Implicit and Dynamic Trees for High Performance Rendering

Nathan Andrysco* Xavier Tricoche†

Purdue University

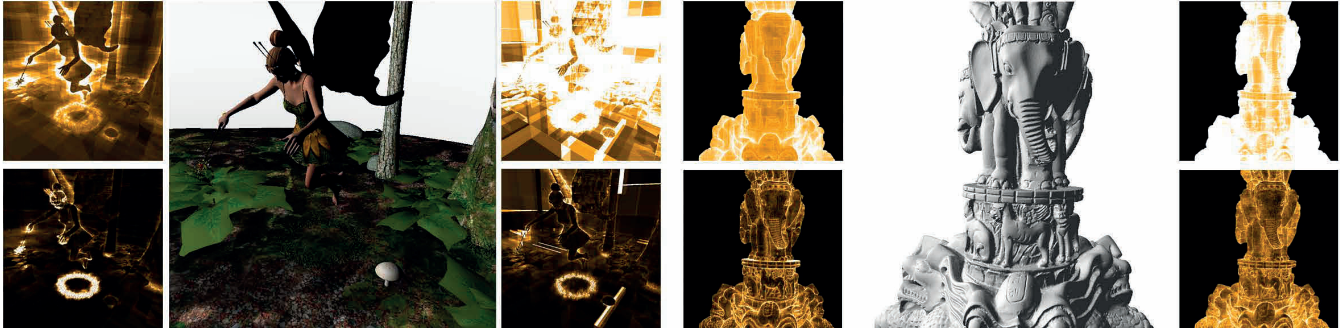

Figure 1: Comparison between using a regular matrix kd-tree (histograms on left of images) and a SAH built kd-tree (right) for ray tracing. In the fairy forest scene, the matrix tree traversed 53% fewer tree nodes (top row of images) and tested 23% fewer triangles (bottom) than its SAH counterpart. Similarly for the Thai statue which is composed of a million small triangles, the matrix tree required 51% fewer traversals and 39% fewer triangle intersection tests. For each pixel, pure black indicates zero traversals/tests and pure white indicates either 100+ node traversals or 250+ triangle tests. Because of these fewer node traversals and triangle tests, our method allows for speed-ups of over 200%.

Abstract

Recent advances in GPU architecture and programmability have enabled the computation of ray casted or ray traced images at interactive frame rates. However, the rapid performance gains of the hardware cannot by themselves address the challenge posed by the steady growth in the geometric and temporal complexity of computer graphics datasets. In this paper we present a novel versatile tree data structure that can accommodate both sparse and dense data sets and is more memory efficient than state-of-the-art representations. A key feature of our data structure for rendering applications is that it fully supports efficient, parallel building. As a result, our implicit tree representation significantly outperforms existing techniques in the rendering of time-varying scenes. We show how this data structure can be extended to encode other classic representations such as BSP-trees and we discuss the high-performance implementation of our general approach on the GPU.

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

1 Introduction

Recent advances in multicore architectures and GPUs, as well as in corresponding programming languages have brought researchers closer to the long standing goal of creating interactive high-quality renderings of realistic scenes. The performance of the corresponding algorithms crucially depends on the underlying data structures, which support efficient search and data access. However, as datasets increase both in geometric and temporal complexity, the build and update times of these data structures are becoming as important as the structure's spatial query performance in real-time applications. This problem is particularly pressing in the context of ray tracing. Indeed, ray tracing has long been a completely offline algorithm and it has yet to become a fully interactive technique. While interactive ray tracing has been shown to be possible for fairly simple scenes, large and dynamic scenes continue to pose a performance challenge that is not properly addressed by existing strategies. In scenes where the geometry undergoes significant changes between frames, as is the case in video games, the build times of currently available data structures can easily outweigh the ray tracing time and constitute a primary bottleneck in the overall rendering performance.

In this paper we propose a novel spatial data structure that is suited for both interactive rendering and parallel construction on the GPU. By leveraging a tree representation built upon sparse matrices, we enable faster tree traversals than previous data structures while achieving a smaller memory footprint. This data structure can be built very efficiently in parallel and it significantly improves upon the current state-of-the-art in rendering. In particular, we show that our implicit tree solution clearly outperforms the surface area heuristic (SAH), which constitutes the de facto standard. A visual comparison of render cost between our method and SAH is shown in Figure 1. In short, our proposed tree structure leads to a much smaller number of examined nodes and triangles on average, which results in a significant decrease in overall rendering time. In addition, the construction of our trees is extremely well suited for GPU architectures and exploits the massively parallel nature of the hardware to achieve a sharply reduced build time compared to its SAH counterpart.

A basic aspect of our solution consists in using sparse matrix representations to efficiently encode spatial datasets, as was recently proposed in the visualization literature [2]. In contrast to previous work, we propose here a data structure that is very flexible, naturally accommodates the types of trees used in high-performance rendering applications, has a small memory footprint in the context of graphics datasets, and can be built very efficiently in a dynamic

*e-mail:nandrysc@cs.purdue.edu

†e-mail:xmt@cs.purdue.edu

Graphics Interface Conference 2011
25-27 May, St. John's, Newfoundland, Canada
Copyright held by authors. Permission granted to
CHCCS/SCDHM to publish in print form, and
ACM to publish electronically.

setting. Specifically, we consider in the following sections octrees, kd-trees, and BSP-trees and discuss their encoding in the proposed framework. Motivated by the fact that no single data structure offers an optimal solution to all rendering scenarios [30], our focus in this paper is on a versatile data structure that can be used as a building block for a wide range of spatial data representations and lends itself to a host of fine-grained optimizations strategies.

