SUPERCOMPUTING 2006 Tutorial on GPGPU, Course Notes (2006), `http://www.gpgpu.org/sc2006/slides/10.houston-understanding.pdf`
10. Möller, T., Trumbore, B.: Fast, minimum storage ray triangle intersection. Journal of Graphics Tools 2(1), 21–28 (1997)
11. NVIDIA. The CUDA homepage, `http://www.nvidia.com/cuda`
12. Popov, S., Günther, J., Seidel, H.-P., Slusallek, P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. Computer Graphics Forum 26(3), 415–424 (2007) (Proceedings of Eurographics)
13. Purcell, T.J., Buck, I., Mark, W.R., Hanrahan, P.: Ray tracing on programmable graphics hardware. ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH), 703–712 (2002)
14. Shoemake, K.: Plücker Coordinate Tutorial. Ray Tracing News (1998), `http://tog.acm.org/resources/RTNews/html/rtnv11n1.html#art3`
15. Thrane, N., Simonsen, L.O.: A Comparison of Acceleration Structures for GPU Assisted Ray Tracing. Master's thesis, University of Aarhus (2005)
16. Wald, I., Slusallek, P., Benthin, C., Wagner, M.: Interactive rendering with coherent ray tracing. Computer Graphics Forum 20(3), 153–164 (2001)
17. Wald, I.: Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Saarland University (2004)
18. Whitted, T.: An Improved Illumination Model for Shaded Display. CACM 23(6), 343–349 (1980)
19. Woop, S., Schmittler, J., Slusallek, P.: Rpu: a programmable ray processing unit for realtime ray tracing. ACM Trans. Graph. 24(3), 434–444 (2005)