Parallel On-Demand Hierarchy Construction on Contemporary GPUs

Marek Vinkler[1] Vlastimil Havran[2] Jiří Bittner[2] Jiří Sochor[1]
[1]Faculty of Informatics, Masaryk University, Czech Republic
[2]Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic
E-mail: xvinkl@fi.muni.cz, havran@fel.cvut.cz, bittner@fel.cvut.cz, sochor@fi.muni.cz

Abstract—We present the first parallel on-demand spatial hierarchy construction algorithm targeting ray tracing on many-core processors such as GPUs. The method performs simultaneous ray traversal and spatial hierarchy construction focused on the parts of the data structure being traversed. The method is based on a versatile framework built around a task pool and runs entirely on the GPU. We show that the on-demand construction can improve rendering times compared to full hierarchy construction. We evaluate our method on both object (BVH) and space (kd-tree) subdivision data structures and compare them mutually. The on-demand method is particularly beneficial for rendering large scenes with high occlusion. We also present SAH kd-tree builder that outperforms previous state-of-the-art builders running on the GPU.

Index Terms—GPU, bounding volume hierarchies, kd-trees, lazy build, ray tracing.

1 Introduction

Ray tracing is a fundamental algorithm used in the global illumination methods rendering high quality images. Most of these methods sample the illumination by tracing rays inside the scene. These rays solve visibility queries by computing their nearest intersections with the scene. Hierarchical data structures are commonly used in order to reduce ray tracing times. The performance of ray tracing is then directly related to the quality of these data structures. However, building a data structure for a scene in an efficient way is a time consuming task. Until recently, only static scenes could be ray traced at interactive rates because of the required preprocessing time. With the advent of fast parallel processors, algorithms have been proposed allowing for interactive rendering of moderately sized dynamic scenes.

In this paper, we propose a novel method for building the hierarchical data structures in parallel with tracing rays, while building only those hierarchy nodes that are needed during the traversal. To the best of our knowledge, this is the first method of its kind (called "on-demand" below) targeting many-core architectures such as GPUs. The method features complex synchronization and load balancing to run correctly and efficiently on these parallel processors.

The result of this approach are faster build times for both bounding volume hierarchy (BVH) and kd-tree data structures. On average the on-demand method saves 50% of the time to image compared to full hierarchy build. The savings are higher for larger scenes with more occlusion (up to 89%), especially for the kd-tree data structure. As an additional benefit the constructed data structures have smaller memory footprint because the nodes not needed for the ray traversal are not built. Moreover, the method runs entirely on the GPU leaving the CPU free for other computational tasks such as animating the scene geometry.

Our single kernel kd-tree builder is also currently the fastest algorithm for building kd-trees using the surface area heuristic (SAH) on a GPU.

2 Related Work

Bounding Volume Hierarchies on the GPU. The first Bounding Volume Hierarchy (BVH) build method harnessing the high bandwidth and floating point performance of GPUs was proposed by Lauterbach et al. [1]. In their method, the primitives are sorted based on their Morton code and the BVH is built level-by-level, i.e., bit-by-bit of the Morton code. This corresponds to building the BVH with spatial median splitting. To improve the traversal performance of hierarchies built with spatial median, they propose to build the lower levels with SAH.

This method was improved by Pantaleoni and Luebke [2] who proposed to build three levels of the BVH at the same time. Unlike the previous method, they propose to build the top levels using the SAH and use their hierarchical linear BVH (HLBVH) build algorithm for the bottom levels of the tree. This choice improves the traversal performance.

The HLBVH method was further optimized by Garanzha et al. [3] who used a faster radix sorting algorithm and replaced the complex pipeline of the previous method, requiring multiple kernel launches by a set of work queues.

Karras [4] noticed that when ordering the primitives along a space-filling curve, the entire tree can be built in parallel instead of level-by-level. This promotes the scalability of the method making it particularly well suited for modern many-core architectures, but only low quality spatial median hierarchies can be built.

Garanzha et al. [5] proposed to build the entire BVH with an approximate SAH. A grid is constructed on the triangle references and a mip-map of the number of triangles in the cells of the grid is computed. The SAH is then evaluated using the mip-map for the number of primitives. Thus, only splits that coincide with the border between the grid cells on the current mip-map level are evaluated.

Bounding Volume Hierarchies on other parallel platforms. Wald [6] presented a parallel version of a SAH BVH builder with binning for the Intel Many Integrated Core (MIC) architecture. A hardware architecture for building BVHs with binned SAH was proposed by Doyle et al. [7].

Improving Bounding Volume Hierarchies quality. The efficiency of updating BVHs in the context of animations was studied by Kopta et al. [8]. They propose to extend simple refitting, that fails in the presence of incoherent motion with more complex subtree rotations and leaf splitting and merging in order to keep high traversal efficiency during the entire animation.

More recently, Gu et al. [9] presented a method to build the BVH with the use of agglomerative clustering that allows for setting a tradeoff between build time and efficiency. Bittner et al. [10] and Karras and Aila [11] proposed methods that optimize an already built BVH. These optimization methods, however, work on the entire tree without knowing which of its branches are needed for traversal in contrast to on-demand building.

Kd-trees on the GPU. The performance of GPUs was first leveraged for kd-tree construction by Zhou et al. [12]. The algorithm was however limited by the capabilities of the GPUs at that time to spatial median splitting only. Danilewski et al. [13] presented a scalable GPU algorithm using binning for construction of SAH kd-trees. Wu et al. [14] proposed a GPU kd-tree builder that evaluates all the splitting positions by parallelizing the algorithm of Wald and Havran [15]. A hybrid CPU-GPU implementation of the same baseline algorithm was proposed by Roccia et al. [16], that outperforms the previous approaches. The parallel method of Karras [4] can also be used for construction of kd-trees with spatial median splitting.

On-demand data structures for ray tracing. The idea of on-demand construction of spatial hierarchies for ray tracing has been around for over two decades. Hook and Forward [17] presented an approach that for the BVH suggests to mark the nodes as leaf, split-table leaf, and inner node. The split-table leafs when visited by a ray during traversal are further subdivided. A similar idea was presented for BSP trees by Ar et al. [18]. The idea of on-demand construction has been further used for the Bounding-Interval-Hierarchy by Wächter and Keller [19]. We are not aware of any work dealing with on-demand acceleration data structure construction for ray tracing targeting massively parallel GPU architectures.

Task management on the GPU. Several frameworks have been recently proposed that limit the management overhead of launching many kernels to solve complex computations on the GPU. These run in a single kernel and manage the computation using a task pool which schedules the work. The Softshell framework of Steinberger et al. [20] defines tasks that are independently solved by computational units on the GPU. The tasks are stored in fixed size ring buffer used as a queue. The method of Vinkler et al. [21] enables communication between computational units and computational dependencies between individual tasks allowing for a broader class of algorithms to be implemented inside a single kernel. To this end the tasks are stored in a memory pool, with task at any position being eligible for computation. We adopt the approach of Vinkler et al. because the ability of computational unit cooperation is crucial for our method.

The paper is further structured as follows: the core of the proposed method is presented in Section 3. Section 4 describes several optimization techniques of the basic method. Section 5 presents the results and their discussion. Finally, the conclusion is in Section 6.

3 On-Demand Construction

This section presents an overview of the proposed algorithm, discusses the GPU framework supporting our method, and provides a detailed description of the extended traversal and building algorithms. Note that although being mostly generic, the details of our method reflect the properties of CUDA [22] parallel programming environment.

3.1 Algorithm Overview

The main idea of the proposed method is to perform ray tracing and on-demand construction on a massively parallel architecture in an interleaved manner. The input of our algorithm are two arrays: the array containing rays to be shot and the array with geometric primitives (triangles in our case) defining the scene geometry. The algorithm proceeds by traversing a ray through the data structure using the traditional traversal code until a node is reached which should be further subdivided (e.g. it contains more geometric primitives than a given threshold). In that case, the traversal is suspended until the node is subdivided. The subdivision of nodes runs in parallel with the tracing of rays that traverse the already constructed parts of the tree.

In order to increase the utilization of the parallel processing units, the nodes which were not yet visited by any ray may be subdivided speculatively. Such a speculative subdivision reduces the number of traversal stalls and their latency, since a node may already be subdivided when visited by a ray for the first time. On the other hand, a too aggressive speculative tree construction may overload the processing units with building such parts of the tree, which will never be visited during the ray traversal. Our method uses a simple priority mechanism in which the nodes that caused a traversal stall are processed with higher priority. When some processing units are idling because no traversal progress can be made and the nodes which caused a traversal stall are processed by other units, the speculative building of some other nodes is performed. An illustration of the method is shown in Figure 1, which depicts a part of the tree already visited by the ray traversal, a part of the tree built speculatively and a node at which the traversal is suspended.

