EUROGRAPHICS 2007 / D. Cohen-Or and P. Slavík
(Guest Editors)

Volume 26 (2007), Number 3

Stackless KD-Tree Traversal for High Performance GPU Ray Tracing

Stefan Popov[†] Johannes Günther[‡] Hans-Peter Seidel[‡] Philipp Slusallek[†]

[†]Saarland University, Saarbrücken, Germany
[‡]MPI Informatik, Saarbrücken, Germany

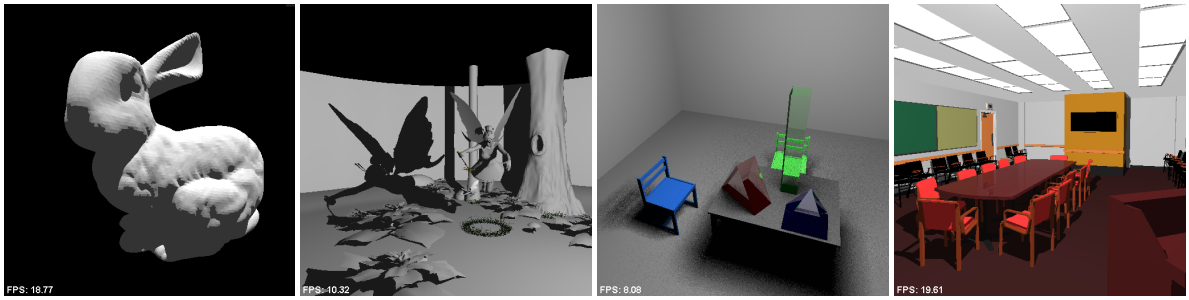

Figure 1: *Our test scenes, from left to right:* BUNNY, FAIRYFOREST, GLASSSHIRLEY6, *and* CONFERENCE. *Our novel GPU ray tracer can render them at 18.7, 10.3, 8, and 19.6 fps, respectively, at a resolution of 512 × 512. We support ray traced shadows from point lights and soft shadows from area light sources (9 samples / pixel), reflections, and refractions.*

Abstract

Significant advances have been achieved for realtime ray tracing recently, but realtime performance for complex scenes still requires large computational resources not yet available from the CPUs in standard PCs. Incidentally, most of these PCs also contain modern GPUs that do offer much larger raw compute power. However, limitations in the programming and memory model have so far kept the performance of GPU ray tracers well below that of their CPU counterparts.

In this paper we present a novel packet ray traversal implementation that completely eliminates the need for maintaining a stack during kd-tree traversal and that reduces the number of traversal steps per ray. While CPUs benefit moderately from the stackless approach, it improves GPU performance significantly. We achieve a peak performance of over 16 million rays per second for reasonably complex scenes, including complex shading and secondary rays. Several examples show that with this new technique GPUs can actually outperform equivalent CPU based ray tracers.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Ray tracing I.3.6 [Computer Graphics]: Graphics data structures and data types

1. Introduction

Ray tracing is well known for its advantages in high quality image generation and lighting simulation tasks. The recent development of highly optimized realtime ray tracing algorithms makes this technique an interesting alternative to rasterization. While CPU performance has increased dramatically over the last few years, it is still insufficient for many ray tracing applications. Existing deployments in industry, such as several large visualization centers, are typically based on CPU clusters to achieve the necessary performance.

† {popov,slusallek}@cs.uni-sb.de
‡ {guenther,hpseidel}@mpi-inf.mpg.de

Increasingly, realtime ray tracing is being discussed also for gaming platforms, e.g. [FGD*06]. These systems already have one or more modern GPUs that offer significantly higher raw compute power than CPUs due to their highly parallel architecture. This also makes them a very attractive compute platform for ray tracing.

However, today's efficient ray tracers are based on hierarchical acceleration structures (typically kd-trees) and corresponding stack-based traversal algorithms. Unfortunately, even the latest GPU architectures are poorly suited for implementing such algorithms.

To remove these limitations we present three main contributions in this paper: (1) We review and adapt an efficient, stackless ray traversal algorithm for kd-trees [HBŽ98]. (2) Based on this algorithm we present a *novel, stackless packet traversal algorithm* that supports arbitrary ray bundles and can handle complex ray configurations efficiently. (3) We present a *GPU implementation* of these algorithms that achieves higher performance than comparable CPU ray tracers. The GPU implementation uses the Compute Unified Device Architecture (CUDA) framework [NVI] to compile and run directly on the latest generation of GPUs.

2. Previous Work

Currently, the best known acceleration structure for ray tracing of static scenes remains the kd-tree [Hav01] built according to the surface area heuristic (SAH) [MB89]. In practice the recursive ray segment traversal algorithm [HKBŽ97] is widely used due to its high efficiency. In order to increase memory coherence and save per-ray computations, this algorithm has been extended to also support tracing of entire ray packets using the SIMD features of modern CPUs [Wal04] or even to larger groups of rays with the help of frustum traversal [RSH05, Ben06]. These algorithms also form the basis for custom ray tracing chips, like the RPU [WSS05] and its dynamic scene variant D-RPU [WMS06].

2.1. Ray Tracing on GPUs

The first step toward GPU ray tracing was made in 2002 with the Ray Engine [CHH02]. It implemented only the ray-triangle intersection on the GPU while streaming geometry from the CPU. This division of labor resulted in high communication costs, which greatly limited performance.

