

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/221306630>

kD-Tree Traversal Implementations for Ray Tracing on Massive Multiprocessors: A Comparative Study

Conference Paper · October 2009

DOI: 10.1109/SBAC-PAD.2009.25 · Source: DBLP

CITATIONS

9

READS

374

5 authors, including:



[Artur Lira dos Santos](#)

Federal University of Pernambuco

8 PUBLICATIONS 54 CITATIONS

[SEE PROFILE](#)



[Joao Teixeira](#)

Federal University of Pernambuco

46 PUBLICATIONS 120 CITATIONS

[SEE PROFILE](#)



[Thiago Farias](#)

Universidade de Pernambuco

20 PUBLICATIONS 74 CITATIONS

[SEE PROFILE](#)



[Veronica Teichrieb](#)

Federal University of Pernambuco

139 PUBLICATIONS 510 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



MAR Tracking System [View project](#)



ARBlocks: Augmenting Education [View project](#)

All content following this page was uploaded by [Judith Kelner](#) on 15 May 2014.

The user has requested enhancement of the downloaded file.

kD-Tree Traversal Implementations for Ray Tracing on Massive Multiprocessors: a Comparative Study

Artur L. dos Santos, João Marcelo X. N. Teixeira, Thiago S. M. C. de Farias, Veronica Teichrieb, Judith Kelner
Federal University of Pernambuco, Computer Science Center
Virtual Reality and Multimedia Research Group
Recife, Brazil
{als3, jmxnt, tsmcf, vt, jk}@cin.ufpe.br

Abstract—Current GPU computational power enables the execution of complex and parallel algorithms, such as Ray Tracing techniques supported by kD-trees for 3D scene rendering in real time. This work describes in detail the study and implementation of five different kD-Tree traversal algorithms using the parallel framework NVIDIA Compute Unified Device Architecture (CUDA), in order to point their pros and cons regarding adaptation capability to the chosen architecture. In addition, a new algorithm is proposed by the authors based on this analysis, aiming performance improvement. A performance analysis of the implemented techniques demonstrates that two of these algorithms, once adequately adapted to CUDA architecture, are capable of reaching speedup gains up to 15x when compared to former CPU implementations and up to 4x in comparison to existing and optimized parallel ones. As a consequence, interactive frame rates are possible for scenes with 1376×768 pixels of resolution and 1 million primitives.

Keywords—Ray Tracing; kD-Tree; Traversal; CUDA;

I. INTRODUCTION

Ray Tracing has become a very interesting topic for the computer graphics community in last decades. Several scientific works were produced showing images generated based on natural phenomena, usually optics, that can reach real world appearance [1]. On the other hand, Ray Tracing techniques are also distinguished as computationally expensive, since their workflow consists in casting rays from every image pixel and following these rays to decide how to paint that pixel, according to the scene, light sources and material properties.

For many years, Ray Tracing generated images were mainly explored by animation movies and offline content generation systems. Many algorithms used in computer graphics to simulate behaviors such as refraction, reflection, shadow generation and self shadowing, global illumination, color bleeding, sub surface scattering, and even simple shading schemes (such as Gouraud and Phong), are easier to code in straight ray based implementations.

However, spatial data structures allowed Ray Tracing to walk towards real time rendering. They have to be applied to the application scene graph in order to reduce time spent by the ray casting and primitive structures (such as triangles, spheres, implicit surfaces, etc) intersecting procedures.

Ray Tracing techniques are very parallelizable, since the screen can be split to distribute rays over parallel processing units. The movie industry uses render clusters, usually named render farms, to generate complex ray traced movie scenes. Another option is taking advantage of multicore processors and/or coprocessors, to run the entire solution on a single machine still having a good result.

Recent studies report performance improvements in ray tracers by the use of GPUs as coprocessors. Specifically, the NVIDIA CUDA platform [2] can be successfully applied to Ray Tracing application development. Its architecture delivers a many core solution containing parallel stream processors that may be used to general purpose programming. That strategy was exploited in this work, which adapts existing spatial queries data structures to run in parallel achieving real time frame rates.

Initially, this work performs a review of the most used algorithms in the Ray Tracing research area. Five common used techniques were implemented in CUDA in order to create a comparative grid of such approaches, and then a highly optimized GPU real time ray tracer for static scenes, taking benefit of the advantages of the studied algorithms, is proposed. As far as we know, our new implementation surpasses the performance of current GPU counterparts.

II. RELATED WORK

Recent efforts regarding Ray Tracing have been about optimizing existing algorithms and data structures to achieve the best image quality/speed on given hardware architectures. The modification of these algorithms guides optimizations on such hardware architectures to better suit the needs of computer graphics developers.

As graphics hardware has become increasingly programmable and suited to more general-purpose computation, developers were attracted to this platform. By bringing hardware and software together, it is possible to explore how to best design hardware as well as software systems and what are the best interfaces between them. As witnessed in recent years, the cycle of rapid innovation in hardware systems enabling new software approaches, and innovative new graphics software driving the hardware architectures

of the future has brought great benefit to the area of high performance graphics.

