To appear in the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007.

Realtime Ray Tracing on GPU with BVH-based Packet Traversal

Johannes Günther* MPI Informatik

Stefan Popov† Saarland University

Hans-Peter Seidel* MPI Informatik

Philipp Slusallek† Saarland University

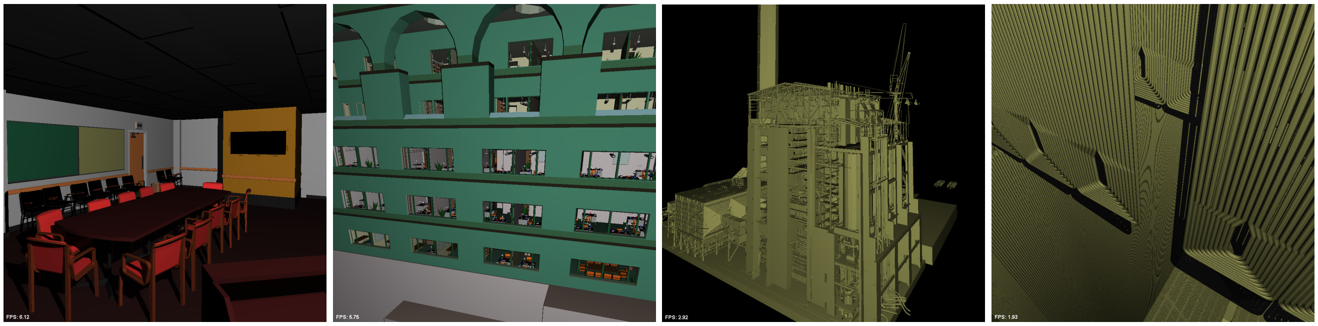

Figure 1: The CONFERENCE, SODA HALL, POWER PLANT from outside, and POWER PLANT furnace scenes. Using our new BVH-based GPU ray tracer, we render them at 6.1, 5.7, 2.9, and 1.9 fps, respectively, at a resolution of 1024×1024 with shading and shadows from a single point light source.

ABSTRACT

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

In this paper we present a BVH-based GPU ray tracer with a parallel packet traversal algorithm using a shared stack. We also present a fast, CPU-based BVH construction algorithm which very accurately approximates the surface area heuristic using streamed binning while still being one order of magnitude faster than previously published results. Furthermore, using a BVH allows us to push the size limit of supported scenes on the GPU: We can now ray trace the 12.7 million triangle POWER PLANT at 1024×1024 image resolution with 3 fps, including shading and shadows.

Index Terms: I.3.6 [Computer Graphics]: Methodology and Techniques Realism—Graphics data structures and data types I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1 INTRODUCTION

Lately, ray tracing systems running on graphics hardware have developed to a serious alternative to CPU-based ray tracers [PGSS07, HSHH07]. However, even though optimized for the GPU architecture, these implementations can still not utilize the full power of modern GPUs.

To gain maximum performance from the GPU, two main problems need to be addressed. First, one needs to keep only a small state per thread to allow for enough active threads to run to keep the GPU busy. The ray tracer of Popov et al. required too many live registers which resulted in a poor GPU utilization of below 33% [PGSS07]. Second, one needs to assure the coherent execution of threads running in parallel, due to the very wide SIMD architecture of current GPUs (32–48 units execute the same instruction [NVI]). Execution divergence (i.e. incoherent branching) can limit performance of ray tracing to around 40% of the graphics board's theoretical potential [HSHH07].

Besides ray tracing performance there are several other issues to keep in mind when designing a ray tracer running on the GPU. Usually there is considerably less memory available on the GPU. While a standard PC has typically 2 GB (and easily up to 8 GB) of RAM the memory of standard graphic boards is still limited to only 512-768 MB. Thus, more compact data structures should be preferred on the GPU. Furthermore, a ray tracing system – aiming at real-time frame rates on the GPU – should support dynamically changing scenes.

In this paper we present a novel GPU ray tracing implementation, addressing the above pointed problems and issues. We use a new, parallel, and coherent traversal algorithm for a bounding volume hierarchies (BVH), based on a shared stack. Our method is suited for the GPU as it requires less live registers and it exhibits coherent branching behavior. Furthermore, the choice of a BVH as an acceleration structure has the additional advantage of requiring less memory than the previously used kd-trees (especially when using ropes [PGSS07]), and BVHs seem to be better suited for dynamic scenes [WMG*07] and for handling secondary rays [BEL*07].