In the same year Purcell et al. [PBMH02] showed through hardware simulation that it is possible to overcome this limitation by moving essentially all computations of ray tracing onto the GPU. The GPU was treated as a stream processor and each of the different tasks – primary ray generation, acceleration structure traversal, triangle intersection, shading, and secondary ray generation – were implemented as separate streaming kernels. Due to the difficulty of implementing efficient kd-trees they chose a simple regular grid as their acceleration structure.

Later a concrete implementation on a GPU was able to achieve a performance of roughly 125k rays/s for non-trivial scenes [Pur04]. Its main bottlenecks were the sub-optimal acceleration structure as well as the high bandwidth requirements. This basic approach to ray tracing on the GPU was the base for several other implementations, including [Chr05, Kar04].

Carr et al. implemented a limited ray tracer on the GPU that was based on geometry images [CHCH06]. It can only support a single triangle mesh without sharp edges, which is difficult to create from typical models. The acceleration structure they used was a predefined bounding volume hierarchy. Due to the limited model support comparing performance is difficult, but for reasonable model sizes it is not much higher than the above approaches.

The high computation power of the GPU was also utilized to implement ray casting of piecewise quadratic surfaces [SGS06] and NURBS [PSS*06]. However, these papers do not use an acceleration structure.

2.2. KD-Tree Traversal on the GPU

For static scenes, kd-trees are regarded as the most efficient acceleration structure. Therefore, several attempts exist to implement kd-trees on the GPU. In 2004 Ernst et al. [EVG04] showed an implementation of a (parallel) stack for kd-tree traversal on the GPU using several kernels executed in a multi-pass fashion. Frequent kernel switches introduced a high overhead and huge memory bandwidth requirements for storing intermediate results. Thus, the resulting frame rates were much too low for interactive ray tracing even for small scenes. Additionally, the parallel stack consumed large amounts of memory.

One year later, Foley and Sugerman [FS05] presented two implementations of stackless kd-tree traversal algorithms for the GPU, namely *kd-restart* [Kap85] and *kd-backtrack*. Both algorithms clearly outperformed regular grids on the GPU. However, despite the high GPU compute power and despite the efficient acceleration structure, they were still not able to outperform the CPU implementations, achieving a peak performance of around 300k rays/s for reasonably complex scenes. One reason for the relative low performance is the high number of redundant traversal steps.

Concurrently to our work, Horn et al. [HSHH07] have independently developed an interactive GPU ray tracer achieving similar high performance with 15–18M rays/s. By adding a short stack to the kd-restart algorithm they avoid some but not all of the redundant traversal steps.

For bounding volume hierarchies Thrane and Simonsen [TS05] showed that stackless traversal allows for efficient GPU implementations. They outperformed both regular grids and the kd-restart and kd-backtrack variants for kd-trees. However, performance was still fairly limited, to a large degree due to bandwidth limitations of the algorithm.

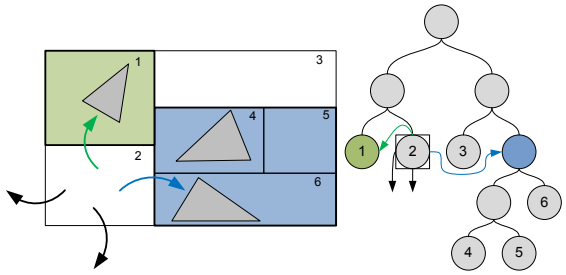

Figure 2: *A kd-tree with "ropes". Ropes link each leaf node of the kd-tree via its six faces directly to the corresponding adjacent node of that face or to the smallest node enclosing all adjacent nodes if there are multiple.*

3. Efficient Stackless KD-Tree Traversal

As demonstrated above, implementing an efficient ray tracer on the GPU that takes full advantage of its raw processing power is challenging. In particular, an efficient implementation needs to be based on a *stackless design* to avoid the issues discussed above. While latest GPUs would allow for a stack to be implemented in a fast but small on-chip memory (a.k.a. shared memory on the NVIDIA G80 architecture), the memory requirements of such an implementation would most likely prohibit good parallelism. A viable stackless algorithm should outperform existing algorithms, and should be simple and small enough to comfortably be implemented in a single GPU kernel in order to avoid bandwidth and switching overhead of multi-pass implementations. Additionally, register usage should be minimized such that optimal parallelism can be achieved on the latest GPUs.

We base our approach of high performance GPU ray tracing on seemingly little noticed previous publications on stackless traversal of spatial subdivision trees, which use the concept of neighbor cell links [Sam84, Sam89, MB89], or *ropes* [HBŽ98]. In the following, we first discuss the kd-tree with ropes as the basis for a single ray traversal algorithm. Later, we present our new extension that efficiently supports the stackless traversal of kd-trees with packets of rays.

3.1. Single Ray Stackless KD-Tree Traversal

The main goal of any traversal algorithm is the efficient front-to-back enumeration of all leaf nodes pierced by a ray. From that point of view, any traversal of inner nodes of the tree (also called "down traversal") can be considered overhead that is only necessary to locate leafs quickly.