Horn et al. [3] take advantage of the GPU processing power to perform Ray Tracing on an X1900 XTX. Their implementation uses ATI's CTM (Close To the Metal) toolkit and pixel-shader 3.0. The entire process runs at 12-14 frames per second, over 1024×1024 scenes with shadows and Phong shading.

Popov et al. [4] propose a stackless packet traversal algorithm that supports arbitrary ray bundles and handles complex ray configurations efficiently. Their GPU implementation uses CUDA and runs on a GeForce 8800 GTX. Allowing primary rays only, they achieve about 15 frames per second over scenes with 1024×1024 pixels of resolution.

Gunther et al. [5] present a bounding volume hierarchy (BVH) ray tracer with a parallel packet traversal algorithm. Using a GeForce 8800 GTX, they achieve 3 frames per second for the well-known Power Plant scene, with 12.7 million triangles and 1024×1024 pixels of resolution.

All these works converge taking advantage of the capabilities of hardware systems. Exploiting existing hardware running novel algorithms and data structures, improving both the hardware and the software, and discussing the best applicable programming models and interfaces to hardware capabilities are key challenges facing all of us.

III. CUDA FRAMEWORK

CUDA is a framework released by NVIDIA that allows seamless general purpose parallel programming in GPUs. CUDA compatible GPU cards act as a manycore SIMT (Single Instruction, Multiple Thread)[2] processor, which can process millions of threads using a lightweight scheduling model.

CUDA offers a stratified memory model that allows the use of different levels of memory, which varies in latency, size and amount [2]. The CUDA execution model groups threads into blocks, and blocks into a grid. These blocks can be spatially distributed in 1, 2 or 3 dimensions, according to the programmer needs. Functions implemented in GPU (known as kernels) can be directly called from CPU side, unifying the GPU-CPU programming.

IV. ACCELERATION STRUCTURES

Ray Tracing techniques commonly present a high computational cost since they need to perform, for each ray \mathbf{r} emitted into the scene, a search for the closest geometric primitive \mathbf{p} that is intersected by \mathbf{r} . A simple way of solving this problem is to test intersections between \mathbf{r} and every primitive in the scene [6]. Since this approach passes through all scene objects, this algorithm makes the search complexity linear [7].

Due to the fact that ray-object intersection involves dozens of floating point arithmetical operations, it is costly in most computer architectures. Consequently, this approach

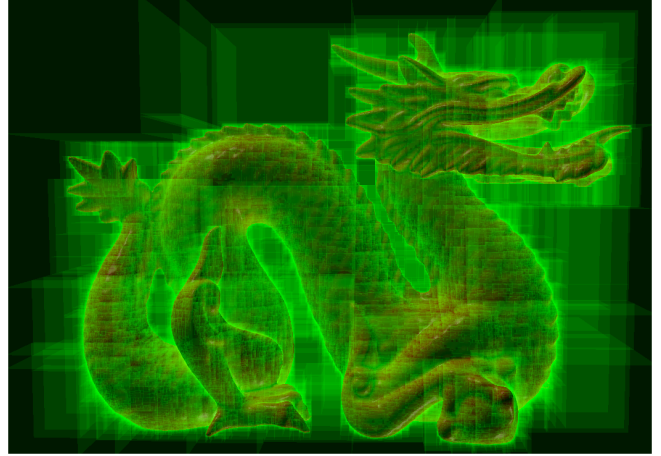


Figure 1. Stanford Dragon model subdivided into small bounding boxes. Light areas represent high density kd-tree nodes

becomes prohibitive for real time Ray Tracing of scenes containing millions of geometric primitives. Using this method, even for scenes with a few hundred primitives, about 95% of processing time is spent with intersection calculations [6].

This problem can be minimized by the use of special data structures capable of spatially organizing or grouping scene objects, in order to considerably reduce the amount of intersection tests involved in the search. This way, instead of following a “blind” search for the next closest intersection, only objects with a high probability of intersection with the ray are tested, discarding the remaining ones (like those placed far away from the ray’s path). Since these remaining objects comprise the most part of the scene, the increase in performance is significant.

Most of the data structures used are adaptations of search trees. Such structures help reducing search complexity to a sub-linear one [7]. They can be created by using concepts such as spatial scene subdivision (Partitioning Space), subdivision involving sets of objects (Partitioning Object List), or hybrid approaches taking benefit of both previous concepts [8]. Among many existing acceleration structures, the ones most used in Ray Tracing are related to Binary Space Partitioning Trees (BSP Trees) [9]. Such tree recursively divides the scene at each branch, separating all scene objects into sub-groups, as shown in Figure 1.

