Memory-Scalable GPU Spatial Hierarchy Construction

Qiming Hou, Xin Sun, Kun Zhou, Christian Lauterbach, and Dinesh Manocha

Abstract—Recent GPU algorithms for constructing spatial hierarchies have achieved promising performance for moderately complex models by using the breadth-first search (BFS) construction order. While being able to exploit the massive parallelism on the GPU, the BFS order also consumes excessive GPU memory, which becomes a serious issue for interactive applications involving very complex models with more than a few million triangles. In this paper, we propose to use the partial breadth-first search (PBFS) construction order to control memory consumption while maximizing performance. We apply the PBFS order to two hierarchy construction algorithms. The first algorithm is for kd-trees that automatically balances between the level of parallelism and intermediate memory usage. With PBFS, peak memory consumption during construction can be efficiently controlled without costly CPU-GPU data transfer. We also develop memory allocation strategies to effectively limit memory fragmentation. The resulting algorithm scales well with GPU memory and constructs kd-trees of models with millions of triangles at interactive rates on GPUs with 1 GB memory. Compared with existing algorithms, our algorithm is an order of magnitude more scalable for a given GPU memory bound. The second algorithm is for out-of-core bounding volume hierarchy (BVH) construction for very large scenes based on the PBFS construction order. At each iteration, all constructed nodes are dumped to the CPU memory, and the GPU memory is freed for the next iteration's use. In this way, the algorithm is able to build trees that are too large to be stored in the GPU memory. Experiments show that our algorithm can construct BVHs for scenes with up to 20 M triangles, several times larger than previous GPU algorithms.

Index Terms—Memory bound, kd-tree, bounding volume hierarchy.

1 Introduction

Current many-core GPUs have evolved into incredible computing processors for general-purpose computation, and this evolution is likely to continue in the future. Recently, GPU construction of hierarchical data structures, such as kd-trees [1] and BVHs [2], has shown great promise in a variety of applications, including ray tracing, photon mapping, point cloud modeling, and simulations. Unlike traditional CPU-based algorithms, which build hierarchical data structures following the depth-first search (DFS) order, the GPU algorithms achieve interactive construction by using the breadth-first search (BFS) order, which best exploits the massive parallelism on the GPU. These algorithms exploit the multiple cores and high memory bandwidth in terms of building hierarchies of moderately complex models at interactive rates. Unfortunately, this parallel computation comes at the cost of excessive memory consumption overhead because the GPU algorithms need to maintain and process a large amount of data simultaneously. This becomes a serious issue for interactive applications involving complex models with more than a few million triangles [1], [2]. Current GPUs have a different memory architecture than CPUs. The on-board memory on GPUs is limited to a few GBs. Moreover, GPUs have high memory bandwidth, much smaller per-thread caches and GPU's memory limitation cannot be virtualized by on-demand paging. As a result, it is important to design GPU-based algorithms that can cope with these memory architecture characteristics of GPUs for interactive applications.

An important characteristic of many-thread algorithms running on parallel processing platforms is that the memory consumption is correlated with the level of parallelism. GPU's architecture exaggerates this issue as it requires significantly more parallel threads than physical execution units to perform efficiently. Executing more computations in parallel requires simultaneously maintaining more intermediate data and thus consumes more memory. The key idea of this paper is to make proper trade-offs between memory consumption and level of parallelism to control memory consumption while maximizing performance. For hierarchy construction, such trade-offs are facilitated by the partial breadth-first search (PBFS) order. Unlike the BFS and DFS, the PBFS allows the set of tree nodes being processed simultaneously to be explicitly controlled in each iteration, and thereby enables management of the memory consumption and level of parallelism. By carefully tuning the set of nodes being processed simultaneously, we can achieve a good balance between them. Note that the PBFS only affects the order of node processing and does not impact the quality of the resulting hierarchy.

- *Q. Hou is with Tsinghua University, Beijing, China, and Microsoft Research Asia, 5036, Sigma Building, Zhichun Road 49, Haidian District, Beijing, China 100190. E-mail: hqm03ster@gmail.com.*
- *X. Sun is with State Key Laboratory of CAD&CG, ZiJinGang Campus, Zhejiang University, HangZhou, China 310058, and Microsoft Research Asia, 5081, Sigma Building, Zhichun Road 49, Haidian District, Beijing, China 100190. E-mail: sunxin@microsoft.com.*
- *K. Zhou is with State Key Laboratory of CAD&CG, ZiJinGang Campus, Zhejiang University, HangZhou, China 310058. E-mail: kunzhou@acm.org.*
- *C. Lauterbach and D. Manocha are with the Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175. E-mail: {cl, dm}@cs.unc.edu.*

Manuscript received 23 Aug. 2009; revised 7 Feb. 2010; accepted 1 May 2010; published online 3 June 2010.
Recommended for acceptance by P. Slusallek.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2009-08-0183.
Digital Object Identifier no. 10.1109/TVCG.2010.88.

We apply the PBFS order to two hierarchy construction algorithms. The first algorithm is a GPU kd-tree algorithm that achieves superior performance for a given memory bound. The algorithm uses PBFS to automatically adapt the level of parallelism based on available memory and thus allows the peak memory consumption to be controlled without swapping any data out of the GPU. On an NVIDIA GeForce GTX 280 GPU with 1 GB memory, we can construct kd-trees of scenes with up to several million triangles at interactive rates. The second algorithm is an out-of-core BVH construction algorithm on the GPU. Compared to kd-tree construction, BVH construction has a relatively small memory overhead. It does not split triangles and does not need to dynamically allocate GPU memory. Consequentially, the primitive storage remains static throughout the construction and the final tree size can be bounded prior to construction. However, the memory consumption will still exceed the available GPU memory for very large scenes. We use PBFS to extend BVH construction to handle such scenes. At each PBFS iteration, all constructed nodes are dumped to the CPU memory or disk, and the GPU memory is freed for the next iteration's use. In this way, the algorithm is able to build trees that are too large to be stored in the GPU memory. Our algorithm can construct BVHs for scenes with up to 20 M triangles.

As far as we know, ours are the first GPU hierarchy construction algorithms that are designed with a memory bound in mind. Our methods can handle scenes nearly an order of magnitude larger than previous GPU methods. For small scenes that previous GPU methods can handle, our algorithm achieves similar construction performance. For large scenes, our method performs comparably to the state-of-the-art multicore CPU algorithms in terms of construction time while maintaining tree quality similar to high quality methods. In general, our methods scale well with respect to the amount of available memory, and hierarchy construction can be performed within user-specified memory bounds at a modest performance cost.