Kd-trees with ropes augment the leaf nodes with links, such that a direct traversal to adjacent nodes is possible: For each face of a leaf cell they store a pointer to the adjacent leaf, or, in case there is more than one adjacent leaf overlapping that face, to the smallest node containing all adjacent

Algorithm 1: *Single Ray Stackless KD-Tree Traversal*

1: $R = (O, D)$ ▷ The ray
2: $N \leftarrow$ the root node ▷ $N \equiv$ current traversed node
3: $\lambda_{entry} \leftarrow$ Entry distance of R in the tree
4: $\lambda_{exit} \leftarrow$ Exit distance of R in the tree

5: **while** $\lambda_{entry} < \lambda_{exit}$ **do**
6: ▷ Down traversal
7: $P_{entry} \leftarrow O + \lambda_{entry} \cdot D$
8: **while** $\neg$ *is-leaf*(N) **do**
9: $N \leftarrow \begin{cases} N_{\text{left-child}} & \text{, if } P_{entry} \text{ on left of split} \\ N_{\text{right-child}} & \text{, else} \end{cases}$
10: **end while**

11: ▷ At a leaf
12: ▷ Check for intersection with contained triangles
13: **for all** $T \in N_{triangles}$ **do**
14: $I =$ *intersect-in-range*$(R, T, \lambda_{entry}, \lambda_{exit})$
15: **if** *intersection-found*(I) **then**
16: Update the current best result
17: $\lambda_{exit} \leftarrow$ *intersection-distance*(I)
18: **end if**
19: **end for**
20: ▷ Exit the leaf.
21: $\lambda_{entry} \leftarrow$ *exit-distance*$(R, AABB(N))$
22: $N \leftarrow$ *exit-rope*(R, N)
23: **return** *no-intersection*, **if** $N \equiv$ **nil**
24: **end while**

25: **return** Best found result

leafs (see Figure 2). Faces that do not have adjacent nodes lie on the border of the scene and point to a special **nil** node.

Single ray traversal of rope trees works as follows (see Algorithm 1). Suppose the traversal currently processes a leaf node. If the ray does not intersect anything in this leaf the algorithm determines the face and the intersection point through which the ray exits the node. Then traversal continues by following the "rope" of this face to the adjacent node. If that node is not itself a leaf node, the algorithm performs a down traversal to locate the leaf node that contains the exit point (which is now the entry point of the new leaf node). If the rope points to the **nil** node, the ray is terminated.

Adding ropes to any kd-tree is fairly simple and can be done in a post processing step (Algorithm 2). First, **nil** ropes are created for all faces of the root node. During kd-tree construction, the ropes of a node are copied to the respective outer faces of its children and the children are then connected to each other as there is a 1-to-1 adjacency across the shared face. Every time the post processing algorithm visits a node, it tries to push it's ropes down the tree as much as possible: Assume we are at a node N with a rope pointing to node R for some side s of N. R is replaced with it's right child R_R if

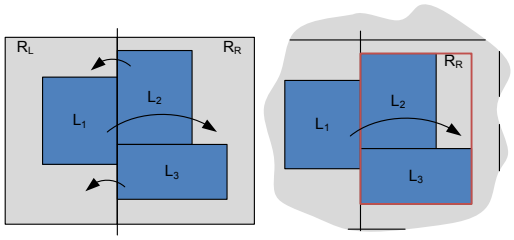

Figure 3: *Left: Un-optimized ropes. The rope for the right face of L_1 links to the root of the subtree adjacent to L_1 from the left. Right: Optimized ropes. The right face points to the deepest node that contains all leafs adjacent to L_1 on the right.*

the split axis of R is parallel to s and $s \in \{left,\ top,\ front\}$, or if the split plane of R is above, to the left or in front of the bounding box of N. Symmetrical rules apply for R and R_L. We refer to that last part of the algorithm as rope optimization. It can be proven that the presented algorithm runs in $O(N)$, with N being the number of nodes in the tree.

Note that kd-tree traversal would still work correctly if the ropes are not pushed down upon visiting a node. The traversal steps in this case would actually match the ones taken in a recursive segment traversal. Also, the resulting ropes in this case point to the root of the neighboring subtrees – an important property used by the packet traversal algorithm as explained below. Thus, for packet traversal, we do not push the nodes down during construction. We will refer to ropes constructed in this manner as "non-optimized".

3.2. Stackless Traversal for SIMD Packets of Rays

The SIMD nature of GPUs suggested the design of a stackless packet traversal algorithm. This new algorithm corresponds to the stack-based packet algorithm used for CPUs by Wald et al. [WSBW01] and later extended by Reshetov [Res06] for incoherent rays. The use of packets reduces off-chip bandwidth, avoids memory fetch latencies, and eliminates incoherent branches on the GPU [Hou06] by exploiting ray coherence whenever present.

The packet traversal algorithm is an extension of the single ray variant. It operates on one node at a time and only processes rays of the packet that intersect the current node. For efficiency reasons, it requires non-optimized ropes.

In a recursive traversal some rays in a packet might become inactive because they do not overlap with a child node. Such rays are again activated when the recursion returns to the parent node. To make stackless traversal of packets efficient, we apply a similar regrouping mechanism by using non-optimized ropes. For a node N, all leafs in its left child N_L adjacent to the split plane of N will link to N_R (the right child). Thus, all rays exiting leafs in N_L through the split plane of N will be collected again in N_R

In order to implement the corresponding traversal algorithm, each ray of a packet maintains a separate state. For a ray R, this state consists of the currently traversed node $N_{current}$ and the entry point P_{entry} of R into $N_{current}$. The packet algorithm operates on one node $N_{traversed}$ at a time by processing all rays of the packet against it. However, only rays that are currently in $N_{traversed}$ (i.e. where $N_{traversed} \equiv N_{current}$) participate in the computations. We name such rays *active rays*.