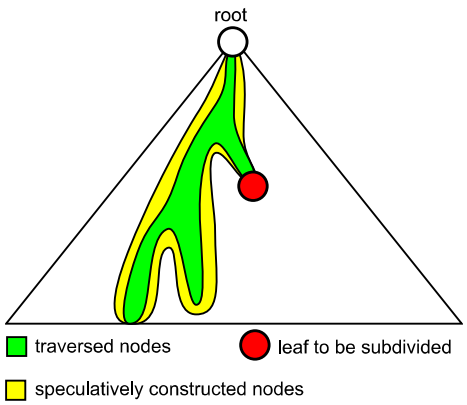

Fig. 1: An example of the situation where ray traversal is suspended because it has reached a node which should be further subdivided (shown in red). Notice that the neighboring nodes of the already visited parts of the tree (green) were built speculatively in the otherwise idle time of the GPU (shown in yellow).

3.2 Task Pool

In order to implement the parallel on-demand algorithm on the GPU, we exploit the previously proposed framework based on the *persistent warps* [23] and the *task pool* [21]. The task pool is responsible for distributing the computation among the computational units, where multiple persistent warps reside. These warps dequeue work to be done from the task pool and optionally enqueue new work into it. The framework is flexible enough to implement the on-demand construction and it also provides seamless way of load balancing between parallel tracing and construction.

The computation is organized on three different levels. The first level consists of a coarse subdivision of work into *tasks* sharing common data. In particular, the subdivision of a node corresponds to a distinct task. On the second level a task is subdivided into a set of *phases* that are executed sequentially (such as finding the splitting plane, partitioning the geometric primitives into the left and right subsets, computing the bounding boxes). The third level is a fine grained subdivision of the task data into warp sized blocks called *work chunks*. A work chunk is the smallest unit of work in our framework and it is processed by a single warp. For the data structure construction, a work chunk consists of 32 geometric primitives contained in the node being subdivided. At any particular moment in time, multiple tasks may be solved in parallel (coarse grained parallelism) and for each task many work chunks of a given phase may be processed in parallel (fine grained parallelism).

The task pool consists of two arrays, a task data array and task header array. The task data array holds all the information necessary to compute the task, while the header array indicates the state of the task. The value of an entry in the task header array represents four different states combined with a state dependent scalar value [21]:

- *Positive* value represents the number of work chunks to be processed on an unlocked task.
- *Zero* value represents a *locked* task due to a maintenance operation such as switching the phase. No warp can take work from a locked task.
- *Large negative* values represent subdivision tasks inserted into the pool but waiting to be unlocked by ray traversal.
- *Maximum negative* value is reserved for tasks that are waiting to be initialized.

As mentioned above, the traversal and building of nodes are interleaved and we do not know in advance, what portion of the persistent warps should be building the nodes and what portion of these warps should be tracing the rays. This problem is resolved dynamically since all the persistent warps can do both these tasks and the ratio changes as needed during the computation. An overview of the main data structures of our algorithm and their relation to the task pool is shown in Figure 2.

3.3 Traversal Loop

Our ray traversal algorithm is an extension of the Fermi optimized code of Aila and Laine [23]. The pseudocode of the algorithm for BVHs is shown in Algorithms 1 and 2.

The `BuildAndTrace` function (Algorithm 1) contains the main loop of the on-demand traversal and building algorithm. Inside this loop the rays to be traced are fetched and they are traced until the traversal is suspended because nodes have to be built. Then the computation proceeds with subdividing these nodes, thus building the BVH on demand. The details of the construction will be presented in Section 3.4.

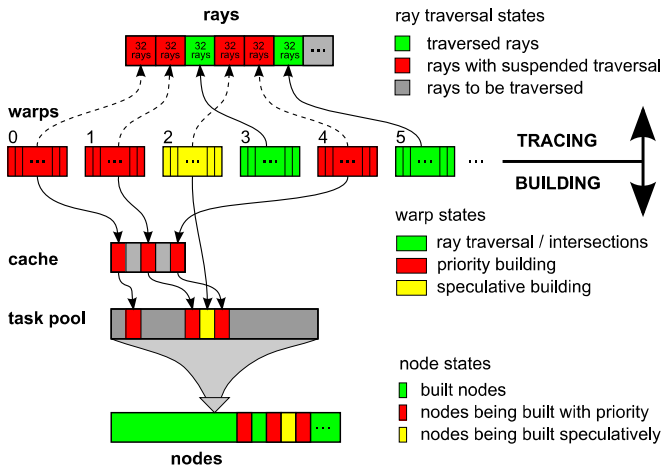

Fig. 2: Overview of the data structures and the main computational dependencies of our algorithm.

Algorithm 1: On-demand method for building BVHs. The parts of the algorithm needed for on-demand BVH construction are shown by red-tinged blocks. The traversal function is defined in Algorithm 2.

```
1 BuildAndTrace(ray array) begin
2   while (any ray is not computed) do
3     Load ray R and set N to the root if
      necessary;
4     Traversal(R, N);
5     if (traversal suspended in a node N) then
6       Take work from task pool;
7       Perform building of node X ;
8
```

The `Traversal` function (Algorithm 2) describes the actual traversal loop.

The modified traversal loop contains four additional conditional branches allowing the code to process yet unbuilt nodes. The first condition in the traversal code (lines 4–5) aims to postpone ray-triangle intersections and perform speculative traversal [23] for the case that the traversal has been restarted at a leaf after waiting for an unbuilt node.

The second and third conditions are inside the inner loop of the node traversal (lines 14–15 and 17–18). These conditions are executed when a child node, which should be traversed by relevant rays, is flagged as unbuilt. Since both children of a node may be traversed and both may be unbuilt, we need two conditions, one for each child. If either of the conditions succeeds, the task in the pool corresponding to that node is unlocked. The address of the unbuilt node loaded from the node array holds the index of the corresponding task. This address is replaced in the thread's traversal state with a new address holding the address of the parent node and the child link offset

Algorithm 2: On-demand traversal function following the traversal algorithm for GPUs [23]. The first intersected leaf is saved in *PostponedLeaf* to maximize SIMD occupancy during ray tracing. The parts of the algorithm needed for on-demand BVH construction are shown by red-tinged blocks.

```
1  Traversal(ray R, node N) begin
2    PostponedLeaf ← NULL;
3    if (N is leaf) then
4      PostponedLeaf ← N;
5      N ← pop(stack) ;
6    while (N is not NULL or
7          PostponedLeaf is not NULL) do
8      while (N is built inner node) do
9        Load N, intersect its children AABBs;
10       if (neither child intersected) then
11         N ← pop(stack);
12       else
13         if (left child intersected and flagged as
           unbuilt) then
14           Mark left child for building;
15           Translate left child address;
16         if (right child intersected and flagged
           as unbuilt) then
17           Mark right child for building;
18           Translate right child address ;
19         Determine nearchild from left and
           right child using the distance along
           the ray;
20         N ← nearchild;
21         if (both children intersected) then
22           push(farchild);
23       if (N is leaf and
24         PostponedLeaf is NULL) then
25         PostponedLeaf ← N;
26         N ← pop(stack);
27       if (all threads have postponed leaves) then
28         break;
29     while (PostponedLeaf is not NULL) do
30       Load geometric primitives and
         compute intersections;
31       if (N is leaf) then
32         PostponedLeaf ← N;
33         N ← pop(stack);
34       else
35         PostponedLeaf ← NULL;
36     if (N flagged as unbuilt for any thread) then
37       Save traversal state;
38       return ; // Suspend to building
39   Save ray intersection data to global memory;
40
```

where to access the result of the build, i.e. the final BVH link to the to-be-built node. The relationship between the node array and the task pool is shown in Figure 3.

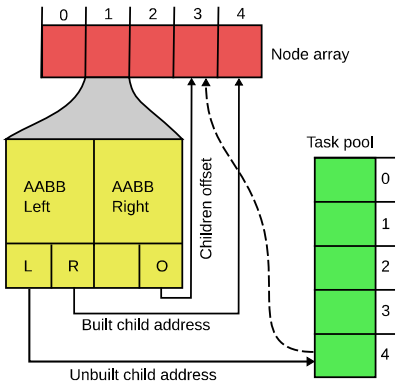

Fig. 3: Diagram of our node layout and the relations between node array and task pool. It depicts a parent node with the right child (pointer R) built and the left child (pointer L) waiting to be built. The O pointer always points to the address of the left child in the node array with the right child having index O+1.

After unlocking the nodes and translation of the node addresses, the traversal continues similarly to the original code, where the intersected child closer to the ray origin along the ray is chosen as the next node to be traversed.

The algorithm exits from the inner and outer traversal loops whenever an unbuilt node has to be traversed next (lines 37–38). In particular, after the inner while-while loops, the algorithm checks whether all rays (threads) want to traverse already built nodes. In that case the traversal continues with the next node. However, if there is at least one ray which aims to traverse an unbuilt node, the traversal is suspended for all rays and the warp proceeds with BVH building (see the next section).

