Fast Construction of SAH-based Bounding Interval Hierarchies using Dynamic Parallelism

Carl-Feofan Matthes, Adrian Kreskowski, Andre Schollmeyer, Bernd Froehlich

Abstract: We present a fast and efficient algorithm for the construction of a SAH-based BIH on the GPU. Our approach uses a novel asynchronous processing scheme which launches kernels for the necessary subtasks directly on the GPU. This avoids the communication overhead between CPU and GPU and optimizes the GPU's workload. The SAH is employed for all hierarchy levels which results in very efficient BIHs. Our results show that our algorithm processes hundreds of thousands primitives at interactive frame rates. This enables interactive ray tracing of dynamic scenes even with changing geometry as is necessary for skeletal animation or adaptive tessellation.

Keywords: GPU, ray tracing, dynamic parallelism, spatial data structures, bounding interval hierarchy

1 Introduction

In virtual reality environments, the spatial perception and the degree of immersion depend on the visual quality of the rendering system. In terms of visual quality, most ray tracing systems outperform even modern rendering systems [LSB+14] [She12] because they provide global illumination effects such as reflections, shadows or caustics. In general, interactive ray tracing systems depend on a spatial data structure to minimize costs for intersection tests. In most cases, hierarchical acceleration data structures such as kd-trees, bounding volume hierarchies (BVH) or bounding interval hierarchies (BIH) are employed. However, the generation of these data structures is a costly task and it is often performed during preprocessing which prevents using ray tracing for dynamic scenes.

There are various approaches to mitigate this limitation. Multi-level hierarchies based on grids [KBS11] or a combination of BVH and kd-trees [RDS+10] have been introduced to

deal with moving, but rigid objects. Most recent approaches use modern graphics hardware to accelerate the generation process. In particular, some subtasks such as sorting can be performed in parallel on the GPU. However, frequent communication with the GPU is necessary to control the subdivision process. This communication interrupts the kernel execution and represents a bottleneck in most approaches.

In this paper, we present a GPU-based approach for the efficient generation of a full SAH-based BIH. Our approach exploits recent graphics hardware developments to eliminate the communication overhead between CPU and GPU. In particular, one of the latest key features of Nvidia graphics processors, *dynamic parallelism*, allows us to spawn new computation kernels directly on the GPU. Our implementation shows that our approach generates very efficient BIHs for hundreds of thousands primitives at interactive frame rates.

The main contribution of our approach is a novel multi-level parallel algorithm for generating a full SAH-based BIH on the GPU. Our asynchronous processing scheme avoids the communication overhead typically arising in other GPU-based implementations. The resulting hierarchies are very efficient for ray tracing because our SAH is used at all hierarchy levels. Our scheme generates BIHs on-the-fly which allows VR-applications to use ray tracing as rendering method for dynamic scenes, even if the object geometry changes, e.g. for skeletal animation or adaptive tessellation. This may highly increase visual quality, spatial perception and the degree of immersion in virtual reality applications.

2 Background

In general, there are a variety of acceleration data structures for ray tracing. Ray classification schemes [AK87] suffer from their enormous memory requirements and are not considered in this work. In most cases, spatial data structures such as kd-trees, BVHs or BIHs are used. However, most algorithms for the on-the-fly generation of these data structures focus either on the BVH or the kd-tree. For constructing a SAH-based BVH, Wald [Wal07] shows a fast, yet not interactive approach. Lauterbach et al. [LGS+09] present a hybrid approach which does not use a full SAH, but results in an almost optimized hierarchy. Furthermore, interactive updates (instead of rebuilding from scratch) have been used to increase performance [WBS07]. For kd-trees, there are also highly parallelized algorithms, both CPU-based [SSK07] as well as GPU-based approaches by Danilewski et al. [DPS10] and Zhou et al. [ZHWG08]. The latter both use different node stages to optimize the amount of parallelism for the corresponding node size.

When constructing an efficient acceleration data structure for ray tracing, it is important to find the best split for each node. Havran [Hav00] shows that clipping empty space in the upper levels of a hierarchy may significantly increase rendering performance. In our algorithm, a Surface Area Heuristic (SAH) is used to find the optimal splitting planes for all node stages.