We will briefly review previous work relevant to fast spatial hierarchy construction in Section 2. In Section 3, we describe our memory-scalable kd-tree construction algorithm. Section 4 describes how to use the PBFS order to support out-of-core BVH construction on the GPU. Finally, we present results in Section 5.

2 Related Work

Several CPU-based algorithms have been proposed for fast construction of surface area heuristic (SAH) kd-trees [3], [4], which are commonly regarded to offer optimal ray tracing performance. Hunt et al. [5] approximated the SAH cost function to achieve subinteractive construction with minimal degradation in tree quality. Shevtsov et al. [6] developed an interactive parallel construction algorithm with a modest memory footprint on multicore CPUs. However, their tree suffers from considerable quality loss. Soupikov et al. [7] recently introduced approximate triangle clipping to compensate for this quality loss within a similar construction time. However, with both algorithms, tests show serious scalability issues at more than a few hundred threads. This makes them inappropriate for massively parallel architectures like GPUs.

Zhou et al. [1] proposed the first kd-tree construction that runs entirely on the GPU. The algorithm maximizes parallelism in the construction process and scales well to GPUs with hundreds of cores. High-quality trees can be constructed in rapid time. However, the high parallelism is achieved at the cost of excessive memory consumption. This results in a scene size limitation one order of magnitude smaller than previous methods. We use the node splitting schemes of [1] to maintain tree quality and construction performance but introduce novel parallelization and memory management techniques to bound the memory consumption.

BVH is an alternative spatial hierarchy for ray tracing that favors build time over tracing performance. Efficient construction has been demonstrated on both CPU and GPU [8], [9], [2]. Recent work also demonstrates ray tracing performance improvement by incorporating kd-tree-like features into BVHs [10]. The state-of-the-art GPU BVH construction algorithm [2] has a workflow resembling GPU kd-tree construction. We apply the PBFS construction order to the hybrid algorithm described in [2] for out-of-core BVH construction of very large scenes.

Wachter and Keller [11] tackled the memory problem of kd-trees from a different perspective. They terminated the splitting node when necessary to bound the final hierarchy size. Their approach puts the tree quality at risk and does not apply to hierarchies with naturally bounded size like BVH. In contrast, our work seeks to control the work memory requirement during construction while maintaining tree quality. Lauterbach et al. [12] reduced the memory consumption by using triangle strips. We still use triangle lists because they are more general and widely used in computer graphics. Paging systems like virtual memory can be used to handle large data within limited physical memory, effectively providing out-of-core support for any algorithm. Built-in virtual memory support can be expected in future GPUs, such as Larrabee [13]. A general paging-like out-of-core system also has been demonstrated on current hardware [14]. While paging systems can be very efficient when handling large input/output, paging intermediate work memory can result in significant performance overhead. Our PBFS aims to overcome this problem by bounding work memory within available physical memory. PBFS can also be used in combination with paging systems to handle out-of-core input/output more efficiently.

Memory-bounded situations have been investigated in traditional parallel programming research [15]. The main focus there is the trade-off between data replication and communication in distributed systems. Our work controls peak memory usage by limiting the creation of new data and does not involve data replication.

3 Memory-Scalable kd-Tree Construction

Most CPU-based kd-tree construction methods follow the natural DFS order. Even multicore CPU algorithms follow the DFS order in the majority of their pipelines. While the DFS order has a small memory footprint, it is difficult to achieve good scalability on more than a few hundred of threads. GPU-based constructors follow the BFS order [1], [2]. The BFS maximizes the number of nodes constructed simultaneously and thus benefits from the high parallelism of the GPU to outperform DFS methods. However, it also results in a significantly larger memory footprint.

During kd-tree construction, each node being split requires storage of extra temporary data for the subsequent

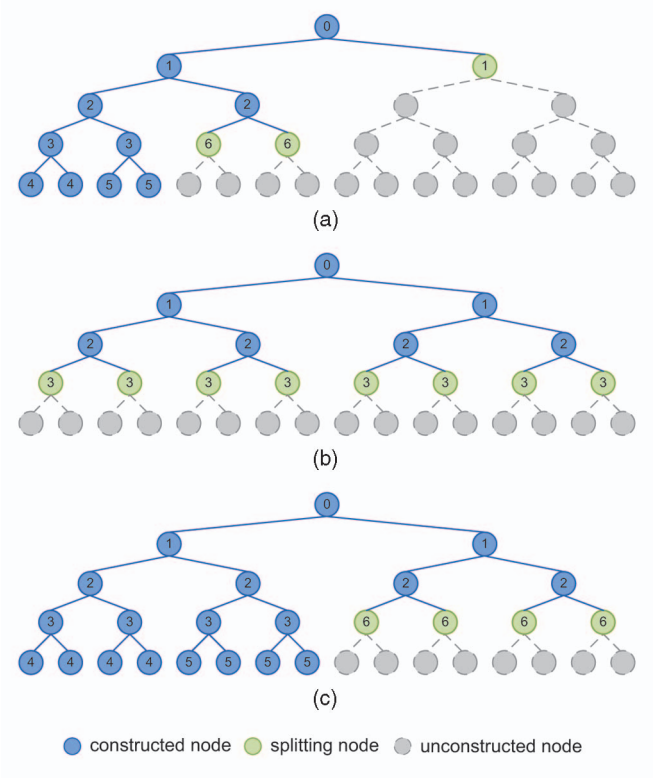


Fig. 1. Different kd-tree construction orders. The number in each node corresponds to the iteration it is created in. (a) DFS kd-tree construction. (b) BFS kd-tree construction. (c) Our PBFS kd-tree construction.

computation. Thus, the memory consumption is proportional to the number of nodes being split simultaneously. Based on this, we can make a rough comparison of the memory cost between DFS and BFS schemes. Fig. 1 illustrates the set of splitting nodes maintained simultaneously in three construction schemes. The number of splitting nodes with the DFS scheme is proportional to the current construction depth, as shown in Fig. 1a. For a scene with n primitives, this depth is $O(\log n)$. In a BFS constructor, the number of splitting nodes grows exponentially with the construction depth and eventually reaches $O(n)$. This is shown in Fig. 1b. This kind of extreme difference leads to a heavy storage load for the BFS construction scheme.