Traversal restart. In order to restart a warp's suspended traversal, the algorithm periodically checks whether the nodes that caused the traversal stall have already been built. This check happens at two places: when the warp has finished the work chunks that it has dequeued from the task pool, and when it has found no work in one scan of the pool and is forced to search the pool from the beginning. The node is already built if its child link reloaded from the parent node no longer contains the unbuilt flag, at which point traversal is continued with the new address.

Terminating the computation. As the computation progresses, at some point there are no more rays to be processed in the ray array. To detect that the computation should terminate, we use a *termination counter* in global memory. The termination counter is initially set to the number of warps. The warp does not finish immediately upon detecting that there are no rays to be processed in the ray array, but instead it atomically decrements the termination counter. The warp then continues building tasks in the pool. When all warps are finished, i.e. the termination counter reaches zero, all warps quit and the kernel terminates. This delayed computation termination is implemented because some warps may still be tracing their rays and may require yet unbuilt nodes. The already finished warps help in building the nodes, which decreases the overall computation time.

3.4 Building BVH Nodes

The BVH traversal algorithm generates requests for constructing yet unbuilt nodes when they are reached. Construction of nodes is initialized for a warp when the BVH traversal of at least one ray from that warp is stalled due to a yet unbuilt node (lines 6-7 of Algorithm 1). Any unlocked node may be picked for construction because a warp may be stalled on up to 32 nodes and it is infeasible to try to build those sequentially.

Subdividing nodes. The BVH construction proceeds in top-down fashion by subdividing the nodes into smaller ones using the Surface Area Heuristic (SAH). The task of subdividing a node consists of three phases — splitting plane computation, partitioning of geometric primitives into left and right subsets, and bounding box computation (AABB). Note that computing the splitting plane using SAH also gives us the bounding boxes. The bounding box computation phase is only needed when SAH fails to subdivide the node and object median splitting is performed.

The on-demand BVH construction implicates several modifications to the full BVH building algorithm. The node array and the task pool have to be interconnected. In particular, an unbuilt node needs to keep a pointer to the task that builds it in order to unlock it, and the task needs to know which node it has built and how to link that node to its parent. Moreover, in the on-demand BVH build the node data are not only written into the node array, but also read from the array during the traversal in the same kernel launch, requiring synchronization. Special care is thus needed when enqueueing children of a node into the task pool and when unlocking a node.

Enqueueing items in the task pool. When inserting tasks into the task pool, we do not know whether their corresponding nodes will get traversed and whether they need to be built. Thus, the tasks are added to the task pool in a locked state, waiting for being unlocked when some ray aims to traverse the corresponding node. A task may never get unlocked if there is no request for building the node. The value used to represent a locked item is a large negative integer as mentioned in the previous section. In particular, we use a value composed of a negative flag, representing that the item is waiting to be unlocked by a ray, and the index into the node array for the node corresponding to the task. Thus, each lock value is unique

preventing an accidental unlocking of a different task when using atomic Compare-and-Swap.

3.5 Building Kd-trees

The kd-tree and on-demand kd-tree construction algorithms are similar to the above described algorithms for the BVH. The only notable difference is the need for dynamic memory allocations since geometric primitives may straddle the splitting planes and thus the number of interior nodes and leaves cannot be predicted in advance. We use the CMalloc memory allocator [24] optimized for GPUs to allocate triangle index arrays for the left and right children of the split node.

To make the kd-tree efficient, we had to change termination criteria for kd-trees built on the GPU architecture compared to those presented in [25] for the CPU. In addition to the maximum number of geometric primitives in a node to declare it a leaf ($n_{max} = 16$, leaf size criterion), we use the maximum depth computed as $k_1.log_2 N + k_2$ for $k_1 = 1.2$ and $k_2 = 2$ (hierarchy depth criterion). The third criterion, the failure of subdivision is then different to the CPU. The node is declared as a leaf upon the first subdivision failure (this corresponds to $F_{max} = 0$ in the paper [25]). We consider a failure if the ratio of the cost of subdividing a node to the cost of declaring the node a leaf is higher than $r_q^{min} = 0.9$.

Note that our implementation does not use split clipping [25] as the technique increases both the build time and the time to image that we target to minimize in this paper.

4 Optimizations

Below, we describe three optimization techniques for the basic algorithm presented in the previous section.

Empty task cache. When storing tasks in the task pool, an empty item has to be found. To accelerate finding of such items for the warps, we use a cache of fixed size which stores indices of recently emptied items in the task pool. These cached items are tested before a full scan of the pool is done accelerating the insertion of new tasks into the task pool. This optimization is beneficial for all methods including the ones that fully build the data structure.

Speculative unlocking. Computation stalls may occur when ray traversal is suspended because of a request for an unbuilt node and there is not enough parallel work in the requested nodes for all warps. To prevent these stalls, we propose a variant of our on-demand method, which unlocks all node building tasks immediately after their insertion into the task pool, increasing the amount of parallel work. These tasks may then be speculatively built in hope they will be later needed for the traversal.

Prioritization cache. In order to avoid the situation that the GPU gets overloaded with speculative construction, we use a simple priority mechanism to favor the subdivision of nodes that cause a traversal stall and thus should be built as soon as possible. To distinguish between requested nodes and speculative nodes, we use a cache of fixed size. In this cache, we store only the nodes requested during the traversal. The other nodes that were unlocked speculatively are not stored in the cache. A warp searching for a task to process first checks the tasks stored in this cache. Only if the warp fails in finding work in the cache, it searches for tasks by scanning the task pool. As long as there are requested nodes in the cache, these will be processed before the speculatively unlocked nodes in the pool.

5 Results

We have measured the behavior of the proposed algorithms on a PC with Intel Core i7-2600, 16 GBytes of RAM and NVIDIA GeForce GTX 680 running 64-bit version of Windows 7. All images in this paper were rendered at the resolution of 1024×1024 pixels and the reported data are averages over four viewpoints for each scene. Five runs are executed for each viewpoint and minimum running times are reported. Two (outer) viewpoints show majority of the scene geometry while for the other two viewpoints (close-up views) some detailed geometry is rendered. The images for these viewpoints are shown in Figure 4.

5.1 Methods

We have measured seven methods in total: fully built HLBVH [5] with $k = 4$ ($\mathrm{HLBVH_4}$), fully built BVH (BVH), on-demand BVH with building only the requested nodes ($\mathrm{BVH_R}$), on-demand BVH with speculative unlocking and prioritization cache ($\mathrm{BVH_A}$), fully built kd-tree ($\mathrm{Kd\text{-}tree}$), on-demand kd-tree with building only the requested nodes ($\mathrm{Kd\text{-}tree_R}$), and on-demand kd-tree with speculative unlocking and prioritization cache ($\mathrm{Kd\text{-}tree_A}$).

The subdivision terminates when the automatic SAH termination criteria are met (except in $\mathrm{HLBVH_4}$ where this is not possible) or when the number of geometric primitives drops below 16 (for both BVHs and kd-trees). This ad-hoc termination criterion was chosen as it minimized the times to render the frame for all methods. The termination criteria are optimized for minimizing the time to image, not for maximizing the traversal performance.

The size of empty task cache and prioritization cache described in Section 4 was set to 1024 entries. Note that the size of the prioritization cache must be large enough to hold all the nodes requested at any time in the $\mathrm{BVH_A}$ and $\mathrm{Kd\text{-}tree_A}$ algorithms. Otherwise, these nodes are not prioritized and the entire tree may be built before the algorithm finishes.