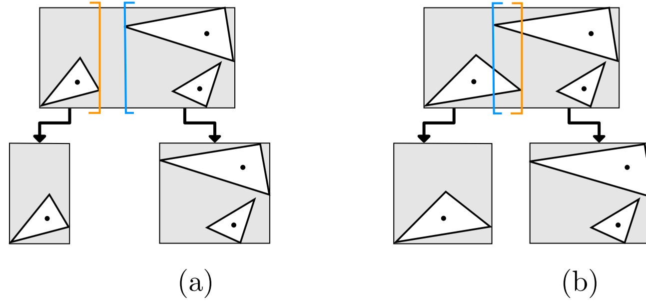


Figure 1: The Figures (a) and (b) illustrate the split of a BIH node. The bounds of the resulting child nodes are spatially adjusted with respect to the split direction and the primitives contained in the child nodes. This avoids instancing of primitives, but may result in overlapping bounding boxes, as shown in (b).

2.1 Surface Area Heuristic

The selection of splitting planes directly affects the quality of the acceleration data structure. The Surface Area Heuristic (SAH) proposed by MacDonald et al. [MB90] is a suitable estimate for the split quality and ray tracing efficiency. Given split candidate x, the cost function $C(x)$ is

$$C(x) = C_t + C_i \frac{A_L(x)N_L(x) + A_R(x)N_R(x)}{A_P} \quad (1)$$

where A_P identifies the surface area of the node to be split. The surface areas of the left and the right child are represented by A_L and A_R, whereas N_L and N_R are the number of primitives in the left and right child. The constants C_t and C_i reflect the costs for traversal and intersection, respectively.

In order to accelerate the evaluation of the SAH, MacDonald et al. allow the use of a limited number of equally spaced splitting splane candidates. In our algorithm, we employ 31 splitting plane candidates per axis in order to determine the best split.

2.2 Bounding Interval Hierarchy

A bounding interval hierarchy is a spatial acceleration data structure which was introduced by Wächter and Keller [WK06]. While its construction properties are similar to a BVH, they report that its traversal is as efficient as for a kd-tree. The construction of BIHs is generally faster compared to other spatial data structures which makes them suitable for ray tracing of dynamic scenes. The main reason for this is that spatial partitioning schemes need to handle primitives which overlap the volume boundaries. In BIHs, the bounding boxes of a node are adjusted to contain all primitives completely, as shown in Figure 1. Thus, instancing is not required which is a major difference in comparison to kd-trees. As a result, the memory requirements can be predicted and primitives can be sorted in-place which minimizes the need for dynamic memory allocation.

```
WHILE nodes IN queue {
  IF split NOT viable {
    store (node)
  } ELSE {
    FOREACH axis {
      FOREACH primitive IN node {
         bin_primitive()
      }
    }
    FOREACH bin {
      evaluate_costs()
    }
    split(node)
    FOREACH primitive IN node
      assign primitive to left_child or right_child

    FOREACH primitive IN left_child
      update_max_bound(left_child)

    FOREACH primitive IN right_child
      update_min_bound(right_child)

    enqueue_child_nodes()
  }
}
```

Figure 2: General pseudo code description for building a SAH-based BIH.

The construction of a BIH starts with a single node containing all primitives. This node is recursively subdivided until the termination criteria are satisfied. The pseudo code in Figure 2 summarizes the tasks which have to be performed, most of which can be run in parallel as separate GPU kernels.

2.3 Dynamic Parallelism

For better understanding, we give a brief overview of the terminology used to describe our asynchronous processing scheme.

A *thread* is the smallest working unit, which maps directly to a processing unit of the GPU. Threads execute programs, referred to as *kernels*, in parallel. A collection of threads is called a *block*. In order to run kernels, one or more blocks are launched together in a *grid*. An important unit of the CUDA programming model [NVI14] that we utilize in our algorithm is the *warp*. A warp is a small set of threads (currently 32) within a block which perform the same operations in parallel at the same time and are therefore synchronized implicitly.