The recursive subdivision is originated from a chosen splitting plane located on each branch of the tree. The splitting plane choice is restricted to one of the coordinate axes, thus simplifying the traversal algorithm. This specific tree is known as kd-tree (k - Dimensions tree). On a kd-tree, every internal node stores the dimension and position of the splitting plane that will divide the scene in two sub-spaces, and the node’s children function as representatives of such sub-spaces. The leaves represent the smallest sub-spaces of the tree and store a list of objects located within

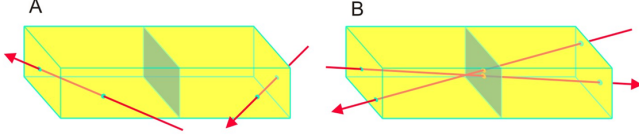


Figure 2. Ray-Node intersection cases a) one child; b) both children

them. Since the normals of kD-tree splitting planes are always canonical vectors of world coordinates, each scene sub-space is represented by Axis-Aligned Bounding Boxes (AABBs). In this case, the tree’s root node represents the scene bounding box. This way, every traversal algorithm is reduced to ray-AABB intersection tests until intersecting an object inside the AABB, or the ray leaves the scene bounding box without intersecting any object. Since a single axis is used as splitting plane support for each node, the ray-AABB intersection test is again simplified, now to one dimension, which considerably reduces the number of arithmetical operations used.

When intercepting one node, a ray can traverse just one or both children, as shown in Figure 2. For traversing both children, in the standard traversal algorithm [7], a stack is needed in order to store the child-node located most far away (far child). This information may be used in a posterior search, since the traversal first follows the near child path, and later the far child one. The stack guarantees that the traversal occurs according to the order in which bounding boxes are intercepted by the ray, visiting a single leaf node at a time.

During last years, the fast computational power growth of graphics processors has attracted the attention of researchers seeking for efficient ray tracer GPU implementations. This search was also focused on acceleration structures such as kD-trees. Foley et al. [10] have adapted the standard traversal algorithm, creating two new traversal algorithms that do not require a stack for the search, reducing “memory footprint” and consequently the number of accesses to GPU’s primary memory.

The so called kD-Restart algorithm starts over the traversal to the tree root every time it reaches a leaf of the tree. This “restart” operation continues until an intersection with any primitive is found or the ray leaves the scene bounding box. When the process is restarted to the root, it is not more necessary to store its children intersection tests, which eliminates the need for the stack required by the standard algorithm. However, the average cost of this search becomes considerably higher than the standard one, since many nodes are revisited when search goes back to the root.

In order to reduce the amount of revisited nodes, the second algorithm called kD-Backtrack stores in each node its bounding box (AABB) and a pointer to its parent. Consequently, it is not necessary anymore to return to the root every time a leaf is found; the algorithm just needs to

return to a node that enables traversing other unvisited nodes. In spite of reducing the number of visited nodes, the kD-Backtrack algorithm demands about an order of magnitude of more memory to store the entire tree in comparison to the kD-Restart algorithm, due to the necessity of storing information about parents pointers and AABBs. Therefore, kD-Backtrack algorithm not suitable for complex scenes, with more than 10 million polygons.

In order to obtain a performance comparable to the original traversal algorithm one, Horn et al. [3] proposed a few modifications to the kD-Restart algorithm to reduce the total amount of visited nodes, through push-down and short-stack algorithms. The push-down technique keeps the last visited node n that can be considered as query root; in such node, the ray hits only one of the node’s children. This way, whenever a restart event occurs, instead of returning to the root, the search restarts at node n , which reduces the amount of revisited nodes.

Besides the push-down technique, Horn et al. [3] proposed the short-stack algorithm, which uses a circular stack with about 10% the size of the stack used in the original algorithm. In case the stack empties and the ray did not hit any primitive, leaving the scene bounding box, the restart event is processed in the same way of the kD-Restart, redirecting the query to the root.

Push-down and short-stack algorithms optimize different parts of kD-Restart and can be used together. In this case, if a restart event is raised in short-stack, instead of returning to the root, the search goes back to the node previously stored by push-down.

Popov et al. [4] demonstrate a CUDA implementation of a stackless traversal using the concepts of ropes [11], in which each leaf stores its corresponding bounding box and 6 pointers (ropes) to the neighbor nodes of the faces of its bounding box. This way, when a leaf is reached, instead of returning to a node stored in the stack, one of the 6 stored pointers is used. Consequently, the traversal using ropes visits fewer nodes than most of the algorithms cited before, with the cost of having to perform more memory accesses for fetching bounding boxes and rope information.

Our work proposes some improvements and adaptations to the GPU stackless algorithm proposed by Popov et al. in order to minimize the “register pressure” effect, which occurs when there are more variables to allocate than registers available. It also aims reducing costly accesses to global memory, making better use of CUDA’s architecture.