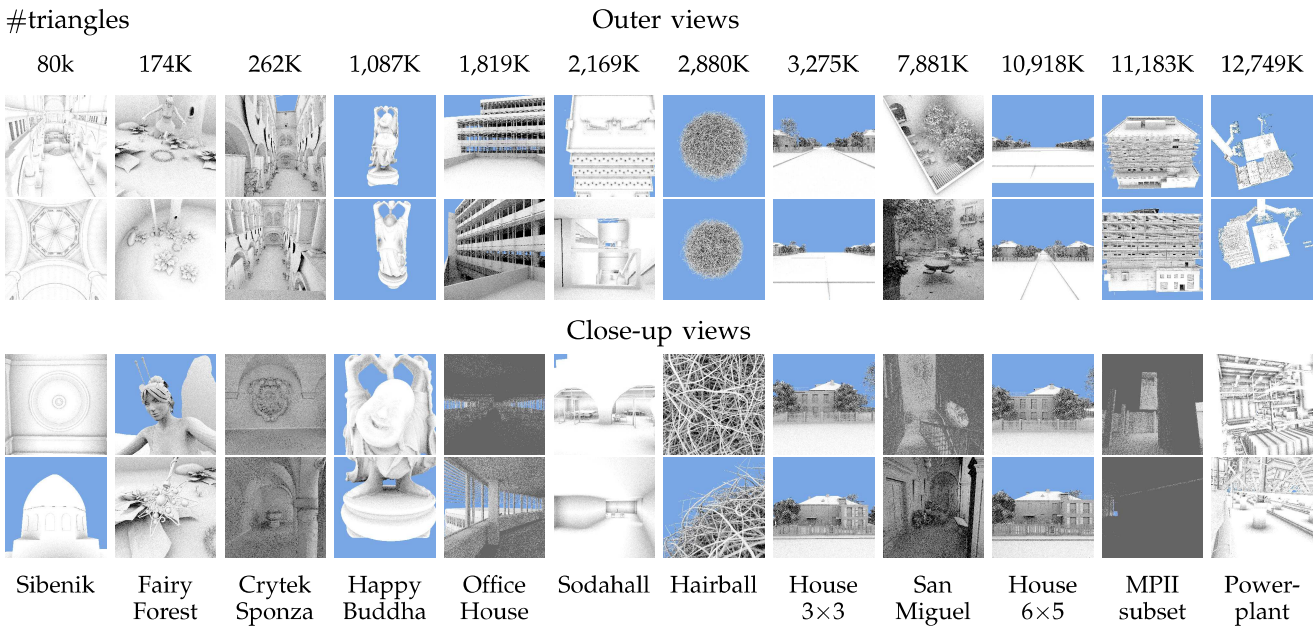

Fig. 4: Rendered images from twelve test scenes and four viewpoints. The image resolution was set to 1024×1024 pixels and eight ambient occlusion samples per primary ray were used. The MPII subset is composed of layers #1,4,7-10 and 12-13.

To compare fairly the on-demand methods with the ones that fully build the tree, ray generation and shading are handled by separate kernel calls and are not measured. For example, when rendering with eight ambient occlusion samples per primary ray using the full BVH, ten kernel calls are measured: BVH build, tracing primary rays, and eight times tracing ambient occlusion rays. For the on-demand methods one less kernel call is necessary as the build is done during primary and ambient occlusion rays tracing. It would be beneficial for the on-demand methods to trace all the ambient rays in a single batch. However, the ray data would then take a lot of memory allowing only small scenes to be rendered. We are using batching in the same way as for the methods with fully built hierarchy.

Unless otherwise stated, reading data from global memory bypasses the L1 caches by using a compiler switch. This is necessary for the pool based methods because the L1 caches are incoherent on the current hardware. Imagine a case where a single address is written by thread T1 and then read by thread T2 running on a different multiprocessor. If incoherent L1 cache is used, the value written by T1 may be written only into the L1 cache or T2 may read stale L1 cache data instead of the value written by T1. This leads to errors since warps from different multiprocessors may cooperate on building a single task, or may traverse nodes created by other warps. In our measurements, only the HLBVH build uses the L1 cache, other kernels are compiled to bypass the L1 cache.

5.2 Measurements

Tables 1 and 2 show a detailed comparison of the seven methods for tracing primary rays. The on-demand methods save significant number of operations used in geometric primitives sorting, in particular when shooting only the primary rays. For the $\mathrm{BVH_R}$ and $\mathrm{Kd\text{-}tree_R}$ methods, waiting for requested nodes to be built may incur stalls. These stalls are reduced in the $\mathrm{BVH_A}$ and $\mathrm{Kd\text{-}tree_A}$ methods at the cost of processing nodes that are not required. Building extra nodes is usually preferable when more of the scene geometry is traversed. Savings in the number of operations used in sorting strongly correlate with the decreases in build times. The Pearson correlation coefficient is 0.88 for the $\mathrm{BVH_A}$ and 0.98 for the $\mathrm{Kd\text{-}tree_A}$ methods. For the $\mathrm{BVH_R}$ and $\mathrm{Kd\text{-}tree_R}$, the correlations are weaker because of the stalls (0.76 and 0.81 respectively).

For kd-trees, where the traversal is immediately ended on the first hit, speculative building is less beneficial since less nodes are traversed. For the San Miguel and Powerplant scenes $\mathrm{HLBVH_4}$ has the fastest build, but only at the cost of significant slowdown in the ray tracing times.

These savings in computation time are significant for large scenes with viewpoints that do not show the entire geometry or highly occluded scenes (Sodahall, House 6×5, and Powerplant). The $\mathrm{Kd\text{-}tree_A}$ algorithm saves 89% of the time to image compared to the fully built kd-tree for the scene House 6×5 with primary rays. For smaller scenes with low occlusion, the proposed on-demand algorithms with speculative

Scene	T_{image}				$T_{primary}$				T_{build}				S_t		
	BVH	$\mathrm{BVH_R}$	$\mathrm{BVH_A}$	$\mathrm{HLBVH_4}$	BVH	$\mathrm{BVH_R}$	$\mathrm{BVH_A}$	$\mathrm{HLBVH_4}$	BVH	$\mathrm{BVH_R}$	$\mathrm{BVH_A}$	$\mathrm{HLBVH_4}$	BVH	$\mathrm{BVH_R}$	$\mathrm{BVH_A}$
	[ms]	[%]	[%]	[%]	[ms]	[%]	[%]	[%]	[ms]	[%]	[%]	[%]	[-]	[%]	[%]
Sibenik	17.8	+69	-31	+141	5.0	+2	+4	+6	12.7	+96	-45	+195	1.3	-54	-10
Fairy Forest	22.8	+62	-13	+121	7.1	-1	+0	+25	15.6	+92	-18	+165	2.8	-30	-11
Crytek Sponza	29.7	+59	-13	+67	7.8	-4	-1	+0	21.7	+82	-16	+93	4.6	-42	-19
Happy Buddha	73.0	-0	-19	+11	5.0	+20	+22	+32	67.9	-2	-22	+10	18.4	-38	-29
Office House	151.5	-32	-52	-27	15.9	+11	+14	+53	135.5	-37	-59	-37	39.2	-66	-61
Sodahall	156.2	-63	-65	-36	3.8	+13	+18	+5	152.3	-65	-67	-37	45.0	-73	-69
Hairball	206.6	-16	-31	-20	16.5	+18	+20	+24	190.1	-19	-35	-24	54.4	-46	-39
House 3×3	232.3	-55	-59	-42	6.8	+13	+15	+44	225.4	-57	-61	-44	66.4	-65	-63
San Miguel	621.1	-45	-53	-58	17.2	+9	+10	+47	603.8	-46	-55	-61	181.8	-57	-56
House 6×5	804.5	-69	-71	-55	9.1	+13	+14	+81	795.4	-70	-71	-57	240.0	-72	-71
MPII subset	817.7	-66	-67	-62	7.8	+6	+6	+37	809.8	-66	-68	-63	247.2	-67	-66
Powerplant	1027.3	-58	-59	-66	16.0	+14	+15	+77	1011.2	-59	-61	-68	310.6	-59	-58
Average	346.7	-18	-44	-2	9.8	+10	+11	+36	336.8	-13	-48	+6	101.0	-56	-46

TABLE 1: Comparison of four methods for building BVHs. T_{image} is the time to image (build time plus time for tracing 1024×1024 primary rays). $T_{primary}$ is the time for tracing primary rays. T_{build} is the build time ($T_{image} - T_{primary}$ for on-demand methods) and S_t is the number of operations used in geometric primitives sorting in millions (not reported for HLBVH as it is pointless for this method). Fully built BVH is the reference method (+0%) presenting measured values. For other methods, the difference as signed percentage is reported. Averages over the tabled data are given in the last row.

Scene	T_{image}			$T_{primary}$			T_{build}			S_t		
	Kd-tree	$\mathrm{Kd\text{-}tree_R}$	$\mathrm{Kd\text{-}tree_A}$	Kd-tree	$\mathrm{Kd\text{-}tree_R}$	$\mathrm{Kd\text{-}tree_A}$	Kd-tree	$\mathrm{Kd\text{-}tree_R}$	$\mathrm{Kd\text{-}tree_A}$	Kd-tree	$\mathrm{Kd\text{-}tree_R}$	$\mathrm{Kd\text{-}tree_A}$
	[ms]	[%]	[%]	[ms]	[%]	[%]	[ms]	[%]	[%]	[-]	[%]	[%]
Sibenik	22.9	+41	-9	6.7	+1	+10	16.2	+58	-17	2.1	-64	-12
Fairy Forest	36.9	+34	+3	12.0	+8	+8	24.7	+48	+2	4.8	-42	-7
Crytek Sponza	47.9	+10	-12	9.0	+11	+11	38.8	+10	-17	7.5	-61	-32
Happy Buddha	183.3	-34	-19	10.1	+19	+21	173.1	-37	-21	30.1	-54	-34
Office House	231.9	-60	-62	20.4	+1	+0	211.3	-66	-68	60.0	-76	-71
Sodahall	326.3	-81	-79	5.5	+15	+15	320.8	-82	-81	71.5	-83	-77
Hairball	582.5	-45	-40	56.8	+2	+1	525.6	-50	-45	121.6	-68	-55
House 3×3	650.5	-79	-77	9.5	+18	+16	641.0	-81	-78	113.0	-77	-72
San Miguel	1457.2	-64	-68	24.9	+9	+7	1432.2	-65	-69	272.1	-70	-67
House 6×5	2594.7	-88	-89	9.6	+16	+15	2585.0	-89	-89	397.3	-82	-81
MPII subset	2167.9	-85	-84	7.9	+10	+6	2159.8	-86	-84	412.8	-82	-80
Powerplant	2345.6	-78	-80	18.5	+6	+4	2327.0	-79	-81	503.6	-75	-74
Average	887.3	-44	-51	15.9	+10	+9	871.3	-43	-54	166.4	-69	-55