Dynamic parallelism is an essential feature of Nvidia's graphics processors which was introduced by GK110. In contrast to prior graphics hardware, it allows a kernel to launch an own child kernel with dynamically adjustable parameters such as block and grid size, to distribute the resources according to upcoming task. Parent kernels are able to wait for the results of their own child kernels. We exploit this capability to implement a management

kernel which distributes workload according to the upcoming subtasks without interrupting the GPUs workflow by returning to the CPU to reconfigure launch parameters.

3 BIH Construction Using Dynamic Parallelism

This section describes the asynchronous processing scheme of our algorithm. As shown in Figure 2, the BIH construction consists of a number of parallelizable subtasks, which can be executed as separate kernels. In our system, these kernels are launched by a management kernel which represents the management layer of our algorithm. The workflow of the management kernel is shown in Figure 3.

In our management kernel, we utilize warps as functional entities for several reasons. First, warps are synchronized implicitly. Therefore, instructions can be assumed to be executed in a locked-step fashion for all 32 threads in a warp. Second, specialized intra-warp instructions, e.g. shuffle instructions, can be used to share and compute information within warps efficiently.

All BIH nodes are stored in an array which is organized in blocks of size 32. Henceforth, we will refer to such a block of nodes as *chunk*. Each chunk is associated with a *chunk header* which indicates the actual number of nodes contained in the chunk. All accesses to a chunk are protected by the corresponding chunk header.

In the management kernel, there is no synchronization between warps, which means all warps work independently of each other. Each chunk of nodes is processed by a warp. Within one warp, each thread fetches exactly one node from its chunk. Given the number of management warps W_L, the index of the current chunk $B(w_i)$ to be processed by warp w_i is computed by

$$B(w_i) = W_L H_i + i, \quad 0 \leq i < W_L \tag{2}$$

where H_i is the current iteration of w_i, which is equal to the number of chunks already processed by w_i. After each iteration, a warp moves W_L chunks forward along the queue. Nevertheless, each warp processes only one chunk per iteration.

At the end of each iteration, memory has to be reserved to store the child nodes in consent with all other warps in the management layer. A chunk of size 32 can generate up to 64 child nodes. Therefore, a warp may need to reserve up to two chunks. The reservation of free chunks is realized by means of atomic additions to a global variable which identifies the index of the first free chunk. In general, warps reserve the appropriate number of chunks with respect to the number of child nodes produced.

In order to infer the storage location for child nodes, we compute the prefix sum across the number of child nodes of all threads in a warp. For parallel prefix sums and their applications, we refer to [Ble90]. After child nodes have been stored in the reserved chunks, the corresponding chunk headers are adjusted such that their value represents the number of valid nodes in the chunk. As soon as no more child nodes are produced, reserved chunk headers are finalized and the warp moves to the next chunk.