V. PROPOSED IMPLEMENTATION

Global lighting techniques such as Ray Tracing are based on basic concepts of Optics [1] in order to render more realistic scenes in comparison to rasterized ones. Images with complex visual effects like transparent or reflexive objects are easily obtained with the use of Ray Tracing,

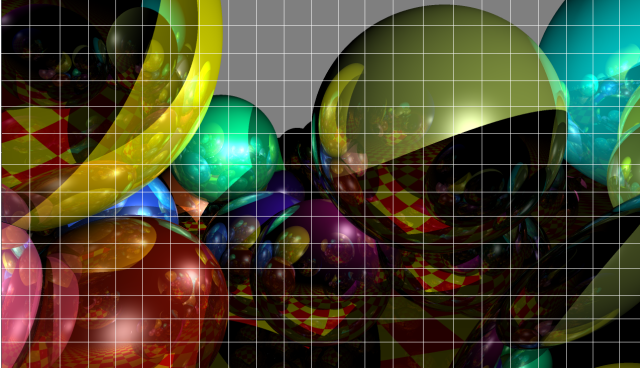


Figure 3. Tiles used to process neighbor pixels

being this technique indicated for more precise renderings, with plenty of details.

Our implemented Ray Tracing system supports reflections and shadows according to the model described by Turner Whitted [6]. Due to hardware limitations, CUDA does not support recursive calls. As a consequence, we adapted the standard recursive algorithm to an iterative one. By using only an instance of the primary ray, whenever an intersection is found, the ray is translated to the intersection point and has its direction modified according to the type of ray to be generated (shadow or reflected ray). The iterations end when the reflected ray does not hit any object or when it reaches the maximum level of reflection previously determined.

A. Ray tracer kernel

Our implementation distributes the workload of each pixel to different threads, in a way that n pixels are processed by n parallel threads. Since CUDA presents itself as a SIMT (Single Instruction, Multiple Thread) architecture, it is important that near threads follow the same sequence of instructions whenever possible, to guarantee that the instruction is simultaneously executed in all threads. Because of that fact, near pixels are treated by near threads, favoring parallel execution.

Since rays generated by near pixels have equal origins and similar directions, there is a high probability that they perform a similar traversal on the scene, intersecting the same objects while executing the same group of instructions. Such alike rays are known as coherent rays.

CPU implementations take benefit of coherent rays when using them with packet traversals [12], by grouping the search computation of such rays in groups of SIMD instructions. However, CUDA’s SIMT architecture seems to be even more adequate to process coherent rays, since the search algorithm does not need to be modified in addition to the fact that no overhead is added, as happens with SIMD packet traversals.

By assigning the Ray Tracing of each pixel to one corresponding thread, it is possible to use a single kernel

for processing the entire algorithm in GPU, including the kD-tree traversal and shading. This way, each CUDA block of threads represents an image sub-region to be rendered, commonly known as tile, as shown in Figure 3. This approach offers less overhead in comparison to multi-kernel implementations, since for each change of pipeline stage it is necessary to store in global memory all the information that must be passed to the next stage.

However, the single kernel approach is not the best choice when dealing with scenes composed by many reflective objects: a significant amount of low coherent secondary rays is generated, harnessing the degree of parallelism due to divergent instructions and random accesses to memory. This is a known problem of CPU implementations that becomes far more serious on GPUs. Besides that, it is not difficult to find cases when only a small amount of the block generates secondary rays, in a way that many threads stay in “idle” state, waiting for the busy ones to complete. GPU resources are only made available when the entire block of threads being used has finished its processing, which does not occur in CPU, where a new pixel is processed every time one is finalized.

Given those problems, the implemented Ray Tracing makes use of a single kernel approach whenever rendering just primary rays with a small amount of reflection points. In case there are many reflective objects, a new approach for dealing with secondary rays based on “screen space” compression is used in order to reduce the amount of time wasted by idle threads.

B. Secondary rays

In order to lessen the secondary rays problem, pixel execution is subdivided into phases, each one corresponding to a new generated reflective ray to be computed.

At first, the kernel responsible for processing all primary rays traversal and shading is executed. A screen space Boolean mapping is created as a result of this first phase, in which pixels with value 1 represent the need for secondary rays. After that, the Boolean map is compressed using a parallel prefix sum technique [13]. Such map enables the execution of a new Ray Tracing kernel in which all active pixels are grouped into contiguous blocks. This guarantees a balanced processing load for every thread in a block, and consequently increases the thread occupancy on the stream processors by reducing the amount of idle threads. Despite the overhead caused by more kernel invocations and more global memory usage, this approach enables gains up to 30% in comparison to the single kernel approach.

C. Memory usage

The entire Ray Tracing scene, represented by its kD-tree, is stored in both CPU and GPU. This occurs due to the fact that the kD-tree construction process is performed by the CPU and the structure built is later sent to the GPU.