As a second main contribution we present a fast BVH construction algorithm for the CPU based on streamed binning [PGSS06].

2 PREVIOUS WORK

While the kd-tree remains the best known acceleration structure for ray tracing of static scenes [Hav01] this is not that clear for dynamic scenes. Currently it seems that bounding volume hierarchies (BVHs) are easier to update after geometry changes [LYTM06, WBS07, YCM07]. Thus BVHs seem to be the better suited acceleration structure for animated scenes [WMG*07]. It has also been shown that BVHs built according to the surface area heuristic (SAH) [MB89] are quite competitive to kd-trees, in particular if groups of rays are traversed together [WBS07].

2.1 Ray Tracing on GPUs

Ever since GPUs started to provide more raw computation power than CPUs researchers tried to leverage this performance for other task than the intended rasterization. Ray tracing has been among

*e-mail:{guenther,hpseidel}@mpi-inf.mpg.de
†e-mail:{popov,slusallek}@graphics.cs.uni-sb.de

these tasks from the very beginning, being both computationally demanding and massively parallel.

The first step toward GPU ray tracing was made in 2002 with the Ray Engine [CHH02], implementing only the ray-triangle intersection on the GPU. Streaming geometry to the GPU became quickly the bottleneck. To avoid this bottleneck Purcell et al. [PBMH02, Pur04] moved essentially all computations of ray tracing to the GPU: primary ray generation, acceleration structure traversal, triangle intersection, shading, and secondary ray generation. This basic approach to ray tracing on the GPU was the base for several other implementations, including [Chr05, Kar04, EVG04]. However, these approaches had limited performance, by far not reaching frame rates of CPU-based ray tracers. The main problem at that time was the limited GPU architecture. Only small kernels without branching were supported, thus many CPU-controlled "rendering" passes were necessary to traverse, intersect and shade the rays.

In particular the traversal of hierarchical acceleration structures was difficult on the GPU, because it usually requires a stack, which is poorly supported on GPUs. Therefore, Foley and Sugerman [FS05] presented two implementations of stackless kd-tree traversal algorithms for the GPU, namely *kd-restart* [Kap85] and *kd-backtrack*. Although better suited for the GPU, the high number of redundant traversal steps lead to relative low performance.

Recently, Horn et al. [HSHH07] reduced the number of redundant traversal steps of kd-restart by adding a short stack. With their implementation on modern GPU hardware they already achieve a high performance of 15–18M rays/s for moderately complex scenes.

Concurrently Popov et al. [PGSS07] presented a parallel, stackless kd-tree traversal algorithm without the redundant traversal steps of kd-restart. With over 16M rays/s on the CONFERENCE scene, their GPU ray tracer achieves similar performance as CPU-based ray tracers. However, both fast GPU ray tracing implementations [PGSS07, HSHH07] demonstrated only medium-sized, static scenes.

Besides grids and kd-trees there are also several approaches that use BVH as acceleration structure on the GPU. Carr et al. implemented a limited ray tracer on the GPU that was based on geometry images [CHCH06]. Therefore, it can only support a single triangle mesh without sharp edges. The acceleration structure they used was a *predefined* bounding volume hierarchy which cannot adapt to the topology of the object.

Thrane and Simonsen [TS05] presented stackless traversal algorithms for the BVH which allows for efficient GPU implementations. They outperformed both regular grids and the plain kd-restart and kd-backtrack variants for kd-trees.

In our approach, we use SAH built BVHs and in contrast to [TS05] we support ordered, view dependent traversal, thus heavily improving performance for most scenes.

3 Modern GPU Design: The G80

Recently, with the introduction of the G80 architecture, GPUs have made a huge step ahead, not only in performance but in programmability as well. Through their new programming platform [NVI], NVIDIA's G80 GPUs are much closer now to a highly parallel general purpose processor with extensions for doing graphics, than to a traditional GPU. Rather than targeting a wide range of compatible graphics hardware, we developed our implementation specifically for the G80 architecture, making use of most of the advanced features it provides. The BVH traversal algorithm, presented in the next section, will also work on any other parallel RAM (PRAM) machine, with processors working in SIMD mode.

3.1 Hardware Architecture

The main computational unit on the G80 is the thread. As opposed to other GPU architectures, threads on the G80 can read and write freely to GPU memory and can synchronize and communicate with each other.