The remainder of this paper is organized as follows. We start by reviewing relevant previous work on data structures for ray tracing in Section 2. We then present the basic sparse data representation that forms the core of our proposed data structures in Section 3. We proceed by describing in detail how various tree types can be built upon this encoding (Section 4). Fast tree construction on parallel architectures and additional implementation considerations are discussed in Section 5 before presenting results in Section 6. Finally, we conclude and comment on interesting avenues to extend this work in Section 7.

2 Prior Work

The high computational cost of ray tracing has led the graphics community to devise numerous acceleration strategies. A complete survey of the corresponding literature is clearly beyond the scope of this paper and we refer the interested reader to Wald's thesis [26], which provides an excellent overview of many of these acceleration methods. Most relevant to the focus of this paper are the data structures that have been proposed to support ray tracing acceleration.

2.1 Acceleration Data Structures

A basic requirement to accelerate ray tracing computation is to leverage data structures that minimize the number of ray-primitive intersection tests. These spatial structures mostly consist of uniform grids, octrees, kd-trees, BSP-trees, or bounding volume hierarchies (BVH). We briefly summarize relevant contributions in the following paragraphs, but more detailed surveys can be found elsewhere [10, 26, 30].

The parallel construction of grids for ray tracing was discussed in an early work by Molnar *et al.* [21] and in the most recent publication of Kalojanov and Slusallek [16]. Grids are generally suitable for scenes that have relatively uniform densities of triangles. When triangles are inhomogeneously scattered throughout the scene, an adaptive data structure typically offers better results. The simplest of these data structures is the octree, which was first used in ray tracing by Glassner [7].

The relatively inflexible nature of octrees has steered research towards other, more adaptable data structures that better fit ray tracing scenes [10]. Kd-trees are the canonical choice in this context, whereby each node is split along an axis-aligned plane. We provide additional detail on the topic in Sections 2.2 and 5.2. A more general form of kd-trees is the binary space partition (BSP) tree, which allows for non-axis aligned split planes. The BSP-tree's increased flexibility makes the optimal placement of the split plane quite expensive such that the corresponding overhead typically outweighs the benefit associated with improved query performance in rendering applications. Even by restricting the number of possible split planes [17, 4], the build times of a BSP tree can be orders of magnitude larger than those of a kd-tree and thus little work has been explored with BSP trees in the context of ray tracing dynamic scenes.

An alternative to subdividing the volume along split planes is to use a hierarchy of bounding objects. Hierarchal grids is one such method, with an early work in the context of ray tracing done by Klimaszewski and Sederberg [18]. A more recent work in this area was proposed by Wald *et al.* [29], who included coherent rays and frustum culling in conjunction with hierarchial grids to achieve high-performance rendering.

Bounding volume hierarchies (BVH), which segment the scene using simple primitives (*i.e.* boxes, spheres, etc.), is an approach that has received a great deal of attention recently. These works on BVHs have contributed support for quick, incremental changes [31], improved splitting schemes for more efficient data structure traversal [25], and support for construction on the GPU [19]. Wald *et al.* [27] provide a comprehensive review of building and traversing BVHs. We note that our proposed data representation could easily be extended to represent BVHs, but their traversal and construction in the context of ray tracing is outside the scope of the paper.

2.2 Surface Area Heuristic

The *surface area heuristic*, or *SAH*, is a popular method for generating kd-tree split planes. This heuristic attempts to generate a tree that hopefully allows for near optimal ray tracing performance. This top-down construction scheme was first proposed by Goldsmith and Salmon in [8]. Its basic idea consists in combining the cost of traversing the tree with the probability and associated cost of intersecting a node:

$$\text{cost} = C_T + C_L \frac{\text{SA}_L}{\text{SA}_P} + C_R \frac{\text{SA}_R}{\text{SA}_P},$$

where C_T is the cost of traversing the node, $C_{L/R}$ are the costs of the two children, and the probability of a ray hitting a node is the ratio of surface area between the children nodes, $\text{SA}_{L/R}$, and the parent node, SA_P. After evaluating all the costs for each candidate split plane, one chooses the split plane associated with the minimum cost. Note that there is a finite set of split plane candidates since the local minima of the cost function are located at triangle vertices.

The time complexity of building these trees is $\mathcal{O}(N \log N)$ [28], where N is the number of triangles in the scene. A number of subsequent papers investigated less expensive heuristics to reduce the build time at the cost of decreased build quality [33, 14, 24].

2.3 Matrix Trees

Our data structure builds upon a recently proposed tree representation called *matrix tree* [2]. In this representation, each level of the tree is encoded as a regular raster grid, or matrix. Yet, when a level of the tree is sparsely populated, a regular encoding is preserved by using a sparse matrix data representation, which significantly reduces the memory footprint. In turn, the structuredness of the encoding enables faster tree traversal than alternative approaches for pointer-based trees. Observe that we discuss fast tree traversal strategies in Section 2.4. Unfortunately, the construction procedure associated with this representation is prohibitively slow and the strong restrictions imposed on the trees by this approach makes it unsuitable for rendering applications. We address these basic limitations by introducing a novel data representation that allows for more efficient access (Section 3), the ability to represent a broader class of trees (Section 4.1), and quick parallel construction (Section 5).