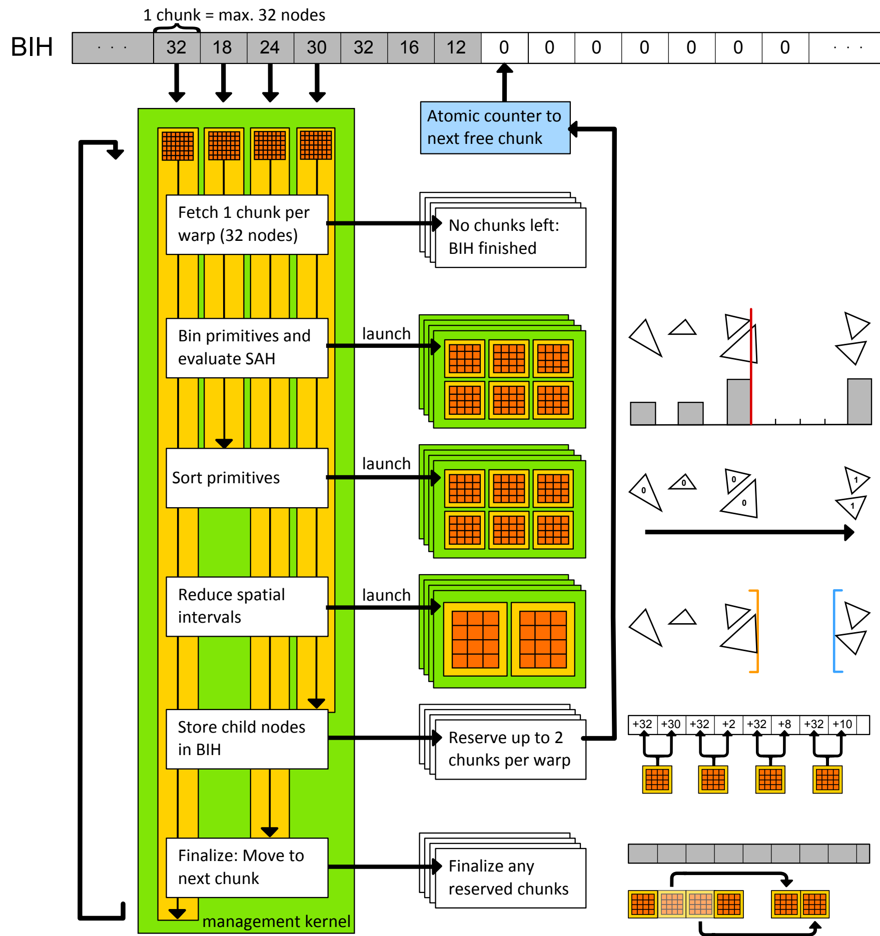


Figure 3: This figure illustrates the workflow of our Large Node Stage. Each kernel (green) consists of thread blocks (orange). In this example, our management kernel is executed with four asynchronously working warps ($W_L = 4$). Each of these warps processes up to 32 nodes in parallel and launches the necessary child kernels. After splitting, the first thread of each warp reserves chunks for storing the child nodes by means of an atomic counter (blue).

Since warps in our management layer work asynchronously and communication takes place only by means of atomic additions, it is possible that one warp stores children in a reserved chunk, while another warp already processes the nodes contained in that chunk. The amount of valid nodes stored in a chunk is determined by the corresponding chunk header. Of course, a thread reads and processes its corresponding node only if it is valid. However, a warp may end the current iteration and advance to the next chunk only if the chunk header of its current chunk has been finalized, indicating that no further nodes will be added to the chunk, and all nodes have been processed.

In summary, warps of our management layer are coupled very loosely and the BIH is constructed in a highly asynchronous manner. Its construction takes place without any explicit synchronization points. As soon as a warp has processed its current chunk, it moves on to the next one immediately. For these reasons, warps in the management layer can be located in different computational blocks and all streaming multiprocessors can be utilized from the beginning.

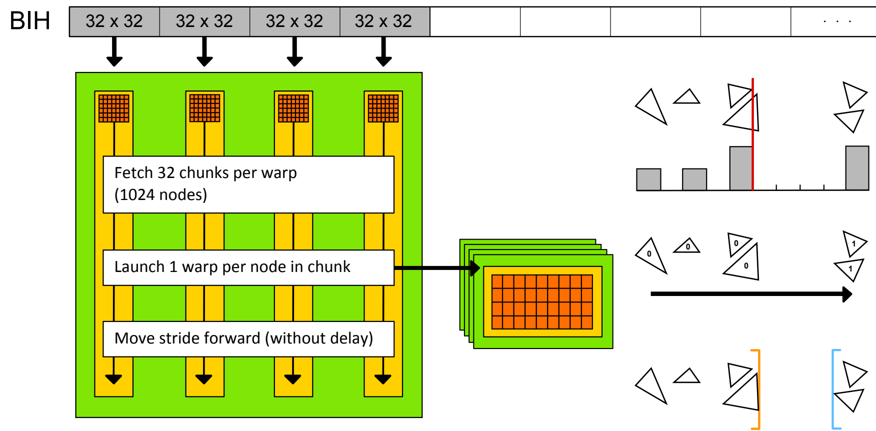


Figure 4: This Figure illustrates the workflow of our Small Node Stage. In contrast to the LNS, each warp processes 32 chunks (1024 nodes). For each node, one single warp is launched which performs all necessary subtasks.