TABLE 2: Comparison of three methods for building Kd-trees. T_{image} is the time to image (build time plus time for tracing 1024×1024 primary rays). $T_{primary}$ is the time for tracing the primary rays. T_{build} is the build time ($T_{image} - T_{primary}$ for on-demand methods) and S_t is the number of operations used in geometric primitives sorting in millions. Fully built Kd-tree is the reference method (+0%) presenting measured values. For other methods, the difference as signed percentage is reported. Averages over data are given in the last row.

unlocking can slightly increase the computation time (3.3% for the $\mathrm{Kd\text{-}tree_A}$ on Fairy Forest with primary rays). This behavior for smaller scenes is not surprising as the algorithm is designed for highly occluded and large scenes.

The build times for kd-trees are much higher than for the BVHs. This is caused by the dynamic allocations of auxiliary arrays containing the lists of geometric primitives and the duplication of primitives that leads to higher numbers of operations used in geometric primitives sorting. Interestingly, in most cases the traversal times are higher as well. We have measured the traversal statistics for the three fully built data structures ($\mathrm{HLBVH_4}$, BVH, and Kd-tree) as shown in Table 3. While the number of intersection tests is lower for the BVH than for the kd-tree on average, the number of traversal steps is higher for the BVH. This is due to the speculative BVH traversal and lower occlusion culling efficiency of the BVH [26].

Table 4 shows the behavior of the methods on the collision detection rays with $\mathrm{HLBVH_4}$ as the reference method. In this test, 100 line segment paths

Scene	N_{it} [−]			N_{ts} [−]		
	HLBVH$_4$	BVH	Kd-tree	HLBVH$_4$	BVH	Kd-tree
Sibenik	22.2	27.8	35.5	62.3	56.6	31.9
Fairy Forest	26.5	27.6	53.7	61.1	56.4	43.1
Crytek Sponza	19.3	30.8	45.9	94.5	87.4	40.4
Happy Buddha	14.1	12.3	23.2	34.4	31.0	25.1
Office House	78.7	90.3	134.8	150.4	129.8	40.9
Sodahall	10.4	21.7	40.5	66.7	57.7	31.0
Hairball	44.8	37.2	133.4	99.0	94.3	62.7
House 3×3	20.3	18.1	32.8	44.9	39.9	24.9
San Miguel	36.7	42.7	54.6	134.2	125.0	67.0
House 6×5	33.6	22.9	27.9	57.6	48.5	28.6
MPII subset	22.4	24.5	28.6	103.9	77.3	29.2
Powerplant	77.4	60.6	76.4	146.1	107.8	43.7
Average	33.9	34.7	57.3	87.9	76.0	39.0

TABLE 3: Traversal statistics per ray for the three data structures. N_{it} is the number of triangle intersections and N_{ts} is the number of traversal steps. The data are for ray tracing 1024×1024 primary rays ($T_{primary}$, T_{build}, and S_t are given in Tables 1 and 2). Averages over the tabled data are given in the last row.

Scene	T_{AVG}						
	HLBVH$_4$	BVH	BVH$_R$	BVH$_A$	Kd-tree	Kd-tree$_R$	Kd-tree$_A$
	[ms]	%	%	%	%	%	%
Sibenik	36.3	-67	-65	-80	-59	-63	-66
Fairy Forest	39.2	-61	-72	-77	-32	-69	-68
Crytek Sponza	40.4	-49	-53	-61	+9	-59	-52
Happy Buddha	71.5	-5	-48	-51	+147	-42	-34
Office House	86.1	+57	-34	-40	+141	-41	-32
Sodahall	96.1	+59	-29	-31	+231	-33	-32
Hairball	147.9	+28	-16	-27	+253	-29	-21
House 3×3	127.8	+77	-20	-25	+400	-24	-23
San Miguel	236.6	+155	-7	-8	+505	-16	-17
House 6×5	344.4	+132	-11	-13	+660	-17	-18
MPII subset	304.6	+169	-3	-4	+628	-14	-17
Powerplant	323.0	+214	-32	-33	+621	-28	-32
Average	154.5	+59	-32	-38	+292	-36	-34

TABLE 4: Comparison of all the methods on collision detection rays. T_{AVG} is the average time per frame for an animation long 20 frames. In each frame, the data structure is built from scratch and collision detection rays are traced (8 rays for each of the 100 agents) for the current agent's position. Averages over the tabled data are given in the last row.

are randomly placed inside the bounding box of the scene. A hypothetical agent moves along each of the paths: in each frame 8 rays are shot from the agent's position into the sphere around it. For the next frame, the agent's position is updated as if the agent has moved. The collision detection rays are bounded by the distance between two points on the path and the closest hit is returned for each ray.

Only a fraction of the scene is needed for the traversal of collision detection rays in each frame because of their number and length. In such scenario, the on-demand methods are more efficient and are always faster even than the fast HLBVH$_4$ method. Interestingly, the performance of the on-demand BVH and on-demand kd-tree methods is similar to each other in this test. This is likely caused by the ability of kd-tree traversal to terminate upon the first hit.

The ability of the on-demand methods to efficiently cull the hidden geometry is clearly shown in Figure 5. For this test, an increasing number of copies of the House model were put in a row in front of the camera. For more than two copies, the added geometry did not influence the final image. This is shown in the graphs for the number of nodes. While the number of nodes grows linearly for the methods that fully build the tree, it stays constant for the on-demand methods. The same behavior is shown in the graph for reference ratio (number of geometric primitive references divided by the number of geometric primitives). This ratio quickly drops near zero for the on-demand methods, since the same number of references is needed regardless of the increase in the number of geometric primitives. The number of operations used in sorting geometric primitives grows linearly for all methods but at different rates. For the on-demand methods, this is caused by processing more geometric primitives near the root while for the fully built hierarchies, more geometric primitives are processed at each level.

While increasing the amount of hidden geometry has only a small impact on the on-demand methods, increasing the amount of visible geometry has a direct influence. To capture this behavior, we have varied the number of diffuse samples used for rendering the image. Figure 6 shows the typical behavior of the methods on more complex scenes. When the number of diffuse samples is low, the on-demand methods can save a significant amount of the build time by skipping the geometry that is not visible. For such scenarios, they often outperform even the fast HLBVH build. As the number of diffuse samples and thus the visible part of the scene increases, the performance of the on-demand methods gradually degrades to the performance of the full build. Eventually, the full build prevails because of its faster traversal as