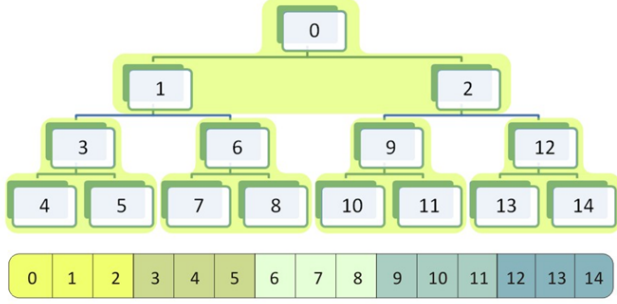


Figure 4. van Emde Boas tree layout

Since this work is focused on real time Ray Tracing of static scenes, and consequently does not require the scene reconstruction every new frame, we adopted a well-known kD-tree CPU implementation. The kD-tree construction utilizes a spatial subdivision technique based on Surface Area Heuristic with perfect splits [14]. As result, it is possible to pre-build a high quality kD-tree for complex scenes, with more than one million objects.

On the GPU side, both kD-tree and geometrical/material properties of objects are stored in global memory. In order to reduce the access penalty to such type of memory, we make use of texture bindings on global memory. This artifice provides us with a 16KB cache system with 256 bytes of cache line size on current graphics cards.

Despite the fact that Ray Tracing presents a high degree of parallelism, there is also a high demand for bandwidth due to the great amount of accesses to global memory. All threads from our implementation realize a considerably high number of memory accesses searching for objects in the scene. In case these accesses are realized inadequately (randomly, for example), the obtained results may not reach performance expectations.

In order to reduce the number of accesses to global memory, our implementation organizes the 3D scene in a way that geometric primitives near in space may be located in neighbor memory blocks. This way, by traversing the scene, the ray will probably intersect near objects based on the principle of data locality. This relationship is analogous, since near objects are stored in memory addresses close to each other, and helps increasing the cache hit rate for such objects.

The scene kD-tree is specially designed to increase the cache hit rates. For that, the tree is structured following the van Emde Boas Implicit Layout [7][15], shown in Figure 4. Such model favors the down traversal by assuring cache hits when descending to an odd level node. Consequently, a traversal would have at least 50% of cache hits, being statistically the best case scenario for random searches on trees, which are frequent in Ray Tracing.

Besides keeping a good cache hit rate, the traversal on a

van Emde Boas tree is considered “a cache-oblivious” algorithm, since it does not require previous knowledge about cache parameters to produce a good rate of cache hits. This constitutes an important advantage of our implementation, by making the code also “cache efficient” in future CUDA architectures.

When CUDA compiler is not capable of reducing the number of necessary variables in a certain scope in order to maintain a fixed number of used registers, all “extra” variables are placed in local memory. It can take about 600 times more to access data in local memory, in comparison to register access. The shared memory can be used as a support for storing variables. This is advantageous since the transfer time between registers and shared memory is about 4 clock cycles [2] for accesses without bank conflicts, which costs the same as a floating point “add” operation.

Therefore, the ray structure, represented by an origin point and a direction vector, is stored in shared memory. This way, ray information is used in every phase of the algorithm and the use of registers is significantly reduced. Another advantage in storing ray information in shared memory space is the ability of using memory indexing, which is necessary in ray-triangle intersection tests and kD-tree traversal. Since in compile time it is not known which axis will be used, it is necessary to represent the axes as a 3-value vector, whose indexes vary from 0 to 2. In case kernel code indexes a vector variable, it will be automatically stored in local memory in order to allow the indexing operation. Using shared memory we are able to overcome this problem, maintaining the indexing and performing faster memory operations.

CUDA constant memory space is used to store most kernel fixed parameters, such as virtual camera configurations and maximum Ray Tracing depth. CUDA compiler automatically stores parameters used by kernel functions in shared memory space. Because of the fact that this memory region is already being used for storing data regarding the rays, we decided to manually place these parameters in constant memory. Since constant memory access in CUDA is cached and consequently fast, it does not present significant performance losses in comparison to shared memory usage.

D. Ropes++

The new traversal approach used in our work, shown in Algorithm 1, reduces the number of global memory reads and arithmetical operations necessary to the leaf traversal. To accomplish that, the intersection algorithm is adapted in a way that the test is reduced to locate the AABB’s exit point by reading only half of the AABB information. This is possible based on the fact that the exit face of a ray-AABB intersection depends only on the ray’s direction, and that there are always three possible intersection faces, one for each ray axis direction. The reduction of memory reads is reached due to the substitution of the access to

six floats, representing the scene bounding box, by the three floating point accesses necessary to the ray-AABB intersection calculations.