To enable communication and synchronization, the threads on the G80 are logically grouped in blocks. Threads in a block synchronize by using barriers and they communicate through a small high-speed low-latency on-chip memory (a.k.a. shared memory).

Physically, threads are processed in chunks of size 32 in SIMD. The G80 consists of several cores working independently on a disjoint set of blocks. Each core can execute one chunk at any point of time, but can have many more on the run and can switch among them (hardware multi-threading). By doing this, the G80 can hide various types of latencies, introduced for example by memory accesses or instruction dependencies. Threads never change cores, and one block is always executed by the same core, until all threads in it terminate. The chunks are formed deterministically, based on the unique ID (number) of the threads in them. Thread IDs are assigned sequentially by the hardware.

The memory of the G80 consists of a rather large on board part (global memory), used for storing data and textures and small on-chip parts, used for caching and communication purposes. Accessing the global memory is expensive in terms of the introduced latency. However, if the consecutive threads of a chunk access consecutive memory addresses, the memory controller does a single request to global memory and brings in a whole line, thus paying the latency cost only once. The on-chip memory is divided between the shared memory and the register file. Each core has its shared memory and accessing shared memory is as fast as using a register, given that it is addressed properly. The shared memory is partitioned among the blocks of threads local to a core. Each thread of a block can access any memory element of its block's partition, but can not access the shared memory of other blocks. The register file is partitioned among all the threads running on a core and each thread has exclusive access to its partition.

The number of running threads (chunks) on a core is determined by three factors: the number of register each thread uses, the size of the shared memory partition of a block and the number of threads in a block. Using more registers or larger shared memory partitions limits the total number threads that a GPU can run, which in turn impacts the performance, since multi-threading is the primary mechanism for latency hiding on the GPU. An explanation of how to best choose the block size, as well as an in-depth description of the G80 architecture is available in [NVI].

The currently available consumer high-end G80 GPUs (GeForce 8800GTX) have 16 cores, an on-board memory of 768 MB and 16 kB of shared memory per core. Each core can run at most 768 threads and the maximum number of threads for the GPU can not exceed 12k. The register file of each core can hold 8k scalar registers and 100% utilization can be accomplished if each thread does not use more than 10 scalar registers and 5 words of shared memory.

Because the threads in a chunk are executed in SIMD, their memory accesses are implicitly synchronized. Thus, the G80 can be viewed as a CRCW PRAM machine [FW78] with 32 processors. All algorithms with coherent branch decisions designed for a PRAM machine can directly be implemented on the G80.

3.2 Implications on Algorithm Design

To achieve full performance on the G80, algorithms should be able to exploit fully its parallelism. Thus an algorithm should be able to benefit from running with tens of thousands of threads. Furthermore, each thread should use as few resources as possible in order to not limit the parallelism of the GPU. Because threads get executed in SIMD chunks, they need to have coherent branch decisions within a chunk. Otherwise, both branches will be executed by the whole chunk.

For optimal latency coverage, the threads of the GPU need to be compute intensive. Also, care should be taken when reading or writing to memory, to exploit the grouping mechanism in the memory controller of the GPU.

In this context, ray tracing can map very well to the parallelism requirement of the GPU. On the other hand, ray tracing relies on a precomputed spatial indexing structure used to accelerate ray-scene intersections. Traversing the structure usually requires a per-ray stack, which increases the per-thread state considerably. Thus, a direct implementation of stack-based traversal on the GPU will be slow and inefficient.

4 GPU Ray Tracing Using Parallel BVH Traversal

To avoid the per-ray stack, previous GPU ray tracing implementations augmented the spatial indexing data structure in a way [PGSS07, TS05] such that they can directly traverse from one node to another along the ray direction. Alternatively, they needed to restart traversal after each visited leaf [FS05]. This resulted in either a large spatial indexing structure [PGSS07] or sub-optimal traversal [FS05].

4.1 Traversal Algorithm

We solve the above problems by taking a different approach. Instead of fully removing the stack, we trace packets of rays and amortize the stack storage over the whole packet. We use a BVH as an acceleration structure, because it is the only hierarchical structure that allows us to discard the per-ray entry and exit distances (points), instead of storing them onto a per-ray stack.