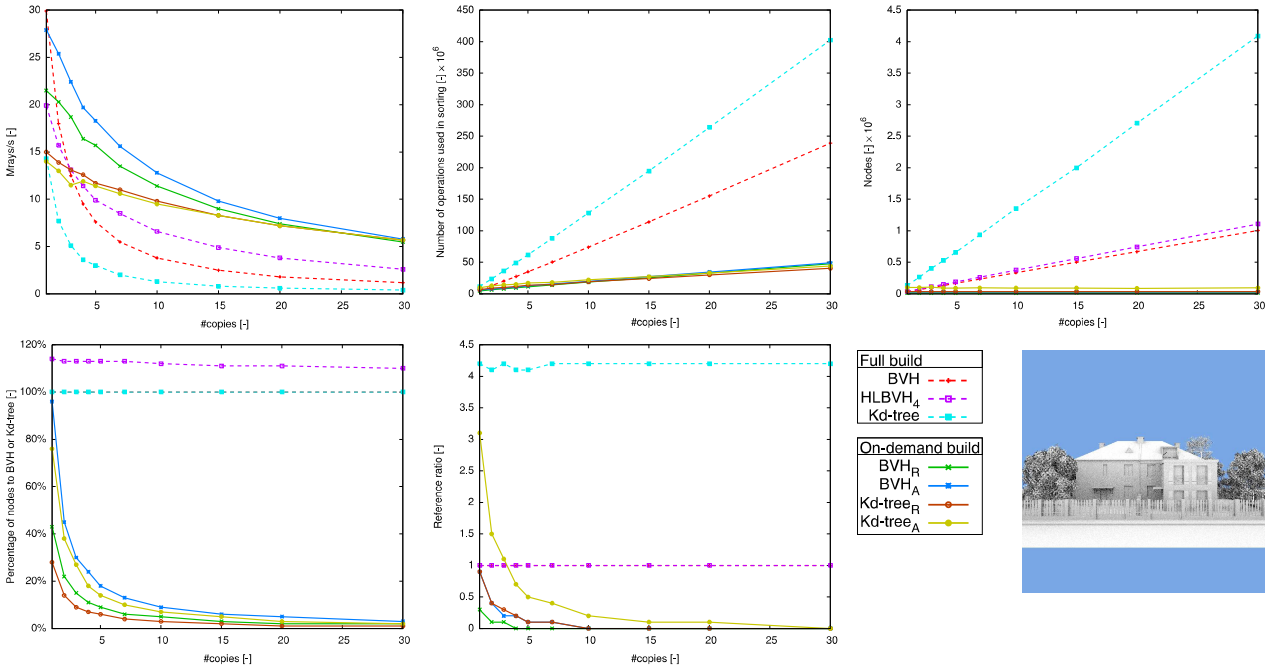

Fig. 5: The influence of increasing the amount of hidden geometry when tracing primary rays. Varying number of copies of the scene "House" are arranged in a row such that most of the view is taken by the front house (bottom-right image). The graphs show in the reading order: the performance of the methods (computed from T_{image} in Mrays/s), number of operations used in geometric primitives sorting in millions, number of nodes in millions, percentage of the number of nodes to the fully built reference hierarchy, and the reference ratio (number of geometric primitive references divided by the number of geometric primitives). For graphs with ratio or percentage, BVH method is the reference for BVHs and Kd-tree method is the reference for kd-trees.

elaborated in the following section. The cross-over points of the on-demand and full build methods are different for BVHs and kd-trees. This is caused by the longer build time of kd-trees and their ability to terminate traversal on the first hit without checking the nodes on the traversal stack as for BVHs.

We have also measured the performance of the compared methods for animated datasets. For each frame the data structure is completely rebuilt to show the performance that can be expected in the presence of arbitrary animations. Table 5 shows the average per frame time for the primary, ambient occlusion and diffuse rays. Since the efficiency of the on-demand methods is dependent on the occlusion present in the rendered scene, we use 3×3 instances of the same dataset of the animated models. While the on-demand methods perform rather poorly on the Fairy Forest scene, where almost the entire scene is built, for the 3×3 instances they perform very well even for the incoherent diffuse rays. This is caused by the large occluding wall in the Fairy Forest scene.

For the Dragon Bunny scene on-demand methods are less beneficial because there is almost no occlusion present after the dragon explodes. The situation is similar for the Break Lion scene. The on-demand methods based on the kd-tree are, however, better suited for exploiting the occlusion still present in the rubble after the lion has collapsed leading to a higher speedup. On average the on-demand methods are more beneficial for coherent primary rays and ambient occlusion rays with limited length because fewer of the data structure nodes are traversed and thus built for these ray distributions.

The entire course of the selected animations can be seen in Figure 7. For the Fairy Forest 3×3 scene the performance stays almost constant because the other instances are effectively culled away for every frame. For the Dragon Bunny 3×3 scene a steady decline of the performance of the on-demand methods can be seen as the dragon explodes.

We also compare the performance of our GPU kd-tree builder with the state-of-the-art method of Roccia et al. [16] (referred to as Roccia's method) in Table 6. The results from the table are not directly comparable since they were measured on two different GPU models. Given our kernels are memory-limited, we can estimate the performance difference from memory bandwidths. GeForce GTX 680 has 25% or 50% higher memory bandwidth than GTX 560 (depending on the version of the GTX 560). Even after taking these ratios into account, the build times for our method are still significantly smaller than for the Roccia's method. The traversal performance on the other hand is roughly the same since both the CPU and the GPU are used for ray tracing in the Roccia's method. On the Sodahall scene, the traversal performance of our

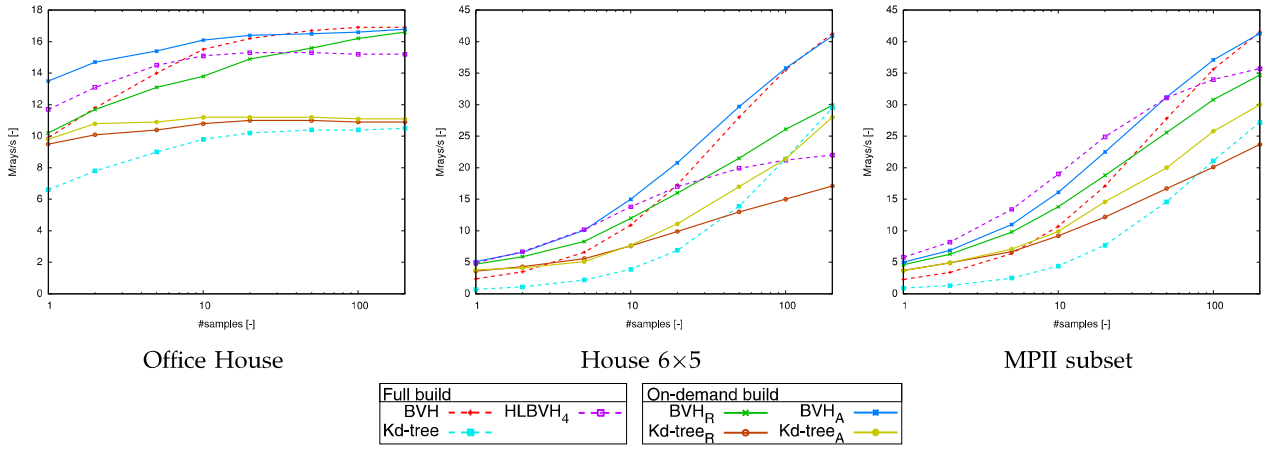

Fig. 6: The influence of increasing the number of diffuse samples on the rendering performance (computed from T_{image} in Mrays/s) when tracing diffuse rays. The results are shown for fully and on-demand built hierarchies on three test scenes with higher occlusion.

Scene	Ray type	T_{AVG}						
		BVH	BVH_R	BVH_A	$HLBVH_4$	Kd-tree	$Kd\text{-}tree_R$	$Kd\text{-}tree_A$
		[ms]	%	%	%	[ms]	%	%
Fairy Forest	P	22.5	+69	-4	+120	34.9	+61	+12
	AO	56.4	+49	+12	+55	96.7	+42	+19
	D	168.6	+14	+2	+22	379.2	+25	+19
Fairy Forest 3×3	P	125.6	-52	-59	-24	202.0	-63	-69
	AO	190.8	-27	-31	+2	303.4	-33	-38
	D	327.9	-16	-20	+24	557.2	-8	-11
Dragon Bunny 3×3	P	164.3	-26	-29	-38	258.8	-16	-25
	AO	192.0	-9	-17	-30	303.5	+9	-9
	D	212.1	+62	+14	-22	362.8	+112	+21
Break Lion 3×3	P	1104.6	-41	-49	-60	11099.1	-88	-93
	AO	1191.9	-28	-40	-55	11303.6	-81	-91
	D	1294.5	+82	-2	-45	12079.1	-23	-83
Avg	P	354.3	-13	-35	-1	2898.7	-27	-44
	AO	407.8	-4	-19	-7	3001.8	-15	-30
	D	500.8	+35	-2	-5	3344.6	+27	-14

TABLE 5: Comparison of all seven methods for animated data. T_{AVG} is the average time to image per frame for a 100 frame long animation. In each frame, the data structure is built from scratch. Results for primary (P), ambient occlusion (AO) and diffuse (D) rays are shown in separate rows for each scene (8 samples were used for ambient occlusion and diffuse rays). Fully built BVH and Kd-tree are the reference methods (+0%) presenting measured values. For other methods, the difference as signed percentage is reported. Averages over the tabled data are given at the bottom of the table.

method is significantly higher, which may indicate a scaling problem in the Roccia's method.

Finally, we briefly compare our method to the state-of-the-art commercial ray tracers on both the CPUs (Embree) and GPUs (OptiX). The Embree [27] achieves similar rendering performance for primary rays on two Sandy Bridge CPUs (234 Mrays/s) on the Asian Dragon model as our full BVH builder when optimized for maximal rendering performance (220.7 Mrays/s) on the GTX TITAN Black. The build performances are also comparable, start times of 0.7s – 1.5s compared to just the build time of 0.29s for our BVH builder. The OptiX framework features the method of Karras and Aila [11] and achieves build 30 to 60% faster than our full BVH builder, but about 40% slower than our on-demand BVH_A builder when using primary rays. The rendering performance is higher for OptiX, which is probably due to exploiting L1 cache and constructing BVH for maximum ray tracing performance.