2.4 Tree Traversal

The benefits of a tree in a ray tracing scenario is directly linked to the efficiency of the associated traversal scheme. A stack, which is one of the most common methods used to perform ray-tree traversal, records nodes that will need to be revisited if a dead-end is reached during traversal. However, a stack is a suboptimal choice on the GPU as pointed out by Foley and Sugerman [6]. The fact that each ray in a parallel computation block requires memory allocated for its entire stack leads to a large increase in the number of register spills and/or global memory accesses. The authors proposed two stackless traversal algorithms better suited for the GPU, called *KD-Restart* and *KD-Backtrack*. In KD-Restart, each traversal that reaches a dead-end proceeds from the root of the tree. This

method avoids the stack-based traversal problems, but does so at the expense of a large increase in nodes visited. In KD-Backtrack, one incrementally steps back towards the root after reaching a dead-end. This requires that the pointers from children to parents be either stored (which can be very costly) or implicitly derived. Another recently proposed GPU-friendly traversal method is *KD-Jump* [13], which maintains a stack-like structure using only a few integers.

A hybrid approach of stack and KD-Restart traversals is the *short stack* [12], which maintains a small stack that does not introduce large memory overhead. If the stack becomes empty and a dead-end is reached, one reverts to the KD-Restart algorithm since the stack is not large enough to record all the nodes visited. A balance must therefore be found between large stacks (increased memory dependency) and small stacks (too many restarts), whereby the authors recommend a stack size of 3.

All of the aforementioned approaches make use of leaf and neighbor finding tree operations with linear complexity. The Matrix trees method [2] in contrast, achieves sublinear traversal times thanks to the ability of matrix trees to leverage their regular encoding to hash directly into any given node. As a result, leaf finding can be performed in $\mathcal{O}(\log h)$ time, as opposed to $\mathcal{O}(h)$ for previous methods, where h is the height of the tree. Specifically, one first hashes into the node corresponding to the leaf level. In the case of a non-existent leaf, this hash yields a root-to-leaf path on which a binary search can be performed. Similarly, the algorithm makes use of the hash and binary search to efficiently determine where a neighbor resides in memory. Matrix trees also allow for efficient GPU code since node information is implicity defined. As a result, node values do not need to be maintained in local variables or retrieved from global memory, which results in inherent speed improvements due to decreased register pressure and no global memory accesses.

3 Improved Sparse Matrix Representation

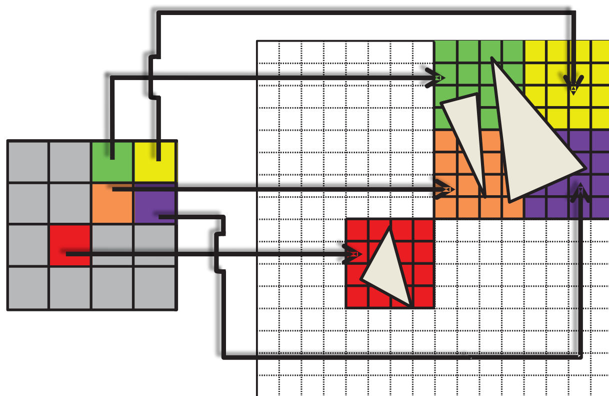

Figure 2: An example of the sparse block matrix representation. The only nodes we allocate are those contained within the blocks that intersect the triangles. The left matrix is fully allocated and is used to store pointers to the blocks in the right matrix.

As noted earlier, matrix trees are prohibitively slow to construct due to the very nature of their sparse encoding strategy. This shortcoming, in turn, makes them completely unsuitable for the ray tracing of dynamic scenes. In fact, the Compressed Sparse Row (CSR) format used [3], while optimized for the multiplication of matrices, is comparatively inefficient for spatial queries in general purpose graphics. Specifically, a random insertion into the data structure requires $\mathcal{O}(n^2)$ time, where n is the number of nodes along a dimension of the matrix. Random access on the other hand requires $\mathcal{O}(\log_2 m)$ time, where m is the average number of non-zero entries along a dimension of the matrix.

We propose the use of a block based scheme [3] that has $\mathcal{O}(1)$ insertion time and $\mathcal{O}(1)$ access time. In this sparse matrix representation, we decompose the overall matrix into a limited number of smaller matrices. In empty areas we do not need to allocate one of these smaller matrices. In graphics applications the interesting data tends to be organized in clusters, which makes this data structure very efficient. We note that recent GPU hashing data structures would provide similar data structure size and speed. However, Lefebvre and Hoppe's approach [20] has a rather costly build time which is not suitable for dynamic scenes. Alcantara *et al.* [1] demonstrated a quicker construction method for their hash data structure, but the $\mathcal{O}(\log_2 n)$ time for each insertion is expensive compared to our proposed method.

Figure 2 shows a schematic of our data structure. In this figure, the left matrix is used to store pointers to blocks. We will hereby refer to this matrix as the *pointer matrix*. The right matrix is comprised of multiple smaller matrices, or blocks. For sparse data sets, the majority of this *block matrix* will not be allocated, which results in efficient storage in memory while maintaining constant access time.