We introduce a partial breadth-first search solution to compromise between parallelism and the size of the peak memory footprint. We control the peak memory by tuning the number of nodes being split simultaneously. Compared to the exhaustive BFS, the PBFS only splits part of the nodes at a time. This is illustrated in Fig. 1c. When some trunks of the tree are completely constructed, the corresponding memory is released so that we can split the remaining nodes.

In the following, we first briefly review the BFS-based construction algorithm of [1] in Section 3.1. We then present our PBFS scheme in detail in Section 3.2. Our antifragmentation dynamic buffer management scheme is introduced in Section 3.3. Section 3.4 describes how we handle memory issues related to triangle clipping.

3.1 Review of BFS kd-Tree Construction on GPU

The GPU kd-tree construction in [1] mainly consists of two stages. The nodes are divided into two categories, large nodes and small nodes, and are split with different schemes. A node is categorized as large if the number of triangles it contains is greater than a prescribed threshold; otherwise, the node is small. The kd-tree construction starts from the root node. First, a large-node stage is launched to split all large nodes recursively. Small nodes generated by splitting large nodes are stored in a dynamic buffer. After dividing all large nodes, the large-node stage terminates, outputting a buffer of small nodes. Then, a small-node stage is launched to finish the construction by splitting all small nodes recursively. For each large node, which contains more than 64 triangles, the median splitting and "empty space maximizing" are employed to minimize the traversing cost of ray tracing. After node splitting, each triangle intersected by a splitting plane is clipped into two polygons (called *clipped triangles* in the following) and distributed to the child nodes. A dynamic buffer is required to hold the vertices of all the clipped triangles generated in the large-node stage. For each small node, which contains no more than 64 triangles, the splitting plane is determined to minimize the SAH cost to minimize the traversal cost. Triangle clipping is not performed during the small-node stage. Each triangle intersected by the splitting plane is simply distributed to both children.

The SAH cost function is defined as

$$SAH(x) = C_{ts} + (C_L(x)A_L(x) + C_R(x)A_R(x))/A,$$

where C_{ts} is the constant cost of traversing the node itself, $C_L(x)$ is the cost of the left child given a split position x, and $C_R(x)$ is the cost of the right child given the same split. $A_L(x)$ and $A_R(x)$ are the surface areas of the left and right children, respectively. A is the surface area of the node. $C_L(x)$ and $C_R(x)$ are usually evaluated as the number of triangles in the two children. For each small node, the splitting plane candidates are restricted to planes containing the faces of the axis-aligned bounding boxes (AABBs) of the clipped triangles contained in the node.

Zhou et al. [1] also provide a data structure for storing the triangles in small nodes as bit masks. All small nodes whose parent nodes are large nodes are called small roots. The triangle set contained in each small node is then stored as a bit mask representing a subset of its small root. For each small root, the triangle sets contained on both sides of each splitting plane candidate are also precomputed as bit masks. For each small node, with its triangle mask and the precomputed split triangle sets of its small root, $C_L(x)$ and $C_R(x)$ can be computed efficiently with bitwise operations.

3.2 PBFS Construction

Note that in the above kd-tree algorithm, small nodes consume much more memory than large nodes because the number of small nodes is much greater than that of large nodes. In particular, the precomputation data of all small roots consume most of the temporary data in the tree construction. Because the data of each small root are needed by all of its descendant nodes, the data can only be freed after all descendants of the small root are completely constructed. Therefore, the key point to consider in designing the PBFS strategy is to find an inexpensive way to control the number of small nodes (including small roots) being processed simultaneously.

Our solution is to alternate between large-node and small-node construction, as shown in Fig. 2. Our observation

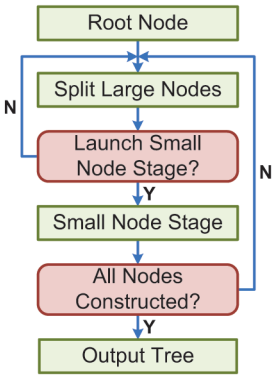


Fig. 2. Our alternating kd-tree construction pipeline: the large-node stage and small-node stage are launched in alternation.

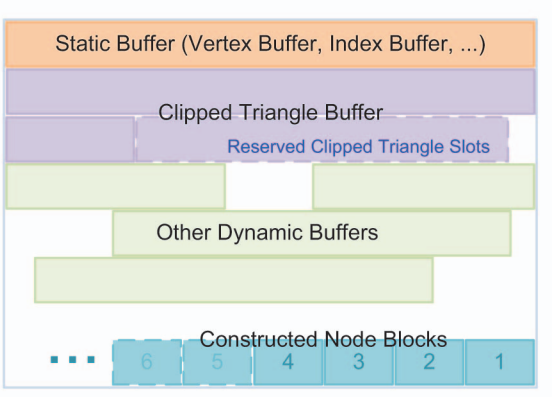


Fig. 3. Memory pool layout in our dynamic buffer management scheme.

is that it is unnecessary to wait until all small roots are generated since the small roots are continuously generated throughout the large-node stage. At any time if we find the small roots are too numerous to be split simultaneously, we should launch a small-node stage to complete the construction of as many small roots as available memory allows. After this visit to the small-node stage, all temporary data associated with the completed nodes are discarded. We can then return to the large-node stage to continue generating small roots.

The above solution needs to compute the maximal number of small roots that the algorithm can process simultaneously under a memory bound. In other words, we need to compute the memory cost for building the subtree under a small root. Unfortunately, there is no theoretical peak memory usage for the SAH-based kd-tree construction because the tree depth is uncertain. We thus need a tight estimation. Observing that the precomputation data of small roots take most of the peak memory usage, we calculate the size of precomputation data exactly and estimate the remaining memory usage as a constant factor times the number of small roots. We set this factor to a very conservative value at first and update it after each launched small-node stage. The number of small roots that can be handled under a memory bound can be easily computed by dividing the memory bound by the estimated per-node memory usage.

Each small-node stage begins with the estimated number of small roots to handle. If the number is overestimated and the stage fails due to insufficient memory, we rollback all operations completed during this stage and try again with half of the original small roots. While this approach is robust, the rollback mechanism is costly. In practice, we find that the memory cost estimation is accurate enough to entirely avoid the costly rollback in all our experiments.

3.3 Dynamic Buffer Management

Dynamic buffers are constantly used throughout the kd-tree construction process for maintaining splitting nodes and storing constructed nodes. They inevitably lead to memory fragmentation. If there are a few memory fragments left in the middle of an available memory region, allocating a large buffer could fail, as often happens when working on large scenes. Therefore, we need efficient dynamic buffer management to reduce fragmentation. For this purpose, we reserve all available memory as a pool at the beginning of the kd-tree construction, and allocate memory from the pool using special strategies.