5.3 Discussion

Time complexity analysis. Here, we analyze the properties of the ray tracing algorithms with building hierarchies in on-demand fashion using the cost model. The standard method fully building a data structure over n geometric primitives and tracing R rays will need total computation time $T_{full} = c_B.n.\log_2 n + c_T.R.\log_2 n$, where c_B is the cost for a single build step and c_T is the cost for a traversal step. The on-demand algorithm results in different time $T_{ondemand} = c_B.n'.\log_2 n' + n'.c_{switch} + c'_T.R.\log_2 n$. We assume here that the constant c_B is the same for both solutions and that the actively visited parts of the data structure are built in the same way, therefore visiting the same number of nodes (factor $R.\log_2 n$). In on-demand building, the number of nodes built will only be n' ($n' \leq n$ and for highly occluded scenes $n' \ll n$).

Scene	T_{build} [ms]						$P_{diffuse}$ [Mrays/s]			
	Roccia	Roccia	Roccia	NoDup(16)	DupClip(16)	DupClip(8)	Roccia	NoDup(16)	DupClip(16)	DupClip(8)
	CPU^A	GPU^A	hybrid	GPU^B	GPU^B	GPU^B	hybrid	GPU^B	GPU^B	GPU^B
Dragon	770.0	2670.0	430.0	137.5	155.4	265.5	108.5	72.3	106.4	111.4
Happy Buddha	920.0	2880.0	500.0	173.2	199.3	348.7	112.3	68.3	104.2	105.5
Sodahall	2680.0	2240.0	1030.0	314.6	355.5	443.9	56.0	72.7	101.5	112.7

TABLE 6: Comparison of six methods for building kd-trees. Three methods of Roccia et al. [16], fully built kd-tree without duplicating primitives (NoDup(16)), fully built kd-tree with primitive duplication and triangle clipping during partition (DupClip(16), DupClip(8)). The numbers in brackets give maximum number of triangles per leaf. T_{build} is the build time in milliseconds and $P_{diffuse}$ is the performance in Mrays/s for tracing four diffuse rays per pixel. CPU^A is Intel Core i7 920, GPU^A is GeForce GTX 560, hybrid is combination of both, and GPU^B is GeForce GTX 680.

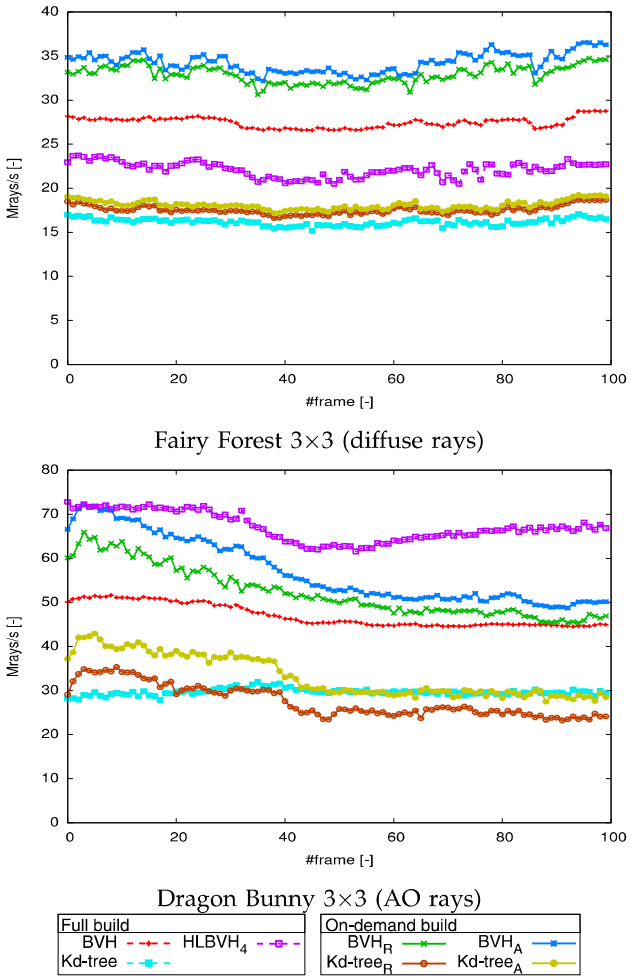

Fig. 7: The performance of the methods (computed from T_{image} in Mrays/s) during the 100 frame long animation.

The on-demand traversal algorithm has some overhead as it contains more conditions to be evaluated, therefore $c'_T > c_T$. The constant c_{switch} represents the overhead of switching between the traversal and building in the on-demand algorithm. Assuming the c_{switch} can be neglected for higher number of rays, we can compare T_{full} and $T_{ondemand}$ and provide the following analysis: (a) with number of rays R going to infinity the fully built data structure is faster, (b) for highly occluded scene or detailed views of the scenes with $n' \ll n$ the on-demand algorithm is faster, (c) for $n' \approx n$ and R not large both algorithms will result in a similar performance. This behavior is shown in the measurements.

On-demand algorithm utility. From the results measured on complex scenes (Figures 5 and 6), we can observe that the BVH_A on-demand method provides the most stable performance across a wide range of the number of cast rays. For a small number of rays the BVH_A is usually close to the fast build $HLBVH_4$ method, while when the number of rays is large, it is close to the high quality BVH method. Although there is a variable size interval of the number of cast rays in which the BVH_A method achieves the highest performance, overall, it is the best method in the comparisons when considering both smaller and higher numbers of cast rays. This is advantageous when the number of rays is not known in advance.

Hardware limitations. When comparing the performance of on-demand methods with methods fully building the acceleration data structure, there are several hardware limitations that must be considered. We have already pointed out the incoherence of L1 caches on current generations of the NVIDIA GPUs. Disabling L1 caching using a compiler switch (but identical source code) produces about 20% slower traversal code. If we could enable L1 cache, the on-demand methods could have faster traversal than the full build methods because the computations are more localized and thus should have a higher L1 hit ratio. This problem is not inherent to the parallel on-demand methods but it is hardware dependent.

Combining build and traversal codes into one kernel leads to a longer code, often demanding more resources for that kernel than for the separate kernels. In our kernels, this leads to registers spilling to local memory. This problem may be mitigated by a more optimized implementation. Moreover, launching the on-demand kernel just for traversing rays using already built data structure has higher launch overhead than launching a specialized traversal kernel. This is the reason why the on-demand traversal is slower

than traversing fully built hierarchy.

BVH vs kd-tree. The space subdivision property of kd-tree is beneficial for on-demand build. As nodes on the traversal stack at the time of the first intersection need not be traversed, they neither need to be built. Compared to the BVH, this saves building nodes that do not contribute to the final image. Given our BVH traversal codes employ speculative traversal [23], the reduction in the built nodes is further amplified.

We have observed an interesting behavior in the on-demand kd-trees that the speedup in the build time compared to the full kd-tree can be higher than the ratios of number of operations used in sorting geometric primitives. This is likely caused by requesting fewer dynamic memory allocations, which is also a major cost in the kd-tree build.

Ray distributions. The performance of the on-demand methods depends on the ray distribution in space. When majority of the scene geometry is accessed, there is little potential benefit of the on-demand methods. On the other hand, when only a fraction of the scene geometry is intersected by shot rays, for example when a close-up view is taken, large parts of the hierarchy may be left unbuilt. For large scenes, the number of operations used in sorting was 40 to 60% smaller than for the full build, indicating the method's potential for rendering massive data sets. The on-demand methods can also take advantage of the ray length, as commonly defined in shadow or ambient occlusion rays.

We have measured our results on primary, ambient occlusion, diffuse and collision detection rays to study the dependence of the on-demand methods on ray distribution. The correlation coefficient between the number of operations used in sorting geometric primitives and build times stayed similar for all distributions. On-demand methods performed best on primary and collision detection rays where the coherency of the rays or their number and length caused large parts of the scenes not to be traversed.

There is also an interesting relation between ambient occlusion rays and diffuse rays which differ only in the ray length. This accounts for an average saving of 25% in the build time for the bounded ambient occlusion rays. For small scenes and diffuse rays, the entire scene or its majority was often traversed rendering the on-demand methods useless.

Memory. For the on-demand builders, less memory is consumed by the acceleration data structure since some parts of the tree are not traversed and built. Moreover, if geometric primitives are converted to some efficient traversal format such as Woop representation [28], savings in triangle data are also possible. For large scenes less than 5% of nodes and references to geometric primitives are created (see Figure 5). Currently, all GPU memory has to be preallocated from the CPU side including the heap for the dynamic memory allocator. Thus, the same amount of preallocated memory for node and triangle arrays has to be used as in the full BVH or kd-tree build, regardless of the actual size of the output. However, with GPU hardware steadily progressing towards more programmability, the ability to save memory through dynamic allocation can become a major advantage of the on-demand methods.

6 Conclusions

We have proposed, implemented and tested a novel proof-of-concept algorithm for the simultaneous ray tracing and on-demand BVH/kd-tree construction on the GPUs. This algorithm solves the computational dependencies and load balancing between building the data structure and tracing the rays. We have also presented a SAH kd-tree builder, that outperforms the previous state-of-the-art approach.