A *down traversal step* proceeds as follows: First, the current node of all active rays is advanced to either the left (N_L)

Algorithm 2: *Rope Construction*

```
 1: procedure OPTIMIZE(R, S, AABB)
                          ▷ R ≡ the rope, passed by reference
 2:     while R is not leaf do
 3:         R ← R_L or R_R based on S,  if split-axis(R) ∥ S
 4:         R ← R_R  if split-plane(R) above AABB_min
 5:         R ← R_L  if split-plane(R) below AABB_max
 6:         brake,  else
 7:     end while
 8: end procedure

 9: procedure PROCESSNODE(N, RS, AABB)
                          ▷ N ≡ current node, RS ≡ ropes of N
10:     if is-leaf(N) then
11:         N_ropes ← RS
12:         N_bounding-box ← AABB
13:     else
14:         if single ray case then
15:             S ← {left, right, top, bottom, front, back}
16:             OPTIMIZE(RS[s], s, AABB), ∀s ∈ S
17:         end if
18:         (S_L, S_R) ← { (left, right)   , if N_split-axis = X
                         { (front, back)   , if N_split-axis = Y
                         { (top, bottom)   , if N_split-axis = Z
19:         V ← N_split-plane-position

20:         RS_left ← RS,  RS_left[S_R] ← N_R
21:         AABB_left ← AABB,  AABB_left[S_R] ← V
22:         PROCESSNODE(N_L, RS_left, AABB_left)

23:         RS_right ← RS,  RS_right[S_L] ← N_L
24:         AABB_right ← AABB,  AABB_right[S_L] ← V
25:         PROCESSNODE(N_R, RS_right, AABB_right)
26:     end if
27: end procedure
28:
29: PROCESSNODE(root-node, {nil, ...nil}, AABB)
                                  (6)
```

Algorithm 3: *PRAM Stackless Packet Traversal of KD-Trees*

```
1:  function FINDINTERSECTION(ray, tree)
2:      return No intersection, if ray ∩ AABB(tree) = ∅
3:      (λ_entry, λ_exit) ← ray ∩ AABB(tree)
4:      P_entry ↔ λ_entry
5:      N_current ← Root(tree)
6:      loop
7:          N_current ← nil, if λ_entry ≥ λ_exit
8:          N_traversed : shared ← arg P_MAX(&N_current)
9:          break, if N_traversed ≡ nil

10:         loop                                  ▷ The down traversal
11:             break, if IsLeaf(N_traversed)
12:             if N_current = N_traversed then
13:                 (N_L, N_R) ← Children(N_traversed)
14:                 on-left ← P_entry is left of split
15:                 N_current ← { N_L , if on-left
                                { N_R , else
16:             end if
17:             axis ← SplitAxis(N_current)
18:             b ← P_entry on left of split
19:             b1 ← P_OR (active ∧ b ∧ ray_d[axis] > 0)
20:             b2 ← ¬P_OR (active ∧ ¬b)
21:             b3 ← P_SUM (P_entry on left ? −1 : 1) < 0
22:             N_traversed ← { N_L , if b1 ∨ b2 ∨ b3
                              { N_R , else
23:         end loop
                                              ▷ The exit step
24:         Intersect ray with contained geometry
25:         if N_current = N_traversed then
26:             Update P_entry and N_current as with single ray
27:         end if
28:     end loop
29:     return Best found intersection
30: end function
```

or right (N_R) child of $N_{traversed}$, depending on whether its entry point is to the left or right of the split plane respectively. Next, $N_{traversed}$ is advanced to either N_L or N_R, according to the following rules (see also Figure 4):

- If the entry point of all active rays lies on the same side of the split plane, we choose that node.
- If the directions of all active rays that have their entry point in child node A point toward the other child B with respect to the splitting dimension, we choose A and vice versa.
- Otherwise, we choose the node containing more entry points.

In case of the first two rules we are handling a coherent set of active rays and can guarantee optimal traversal (for a proof see Appendix A). If a coherent packet is split at some point, it will join again later as shown in Figure 4. In case

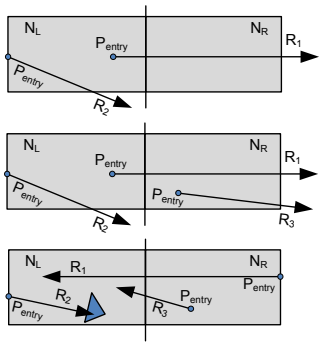

Figure 4: *Top: All rays have an entry point in the left child, so we take it as the next node in the down traversal. Middle: All rays with entry points on the left have positive direction along the split axis. We descend into the left child, so rays can rejoin in the right. Bottom: It is unclear which node should be processed first, so we take the one that contains more ray entry points. Thus, we increase the chance that we do not have to revisit it later.*

of the third rule we have to handle an incoherent case. We choose the larger packet first, hoping that the other group of rays terminates before we have to traverse the initially chosen node again.

Down traversal stops once we reach a leaf, where we intersect all active rays with its geometry. $N_{current}$ of any active ray that terminates in this leaf is set to **nil**. Unless all rays have terminated, we can never traverse to **nil** with the packet (see below). Thus, terminated rays can never become active again. As in the single ray case, we determine the exit point and exit node for each active ray by intersecting it with the axis-aligned bounding box of the leaf and by following the rope of the exit face. This defines the new entry points P_{entry} and new current nodes $N_{current}$ for all active rays.

We now have to perform a *horizontal (or up) traversal step*. In general, the active rays will not all leave through the same face and we need to choose the next node to be processed $N_{traversed}$. Obviously, we need to choose from the set S of current nodes of all non-terminated rays ($N_{current} \neq$ **nil**). If $S = \emptyset$, we can immediately terminate the traversal for the whole packet.