To achieve $\mathcal{O}(1)$ access time, we first compute the corresponding index position in the pointer matrix by dividing the index of the block matrix by the size of a block. Using this computed index, we look up the value at the corresponding position in the pointer matrix. This value will be either a NULL value, meaning the corresponding block does not exist, or a pointer value to an allocated block. If we have a valid block, we make another constant time calculation to get the corresponding index within this block, which is the final location of our data. The overall cost is some minor computation and two memory access, one for the pointer matrix and one for the block. Since we are only accessing global memory twice, as opposed to $\mathcal{O}(\log_2 m)$ times using the CSR data representation, we achieve a dramatic speedup on the GPU since we avoid the prohibitive memory latency on this architecture. For the volume ray casting test datasets used in [2], we achieved a ~10 FPS speed increase across the board, which amounts to a ~75-280% speedup over that previous work. These results indicate that the traversal algorithm associated with the matrix tree technique is limited by memory bandwidth. The matrix representation that we are proposing alleviates this problem by taking advantage of the typical spatial distribution of graphical primitives in rendering scenarios. We also observed speed improvements on the CPU, though less significant than on the GPU due to the much larger memory access penalty on the graphics hardware.

4 Implicit KD-Trees and BSP-Trees

As mentioned previously, the matrix tree technique [2] achieves very efficient data access by leveraging the uniform and isotropic nature of the tree hierarchy to hash directly into any given node. This requirement on the tree structure imposes severe restrictions on the types of trees that can be accommodated by this approach. In particular, kd-trees could only have splits at node centers and a single split direction is allowed per level. These kd-trees are hereafter referred to as *regular* kd-trees, in contrast to *general* kd-trees. Our proposed solution enables an implicit encoding of a much broader class of trees without sacrificing the performance of random queries. Specifically, we store the (arbitrary) split plane information at each node. In this setup it is no longer possible to hash directly into a target node. However, we can still implicitly derive parent and child location at any node. As a result, we are able to represent BSP-trees without explicitly storing pointers, which helps us greatly reduce the tree's memory footprint. Below we explain how to represent general kd-trees and BSP-trees using this pointerless representation. To our knowledge, this implicit and

GPU-friendly encoding of general kd-trees or BSP-trees is the first of its kind. Furthermore, we believe using a similar scheme to the one provided would be able to efficiently and implicitly represent trees outside of the graphics scope.

4.1 General KD-Trees

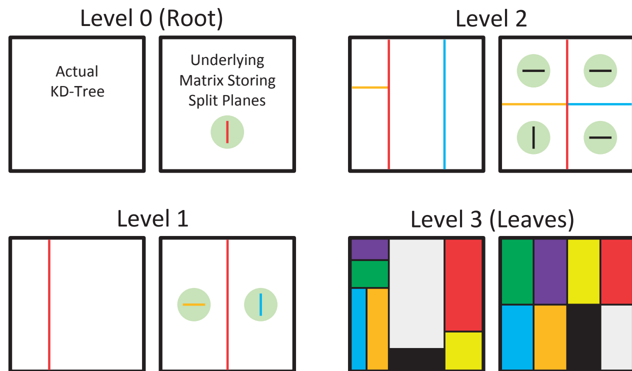

Figure 3: A side-by-side comparison between an example kd-tree (left) and its representation in memory (right). Split planes (represented as circles) are stored at each non-leaf node in the tree. To help the reader better visualize the data structure, each level of the example kd-tree uses the split planes from previous levels, but it should be noted that the split planes are only stored at the level in which they are defined.

A key observation underlying our proposed kd-tree encoding solution is that the parent/child relationships can be maintained regardless of the direction chosen to split the underlying matrix at each level. Refer to Figure 3. It follows that we can arbitrarily choose a direction of a submatrix that we double between levels of the tree. By alternating split directions between levels, we keep the matrix as close to a cube as possible, and the block based sparse matrix representation that we outlined in Section 3 remains well suited for representing the data. However, the matrix can no longer serve as a basis for deriving spatial information. To address this problem, we simply store the split direction and position at each node. By storing our general kd-trees in this outlined manner, we effectively cut the storage requirements in half due to a pointer-based node representation requiring a minimum of 4 bytes to address its children. One can no longer use the full sub-linear matrix tree traversal scheme [2], but due to the underlying data structure's ability to derive memory locations, one can use other GPU friendly schemes such as KD-Jump [13] or the binary search between a node and the root.

4.2 Arbitrary Split Planes

The basic difference between general kd-trees and BSP-trees is that kd-trees have axis-aligned split planes while BSP-trees have split planes with arbitrary orientation. For BSP-trees, we follow the same storage scheme as in the previous section describing general kd-trees. Since split planes are not axis-aligned we need to store more information at each node, namely the plane's normal and distance from the origin. Similar to general kd-trees, BSP-trees are not amenable to the matrix tree traversal algorithm. The initial leaf finding operation essentially amounts to the less efficient *KD-Restart* algorithm [6]. Again, it should be noted that this data structure saves at least 4 bytes per node over previous methods and gains the ability to do a binary search from any given node to the root, which can be leveraged to speed up an algorithm like *KD-Backtrack* [6].

5 Parallel Tree Construction

We present in this section a parallel tree building algorithm suited for parallel architectures such as multi-core CPUs and GPUs. Although our construction algorithm can be equally effective on the CPU, we focus our presentation on building the trees directly on the GPU. The algorithm is a bottom-up regular tree build, suitable for either regular octrees or kd-trees. It should be noted that our algorithm outperforms SAH trees in both build times (Section 5.2) and rendering (Section 6).