The algorithm maps one ray to one thread and a packet to a chunk. It traverses the tree synchronously with the packet. The algorithm works on one node at a time and processes the whole packet against it. If the node is a leaf, it intersects the rays in the packet with the contained geometry. Each thread stores the distance to the nearest found intersection. If the processed node is not a leaf, the algorithm loads its two children and intersects the packet with both of them to determine the traversal order. Each ray determines which of the two nodes it intersects and in which it wants to go first by comparing the signed entry distances of both children. If an entry distance of a node is beyond the current nearest intersection, the ray considers the node as not being intersected. The algorithm then makes a decision in which node to descend with the packet first by taking the one that has more rays wanting to enter it. If at least one ray wants to visit the other node then the address of this other node is pushed onto stack. In case all rays do not want to visit both nodes or after the algorithm has processed a leaf, the next node is taken from the top of the stack and its children are traversed. If the stack is empty, the algorithm terminates.

The decision, which node has more rays wanting to traverse it first, is made using a PRAM sum reduction. Each thread writes a 1 in an own location in the shared memory if its ray wants to visit the right one first, and -1 otherwise. Then, the sum of the memory locations is computed in $O(\log N)$ – that is in 5 steps with 32 wide chunks. The packet takes the left node if the sum is smaller than 1 and the right one otherwise.

We use the general packet intersection algorithm presented in [KS06] for intersecting a ray with a triangle. We carry out all ray independent pre-computations of the algorithm in 6-wide SIMD. Working directly on the geometry allows us to discard the per-triangle pre-computed data, used in conjunction with the fast projection intersection test [WSBW01]. Although this decreases rendering speed by ca. 20%, it allows us to ray trace deformable scenes as well as to store larger scenes in the GPU memory.

Algorithm 1: Shared Stack BVH Traversal

```
$R = (O,D)$                                   ▷ The ray
$d \leftarrow \infty$                         ▷ Distance to closest intersection
$N_P \leftarrow$ pointer to the BVH root

$N_L, N_R$ : **shared** $\equiv$ Shared storage for $N$'s children
$M[]$ : **shared** $\equiv$ Reduction memory
$S$ : **shared** $\equiv$ The traversal stack
$P_{ID}$ : **const** $\equiv$ The number of this processor

**loop**
    **if** $N_P$ points to a leaf **then**
        Intersect $R$ with contained geometry
        Update $d$ if necessary
        **break**, **if** $S$ is empty
        $N_P \leftarrow pop(S)$
    **else**
        **if** $P_{ID} < size(N_L, N_R)$ **then**         ▷ parallel read
            $(N_L, N_R)[P_{ID}] \leftarrow children(N_P)[P_{ID}]$
        **end if**

        $(\lambda_1, \lambda_2) \leftarrow intersect(R, N_L)$
        $(\mu_1, \mu_2) \leftarrow intersect(R, N_R)$
        $b_1 \leftarrow (\lambda_1 < \lambda_2) \wedge (\lambda_1 < d) \wedge (\lambda_2 \geq 0)$
        $b_2 \leftarrow (\mu_1 < \mu_2) \wedge (\mu_1 < d) \wedge (\mu_2 \geq 0)$

        $M[P_{ID}] \leftarrow$ **false**, **if** $P_{ID} < 4$
        $M[2b_1 + b_2] \leftarrow$ **true**
        **if** $M[3] \vee M[1] \wedge M[2]$ **then**          ▷ Visit both children
            $M[P_{ID}] \leftarrow 2(b2 \wedge \mu_1 < \lambda_1) - 1$
            $\textsc{ParallelSum}(M[0\,..\,\textit{processor-count}])$
            $(N_N, N_F) \leftarrow pointer\text{-}to \begin{cases} (N_L, N_R) & , \textbf{if } M[0] < 0 \\ (N_R, N_L) & , \textbf{else} \end{cases}$
            $push(S, N_F)$, **if** $P_{ID} = 0$
            $N_P \leftarrow N_N$
        **else if** $M[1]$ **then**
            $N_P \leftarrow pointer\text{-}to(N_L)$
        **else if** $M[2]$ **then**
            $N_P \leftarrow pointer\text{-}to(N_R)$
        **else**
            **break**, **if** $S$ is empty
            $N_P \leftarrow pop(S)$
        **end if**
    **end if**
**end loop**
```

We implemented the above algorithm as part of a ray tracing system, using NVIDIA's CUDA [NVI]. We used a single kernel for the whole ray tracing pipeline. Even though the CUDA compiler was still in beta and did not aid us too much in reducing the register count (as also reported by [PGSS07]), we were able to reach 63% occupancy of the GPU for primary rays with eye light shading and 38% with full Phong shading with shadows and mulitple light sources. We did not tune our code additionally to reduce the register count.