Algorithm 1 Modified Traversal with ropes (ropes++)

```

1: Ray  $r = (\text{org}, \text{dir})$ ; {Origin Point + Direction Vector}
2: Node  $cNode = \text{root}$ ; {only the entry distance is necessary at this point}
3:  $tEntry = \text{Entry distance of } r \text{ in the tree}$ ;
4: Object  $intersected$ ; {possible intersected object}
5:  $tDistance$ ; {distance ray-object}
6: repeat
7:   Point  $p = r.\text{org} + tEntry * r.\text{dir}$ ;
8:   while  $\neg cNode.\text{isLeaf}()$  do
9:     if  $pEntry$  is on left of  $cNode.\text{splitPosition}$  then
10:       $cNode = cNode.\text{leftChild}$ ;
11:     else
12:       $cNode = cNode.\text{rightChild}$ ;
13:     end if
14:   end while
15:   {AABB represented by 6 floats (min,max 3D point)}
16:   {We only need to test against 3 parameterized values}
17:    $localFarX = (cNode.AABB[(r.\text{dir}.x \geq 0)*3] - r.\text{org}.x)/r.\text{dir}.x$ ;
18:    $localFarY = (cNode.AABB[(r.\text{dir}.y \geq 0)*3 + 1] - r.\text{org}.y)/r.\text{dir}.y$ ;
19:    $localFarZ = (cNode.AABB[(r.\text{dir}.z \geq 0)*3 + 2] - r.\text{org}.z)/r.\text{dir}.z$ ;
20:    $tExit = \min(localFarX, localFarY, localFarZ)$ ;
21:   if  $tExit \leq tEntry$  then
22:     return MISS;
23:   end if
24:   for all object  $O$  in  $cNode$  do
25:      $I = \text{Intersection}(r, O, tEntry, tExit)$ ;
26:     if  $I \neq \text{null}$  then
27:       Update  $tDistance$  and  $intersected$ , using current best result;
28:     end if
29:   end for
30:   if  $tDistance \geq tEntry \wedge tDistance \leq tExit$  then
31:     return HIT( $intersected, tDistance$ );
32:   end if
33:    $cNode = cNode.\text{ropes}[(r.\text{dir}[\text{minLocalFarAxis}] \geq 0)*3 + \text{minLocalFarAxis}]$ ;
34: until  $cNode == \text{null}$ 
35: return MISS;

```

In order to reduce the register pressure, our rope traversal also performs a simplification of the initial ray-scene intersection test, initially calculating the entry point distance. Using the fact that in case a ray does not intersect the scene AABB it will not intersect a sub-scene volume either, the ray-scene test can use as exit point distance the intersected

exit face of the first traversed leaf, computation that is always necessary for ray-scene hit cases. Although increasing the computation time when the ray misses the scene, this loss becomes small when considering the hit cases, because they have a higher probability of happening. Since it is not necessary anymore to the kernel to store the scene exit point distance, the register pressure is slightly reduced, enabling a higher number of threads effectively processed in parallel.

VI. COMPARATIVE ANALYSIS AND RESULTS

Despite each traversal algorithm follows a different search concept, it is possible to notice a certain similarity in the structures of all of them. Initially, before the search itself, all algorithms execute an intersection test between ray and scene bounding box. This operation is performed in such a way that, in case the ray does not hit the scene bounding box, there is no meaning to perform the kD-tree traversal. Another reason for this initial test is that most of traversal algorithms store both entry and end intersection points distances, for later use as traversal end condition.

Our standard traversal algorithm implementation stores in local memory its stack. Such memory, in spite of not being cached and having the same high latency that global memory accesses, guarantees an “automatic coalesced access” of local variables for threads located in the same half-warp, taking more benefit of CUDA’s SIMT architecture.

Table I
KD-TREE MEMORY USAGE VS. AVERAGE NUMBER OF VISITED NODES

	Memory usage			# of visited nodes		
	DRAGON	BUNNY	ALIEN	DRAGON	BUNNY	ALIEN
Standard	17.2 MB	7.7 MB	6.3 MB	38.4	18.3	13.1
Restart	15.3 MB	5.8 MB	4.4 MB	140.6	63.3	30.2
PushDown	15.3 MB	5.8 MB	4.4 MB	120.2	55.2	27
ShortStack	15.3 MB	5.8 MB	4.4 MB	80.3	43.3	24.3
PD & SS	15.3 MB	5.8 MB	4.4 MB	60.5	38.5	18.2
Ropes[4]	47.1 MB	19 MB	13 MB	38.4	18.6	13.1
Ropes++	47.1 MB	19 MB	13 MB	38.4	18.6	13.1




The kD-Restart algorithm does not need a stack, which makes the code simpler and with less variables reducing the number of registers per thread. However, when the search reaches a leaf and is restarted to the root, many nodes are revisited, resulting in a higher amount of global memory reads, as shown in Table I.

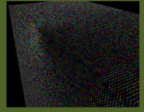


The kD-PushDown reduces the number of visited nodes when executing kD-Restart by adding control structures to the code in order to avoid returning the search to the tree’s root. This generates divergent cases and decreases performance, as shown in Table II.