4 Node Stages

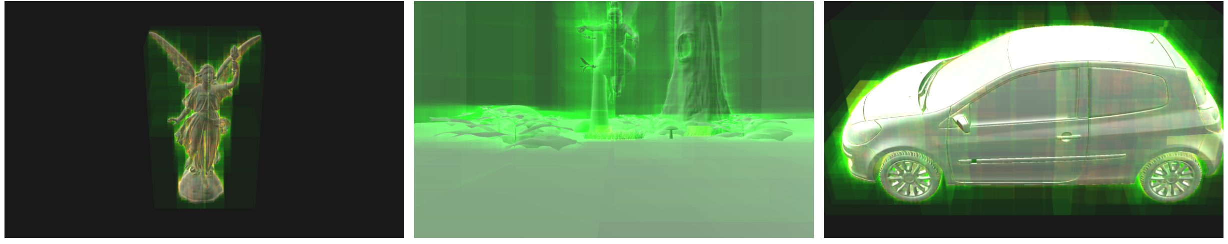
As the depth of the acceleration structure increases, the amount of nodes which have to be processed grows exponentially. At the same time, the number of primitives per node decreases quickly. For this reason, we adjust the level of parallelism in order to better distribute the parallel processing power to the granularity of the problem. Similar to [ZHWG08], our management layer is organized in two node stages – a *Large Node Stage (LNS)* and a *Small Node Stage (SNS)*. Our Large Node Stage is designed to work on a relatively small amount of nodes containing many primitives, whereas our Small Node Stage invests its capacities into processing thousands of nodes with a small number of primitives per node in parallel.

4.1 Large Node Stage

The LNS makes extensive use of dynamic parallelism. As there are few nodes in the queue with very high primitive counts, three specialized kernels are launched – one for every major task of the construction process, as shown in Figure 3. The amount of parallelism of these grids is dynamically adjusted to the granularity of the tasks. First, the best split is determined by a kernel specialized to generate and evaluate the SAH for 31 splitting plane candidates per axis. This results in 32 bins per axis which can be sorted efficiently using intra-warp instructions. Similarly, a specialized kernel is launched to parallel sort all primitives of a node with respect to the splitting plane and the splitting axis. The third kernel is optimized for the parallel adjustment of the child nodes' bounding boxes.

4.2 Small Node Stage

The most important difference between our two node stages is that each warp in the management layer of the Small Node Stage fetches 32 chunks. This means that every management thread is responsible for one chunk (32 nodes) per iteration.



(a) *Lucy* (b) *Fairy Forest* (c) *Clio*

Figure 5: This Figure shows the models used for performance tests.

In order to process all nodes in the chunk at once, each thread launches one single grid. The number of warps in that grid equals the number of nodes in the chunk. In contrast to the LNS, all major tasks of the construction are performed in this grid, as indicated in Figure 4.

If W_S represents the number of management warps in the Small Node Stage, $1024 \cdot W_S$ nodes are processed in each iteration. In comparison, the SNS processes considerably more nodes at once than the LNS. However, for reasons of efficiency, nodes may only enter our Small Node Stage if the number of primitives per node is less than 1024.

Another important difference between our two node stages is the location where administrative tasks, e.g. storing child nodes, are performed. In the Small Node Stage, these administrative tasks are moved into the newly launched child grid. This has the advantage that warps in the management layer can proceed to the next chunk immediately after the child grid is launched.

5 Results and Discussion

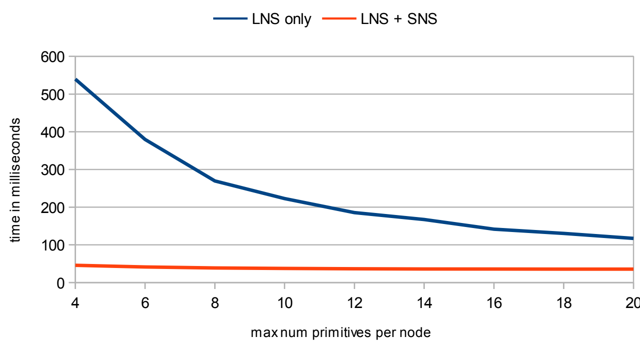
All of our results were measured on an Intel Core i7-3820 CPU with 32 GiB of RAM and a Nvidia GTX Titan GPU. For ray casting, a naïve GPU implementation with no further optimizations was used. The window resolution was set to 1664x1024.