5.1 Octrees and Regular KD-Trees

A tree composed of regular matrices allows for an efficient bottom-up build. To construct the bottom level, we use an algorithm similar to the one proposed by Kalojanov and Slusallek [16] (with a variant done by Ivson *et al.* [15]). In their work, the authors are building uniform grids in parallel for the ray tracing of triangle primitives. Applying this approach for a regular tree structure, we construct a uniform grid at each level. Unlike prior work, however, our levels have sparse encodings, which requires extra build steps and modifications to the purely uniform and dense grid based algorithms. Pseudocode is shown in Figure 4. We further elaborate on each step of the algorithm below.

1. Besides the triangles in the scene and their bounding box, the only input of our algorithm is an upper bound on the tree height. From this, we can easily compute the basic properties of each matrix (*e.g.,* node sizes, matrix dimensions, etc.) at each level of the tree [2]. Since kd-trees and octrees have relatively limited heights (typically less than 25 due to leaf sizes approaching the sizes of the triangles themselves), this loop is best executed using a single-core. This is the only serial step of the algorithm, but it can be computed very quickly.

2. We invoke a parallel kernel that operates on each triangle of the scene. For each triangle, we count the number of nodes it intersects and also mark the blocks of our underlying sparse matrix data structure that contain these triangles. All write conflicts are avoided since each triangle has its own counter and we are simply setting a bit for each block.

3. We use an exclusive sum scan on the marked block vector. Our implementation uses the scan from the CUDPP library [9]. The output of the scan is used to determine how many blocks need to be allocated to represent our tree.

4. Another kernel is invoked that assigns pointers going from the pointer matrix to the allocated blocks. A more detailed description of our block based storage scheme is found in Section 2. These assignments are done by examining the marked block vector (step 2) and the output of the scan of this vector (step 3). If the block is marked, we simply use the value at the same position in the scan vector as the offset that addresses the appropriate block.

5. Similarly to step 3, we scan the triangle-node intersection counter vector, which tells us how much memory must be allocated to store all triangle pointers. This computation is analogous to the scheme used in [16].

6. Another kernel operates on each triangle. In this kernel we output a list of pairs that records both the triangles and the nodes they intersect. Unlike the work done in [16], we pass in the result of the scan from step 5 to avoid the need of shared memory or atomic writes. We observe that from this scan, we know exactly where in memory that each triangle should begin writing its node-triangle pairs. From this memory offset, we can use a local counter to increment our position in memory as we calculate triangle node intersections.

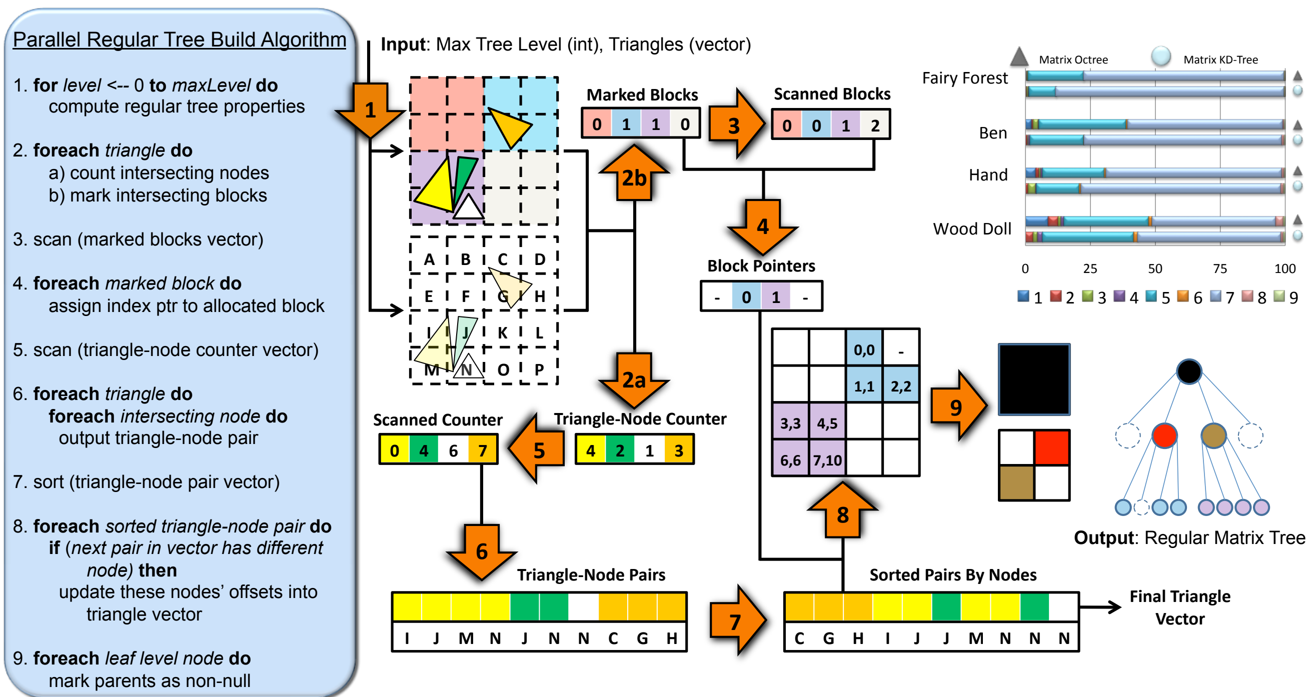

Figure 4: Pseudocode and visual explanation of our parallel algorithm for regular tree construction. The graph shows the percentage of time spent in each phase of the algorithm.