Regarding kD-ShortStack, our implementation allocates the stack in shared memory, allowing a maximum size of 3 nodes, without the possibility of stack overflows, due to the circular nature of the structure. Accesses to this short stack are faster than the standard algorithm since it is located

Table II

PERFORMANCE COMPARISON FOR DIFFERENT SCENARIOS (AVERAGES FROM 3 FIXED VIEWPOINTS PER SCENE) ON 1376X768 RESOLUTION USING A 9800 GX2 ON SINGLE-GPU MODE

	DRAGON Ray Casting 2 Point Lights Phong Shading	BUNNY Ray Casting 2 Point Lights Phong Shading	ALIEN Ray Casting 2 Point Lights Phong Shading
			
	1.1 million triangles	69k triangles	32k triangles
Standard	45.52 ms	25.30 ms	22.41 ms
Restart	67.11 ms	35.76 ms	29.68 ms
PushDown	67.13 ms	35.93 ms	30.15 ms
ShortStack	59.25 ms	31.98 ms	26.87 ms
PD & SS	54.06 ms	29.76 ms	24.70 ms
Ropes[4]	-	78.74 ms(8800 GTX)	-
Ropes++	40.36 ms	22.15 ms	19.69 ms

	SPHERES 1 Ray Casting 1 Point Light Phong Shading	SPHERES 2 Ray Tracing Max Depth: 6 2 Point Lights Phong Shading	SPHERES 3 Ray Tracing Max Depth: 6 2 Point Lights +Shadows Phong Shading
			
	1.2 million spheres	5k spheres	1k spheres 1 plane
Standard	11.62 ms	107.92 ms	120.11 ms
Restart	17.13 ms	153.80 ms	172.90 ms
PushDown	17.54 ms	155.20 ms	176.32 ms
ShortStack	15.01 ms	124.15 ms	140.67 ms
PD & SS	13.89 ms	112.40 ms	128.92 ms
Ropes++	10.20 ms	62.80 ms	65.34 ms

in shared memory. This approach surpasses kD-Restart's performance as well. However, similar to the kD-Restart, the stack's limited size forces a return to the root and as a consequence new searches to already visited nodes are performed. This way, ShortStack performance is lower than the original traversal one.

The kD-PushDown and kD-ShortStack hybrid algorithm (PD & SS) [3] offers a significant gain of performance in comparison to the single kD-ShortStack, even with the addition of kD-PushDown divergent control structures and kD-ShortStack stack operations. This happens due to the fact that kD-PushDown helps limiting the stack use by transferring the restart event to a scene sub-tree, reducing the number of nodes to be stored, which is convenient because of the small stack size.

The rope traversal algorithm proposed by Popov et al. [4]

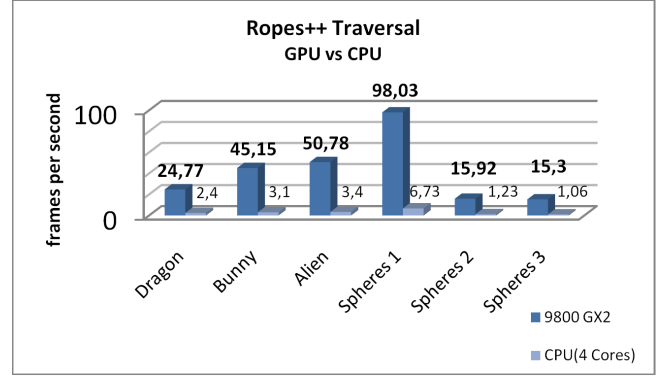


Figure 5. Ropes++ Traversal algorithm GPU vs. CPU performance comparison

for the CUDA architecture results in a lower number of visited nodes in comparison to all previously cited algorithms, excluding the standard traversal. In spite of increasing the amount of arithmetical operations, it is important to notice that, contrary to the other algorithms, the rope traversal does not need to perform costly division operations (about 20 clock cycles at least) for every node traversal. Instead of that, it makes use of 3 multiply-add operations, each one costing 4 clock cycles. However, an additional ray-AABB intersection test is necessary for every visited leaf to define the new rope to be used.

As shown in Table II, the Ropes++ traversal algorithm achieved the best performance results for all tested scenes. The reason for such performance enhancement is directly related to a lower number of visited nodes, which significantly reduces the amount of global memory accesses. A great and inherent advantage of an approach using ropes refers to shadow and reflective rays, that start their traversal in the last intersected leaves, which contain the origin of such rays. However, an algorithm using ropes demands about 3 times more memory space to store the scene, therefore being a prohibitive alternative for scenes with high complexity in low end CUDA enabled devices.

Surprisingly, the algorithm with the second best performance result was the one using standard traversal. Although allocating a stack in local memory, it has the advantage of showing a simple implementation, with few control structures and consequently presenting a low amount of divergences. All the previous developed algorithms based on the standard traversal possess a higher number of visited nodes per traversal, and therefore they showed an inferior performance. Such algorithms performed well on older GPU architectures with specific limitations, but they do not take fully benefit of current GPU technology, since most of these limitations do not exist anymore.