The BIH construction time depends on the number of warps in the management layer. More specifically, we observe that the optimal construction time is achieved when the number of warps is close to the number of streaming multiprocessors (SMX) of the graphics hardware. However, since a thread block can only run on one streaming multiprocessor, we distribute all warps of our managment layer into distinct blocks. In our implementation, the number of management warps for both node stages W_L and W_S was set to 16 (close to the 15 SMX of the GK110).

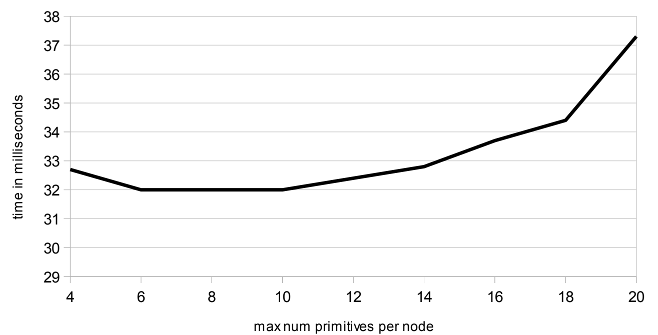
For performance measurements, we used the three models shown in Figure 5. Table 1 shows the corresponding number of triangles and construction times, whereas Figure 6 shows the construction and ray casting performance for different configurations for the model *Fairy Forest*. We used this model to find the optimal settings for the combined performance for construction and ray casting performance. The primitive threshold was used as an additional

Model	#Triangles	[ZHWG08]	[DPS10]	[LGS+09]	Our approach
Lucy	79k	n.a.	n.a.	n.a.	25 ms
Fairy Forest	175k	77 ms	57 ms	488 ms	45 ms
Clio	257k	n.a.	n.a.	n.a.	56 ms

Table 1: This Figure shows the construction times of our algorithm and other approaches. Unfortunately, a direct comparison with the other approaches is not possible, because different data structures (kd-trees and BVH) and heuristics have been used and the tests were performed on slower hardware (GeForce 8800 ULTRA, GTX285 and GTX280). In comparison to our approach, only [LGS+09] and [DPS10] use a SAH for all node stages. Zhou et al. [ZHWG08] use a SAH only in the lower hierarchy levels which significantly reduces costs.



(a) BIH construction using our algorithm



(b) Ray casting performance using the BIH

Figure 6: These diagrams show the times for construction and ray casting of the model *Fairy Forest* in dependence of the maximal number of primitives per leaf node. Obviously, smaller leaf nodes are optimal for ray casting (6b). In addition, we observe that the construction time is almost independent using both node stages (6a) which proves the efficiency of our multi-stage concept.

termination criterion to avoid degeneracies and to evaluate the efficiency of our node stages.

Unfortunately, a direct comparison to existing work is quite involved. A performance comparison would require a reimplementation of other approaches, because the used graphics hardware lacks the necessary capabilities we require. Furthermore, the acceleration data structures produced by these approaches are different. Danilewski et al. [DPS10] utilize five highly optimized node stages, which seems advantageous compared to our system. However, our system avoids the communication overhead, but measuring this performance gain is also quite involved and remains future work. Zhou et al. [ZHWG08] use much slower hardware, but refrain from using a SAH for all hierarchy levels which highly reduces construction costs.

An interesting observation is that the frame rate is almost entirely dependent on the ray casting times for our models. This is because the BIH construction performs almost independent of the maximal number of primitives per node. Furthermore, we evaluated the efficiency of our node stage concept. We limited the number of maximal primitives per

node to measure the scaling of the Large Node Stage. Using only the Large Node Stage, the construction times increase exponentially, which indicates that too few threads need to handle an exponentially growing number of child nodes. Using both node stages, the construction time is almost constant with respect to the number of primitives in the leaf nodes which shows that the Small Node Stage is capable of dealing with a larger number of nodes.

Since the size of a chunk (32 nodes) is chosen relatively large, chunks are usually not completely filled. The reserved chunks of a warp are filled entirely only if the number of child nodes of an iteration is a multiple of the chunksize or zero. Therefore, the efficiency of our asynchronous management scheme comes with increased memory consumption. An evaluation of our memory requirements revealed that the overhead introduced by our management layer was in average 44% for our models.