7. We sort the node-triangle pairs by node through CUDPP's radix sort [23]. The triangle portion can be directly copied to texture memory to yield the final triangle pointer vector. As in [16], this is by far the most expensive computational step.

8. The sorted nodes from the previous step are passed onto another kernel. We pass in another vector to record the node offsets into the triangle vector. At each position in the sorted node vector, we simply examine the neighboring values to determine the appropriate offsets into the triangle vector that belong to each node. If the node value at the neighbor differs from the node at our current location, we have found the border in the triangle vector that separates which triangles belong to the two nodes. From this, we update our nodes' offsets into the triangle vector with the index of the current thread.

9. The bottom level of the tree is now built and the corresponding information must be propagated up the tree. This is done by invoking one final kernel that examines each of the nodes' offsets from the previous step. If a node's offset indicates that the node is valid, we mark all the nodes up to the root as valid. A node is considered valid if it was updated in the previous step. To help distinguish between existing and non-existing nodes, we initialize these offsets to some NULL value prior to step 8. If no triangles intersect a particular node that node's offset will remain NULL and we know this node is not part of our tree.

5.2 General KD-Trees

General kd-trees have splits that do not cross the node center and have different split directions at each level. In addition to an increased memory footprint per node, the lack of regular structure also prevents the efficient bottom-up tree build discussed in Section 5.1. Previous work [10, 26, 30] has established general kd-trees based upon the surface area heuristic as the de-facto state-of-the-art for fast ray tracing. The state-of-the-art algorithms for the building of SAH kd-trees on the GPU [33] and the CPU [5] are more algorithmically complex and are inherently slower. These algorithms require incremental, segmented memory allocation, whereas our method requires few memory allocations and subsequently is able to store information contiguously in memory. Further, these SAH build algorithms require multiple sorts of the data, whereas our method only requires one. We enable the comparison between rendering times by building optimal SAH trees on a single-core CPU and if needed, we transfer it to the GPU.

6 Results

Results were obtained on a Windows 7 64-bit machine with an Intel Core i7 x980 (3.33 GHz) processor and NVIDIA GeForce GTX 480 GPU. Our underlying block based sparse matrix representation uses block sizes of 8x8x8. Tree heights for both the matrix and SAH builds were chosen to get the best rendering times possible. On the CPU we use OpenMP for parallel ray tracing, but do not take advantage of any available SIMD extensions. CUDA was used for all GPU code, where each thread represents a single ray. Times shown are for both primary and shadow rays.

We emphasize that the focus of our results is on the construction and the use of the spatial data structures that accelerate ray tracing. Our test framework is setup in such a way that only minimal code changes are needed to test the different tree constructions with multiple traversal methods. Even though our test setup lacks certain performance optimizations (*i.e.* ray packets, persistent threads, etc.) or more sophisticated rendering methods (*i.e.* caustics, anti-aliasing, etc.), we note that these optimizations and more complex rendering methods are not limited by the kinds of trees used in our tests, including our own data structure. For ray-triangle intersection tests, we use the method provided in Wald's Thesis [26].

Figure 6 shows the speed-up of our method over other commonly used algorithms. Absolute construction and rendering times can be