We compactly place all static buffers, such as the vertex buffer and the index buffer, at the beginning of our memory pool. For special reasons to be explained in Section 3.4, we also allocate the buffer of clipped triangles statically, even though it is a dynamic buffer.

The most important dynamic buffer is the buffer of constructed nodes. This buffer is continuously appended throughout the entire construction process and cannot be discarded. Without special handling, allocations made for this buffer can cause permanent memory fragmentation. We observe that the nodes deposited into the buffer are left untouched until the construction is complete. This observation allows us to apply a block-based strategy. We allocate the constructed nodes buffer in 4 MB memory blocks from the high address end of the memory pool. When construction begins, a block is allocated at the highest address. When the buffer becomes full, we allocate another block compactly before the previous one. Allocations for all other dynamic buffers are performed at the low address end. The result is that, as long as the memory pool is not used up, the management of the constructed nodes buffer does not interfere with other memory allocations. This is illustrated as the cyan blocks in Fig. 3.

3.4 Efficient Storage of Clipped Triangles

The large-node stage also takes a considerable portion of the memory because of the clipped triangles contained in the nodes. As shown in Fig. 3, all of these triangles are kept in a buffer. Nodes only maintain the indices of their triangles. Since we clip triangles to nodes, newly clipped triangles maybe added during construction. Therefore, the triangle buffer has to be appended on the fly. Instead of dynamically appending this buffer, we preallocate a static buffer with sufficient size for all triangles.

The triangle buffer differs from the constructed nodes buffer in our PBFS scheme. After precomputation of each small-node stage, the clipped triangles contained in already-processed small roots are no longer useful. We can label them after each small-node stage and reuse the freed memory slots later. As shown in Fig. 4, three slots are freed after a small-node stage. These slots are then reused to store

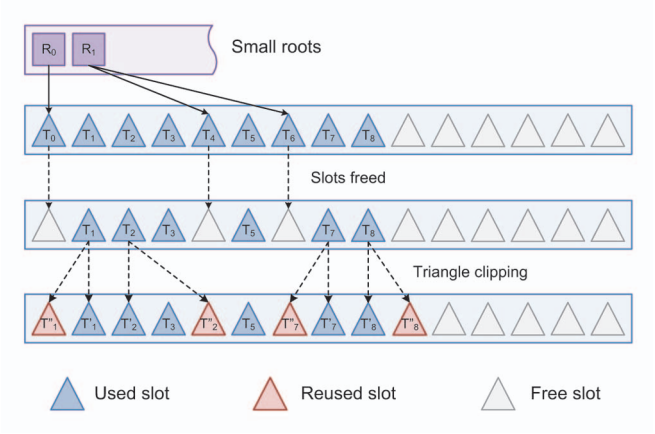


Fig. 4. Reusing the clipped triangle slots. Three slots are freed after a small-node stage. These slots are then reused to store new clipped triangles generated during subsequent triangle clipping.

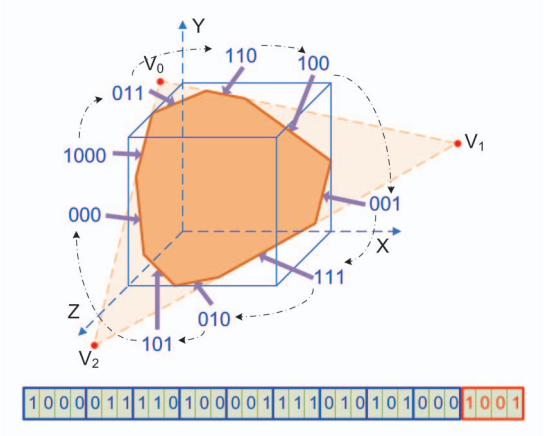


Fig. 5. Packing a clipped triangle shape into a 32-bit integer.

new clipped triangles generated during subsequent triangle clipping. Also, this buffer does not grow as rapidly as that of the constructed nodes. For typical scenes, the analysis in [16] shows that splitting a node with k triangles generates $O(\sqrt{k})$ clipped triangles. By adding up clipped triangles generated at all $O(\log n)$ tree levels, the total number of generated clipped triangles can be expected to be $O(n)$, where n is the number of original triangles. These facts make it more attractive to allocate the triangle list buffer statically. In practice, we find a static triangle list with the capacity of $1.5n$ triangles is sufficient for our test scenes.

Slot reuse is only possible if the information for each clipped triangle can be stored in a fixed-size format. Note that we store the current shape of each clipped triangle. This shape is a triangle-AABB intersection, therefore, a convex polygon of 3 to 9 vertices. Special handling is required to pack it in a compact fixed-size format.

A triangle clipped by axis-aligned planes will result in a polygon with no more than nine vertices and no more than nine edges. The nine edges can come from the original three edges and the six faces of the AABB. We encode each edge as a 3- to 4-bit binary number. Edges of the AABB are labeled from 000 to 101. The three original triangle edges are labeled from 110 to 1,000. The total number of the edges is packed in the four least-significant bits. The edge labels are placed from the most-significant bits to the least-significant bits in either clockwise or counterclockwise order. If the clipped triangle contains the 1,000 edge, this edge is always placed in the four most-significant bits. Fig. 5 illustrates the packing of a nine-edged clipped triangle shape. Since there cannot be two edges with the same label in a polygon, this 32-bit integer is enough for us to recover all edges and vertices of a polygon given its original vertices and the AABB. With this representation, a clipped triangle only needs to keep the AABB, the edge integer, and the index of the original triangle. This representation only takes 32 bytes per triangle and significantly reduces the memory cost. The reconstruction of vertices does not slow down the triangle clipping because of the reduced memory fetching.

3.5 Tree Output

When building the kd-tree with a given memory bound, the output process of the constructed tree merits a bit of discussion. In [1], the constructed tree is converted into a preorder traversal format. However, this conversion is itself a BFS traversal. At its memory peak, the original constructed tree, the preorder traversal, and the node correspondence between them coexist in the memory. This peak is considerably larger than the memory peak in our PBFS construction and has to be avoided. Also, the finalization algorithm of [1] has relatively strict requirements on the processing order of tree nodes and does not fit well in our PBFS scheme.