5 Fast BVH Construction

Inspired by [PGSS06] and [WBS07] we developed a fast, streaming BVH construction algorithm that uses binning to approximate the SAH cost function. The BVH variant we use is simply a binary tree with axis-aligned bounding boxes (AABBs).

The SAH [GS87, MB89] estimates the ray tracing performance of a given acceleration structure. This global cost C_T of a complete kd-tree or BVH T can be computed as

$$C_T = K_T \sum_{N \in Nodes} \frac{SA(V_N)}{SA(V_S)} + K_I \sum_{L \in Leaves} \frac{SA(V_L)}{SA(V_S)} n_L, \qquad (1)$$

where $SA(V)$ is the surface area of the AABB V, V_S is the AABB of the scene, K_T and K_I are cost constants for a traversal and an intersection step, respectively, and n_L is the number of primitives in leaf L.

The goal of building good BVHs is to minimize this cost. However, solving this global optimization problem is impractical even for smallest scenes. Fortunately, a local greedy approximation for a recursive top-down BVH construction works well [WBS07]. For each node N to be split into two child nodes N_l and N_r the cost C_P of each potential partition is computed according to

$$C_P = K_T + \frac{K_I}{SA(N)} \left[n_l SA(N_l) + n_r SA(N_r) \right], \qquad (2)$$

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

This local optimization problem is now much smaller. However, testing all possible $2^{n-1} - 1$ partitions of the primitives of the current node into two subsets is again impractical. Following [WBS07] we use a set of uniformly distributed, axis-aligned planes to partition the primitives by means of their centroids.

5.1 Streamed Binning of Centroids

For each potential partition we need to compute Eq. (2), hence we need to know the primitive counts *and* the surface areas of both children. To compute these counts efficiently, Wald et al. [WH06, WBS07] proposed to sort the primitives. However, a much more efficient method was recently published, which avoids sorting and which additionally features memory friendly access patterns [PGSS06, HSM06]. For our BVH builder, we adapt the streamed binning method of [PGSS06], which was originally proposed for building kd-trees.

The idea is to iterate once over the primitives, to bin them by means of their centroids, and by doing so, to accumulate their count and extend in several bins. The gathered information in the bins is then used to reconstruct the primitive counts and the surface areas on both sides of each border between bins, and thus to compute the SAH cost function at each border plane. Note that accumulating the *extent* in the bins is necessary as well, because – unlike kd-trees – the split plane location alone is not sufficient to compute the surface areas of the child nodes – the AABBs of the children can shrink in all three dimensions.

As Popov et al. [PGSS06] we minimize memory bandwidth by performing the binning in all three dimensions for both children during the split of the parent node.

5.2 Implementation Details

In this section we give some details of our implementation concerning efficiency and robustness. The streamed binning BVH builder is implemented on the CPU to run concurrently to the GPU ray tracer.

We extensively use SIMD operations to exploit instruction level parallelism of modern CPUs, working on all three dimensions at once during binning and during SAH evaluation.

Each bin consists of an AABB and a counter. The primitives are represented by the centroid and the extent of their AABBs. For each primitive we compute the indices of the bins of all three dimensions from its centroid in SIMD. Then, the counters of all three bins are incremented, and their AABBs are enlarged with the primitive's AABB using SIMD min/max operations.

We enhance the resolution of the binning by uniformly distributing the bins over the current interval of all the *centroids* rather than over the the current bounding box of the primitives. This is especially important when there are large primitives.

After binning we evaluate Eq. (2) with two passes over the bins: In the first pass from left to right we compute n_l and $SA(N_l)$ at the borders of the bins by accumulating the counters and by successively enlarge the AABBs of the bins. In the second pass from right to left we reconstruct n_r and $SA(N_r)$, and finally find the index i_{min} and dimension of the bin that has minimal cost C_P.

Computing the split plane position from i_{min} turned out to be surprisingly difficult. Because of floating point precision problems we cannot just invert the linear function used during binning. An inaccurate split plane can not only lead to sub-optimal partitions. In the worst case, an inaccurate split plane can even lead to an invalid partitions (one child is empty) if the split plane is computed to be completely on one side of all centroids. Using double precision only reduces the chances of invalid partitions but does not solve the problem. Our solutions is to not compute the splitting plane from i_{min} at all, but to keep track of the centroids during binning. Therefore each bin additionally stores the minimum of the coordinates of all centroids that fell into it. Using the centroid minimum of bin i_{min} as split plane location then ensures consistent and robust partitioning.