Considering primary rays, the speculative on-demand algorithms ($\mathrm{BVH_A}$ or $\mathrm{Kd\text{-}tree_A}$) can save up to 89% of the time to image compared to the fully built tree on large scenes with high occlusion. On average 50% reduction in rendering time is achieved over the twelve tested scenes. We expect that the time to image savings will become even higher for the future cache-coherent GPU architectures.

The proposed algorithm can also save memory needed by the acceleration data structure as only a small portion of the scene is processed during the traversal. In highly occluded scenes, the memory requirements decrease from 100% for the fully built data structure to less than 5% for the one built on-demand.

In future work, we plan to extend the method for building hierarchies with potentially higher quality, but slower build times such as the general BSP-trees. Combining the method with an out-of-core geometry loading capability would be another interesting avenue for research. This would allow rendering massive scenes while fitting only their visible subset into the GPU memory.

Acknowledgments

We would like to thank Marko Dabrovic for Sibenik model, DAZ3D (www.daz3d.com) for Fairy Forest model, Frank Meinl at Crytek for the Crytek Sponza model, Prof. C. Séquin for Sodahall model, Samuli Laine and Tero Karras for Hairball model, Guillermo M. Leal Llaguno for San Miguel model, the UNC for Powerplant model, project [29] for MPII model, and Stanford repository for other 3D models.

We would also like to thank Tero Karras, Timo Aila, and Samuli Laine for releasing their GPU ray tracing framework. Our research was partially supported by the Czech Science Foundation under research programs P202/11/1883 (Argie) and P202/12/2413 (Opalis) and the Grant Agency of the Czech Technical University in Prague, grant No. SGS13/214/OHK3/3T/13.

REFERENCES

[1] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," *Computer Graphics Forum*, vol. 28, no. 2, pp. 375–384, 2009.

[2] J. Pantaleoni and D. Luebke, "HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry," in *Proceedings of HPG 2010*. Saarbrucken, Germany: ACM SIGGRAPH/Eurographics, 2010, pp. 87–95.

[3] K. Garanzha, J. Pantaleoni, and D. McAllister, "Simpler and Faster HLBVH with Work Queues," in *Proceedings of HPG 2011*. Vancouver, British Columbia, Canada: ACM SIGGRAPH/Eurographics, 2011, pp. 59–64.

[4] T. Karras, "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees," in *Proceedings of HPG 2012*. Paris, France: ACM SIGGRAPH/Eurographics, June 2012, pp. 33–37.

[5] K. Garanzha, S. Premože, A. Bely, and V. Galaktionov, "Grid-based SAH BVH construction on a GPU," *The Visual Computer*, vol. 27, pp. 697–706, Jun. 2011.

[6] I. Wald, "Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 1, pp. 47–57, January 2012.

[7] M. J. Doyle, C. Fowler, and M. Manzke, "A Hardware Unit for Fast SAH-optimised BVH Construction," *ACM Transactions on Graphics*, vol. 32, no. 4, pp. 139:1–139:10, Jul. 2013.

[8] D. Kopta, T. Ize, J. Spjut, E. Brunvand, A. Davis, and A. Kensler, "Fast, effective BVH updates for animated scenes," in *Proceedings of the I3D conference*. New York, NY, USA: ACM, 2012, pp. 197–204.

[9] Y. Gu, Y. He, K. Fatahalian, and G. Blelloch, "Efficient BVH Construction via Approximate Agglomerative Clustering," in *Proceedings of HPG 2013*. New York, NY, USA: ACM SIGGRAPH/Eurographics, July 2013, pp. 81–88.

[10] J. Bittner, M. Hapala, and V. Havran, "Fast Insertion-Based Optimization of Bounding Volume Hierarchies," *Computer Graphics Forum*, vol. 32, no. 1, pp. 85–100, 2013.

[11] T. Karras and T. Aila, "Fast Parallel Construction of High-Quality Bounding Volume Hierarchies," in *Proceedings of HPG 2013*. New York, NY, USA: ACM SIGGRAPH/Eurographics, July 2013, pp. 89–99.

[12] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time KD-tree Construction on Graphics Hardware," in *SIGGRAPH Asia 2008*. New York, NY, USA: ACM, 2008, pp. 126:1–126:11.

[13] P. Danilewski, S. Popov, and P. Slusallek, "Binned SAH Kd-Tree Construction on a GPU," Computer Graphics Group, Saarland University, Tech. Rep., June 2010.

[14] Z. Wu, F. Zhao, and X. Liu, "SAH KD-tree Construction on GPU," in *Proceedings of HPG 2011*. New York, NY, USA: ACM SIGGRAPH/Eurographics, 2011, pp. 71–78.

[15] 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 IEEE/Eurographics Symposium on Interactive Ray Tracing 2006*. Washington, DC, USA: IEEE Computer Society, sep. 2006, pp. 61–69.

[16] J.-P. Roccia, M. Paulin, and C. Coustet, "Hybrid CPU/GPU KD-Tree Construction for Versatile Ray Tracing," in *Eurographics (Short Papers)*. Eurographics Association, 2012, pp. 13–16.

[17] D. Hook and K. Forward, "Using KD-trees to Guide Bounding Volume Hierarchies for Ray Tracing," *Australian Computer Journal*, vol. 27, no. 3, pp. 103–108, Aug. 1995.

[18] S. Ar, G. Montag, and A. Tal, "Deferred, Self-Organizing BSP Trees," *Computer Graphics Forum (Proceedings of Eurographics 2002)*, vol. 21, no. 3, pp. 269–278, 2002.

[19] C. Waechter and A. Keller, "Instant Ray Tracing: The Bounding Interval Hierarchy," in *Proceedings of EGSR 2006*. Nicosia, Cyprus: Eurographics, 2006, pp. 139–149.

[20] M. Steinberger, B. Kainz, B. Kerbl, S. Hauswiesner, M. Kenzel, and D. Schmalstieg, "Softshell: dynamic scheduling on GPUs," *ACM Transactions on Graphics*, vol. 31, no. 6, pp. 161:1–161:11, Nov. 2012.

[21] M. Vinkler, J. Bittner, V. Havran, and M. Hapala, "Massively Parallel Hierarchical Scene Processing with Applications in Rendering," *Computer Graphics Forum*, vol. 32, no. 8, pp. 13–25, 2013.

[22] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[23] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," in *Proceedings of HPG 2009*. New Orleans, Louisiana: ACM SIGGRAPH/Eurographics, 2009, pp. 145–149.

[24] M. Vinkler and V. Havran, "Register Efficient Memory Allocator for GPUs," in *Proceedings of HPG 2014*. Lyon, France: Eurographics Association, 2014, pp. 19–27.

[25] V. Havran and J. Bittner, "On Improving KD-Trees for Ray Shooting," *Journal of WSCG*, vol. 10, no. 1, pp. 209–216, February 2002.

[26] M. Vinkler, V. Havran, and J. Bittner, "Bounding Volume Hierarchies versus Kd-trees on Contemporary Many-Core Architectures," in *Proceedings of the 30th Spring Conference on Computer Graphics*, ser. SCCG '14. New York, NY, USA: ACM, 2014, pp. 61–68.

[27] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree: A Kernel Framework for Efficient CPU Ray Tracing," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 143:1–143:8, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2601097.2601199

[28] S. Woop, "A Ray Tracing Hardware Architecture for Dynamic Scenes," Master's thesis, Saarland University, Saarbruecken, Germany, March 2004.

[29] V. Havran, J. Zajac, J. Drahokoupil, and H.-P. Seidel, "MPI Informatics Building Model as Data for Your Research," MPI Informatik, Saarbruecken, Germany, Research Report MPI-I-2009-4-004, Dec 2009.

Marek Vinkler received his Ph.D. degree from the Masaryk University in Czech Republic. He is currently a a postdoctoral research associate at the Max Planck Institute for Informatics, Germany. His primary research interests are high-performance ray tracing, data structure build and parallel programming on many-core processors.

Vlastimil Havran received his Ph.D. degree from the Czech Technical University in Prague. After his Ph.D., he was a postdoctoral research associate at the Max Planck Institute for Informatics in Saarbruecken, Germany. Currently he works as an associate professor at the Czech Technical University in Prague. His research interests rendering topics focused on data structures and geometric range searching.

Jiří Bittner received his Ph.D. degree from the Czech Technical University in Prague. Then he worked as post-doctoral associate at the Vienna University of Technology. Currently he works as an associate professor at the Czech Technical University in Prague. His research interests include data structures for ray tracing and visibility calculations.

Jiří Sochor received his Ph.D. in digital computers from the Czech Technical University in Prague. Currently he works as an associate professor at the Masaryk University in Czech Republic where he leads the Human Computer Interaction research group. His research interests include computer graphics, virtual reality and human-computer interaction.