We chose to use our natural construction layout directly as the final tree node layout and omit the conversion altogether. In theory, our layout may cause a degradation in ray tracing performance. In practice, we found such degradation to be minor. Additionally, this format change allows us to omit the finalization step in [1], resulting in slightly faster tree construction as discussed in the next section.

4 Out-of-Core BVH Construction

In this section, we describe how to use the PBFS construction order to extend the hybrid BVH construction algorithm proposed by [2] to handle very large scenes. The underlying approach consists of two steps. First, several coarsest tree levels are constructed in a bootstrap pass to generate sufficient parallelism, using Linear Bounding Volume Hierarchy (LBVH), a spatial Morton codes-based algorithm. Next, the remaining tree is then constructed in BFS order using SAH-based strategies.

There is a significant difference in memory footprint between BVH and kd-tree construction. BVH construction does not split triangles or create duplicate triangle references. Consequentially, the primitive storage remains static throughout the whole construction and the final tree size can be bounded prior to construction. Based these observations, Lauterbach et al. [2] only allocate memory for primitives and the final tree at the beginning of the construction algorithm. Node splitting and triangle sorting are done in-place and little temporary memory is required for construction. While the memory overhead is relatively small, Lauterbach et al. [2] still cannot build trees that are too large to be stored in the GPU memory (e.g., up to 1.5 M triangles on a 1 GB GPU). An out-of-core solution is necessary to handle such large scenes.

Our BVH construction pipeline is illustrated in Fig. 6. The BVH construction also consists of two phases. First, all

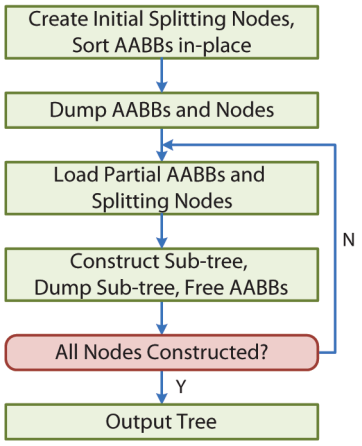


Fig. 6. Our BVH construction pipeline: after generating an initial list of a few thousand splitting nodes, the subtrees of these nodes are constructed iteratively.

primitives are loaded into the GPU memory and the AABBs are computed. The bootstrap pass and a few SAH iterations are performed to generate an initial list of a few thousand splitting nodes. All the AABBs are sorted in-place to match the order of their containing nodes. After that, the AABBs and constructed nodes are dumped to the CPU memory and all GPU memory occupied by phase one are freed.

In the second phase, we iteratively copy continuous portions of the splitting nodes and the AABBs of primitives contained in these nodes to the GPU, and construct subtrees for these nodes. At the end of each iteration, the constructed subtrees are dumped to the CPU memory and the primitive AABBs are freed. We bound the memory consumption of subtrees construction using the total number of primitives in the constructed subtrees. This bound is then used to maximize the number of subtrees constructed simultaneously in each iteration, just like in Section 3.2.

5 Results and Discussion

We have implemented the described algorithms in CUDA on a workstation with Intel Xeon dual-core 3.0 GHz CPU and an NVIDIA GeForce GTX 280 graphics card with 1 GB of memory.

5.1 KD-Tree Construction

In Fig. 7, we show nine test scenes with different scales ranging from 10 K to 7 M triangles. On our hardware, the kd-tree builder in [1] can only handle the first four scenes. It fails for scenes with more than 871 K triangles due to excessive memory consumption. Therefore, our PBFS scheme improves scene scalability by approximately one order of magnitude. Since the first four scenes can be processed in pure BFS order, our algorithm automatically degenerates to a two-stage construction and achieves comparable performance. This is illustrated in Table 1. M_{peak} is the peak memory consumption of our method, including the final kd-tree while excluding the scene data. The slight difference in T_{tree} is mainly due to the fact that we do not convert the constructed tree to a preorder traversal. Note that even in these small scenes, our PBFS scheme has a lower peak memory consumption than that of [1]. This is

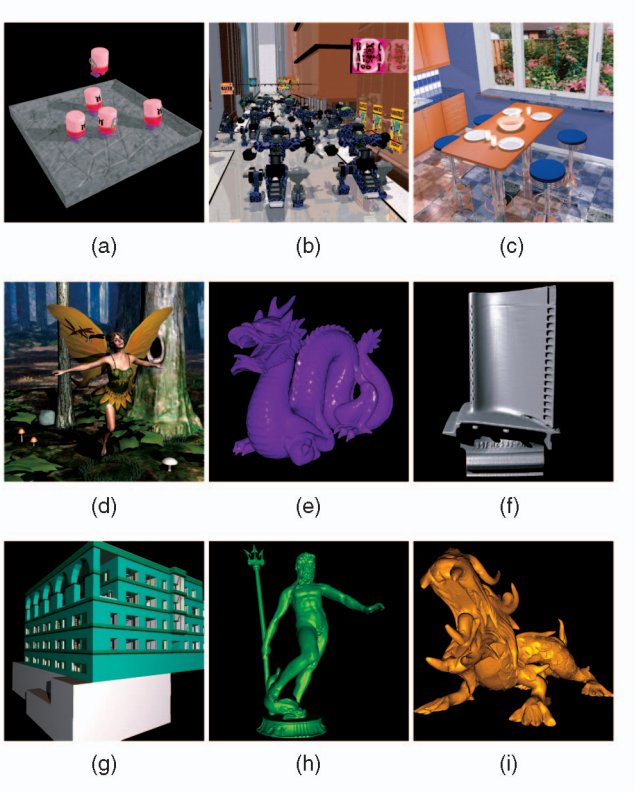


Fig. 7. Test scenes used in this paper. All the images have a resolution of $1{,}024 \times 1{,}024$. The Robots (b) is rendered with three lights and one reflection bounce. The Kitchen (c) is rendered with six lights and eight bounces. The Fairy Forest (d) is rendered with two point lights. All the other scenes are rendered with one point light. (a) Toys, 11 K triangles. (b) Robots, 71 K triangles. (c) Kitchen, 111 K triangles. (d) Fairy Forest, 178 K triangles. (e) Dragon, 871 K triangles. (f) Turbine Blade, 1,765 K triangles. (g) Soda Hall, 2,195 K triangles. (h) Neptune, 4,008 K triangles. (i) Asian Dragon, 7,219 K triangles.

largely due to our efficient clipped triangle storage as described in Section 3.4. An interesting fact is that T_{trace} is about twice as fast as reported in [1]. We attribute this performance divergence to hardware differences. Note that we employed the same ray tracing program as in [1]. Comparing to the GeForce 8800 Ultra GPU used in [1], the GeForce GTX 280 GPU used in this paper has a lower texture unit to core ratio. This may have a significant negative performance impact on the ray tracing kernel which uses textures to access kd-trees and scene data.