The number of bins is a crucial parameter controlling the construction speed and accuracy. The more bins there are, the more accurate is the sampling of the SAH cost function, but the more work has to be done during calculation of the SAH function from the binned data (the binning steps are independent from the number of the bins). There should be at most 256 bins per dimension such that the binning data still fits into 64 kB of L1 cache. Additionally, binning becomes inefficient if the number of bins is close to the number of to-be-binned primitives. Therefore we adaptively choose the number number of bins k per dimension linearly depending on number of primitives n and bin-ratio r: $k = n/r$ and clamp it to $[k_{min}, k_{max}]$. We experimented with different parameter sets representing a trade-off between speed and accuracy. The `default` settings are $k_{max} = 128$, $k_{min} = 8$, and $r = 6$. The `fast` settings are $k_{max} = 32$, $k_{min} = 4$, and $r = 16$.

6 Results and Discussion

For measuring purposes we used an Intel 2.4 GHz Core 2 workstation and a NVIDIA GeForce 8800 GTX graphics card.

6.1 Fast BVH Construction

Streamed binning for BVH construction is more computational demanding than for kd-tree construction, because one needs to keep track not only of the primitive counts, but also of the surface areas of the children. Additionally, the surface area cannot be incrementally computed, because the AABBs may have changed in all three dimensions. Nevertheless, constructing an SAH BVH with streamed binning can still be faster than constructing an SAH kd-tree for the same scene: Because a BVH does not split primitives, less nodes need to be created; and because a BVH node bounds in three dimensions whereas a kd-tree node bounds only in one dimension, there are usually less tree levels in a BVH (given the same SAH termination parameters), and thus the number of splits and binning steps is lower.

These considerations are backed up by our measurements in Table 1, where we compare, among others, the construction time of kd-trees and BVHs. With our BVH builder we consistently outperform published constructions times for kd-trees that also use the scanning/binning approach [PGSS06, HSM06], even though [HSM06] used significantly fewer primitives (because they do not tessellate quads into triangles).

scene	#tris	kd-tree size	with ropes	BVH size
SHIRLEY6	804	82.4 kB	266.3 kB	21.0 kB
BUNNY	69,451	6.9 MB	23.0 MB	2.14 MB
FAIRY FOREST	174,117	14.9 MB	47.9 MB	4.78 MB
CONFERENCE	282,641	27.8 MB	85.0 MB	7.62 MB
SODA HALL	2,169,132	—	—	55.0 MB
POWER PLANT	12,748,510	—	—	230 MB

Table 2: Comparing the size of different acceleration structures for GPU ray tracing for several scenes. We list the sizes for a kd-tree, a kd-tree with ropes (data from [PGSS07]), and for a BVH – all constructed according the greedy SAH cost function. A BVH needs only ${}^{1}/_{3}$–${}^{1}/_{4}$ of the space of a kd-tree and is one order of magnitude smaller than a kd-tree with ropes. Thus even the 12.7 million triangle POWER PLANT fits into graphics memory.

Our measurements in Table 1 include absolute construction time and relative BVH quality (in SAH cost, Eq. (1)) for both, the `default` and the `fast` parameter settings (see Section 5.2). Using the the `fast` settings BVH construction is about 20% faster at the cost of slightly decreased BVH quality.

Comparing to previously published data of a sweep-based SAH BVH builder [WBS07] our streamed binning approach is one order of magnitude faster at almost the same BVH quality.

For their BVH-based ray tracer Lauterbach et al. [LYTM06] favored construction speed over ray tracing performance to support dynamic scenes. With split-in-the-middle they chose the probably fastest approach to select a partition plane during BVH construction, which unfortunately also decreases ray tracing performance to 50%–90% compared to building the BVH according the SAH [LYTM06]. Approximating the SAH with our binning approach achieves faster construction times (also due to faster hardware) while retaining high ray tracing performance.

Interestingly, for some scenes the binning approximation of the SAH cost function results in even *better* BVHs (quality > 100% in Table 1) than when exactly evaluating Eq. (2). This is a strong confirmation that the local greedy SAH function is exactly that, a local greedy optimization, failing to provide the global minimal SAH cost (Eq. (1)).