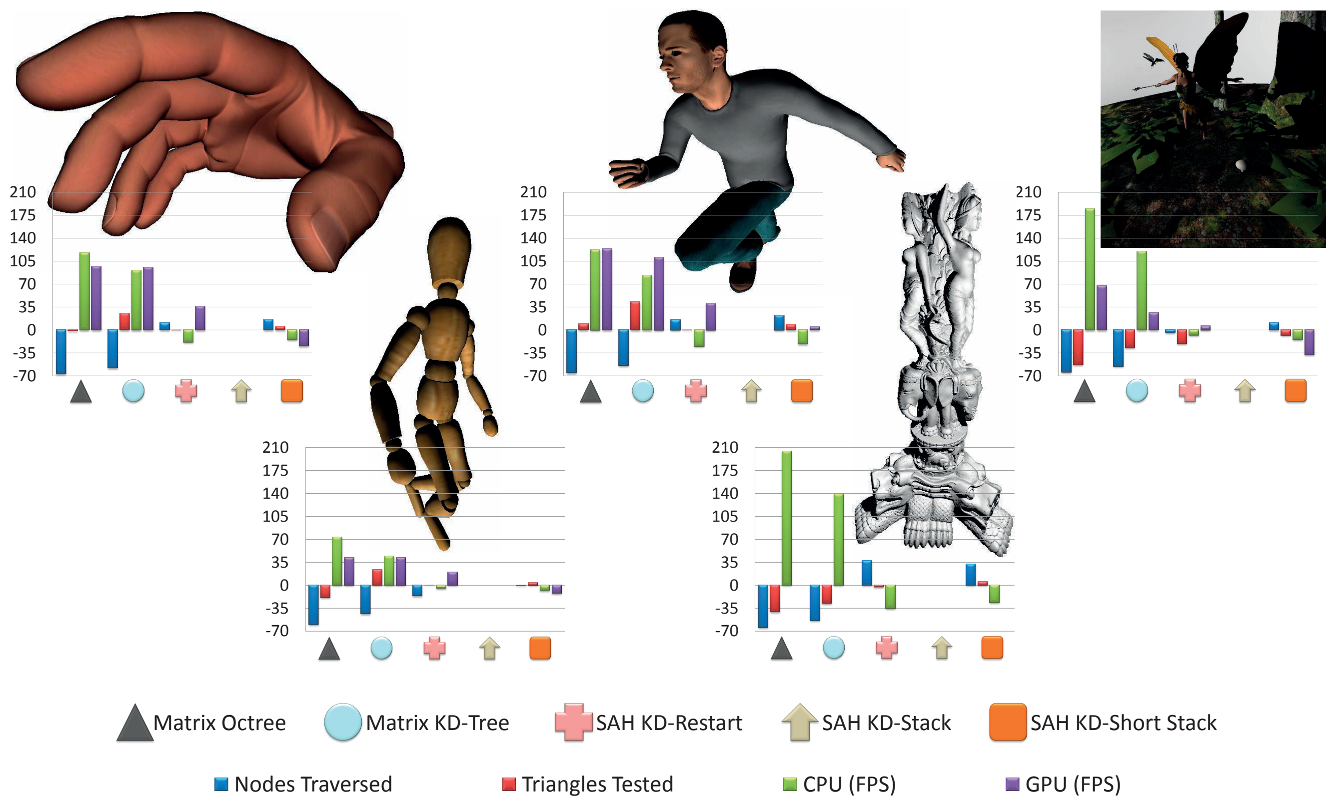

Figure 6: Images of the test data sets with graphs showing the percentage increase/decrease of various methods compared to the previous state-of-the-art, SAH built kd-trees using stack traversal. The decreased node/triangle traversals in the matrix tree method results in significant speed increases during rendering. Results reported are for both primary and shadow rays. Refer to table 1 for the data sets used and their geometric complexity.

found in Table 1. Most data sets were time varying, but we also included a large static scene composed of tiny triangles (Thai statue) to show our method does not suffer when complex geometry is introduced. The results were averaged over multiple view points at multiple time frames. We show results using our data structure with the matrix tree traversal method [2], which uses efficient, sub-linear leaf and neighbor finding. For SAH built trees, we show results for KD-Restart, stack, and short stack (stack size of 3) traversals. Descriptions of the traversal methods can be found in Section 2.4.

By keeping the framework the same and only varying the tree/traversal combination, we observed that our data structures showed speedups of 45-200% on the CPU and 40-125% on the GPU. These speedups can be directly attributed to the sub-linear traversal of the matrix trees, which results in visiting fewer nodes and triangles during rendering. Additionally, since our data structures simply derives all needed information, we avoid potential memory bottlenecks, which helps improve rendering times. We also noticed that the short stack's decrease in register pressure (compared to a regular stack implementation) often could not make up for the increased code complexity/branching needed by the method. The Thai statue does not have GPU results since the size of the triangle vector was too large to fit into a texture variable. Note that even though octrees have long been thought to be an inferior data structure compared to kd-trees for ray tracing, we show that our octree method is a viable alternative. This is most likely because octrees will be about $1/3$ the height of a comparable kd-tree build, which makes for more efficient traversals. Octrees outperforming regular kd-trees may not always be the case, as we have experienced better results with regular kd-trees in the context of point location in very large unstructured meshes of CFD simulations. Also of note, we experimented with our method on lesser hardware (*e.g.* Intel Xeon X5460 and GeForce GTX 280) and we achieved inferior results than those reported here, but still achieved dramatic improvement over SAH-based trees. Having significantly improved results on newer hardware suggests that our method will be even more beneficial on future architectures. This is most likely because the lag between computation and memory throughput is increasing and as previously stated, our method minimizes global memory accesses.

We show two graphs in Figure 5. The top graph compares the computation time during the build to the time it takes for memory transfer, allocation/deallocation, and binding to GPU textures. We see that the total time for our build algorithm is dominated by handling memory, but becomes less so as the scene complexity increases. We note that the time for memory allocation could be lessened by reusing previously allocated memory, using asynchronous memory transfers, or pipelining these operations through another CPU thread dedicated to memory management. The bottom graph shows that the total time from tree construction (computation + memory) to image generation is dominated by rendering, meaning that the bottleneck has shifted from building to rendering [30]. Even though the render times are dependent on the output image resolution (800×800 for our results) and the complexity of the ray tracer, we note that including additional rendering features (*e.g.,* reflections and anti-aliasing) needed in many applications will only further increase rendering times. Even the use of optimizations (*e.g.,* ray packets) will not introduce a significant enough decrease in rendering time as the performance of these optimizations is very