Otherwise, we can choose any node from S different from **nil** and still obtain correct traversal behavior. To choose the optimal node, we rely on a property of the construction algorithm: the tree is constructed in depth first order and the two children of a node are stored sequentially in memory. Thus, nodes deeper in the tree are at higher memory addresses than their parents. Choosing the node corresponding to the largest memory address guarantees us (see Appendix A) that once we enter a node N with a set of active rays A, we will only process nodes from the subtree of N, until all rays in A exit N. As a consequence, coherent rays will be rejoined the same way as in recursive traversal.

Having chosen the next $N_{traversed}$, we proceed with down traversal again unless the node already is a leaf node. To simplify the termination criteria, we give **nil** an address in memory that is smaller than the address of the root of the tree and thus we can skip the check of whether S is empty. A packet terminates exactly when it traverses to the special node **nil**.

The algorithm was designed for the latest GPU architecture and we use the PRAM programming model for a Concurrent Read Concurrent Write (CRCW) machine [FW78] to describe it (see Algorithm 3). As we will see (Section 4), this model is very close to the actual hardware implementation of the latest GPUs. In the algorithm, we make use of standard PRAM reduction techniques (Algorithm 4) to perform parallel OR, parallel SUM, and parallel MAX. The parallel OR returns the disjunction of a given condition over all processors of the PRAM machine and runs in $O(1)$. The other two reduction operations return the sum respectively the maximum of a given value over the processors and run in $O(\log P)$, with P being the number of processors.

Algorithm 4: *Standard PRAM reduction algorithms*

```
1:  function P_OR(condition)
2:      sharedCondition : shared ← false
3:      sharedCondition ← true, if condition
4:      return sharedCondition
5:  end function

6:  function P_REDUCE(value, op)
                                  ▷ op ≡ the reduction operator
7:      m[] : shared              ▷ Shared memory for reduction
8:      m[processorID] ← value
9:      for i = 0 .. log2(#processors) − 1 do
10:         a1 ← 2^(i+1) processorID , a2 ← a1 + 2^i
11:         m[a1] = op(m[a1], m[a2]) , if a2 < #processors
12:     end for
13:     return m[0]
14: end function
15: P_MAX(value) ≡ P_REDUCE(value, max)
16: P_SUM(value) ≡ P_REDUCE(value, +)
```

4. GPU Implementation

We implemented two variants of the ray tracing algorithm: One based on single ray kd-tree traversal and one based on packet traversal. Both variants were implemented on top of CUDA [NVI] and as a proof of concept we implemented the single ray variant on top of DirectX 9. The CUDA implementations have the entire ray tracing routine in a single kernel, whereas the kernels of the DX9 implementation are ray generation, ray-scene intersection, and shading. The implementation of the single ray traversal follows Algorithm 1 literally. Before discussing the implementation of the packet traversal algorithm, we will have a closer look at the hardware architecture of the GPU used for the implementation – an NVIDIA GeForce 8800 GTX (G80).

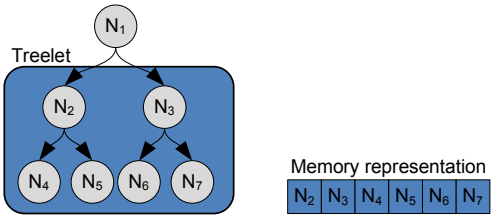

Figure 5: *Organization of kd-tree nodes into tree-lets in memory allows for higher memory coherence.*

The G80 is a highly parallel processor working on many threads simultaneously. The threads are scalar programs and the GPU processes them in SIMD groups – a.k.a. chunks or warps. It possesses multiple independent cores, each of which can process a single chunk at a given moment of time. Once a chunk is assigned to a core, it stays there until it terminates. Each core runs a number of chunks in a multi-threaded fashion and switches among them to hide various types of latencies. Additionally each core has a small amount of on-chip memory that is shared between the threads it runs and that can be used for inter-thread communication. Its size is small and the number of chunks that can be run on a single core in a multi-threaded manner is limited by the amount of shared memory each chunk uses. Thus, implementing a per-ray stack for kd-tree traversal using the shared memory would still be infeasible.

Because the threads are executed in SIMD manner and are thus implicitly synchronized within a chunk, one can look at each core of the GPU, together with its shared memory, as a PRAM CRCW machine, given that the threads make uniform branch decisions. Thus, we can directly implement Algorithm 3 on the GPU.

Since cache support was not exposed in CUDA at the time of writing, we implemented read-ahead to shared memory to reduce the number of round trips to off-chip memory in the packet traversal. A block of data is read simultaneously by all threads of a chunk, by reading consecutive addresses in consecutive threads (base address + thread ID in chunk). In this case, the memory controller can make a single large request to off-chip memory, whose latency is hidden by the GPU with calculations from other threads. We always read blocks of size equal to the chunk size of the GPU.