In Table 2, we compare our algorithm with the state-of-the-art multicore CPU kd-tree algorithms. The statistics of CPU methods are directly taken from [7] and [6] with the latter marked with superscript *. The CPU methods make different trade-offs between construction time and tree quality. We

TABLE 1
Comparison with the BFS Construction Order [1]

Scene	Our method			BFS construction		
	T_{tree}	T_{trace}	M_{peak}	T_{tree}	T_{trace}	M_{peak}
Fig. 7(a)	0.015s	0.026s	3 MB	0.012s	0.026s	8 MB
Fig. 7(b)	0.037s	0.085s	29 MB	0.038s	0.075s	50 MB
Fig. 7(c)	0.042s	0.332s	60 MB	0.043s	0.329s	90 MB
Fig. 7(d)	0.058s	0.127s	68 MB	0.065s	0.125s	123 MB

TABLE 2
Comparison with the Multicore CPU Methods

Scene	Our method			CPU methods	
	T_{tree}	T_{trace}	M_{peak}	$T_{tree}^{\min}$	$T_{trace}^{\min}$
Fig. 7(e)	0.170s	0.020s	272 MB	n/a	n/a
Fig. 7(f)	0.287s	0.041s	550 MB	0.690s*	0.091 s
Fig. 7(g)	0.461s	0.036s	746 MB	0.450s	0.040 s
Fig. 7(h)	0.849s	0.074s	747 MB	n/a	n/a
Fig. 7(i)	1.428s	0.108s	715 MB	1.600s	0.200 s

compare our tree construction time with the fastest construction method and compare our trace time with the highest tree quality method. M_{peak} is the peak memory consumption of our algorithm. It includes the final kd-tree but not the scene data. As shown, our algorithm can achieve comparable tree construction performance to these methods while providing higher quality trees with less ray tracing time.

An important feature of our algorithm is that, instead of using up all available GPU memory, the user can choose to specify a memory bound for kd-tree construction. In many practical applications, not all GPU memory can be used for tree construction—some memory has to be reserved for other data (e.g., animation data) or tasks (e.g., simulation). Our memory scalable algorithm is very useful in these types of situations. We tested three scenes under different memory bounds as shown in Table 3. "Unbounded" means the memory bound is taken as all available GPU memory, namely the total GPU memory minus the memory reserved for scene geometry, rendering, and the operating system. #SNS is the number of small-node stages launched during construction. As the memory bound decreases, the construction has to be split into more small-node stages to reduce peak memory consumption and results in less parallelism in individual small-node stages. For small scenes, this causes underutilization of the GPU, and slows down construction performance. For the Dragon scene, restricting the memory bound to less than half of the memory peak in the unbounded case results in a 10 percent performance loss. However, for larger scenes, even a small fraction of the intrinsic parallelism is sufficient to achieve full GPU utilization. For the Blade and Neptune scenes, the performance loss is only about 6 percent. "Minimum" means the minimum memory required by our algorithm to run, which is the total size of clipped triangles and the final constructed tree. This value is equivalent to the memory consumption of construction on CPUs. Working under this minimum memory on GPUs would lead to degenerate performance due to the lack of parallelism. At least a few more megabytes are required to get practical performance.

TABLE 3
Kd-Tree Construction under Different Memory Bounds

Scene	M_{bound}	#SNS	M_{peak}	T_{tree}
Fig. 7(e)	Unbounded	1	272 MB	0.170 s
	200 MB	3	170 MB	0.187 s
	150 MB	5	131 MB	0.194 s
	100 MB	7	93 MB	0.204 s
	Minimum	–	55 MB	–
Fig. 7(f)	Unbounded	1	550 MB	0.287 s
	400 MB	3	344 MB	0.296 s
	300 MB	5	260 MB	0.306 s
	200 MB	8	184 MB	0.315 s
	Minimum	–	107 MB	–
Fig. 7(h)	Unbounded	4	747 MB	0.849 s
	650 MB	6	646 MB	0.855 s
	500 MB	9	481 MB	0.870 s
	350 MB	18	320 MB	0.904 s
	Minimum	–	255 MB	–

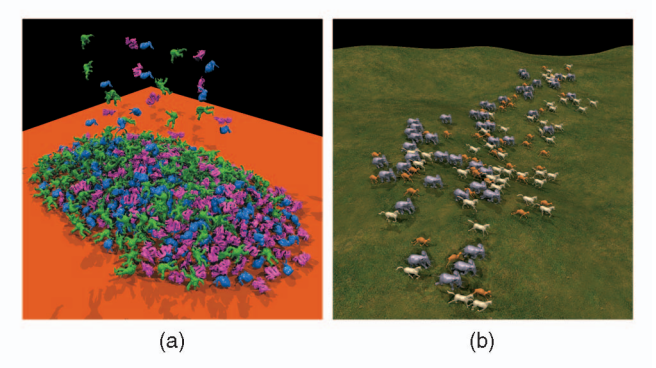


Fig. 8. Kd-tree construction and ray tracing of two large animated scenes. (a) 7,140 K triangles, 612 instances of three models each of which has 5-20 K triangles. (b) 6,763 K triangles, a terrain and 135 instances of three skinning meshes each of which has 17-85 K triangles. For each scene, at each frame, we construct a kd-tree and use it to ray trace the scene completely on the GPU. Images are rendered at $1{,}024 \times 1{,}024$ resolution with four point lights. Note that object instancing is solely used to simplify animation production and is not exploited by the kd-tree constructor. (a) Falling objects. (b) Running animals.

We also tested our kd-tree algorithm using the two large animated scenes shown in Fig. 8. The falling objects animation in Fig. 8a has gradually increasing scene complexity beginning with 560 K triangles and reaching 7,140 K in the end. This scene demonstrates how our performance and memory consumption changes with respect to the scene complexity. As illustrated in Fig. 9a, the memory peak of our construction algorithm exhibits a two-phase behavior. When the scene is small and can fit into the available memory, the peak grows rapidly at a roughly linear speed. As the scene becomes larger, our PBFS scheme takes effect and the memory peak oscillates at a relatively steady level. As the scene size increases further, the memory consumed by the scene geometry increases and the memory available for kd-tree construction decreases.