Even though the 350 million triangle BOEING 777 model currently does not fit into GPU memory, we include construction times showing that even for such a large scene our streamed binning construction algorithm can produce a high quality BVH in less than 10 minutes.

6.2 Memory Requirements

In Table 2 we compare the size of a BVH with the size of both a plain kd-tree and a kd-tree with ropes for stackless traversal on the GPU [PGSS07]. Although a node of a BVH needs 28 Bytes (6 float for the bounds and one pointer for the children) and is therefore larger than a kd-tree node (8 Bytes), a BVH needs fewer nodes than a kd-tree: Being an object hierarchy a BVH does not have empty nodes and has at most as many inner nodes as there are primitives, whereas a kd-tree can potentially finely subdivide the space taken by primitives to cut off empty space. Thus a BVH is much more frugal with memory for the same scene as a kd-tree, not to speak of adding ropes. A kd-tree is between three and four times larger than a BVH, augmenting a kd-tree with ropes adds another factor of three. Given the notoriously stinted memory on GPU boards these numbers strongly advice to use the BVH. Using a BVH with only 230 MB allows us to even ray trace the 12.7 million triangle POWER PLANT scene on the GPU.

6.3 Ray Tracing Performance

Finally, in Table 3 we present the absolute ray tracing performance (excluding BVH construction time) of our BVH-based GPU ray tracer, in comparison with previously published performance data of a kd-tree-based GPU ray tracer [PGSS07] running on the same graphics hardware. Although kd-trees are usually more efficient for ray tracing than BVHs [Hav01] we achieve comparable or even slightly faster frame rates. The reason is that our parallel BVH traversal algorithm is easier to implement and uses less live registers and thus we get a higher GPU utilization of 63% compared to the 33% of [PGSS07] for primary rays.

	[PGSS07]		our GPU ray tracer	
scene	primary	2ndary	primary	+shadow
FAIRYFOREST	10.6	4.0	13.2 (14.6)	4.8
CONFERENCE	16.7	6.7	16 (19)	6.1
SODA HALL	—	—	13.6 (16.2)	5.7
POWER PLANT	—	—	6.4	2.9

Table 3: Absolute ray tracing performance for a 1024×1024 image in fps of our BVH-based GPU ray tracer in comparison to the currently fastest, kd-tree-based GPU ray tracer [PGSS07]. Primary rays are eye-light shaded and additionally we report performance numbers when illuminating with a single point light and tracing shadow rays. The numbers in brackets denote the fps when using a precomputed triangle projection test [WSBW01].

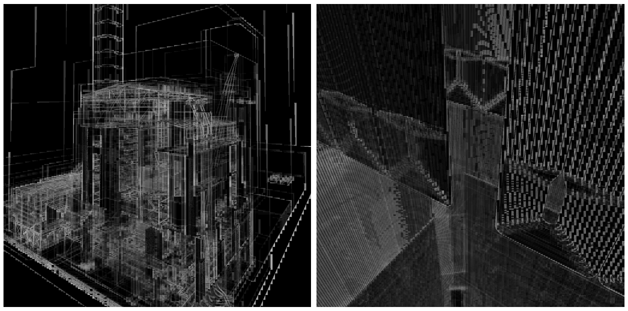

Figure 2: Visualization of SIMD utilization during traversal of the complex POWER PLANT scene for the same views as in Figure 1. The brightness of a pixel indicates the percentage of inactive traversal steps.

The efficiency of our packet traversal algorithm also depends on the coherence of the traversal decisions of the rays in a packet. In Figure 2 we display the ratio of inactive traversal steps of a ray to the number of all traversal steps of its packet. On object boundaries incoherent traversal decisions are clearly visible. For the two shown views of the complex POWER PLANT scene the average SIMD utilization is still about 88% and 85%, respectively.

7 Conclusion and Future Work

In this paper we demonstrated real-time GPU ray tracing with a new, parallel BVH traversal algorithm that is suited for modern graphics hardware. Although BVHs are usually slower for ray tracing than kd-trees we can achieve at least the same performance as kd-tree-based GPU ray tracers running on the same hardware. By exploiting the compactness of BVHs and by directly operating on triangle data without intersection acceleration structures we are able to ray trace large models not seen on a GPU before. Additionally, we presented a construction algorithm for BVHs based on streamed binning that is both very fast and accurate.