We also reorganize the storage of the tree to benefit further from the read-ahead. First, we store the geometry data in a leaf together with its AABB and its ropes, to increase the chance of having the data in shared memory at the beginning of a leaf exit step. Second, we store the non-leaf nodes in tree-lets similar to [Hav97]. A tree-let is a subtree of fixed depth with nodes stored together in memory (see Figure 5).

		kd-tree properties					
scene	#tris	#leaves	#empty leaves	references	size	size with ropes	rope overhead
SHIRLEY6	804	3,923	1,021	1.86	82.4 kB	266.3 kB	3.23
BUNNY	69,451	349,833	183,853	2.53	6.9 MB	23.0 MB	3.30
FAIRYFOREST	174,117	721,083	382,345	3.01	14.9 MB	47.9 MB	3.22
CONFERENCE	282,641	1,249,748	515,970	3.13	27.8 MB	85.0 MB	3.06

Table 1: *Used test scenes together with statistical data of the SAH kd-tree ("references" means the average number of references to triangles per non-empty leaf). Enriching the kd-tree with ropes for stackless traversal increases its size about 3 times.*

algorithm	ray segment		kd-restart		single ray stackless				packet stackless	
scene	down	pops	down	restarts	down	down, ENS	down, opt	exits	down	exits
SHIRLEY6	17.5	3.64	30.9	3.64	17.5	11.5	6.23	3.64	12.0	3.64
BUNNY	24.3	4.82	81.0	4.94	24.5	24.5	20.9	4.94	31.0	7.57
FAIRYFOREST	38.9	7.97	105	7.97	39.0	33.0	25.3	7.97	39.9	10.6
CONFERENCE	33.2	6.64	82.9	6.64	33.2	22.2	13.0	6.64	25.6	7.71

Table 2: *Number of steps for the different kd-tree traversal algorithms: "down" traversal steps and "pops"/"restarts"/"exits" denoting the number of up traversal steps. "ENS" stands for entry node search and "opt" – for rope optimization. All numbers are given averaged per ray.*

Thus, because of the read-ahead optimization, accessing the root of a tree-let in a down traversal, will also bring in the nodes that will be accessed in the next few down traversal steps, saving bandwidth and round-trips to off-chip memory. Even though we change the memory layout of the tree, the proofs in Appendix A still hold, because the root of every sub-tree as well as its sibling still have a smaller address in memory than any nodes in their subtrees.

An important feature of the implementation of the traversal algorithm is that it is transparent to the shader. The shader is written for single ray traversal and the traversal invocation is the same for both packets of rays and single rays. Packets are then formed implicitly by the graphics hardware.

5. Results and Discussion

To evaluate the proposed stackless traversal algorithms we implemented them not only on the GPU but also on the CPU as a reference. For testing purposes we used an AMD 2.6 GHz Opteron workstation and another workstation equipped with a NVIDIA GeForce 8800 GTX graphics card. We tested our implementations using a variety of scenes, ranging from simple to reasonably complex, namely the original SHIRLEY6, BUNNY, FAIRYFOREST, and CONFERENCE. The scenes and the viewpoints for the tests can be seen on Figure 1. More statistical data for the scenes is available in Table 1.

5.1. Memory Requirements

The main disadvantage of stackless traversal seems to be the increased storage requirements for the ropes and the bounding boxes of the leafs.

Assuming a compact representation with 8 bytes per node [Wal04], the kd-tree with ropes can not be more than a factor of 4 larger. To show this, we take the ratio of the size S_{normal} of a kd-tree without ropes to the size S_{ropes} of a kd tree with ropes:

$$1 \leq \frac{S_{ropes}}{S_{normal}} = \frac{48N + 8(2N-1) + 4\sum r_i}{8(2N-1) + 4\sum r_i} < 4 \quad \text{for } \sum r_i > 2$$

N is the number of leafs and r_i is the number of triangles referenced by leaf i. We assume that we need 4 bytes per reference. The first term in S_{ropes} is the overhead of the rope storage.

In practice we encountered a ratio of about 3 (see Table 1). Although this factor seems high in relation to the kd-tree *alone*, this disregards all the other data, such as precomputed data for fast triangle intersections, vertex attributes such as normals, texture coordinates, etc., the textures themselves, and any other scene data. Thus, the memory overhead of storing the ropes is often reasonable in comparison to the overall memory requirements.

5.2. Traversal Steps

Single ray stackless traversal has an important advantage: No stack is required to remember nodes that still need to be visited. Instead the state of the ray only consists of its current node and its entry point. Thus, the traversal can start at any node containing the origin of the ray. More important, it can start directly at a leaf.

For single rays with common origin inside the tree (as in the case of a pinhole camera), we can drastically reduce the number of down traversal steps. Instead of traversing from the root down to a leaf for every ray, we can directly start at

scene	OpenRT primary rays frustum	CPU: Stackless primary rays only		GPU: Stackless primary rays only		GPU: Stackless with 2ndary rays	
		single	packet	single	packet	single	packet
SHIRLEY6	6.6	3.80	3.49	10.6	36.0	4.8	12.7
BUNNY	—	2.16	1.71	8.9	12.7	4.9	5.9
FAIRYFOREST	3.6	1.57	1.27	5.0	10.6	2.5	4.0
CONFERENCE	3.9	2.14	1.78	6.1	16.7	2.7	6.7

Table 3: *Absolute performance for single ray and packet stackless traversal, implemented on the CPU (Opteron@2.6 GHz,* 4×4 *rays/packet) and the GPU (GeForce 8800 GTX,* 8×4 *rays/packet). Performance is given in frames per second at* 1024×1024*, including shading. For secondary rays we use one point light source for all scenes. The* CONFERENCE *was ray traced with a reflective table, taking approximately* $^1/_6$ *of the screen. For comparison we also list the OpenRT performance data with* 4×4 *rays/packet and frustum culling from [WBS07].*

the leaf that contains the origin. Combined with the deeper entry nodes after a leaf exit (optimized ropes), the stackless traversal algorithm saves up to $^2/_3$ of the down traversal steps in practice, compared to its recursive counterpart (see Table 2). Furthermore, we can apply a similar trick for secondary rays – their starting node is simply the leaf where the previous traversal terminated. Thus, we can start traversal for secondary rays directly at a leaf as well.