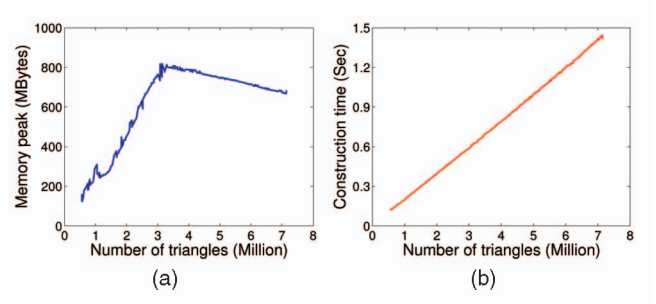


Fig. 9. Memory peak and performance of our construction algorithm for the animated scene shown in Fig. 8a. (a) Memory peak. (b) Construction performance.

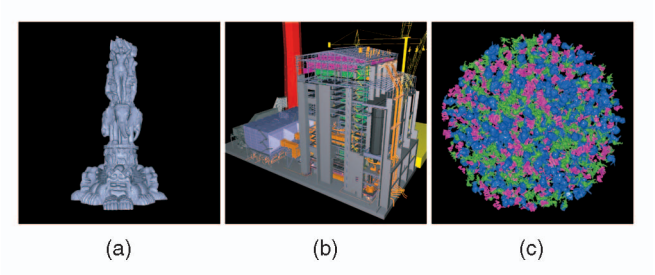


Fig. 10. Test scenes for out-of-core BVH construction. All images are rendered at resolution $1{,}024 \times 1{,}024$ with one point light. (a) Thai Statue, 10,000 K triangles. (b) Power Plant, 12,748 K triangles. (c) Swarm Objects, 20,021 K triangles.

Our construction algorithm thus reduces its memory peak accordingly. Regardless of the memory peak behavior, our construction time grows linearly with the number of triangles, as shown in Fig. 9b. The PBFS scheme successfully controls peak memory consumption with minimal performance penalty.

The example in Fig. 8b demonstrates the potential of our method in handling large animations. The scene geometry and animation consume 248 MB GPU memory. Excluding the memory reserved for rendering and the operating system, only 650 MB memory on the GPU is available for kd-tree construction. Our algorithm can handle that well and achieves interactive performance. Each frame takes approximately 1.84 seconds to render: the kd-tree construction takes about 1.46 seconds, and the remaining time is spent on ray tracing, shading, and animation preparation.

5.2 BVH Construction

Fig. 10 shows three test scenes which cannot be handled by the in-core BFS-based algorithm [2] due to the large memory consumption of geometry and the final tree. Lauterbach et al. [2] only handled scenes with less than 2 M triangles, while our out-of-core algorithm can support scenes with up to 20 M triangles. The statistics of construction timings and hierarchy quality are shown in Table 4. M_{CPU} is the peak memory consumption of an in-core CPU BVH construction algorithm. M_{peak} is the peak memory consumption of our BVH construction algorithm. T_{tree} is hierarchy construction time, including T_{copy}, the GPU-CPU data transfer time. T_{trace} is the relative ray tracing performance on a CPU ray tracer compared to the full SAH solution [8]. For all scenes, our constructed BVHs offer similar rendering performance to the CPU reference results.

The GPU memory bottleneck in our BVH construction algorithm is the AABB computation phase. In that phase, all geometry data and AABBs have to be stored in GPU memory. After the phase, the geometry data maybe freed and the total memory consumption no longer increases. Therefore, for our BVH construction algorithm, the minimum memory requirement $M_{minimum}$ is equal to M_{peak} in Table 4. In terms of CPU memory consumption, our method is exactly the same as a CPU construction algorithm.

Note that for the same tree quality, our out-of-core BVH construction is still slower than the in-core reference algorithm running on a eight-core CPU with 16 GB memory [8]. Even with PBFS, the speed of our BVH construction is still far behind GPU's ideal performance. The in-place BVH construction requires stronger memory consistency than what current GPUs offer and memory barriers have to be added to guarantee correctness. The memory barriers cause suboptimal latency hiding and result in performance degradation. Our main focus is to push the state of the art in the hierarchies that can be built by GPU-based algorithms, based on memory efficiency. Future GPU architectures like Fermi offer write caches and stronger memory consistency, which may result in significant boost of our BVH construction performance. In addition, we plan to use the CPU to construct a portion of the nodes in parallel with the GPU as a future work. Significant potential improvement maybe achieved if workloads can be efficiently balanced between the CPU and GPU. CPU-GPU data transfer time will also be eliminated for nodes constructed by the CPU.

TABLE 4
BVH Construction Timings and Hierarchy Quality

Scene	M_{CPU}	M_{peak}	T_{tree}	T_{copy}	T_{trace}
Fig. 10(a)	1,100 MB	452 MB	4.081 s	1.086 s	93%
Fig. 10(b)	1,430 MB	612 MB	7.561 s	1.429 s	93%
Fig. 10(c)	2,200 MB	897 MB	8.064 s	2.168 s	97%

6 Conclusion and Future Work

We have presented two GPU algorithms for constructing spatial hierarchies with controllable memory consumption, one for in-core kd-tree construction and the other for out-of-core BVH construction. Both algorithms are based on the PBFS construction order, and can handle scenes several times larger than previous GPU methods. The construction time is comparable with the state-of-the-art multicore CPU methods and our tracing performance outperforms these methods.

The PBFS scheme provides an effective approach for balancing memory usage while exploiting the parallelism in general-purpose GPU computation. In the future, we would like to apply this scheme to other GPU algorithms in scientific computations and related applications. Although promising, our kd-tree algorithm still has some limitations—it does not control the final tree size. To cope with available memory less than the tree size, tree-size-controlling techniques as in [11] have to be incorporated into our PBFS scheme.

Data transfer between GPUs and CPUs consumes significant time in the out-of-core BVH construction. In this paper, we focus on using the PBFS scheme to reduce in-core peak memory requirement. Data transfer techniques, like host-mapped GPU memory, are orthogonal to our work. In the future, we would like to incorporate such techniques to improve the overall efficiency of construction.

The problem of memory consumption on GPUs is fundamentally different from its counterpart on single-core or multicore CPUs, because hundreds of thousands of threads are launched simultaneously on GPUs. The same problem should be present on other kinds of many-core platforms, such as Fermi and Larrabee. The PBFS

scheme proposed in this paper is not limited to CUDA, our choice of the implementation language. It is also suitable for other languages such as Compute Shader and OpenCL.