Finally, the GPU and CPU implementations of our Ropes++ Traversal algorithm were compared regarding performance. Figure 5 shows that our GPU implementation

reached speedup gains up to 15x when compared to the CPU one.

All implemented algorithms have been tested on an Intel Core2 Quad 2.66GHz CPU with 4GB of RAM and a NVIDIA GeForce 9800 GX2 graphics card, running Microsoft Windows XP 64-bit Service Pack 2.

VII. CONCLUSION AND FUTURE WORKS

This work presented a compilation of different techniques applied to the development of ray tracers, focusing real time rendering. A considerable number of kD-tree traversal approaches were compared regarding performance, in order to obtain an algorithm capable of offering the lowest execution time. The best achieved result was using our new traversal approach based on the technique proposed by Popov et al. [4], which uses optimized implementations of kD-trees with ropes.

CUDA architecture favors implementations that recently ran only efficiently on CPUs, due to their efficient data structures stored in primary memory. Standard traversal, which uses stacks, were difficult to implement efficiently using shaders. It must be highlighted that all techniques implemented using CUDA architecture are capable of rendering high definition images in real time in a scalable way, which was not the case in the majority of the original implementations discussed in this work.

As future work, we intend to support multi-GPU implementations in order to enhance even more the Ray Tracing performance, since recent graphics cards provide such functionality.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their comments and Jacco Bikker for sharing his kD-Tree construction code, previously presented on Arauna [16] ray tracer. Many thanks to Brian Budge at UC Davis, for sharing his knowledge of ray-traversal optimizations on CUDA.

REFERENCES

- [1] A. Glassner, Ed., *An Introduction to Ray Tracing*. Academic Press, 1989.
- [2] NVIDIA, *NVIDIA CUDA Programming Guide 2.1*, 2009. [Online]. Available: http://www.nvidia.com/object/cuda_develop.html
- [3] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree GPU raytracing," in *S3D*, B. Gooch and P.-P. J. Sloan, Eds. ACM, 2007, pp. 167–174. [Online]. Available: <http://doi.acm.org/10.1145/1230100.1230129>
- [4] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Stackless KD-tree traversal for high performance GPU ray tracing," *Comput. Graph. Forum*, vol. 26, no. 3, pp. 415–424, 2007. [Online]. Available: <http://dx.doi.org/10.1111/j.1467-8659.2007.01064.x>
- [5] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on GPU with BVH-based packet traversal," in *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, Sep. 2007, pp. 113–118.
- [6] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, pp. 343–349, 1980.
- [7] V. Havran, "Heuristic ray shooting algorithms," Ph.D. dissertation, Czech Technical University, Praha, Czech Republic, Apr. 2001, available from <http://www.cgg.cvut.cz/havran/phdthesis.html>.
- [8] C. Wächter and A. Keller, "Instant ray tracing: The bounding interval hierarchy," in *Eurographics Workshop/ Symposium on Rendering*, T. Akenine-Möller and W. Heidrich, Eds. Nicosia, Cyprus: Eurographics Association, 2006, pp. 139–149. [Online]. Available: <http://www.cg.org/EG/DL/WS/EGWR/EGSR06/139-149.pdf>
- [9] F. Jansen, "Data structures for ray tracing," in *Data Structures for Raster Graphics*, L. R. A. Kessener, F. J. Peters, and M. L. P. van Lierop, Eds. Springer-Verlag, 1986, pp. 57–73.
- [10] T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer," in *Graphics Hardware*, M. Meißner and B.-O. Schneider, Eds. Los Angeles, California: Eurographics Association, 2005, pp. 15–22. [Online]. Available: <http://www.cg.org/EG/DL/WS/EGGH/EGGH05/015-022.pdf>
- [11] V. Havran, J. Bittner, and J. Sára, "Ray tracing with rope trees," in *14th Spring Conference on Computer Graphics*, L. S. Kalos, Ed. Comenius University, Bratislava, Slovakia, Apr. 1998, pp. 130–140.
- [12] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive rendering with coherent ray tracing," *Comput. Graph. Forum*, vol. 20, no. 3, 2001.
- [13] M. Harris and M. Harris, "Parallel prefix sum (scan) with cuda," 2007. [Online]. Available: <http://beowulf.lcs.mit.edu/18.337/lectslides/scan.pdf>
- [14] I. Wald and V. Havran, "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*, 2006, pp. 61–69.
- [15] G. S. Brodal, R. Fagerberg, and R. Jacob, "Cache oblivious search trees via binary trees of small height," in *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 39–48.
- [16] J. Bikker, "Arauna real time ray tracing," 2009. [Online]. Available: <http://igad.nhtv.nl/~bikker/>