Similar approaches have also been taken in [RSH05] and later in [Ben06, WIK*06, WBS07], by tracing frustums of rays together and reducing the per ray cost by taking decisions for the whole frustum. In particular, the *entry point search* of [RSH05] is close to the above optimization, however it does not start at a leaf in the general case and it amortizes the down traversal cost over a smaller number of rays. Furthermore, most frustum methods work well for primary and coherent rays only, whereas the stackless traversal works quite well for most types of secondary rays.

For completeness, we also compare our algorithm to the kd-restart algorithm [FS05]. Our experiments show (Table 2) that the single ray traversal with entry node optimizations saves up to $^5/_6$ of all down traversal steps compared to kd-restart. Furthermore, the down traversal of kd-restart/kd-backtrack is more expensive as both algorithms need to intersect the ray with the split plane, in contrast to a simple point location query, performed by our stackless traversal algorithm.

5.2.1. Packet Traversal

Compared to single ray traversal, packets perform more traversal steps (Table 2). On the other hand, packet traversal is characterized by very coherent memory access and by coherent branch decisions, thus outperforming single rays on the GPU for most scenes (Table 3). Its main disadvantage when implemented on a GPU becomes the large packet size, dictated by the chunk size of the GPU.

On the CPU, packet traversal is slower than single ray traversal. One explanation is the relatively large size of the CPU cache and the implicit read-ahead. Thus, a coherent memory access pattern is not as important on the GPU, as long as close rays traverse close nodes. Also, using SIMD for the packets cannot improve performance a lot, since the single ray implementation already uses SIMD instructions where appropriate. Thus, the introduced overhead of a packet becomes an issue.

5.3. Absolute Performance

The absolute performance of our GPU ray tracer in frames per second is summarized in Table 3. We achieve a peak performance of almost 17M rays/s for the non-trivial CONFERENCE scene. If we compare the same stackless traversal algorithms on the CPU and the GPU we clearly show that the GPU would outperform even a four core CPU.

CPU based ray tracers can achieve higher performance by using more advanced traversal methods, most notably frustum culling techniques [RSH05, Ben06, WIK*06, WBS07]. However, these frustum methods are usually not very flexible, e.g. all rays of a packet have to share a common origin or their directions need to have the same sign. Furthermore, even if secondary rays are supported, tracing them with frustums will be much slower than for primary rays [BEL*07].

The GPU ray tracer of [HSHH07] achieves 15.2M primary rays/s in the CONFERENCE scene. Using the same view our GPU implementation traces 22M rays/s. However, note that the difference in speed might be due to the newer hardware we have used.

6. Conclusions and Future Work

In this paper we showed that a stackless traversal algorithm for kd-trees with ropes has several advantages compared to other traversal algorithms: It is simple and reduces the number traversal steps. But more importantly, avoiding the traversal stack leads to a very GPU-friendly implementation. Furthermore, we presented a novel, stackless packet traversal algorithm that boosts the GPU ray tracing performance to a level where the GPU can actually outperform CPU-based ray tracers.

One interesting direction for future work is the development of a stackless traversal algorithm that exploits ideas of frustum test and interval arithmetic to amortize traversal decisions over many rays and thus to further improve performance.

Although we believe a performance of over 16M rays/s to be already quite impressive for the CONFERENCE scene, we expected much higher performance: The G80 with its 128 scalar arithmetic units running at 1.3 GHz should deliver over 160 GFlops, meaning that tracing one ray costs about 10,000 cycles. We suspect the main bottleneck to be the large number of registers in the compiled code, which limits the occupancy of the GPU to less than 33%. Unfortunately, although the program requires much less registers, the CUDA compiler is not yet mature enough and cannot aid in reducing their count. An option would be to rewrite the whole CUDA code in PTX intermediate assembly.

References

[BEL*07] BOULOS S., EDWARDS D., LACEWELL J. D., KNISS J., KAUTZ J., SHIRLEY P., WALD I.: Packet-based whitted and distribution ray tracing. In *Proceedings of Graphics Interface 2007* (May 2007). 8

[Ben06] BENTHIN C.: *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, 2006. 2, 8

[CHCH06] CARR N. A., HOBEROCK J., CRANE K., HART J. C.: Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface* (2006), A.K. Peters. 2

[CHH02] CARR N. A., HALL J. D., HART J. C.: The ray engine. In *Proceedings of Graphics Hardware* (2002), Eurographics Association, pp. 37–46. 2

[Chr05] CHRISTEN M.: *Ray Tracing auf GPU*. Master's thesis, Fachhochschule beider Basel, 2005. 2

[EVG04] ERNST M., VOGELGSANG C., GREINER G.: Stack implementation on programmable graphics hardware. In *Proceedings of the Vision, Modeling, and Visualization Conference 2004 (VMV 2004)* (2004), Girod B., Magnor M. A., Seidel H.-P., (Eds.), Aka GmbH, pp. 255–262. 2

[FGD*06] FRIEDRICH H., GÜNTHER J., DIETRICH A., SCHERBAUM M., SEIDEL H.-P., SLUSALLEK P.: Exploring the use of ray tracing for future games. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames* (2006), ACM Press, pp. 41–50. 2

[FS05] FOLEY T., SUGERMAN J.: KD-tree acceleration structures for a GPU raytracer. In *HWWS '05 Proceedings* (2005), ACM Press, pp. 15–22. 2, 8

[FW78] FORTUNE S., WYLLIE J.: Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing* (1978), ACM Press, pp. 114–118. 6