ACKNOWLEDGMENTS

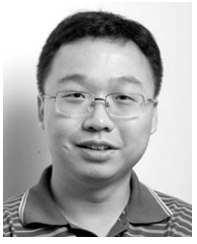
Kun Zhou was partially funded by the NSF of China (No. 60825201) and NVIDIA. Christian Lauterbach and Dinesh Manocha's work is partially supported by ARO Contract W911NF-04-1-0088, the US National Science Foundation (NSF) awards 0636208, 0917040 and 0904990, the US Defense Advanced Research Projects (DARPA)/RDECOM Contract WR91CRB-08-C-0137, and Intel.

REFERENCES

[1] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-Time KD-Tree Construction on Graphics Hardware," *ACM Trans. Graphics,* vol. 27, no. 5, p. 126, 2008.
[2] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," *Proc. Eurographics '09,* 2009.
[3] J. Goldsmith and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE Computer Graphics and Applications,* vol. 7, no. 5, pp. 14-20, May 1987.
[4] J.D. MacDonald and K.S. Booth, "Heuristics for Ray Tracing Using Space Subdivision," *The Visual Computer,* vol. 6, no. 3, pp. 153-166, 1990.
[5] W. Hunt, W.R. Mark, and G. Stoll, "Fast KD-Tree Construction with an Adaptive Error-Bounded Heuristic," *Proc. IEEE Symp. Interactive Ray Tracing '06,* pp. 81-88, Sept. 2006.
[6] M. Shevtsov, A. Soupikov, and A. Kapustin, "Highly Parallel Fast KD-Tree Construction for Interactive Ray Tracing of Dynamic Scenes," *Proc. Eurographics '07,* pp. 395-404, 2007.
[7] A. Soupikov, M. Shevtsov, and A. Kapustin, "Improving Kd-Tree Quality at a Reasonable Construction Cost," *Proc. IEEE Symp. Interactive Ray Tracing '08,* pp. 67-72, Sept. 2008.
[8] I. Wald, "On Fast Construction of SAH-Based Bounding Volume Hierarchies," *Proc. IEEE Symp. Interactive Ray Tracing '07,* pp. 33-40, Sept. 2007.
[9] I. Wald, T. Ize, and S.G. Parker, "Special Section: Parallel Graphics and Visualization: Fast, Parallel, and Asynchronous Construction of BVHs for Ray Tracing Animated Scenes," *Computers and Graphics,* vol. 32, no. 1, pp. 3-13, 2008.
[10] M. Ernst and G. Greiner, "Early Split Clipping for Bounding Volume Hierarchies," *Proc. IEEE Symp. Interactive Ray Tracing '07,* pp. 73-78, Sept. 2007.
[11] C. Wachter and A. Keller, "Terminating Spatial Hierarchies by A Priori Bounding Memory," *Proc. IEEE Symp. Interactive Ray Tracing '07,* pp. 41-46, Sept. 2007.
[12] C. Lauterbach, S.-E. Yoon, M. Tang, and D. Manocha, "ReduceM: Interactive and Memory Efficient Ray Tracing of Large Models," *Computer Graphics Forum,* vol. 27, no. 4, pp. 1313-1321, 2008.
[13] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Trans. Graphics,* vol. 27, no. 3, pp. 1-15, 2008.
[14] B.C. Budge, T. Bernardin, S. Sengupta, K.I. Joy, and J.D. Owens, "Out-of-Core Data Management for Path Tracing on Hybrid Resources," *Proc. Eurographics '09,* 2009.
[15] X.-H. Sun and L.M. Ni, "Scalable Problems and Memory-Bounded Speedup," *J. Parallel and Distributed Computing,* vol. 19, no. 1, pp. 27-37, 1993.
[16] I. Wald and V. Havran, "On Building Fast kd-Trees for Ray Tracing, and on Doing That in O(N log N)," *Proc. IEEE Symp. Interactive Ray Tracing '06,* pp. 61-69, Sept. 2006.

Qiming Hou received the BS degree in the Academic Talent Program of Tsinghua University (Mainland China) in 2006 and is currently working toward the PhD degree in computer science at Tsinghua University (Mainland China). His research interests include general-purpose processing using graphics processors,compiler techniques, realistic rendering, and interactive rendering. For the past three years, he has been an intern consultant in Microsoft Research Asia.

Xin Sun received the bachelor's and PhD degrees in computer science from Zhejiang University, Hangzhou, China, in 2002 and 2008, respectively. After that, he joined Internet Graphics Group in Microsoft Research Asia as an associate researcher. His research interests lies in real-time global illumination rendering and GPU-based photorealistic rendering.



Kun Zhou received the BS and PhD degrees in computer science from Zhejiang University in 1997 and 2002, respectively. He is a Cheung Kong Distinguished professor in the Computer Science Department of Zhejiang University, and a member of the State Key Lab of CAD&CG, where he leads the Graphics and Parallel Systems Group. Prior to joining Zhejiang University in 2008, he was a leader researcher of the Internet Graphics Group at Microsoft Research Asia. His research interests include shape modeling/editing, texture mapping/synthesis, real-time rendering, and GPU parallel computing.

Christian Lauterbach received the diploma in computer science from the University of Bremen, Germany and the PhD degree in computer science from the University of North Carolina at Chapel Hill under the advice of Dinesh Manocha. He is currently working at Google.

Dinesh Manocha received the PhD degree in computer science from the University of California at Berkeley 1992. He is currently the Phi Delta Theta/Mason Distinguished professor of computer science at the University of North Carolina at Chapel Hill. He has received Junior Faculty award, Alfred P. Sloan Fellowship, the US National Science Foundation (NSF) Career award, Office of Naval Research Young Investigator award, Honda Research Initiation award, Hettleman Prize for Scholarly Achievement. Along with his students, he has also received 12 best paper and panel awards at the leading conferences on graphics, geometric modeling, visualization, multimedia, and high-performance computing. He is a fellow of the ACM. He has published more than 280 papers in the leading conferences and journals on computer graphics, geometric computing, robotics, and scientific computing. He has also served as a program committee member and program chair for more than 75 conferences in these areas, and editorial boards of many leading journals. Some of the software systems related to collision detection, GPU-based algorithms, and geometric computing developed by his group have been downloaded by more than 100,000 users and are widely used in the industry. He has supervised 18 PhD dissertations.