As for future work we would like to implement the binning SAH BVH construction on the GPU. Alternatively, we think of refitting the BVH on the GPU to support dynamic scenes and rebuilding the BVH asynchronously on the CPU to counter BVH degradation in the sense of [IWP07].

		published kd-tree data		published BVH data		our BVH measurements				
		2.6 GHz Opteron	2.4 GHz Core 2	2.8 GHz P4	2.6 GHz Opteron	2.4 GHz Core 2				
scene	#tris	[PGSS06]	[HSM06]	[LYTM06]	[WBS07]	exact SAH	binning	quality	fast binning	quality
Bunny	69,451	513 ms	250 ms	90 ms	—	168 ms	48 ms	99.8%	37 ms	98.9%
Fairy Forest	174,117	1.15 s	0.3 s	—	2.8 s	0.47 s	0.12 s	100.2%	0.10 s	98.8%
Conference	282,641	1.41 s	—	—	5.06 s	0.80 s	0.20 s	99.4%	0.15 s	92.5%
Buddha	1,087,716	—	—	1.7 s	20.8 s	4.38 s	0.84 s	100.0%	0.66 s	98.9%
Soda Hall	2,169,132	—	5.14 s	—	53.2 s	8.78 s	1.59 s	101.6%	1.28 s	103.5%
Power Plant	12,748,510	—	—	—	—	119 s	8.1 s	100.5%	6.6 s	99.4%
Boeing 777	348,216,139	—	—	—	—	5605 s	667 s	98.1%	572 s	94.8%

Table 1: Comparing the (re)construction performance for kd-tree and BVH using different construction algorithms on similar hardware. Due to its huge size the Boeing 777 was measured on a 2.0 GHz Opteron with 64 GB RAM, of which 35 GB were consumed during construction. Note that [HSM06] supports quads and thus uses considerable fewer primitives for construction. All acceleration structures are built according to SAH; [LYTM06] is one exception – they use quick split-in-the-middle, which decreases the quality of the BVH and rendering speed to 50%–90% compared to using SAH. The reported quality of our proposed binned BVH construction is measured in SAH cost Eq. (1) and is relative to the exact SAH evaluation. Binned BVH construction is both very fast and accurate.

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). 1

[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), Aka GmbH, pp. 255–262. 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, 3

[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. 2

[GS87] Goldsmith J., Salmon J.: Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications 7*, 5 (May 1987), 14–20. 3

[Hav01] Havran V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001. 1, 5

[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. 1, 2

[HSM06] Hunt W., Stoll G., Mark W.: Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 81–88. 4, 6

[IWP07] Ize T., Wald I., Parker S. G.: Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization* (May 2007). 5

[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

[KS06] Kensler A., Shirley P.: Optimizing Ray-Triangle Intersection via Automated Search. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 33–38. 3

[LYTM06] Lauterbach C., Yoon S.-E., Tuft D., Manocha D.: RT-DEFORM Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 39–46. 1, 5, 6

[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. 1, 3

[NVI] NVIDIA: The CUDA Homepage. `http://developer.nvidia.com/cuda`. 1, 2, 3

[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

[PGSS06] Popov S., Günther J., Seidel H.-P., Slusallek P.: Experiences with Streaming Construction of SAH KD-Trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 89–94. 1, 3, 4, 6

[PGSS07] Popov S., Günther J., Seidel H.-P., Slusallek P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum 26*, 3 (Sept. 2007). (Proceedings of Eurographics), to appear. 1, 2, 3, 5

[Pur04] Purcell T. J.: *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, 2004. 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, 3

[WBS07] Wald I., Boulos S., Shirley P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1 (Jan. 2007), 6. 1, 3, 4, 5, 6

[WH06] Wald I., Havran V.: On building fast kd-trees for Ray Tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 61–70. 4

[WMG*07] Wald I., Mark W. R., Günther J., Boulos S., Ize T., Hunt W., Parker S. G., Shirley P.: State of the Art in Ray Tracing Animated Scenes. In *STAR Proceedings of Eurographics 2007* (Sept. 2007), Eurographics Association. to appear. 1

[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). 3, 5

[YCM07] Yoon S.-E., Curtis S., Manocha D.: Ray Tracing Dynamic Scenes using Selective Restructuring. *Computer Graphics Forum 26*, 3 (Sept. 2007). (Proceedings of Eurographics), to appear. 1