[Hav97] HAVRAN V.: Cache sensitive representation for the BSP tree. In *Compugraphics'97* (Dec. 1997), GRASP – Graphics Science Promotions & Publications, pp. 369–376. 6

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001. 2

[HBŽ98] HAVRAN V., BITTNER J., ŽÁRA J.: Ray tracing with rope trees. In *14th Spring Conference on Computer Graphics* (1998), Szirmay-Kalos L., (Ed.), pp. 130–140. 2, 3

[HKBŽ97] HAVRAN V., KOPAL T., BITTNER J., ŽÁRA J.: Fast robust BSP tree traversal algorithm for ray tracing. *Journal of Graphics Tools 2*, 4 (Dec. 1997), 15–23. 2

[Hou06] 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. 4

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree GPU raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007), ACM Press, pp. 167–174. 2, 8

[Kap85] KAPLAN M. R.: Space-tracing: A constant time ray-tracer. *Computer Graphics 19*, 3 (July 1985), 149–158. (Proceedings of SIGGRAPH 85 Tutorial on Ray Tracing). 2

[Kar04] KARLSSON F.: *Ray tracing fully implemented on programmable graphics hardware*. Master's thesis, Chalmers University of Technology, 2004. 2

[MB89] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. In *Graphics Interface Proceedings 1989* (June 1989), A.K. Peters, Ltd, pp. 152–163. 2, 3

[NVI] NVIDIA: The CUDA homepage. http://developer.nvidia.com/cuda. 2, 6

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) 21*, 3 (2002), 703–712. 2

[PSS*06] PABST H.-F., SPRINGER J. P., SCHOLLMEYER A., LENHARDT R., LESSIG C., FROEHLICH B.: Ray casting of trimmed NURBS surfaces on the GPU. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 151–160. 2

[Pur04] PURCELL T. J.: *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, 2004. 2

[Res06] RESHETOV A.: Omnidirectional ray tracing traversal algorithm for kd-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 57–60. 4

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM Transaction of Graphics 24*, 3 (2005), 1176–1185. (Proceedings of ACM SIGGRAPH). 2, 8

[Sam84] SAMET H.: The quadtree and related hierarchical data structures. *ACM Computing Surveys 16*, 2 (1984), 187–260. 3

[Sam89] SAMET H.: Implementing ray tracing with octrees and neighbor finding. *Computers and Graphics 13*, 4 (1989), 445–60. 3

[SGS06] STOLL C., GUMHOLD S., SEIDEL H.-P.: Incremental raycasting of piecewise quadratic surfaces on the GPU. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 141–150. 2

[TS05] THRANE N., SIMONSEN L. O.: *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*. Master's thesis, University of Aarhus, 2005. 2

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004. 2, 7

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. 6. 8

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics 25*, 3 (2006), 485–493. (Proceedings of ACM SIGGRAPH). 8

[WMS06] WOOP S., MARMITT G., SLUSALLEK P.: B-KD trees for hardware accelerated ray tracing of dynamic scenes. In *Proceedings of Graphics Hardware* (2006). 2

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164. (Proceedings of Eurographics). 4

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: A programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*, 3 (2005), 434–444. 2

Appendix A: Optimality of the Stackless Packet Traversal Algorithm

We define a stackless packet traversal algorithm to be optimal if it is correct and if – for a packet of rays with direction vectors in the same octant – it visits each node of the kd-tree *at most once*. Thus it has to process all rays that intersect a node together. Our algorithm is optimal in this respect. To prove it, we will first prove the following lemma:

Lemma 1 Let T be a subtree and let A be the set of active rays with which we enter T during some down traversal step. Then, the packet traversal algorithm will only visit nodes from T, until all rays of A exit T or terminate.

Proof First we look at the way we store the tree in memory: For a node N we require that its subtree is stored in the order $|N_L|N_R|subtree(N_L)|subtree(N_R)|$, with N_L and N_R being the children of N. Thus, in the down traversal step, the address of $N_{traversed}$ can only increase. Because we started the down traversal with the node with maximum address, it follows that all rays $R \notin A$ will be at nodes with addresses smaller than the one of T_{root}. Let us look now at a leaf L inside T. For a rope of a face of L there are 3 possibilities: It points to a node N_I within T, it points to a node N_O outside of T, or it points to **nil**. N_I has an address larger than the address of the root of T and if we choose it in an exit step, we will stay in T. N_O on the other hand has an address smaller than the address of the root of T. This is a consequence of the way we construct the ropes: The sibling of N_O is actually an ancestor of the root of T. Thus, the algorithm can choose N_O for $N_{traversed}$ after a leaf exit step only if there are no rays waiting at nodes inside T ($N_{current} \notin T$). However, this would mean that all rays of A have exited T. In case the algorithm chooses **nil** after exiting a leaf, the whole traversal stops. Thus, the node we choose at an exit step will always be inside T if some rays of A are still inside T. Which proves the lemma. □

Proof of optimality Let R be the root of the tree. Because we assume that the rays have the same direction along the split axis of R, they will all traverse R's children in the same order and we can classify the children as near and far. According to lemma 1, the rays that intersect the near child will traverse it completely before any ray starts traversing the far child. Thus, the algorithm cannot visit the near child twice, as the entry points of all rays move toward the far child. The same holds for the far child – once all rays finish traversing the far child, no ray will be in the tree anymore. We can apply the reasoning recursively by taking R to be the root of the left respectively right subtree of the root and by limiting the set of rays to only those that intersect the new R. □