Our work focuses on the construction of bounding interval hierarchies for triangle meshes. It is conceivable that our approach can be generalized and used for the construction of different acceleration data structures. Furthermore, our current implementation of the BIH construction can be extended to every primitive type that provides a bounding box, or even whole objects.

6 Conclusion and Future Work

In this paper we explore new possibilities which dynamic parallelism provides for the generation of acceleration data structures. In our work we focus on the management layer we developed. Our algorithm exploits the implicit synchronization of threads within a warp to avoid costly explicit synchronization of thread blocks. The warps of our management layer are loosely coupled which leads to a highly asynchronous and efficient construction of a bounding interval hierarchy. Our results show that our implementation efficiently utilizes dynamic parallelism and constructs SAH-based BIHs for hundreds of thousands of primitives at interactive frame rates. This fast and complete rebuild of the acceleration data structure allows for real-time ray tracing of dynamic scenes.

Although we use only two node stages, we observe that the utilization of an additional Small Node Stage accelerates the BIH construction considerably compared to utilization of a single node stage. We conjecture that employing a higher number of stages, similar to [DPS10], would lead to even better adjustment to node granularities and increased BIH construction performance.

In this work, we construct the BIH per frame from scratch. However, for small changes in scene topology an iterative approach for updating existing hierarchies may suffice for interactive ray tracing performance. Furthermore, it is conceivable to rebuild the acceleration structure only after several frames and to update it on a per-frame basis.

The memory overhead introduced by our algorithm cannot be neglected. Reserving large chunks of memory reduces the propability of concurrent access to the atomic variables, which effectively lowers the waiting time for every working warp. It would be desirable to profile

different chunk sizes in order to find the optimal parameters that balance memory overhead against atomical access collisions.

Our results show that SAH-based BIHs can be generated on-the-fly for medium-sized scenes. This allows for interactive ray tracing of dynamic scenes which may highly increase the visual quality, spatial perception and degree of immersion in virtual reality applications.

References

[AK87] James Arvo and David Kirk. Fast ray tracing by ray classification. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 55–64, New York, NY, USA, 1987. ACM.

[Ble90] Guy E Blelloch. Prefix sums and their applications. In *Synthesis of parallel algorithms*, pages 35—60. Morgan Kaufmann Publishers Inc., 1990.

[DPS10] Piotr Danilewski, Stefan Popov, and Philipp Slusallek. Binned SAH Kd-Tree Construction on a GPU. Technical report, Saarland University, Computer Graphics Lab, 6 2010.

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[KBS11] Javor Kalojanov, Markus Billeter, and Philipp Slusallek. Two-level grids for ray tracing on gpus. *Comput. Graph. Forum*, 30(2):307–314, 2011.

[LGS+09] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David P. Luebke, and Dinesh Manocha. Fast bvh construction on gpus. *Comput. Graph. Forum*, 28(2):375–384, 2009.

[LSB+14] Felix Lauer, Simon Schneegans, Andreas-Christoph Bernstein, Andre Schollmeyer, and Bernd Froehlich. guacamole - an extensible scene graph and rendering framework based on deferred shading. In *7th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS'14)*. Springer, 2014.

[MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, May 1990.

[NVI14] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2014. Version 6.0.

[RDS+10] Martin Reichl, Robert Dünger, Alexander Schiewe, Thomas Klemmer, Markus Hartleb, Christopher Lux, and Bernd Fröhlich. Gpu-based ray tracing of dynamic scenes. *JVRB*, 7, 2010.

[She12] Denis Shergin. Unigine engine render: Flexible cross-api technologies. In *ACM SIGGRAPH 2012 Computer Animation Festival*, SIGGRAPH '12, pages 85–85, New York, NY, USA, 2012. ACM.

[SSK07] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Comput. Graph. Forum*, 26(3):395–404, 2007.

[Wal07] I. Wald. On fast construction of sah-based bounding volume hierarchies. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, pages 33–40, Sept 2007.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1), January 2007.

[WK06] Carsten Wächter and Alexander Keller. Instant ray tracing: The bounding interval hierarchy. In Tomas Akenine-Möller and Wolfgang Heidrich, editors, *Rendering Techniques*, pages 139–149. Eurographics Association, 2006.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 126:1–126:11, New York, NY, USA, 2008. ACM.