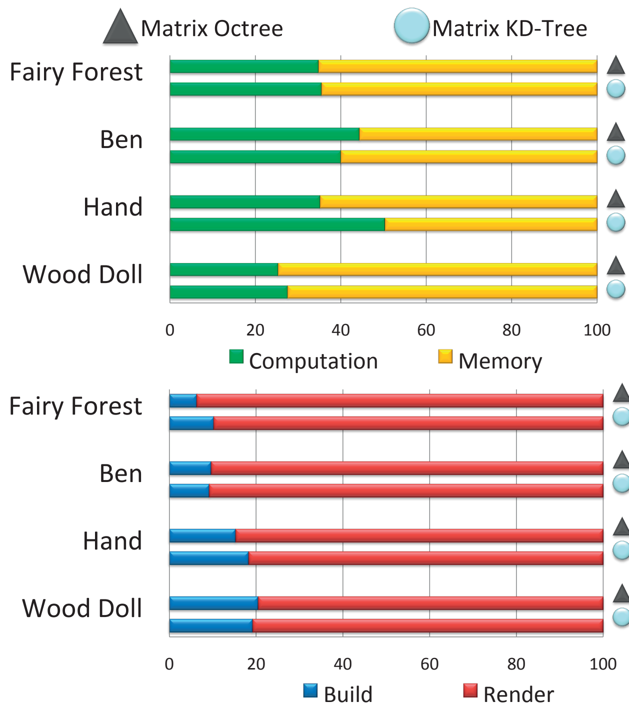

Figure 5: Top graph) Percentage of time spent in computation and memory transfer/allocation/etc. Bottom) Percentage of time it takes to build and render each scene.

Table 1: Build and render times for the scenes (in seconds). The number of triangles and time steps (TS) is included with each data set.

Dataset	Traversal Type		CPU	GPU	Build
Wood Doll 6.6k Tri. 29 TS	Matrix	Octree	0.391	0.023	0.002
		KD-Tree	0.470	0.023	0.002
	SAH	Restart	0.710	0.028	-
		Stack	0.677	0.033	-
		Short Stack	0.735	0.038	-
Hand 17k Tri. 44 TS	Matrix	Octree	0.715	0.036	0.004
		KD-Tree	0.815	0.037	0.002
	SAH	Restart	1.907	0.053	-
		Stack	1.555	0.072	-
		Short Stack	1.831	0.095	-
Ben 78k Tri. 30 TS	Matrix	Octree	0.742	0.079	0.003
		KD-Tree	0.898	0.084	0.004
	SAH	Restart	2.192	0.126	-
		Stack	1.647	0.177	-
		Short Stack	2.099	0.169	-
Fairy Forest 174k Tri. 21 TS	Matrix	Octree	2.072	0.267	0.011
		KD-Tree	2.686	0.355	0.008
	SAH	Restart	6.429	0.423	-
		Stack	5.900	0.447	-
		Short Stack	6.893	0.719	-
Thai Statue 1,000k Tri. 1 TS	Matrix	Octree	1.346	-	-
		KD-Tree	1.714	-	-
	SAH	Restart	6.370	-	-
		Stack	4.091	-	-
		Short Stack	5.564	-	-

dependent on scene complexity and view settings[22].

7 Conclusions and Future Work

We have presented a generic implicit and dynamic tree data structure that outperforms the current state-of-the-art in the context of ray tracing. Trees built with our method not only allow for faster rendering times than their SAH counterparts, they can also be built more efficiently. The data sets shown in the paper are all spatially compact, but we also believe that our method can extend to large, non-uniformly spaced scenes due to the adaptability of the underlying sparse matrix data structures. Additionally, the work presented in this paper has shown improvements in ray casting and allows for an even broader class of trees to be represented implicitly.

The data structure presented in this paper could be directly applied to improve upon other graphics techniques. In particular, the work by Havran and Bittner [11] requires additional information at nodes in order to enable their proposed traversal scheme. We note that all of this stored information can simply be derived by our method. Their method can also benefit from our efficient construction and sub-linear traversal algorithms. Similarly, but in the surface reconstruction context, Zhou *et al.* [32] require recording a *very* large amount of information at each node (236 bytes), which again, can simply be derived by our method.

Since the geometry between key frames typically does not vary much, we would like to update our trees instead of rebuilding them every frame. Updating the tree shape should be relatively straightforward due to the block-based structure. However, maintaining the vector of triangle pointers is not nearly as simple and presents an interesting avenue for future work.

An acknowledged limitation of our work is the memory consumption of a regular matrix tree build compared to SAH based trees due to SAH trees being much sparser. Though the memory required to represent the nodes within the regular tree is minimal (thanks to the sparse encoding), the increased number of nodes in a regular build requires an increased number of duplicated triangle references.

We have implemented a hybrid matrix/SAH build that uses the matrix build at nodes near the root and SAH near the leaves. This hybrid tree allows for the efficient matrix tree traversal at the upper levels while maintaining efficient triangle divisions at the lower levels, which would hopefully allow for the tree to be even more adaptable than either the matrix or SAH trees individually. Though initial tests have proven this theory correct, the improvement in rendering times have been minimal.

Another area of future investigation include balancing work loads on the CPU and multiple GPUs to fully utilize all available computational resources (OpenCL should facilitate this process). Outside of the graphics scope, we believe that matrix trees can be used to efficiently represent non-spatial trees and possibly even graphs.

Acknowledgements

The authors would like to thank Aaron Knoll, the members of the Purdue University Computer Graphics and Visualization Lab, and the reviewers for their discussion and feedback. Data sets were obtained from the Utah 3D Animation Repository and the Stanford 3D Scanning Repository. This work is supported in part by the National Science Foundation award CMMI-1030326 and a gift by Intel Corporation.

References

[1] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the GPU. *ACM Trans. Graph.*, 28:154:1–154:9, December 2009.

[2] N. Andrysco and X. Tricoche. Matrix trees. *Computer Graphics Forum*, 29:963–972(10), June 2010.

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for*

the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition. SIAM, Philadelphia, PA, 1994.

[4] B. Budge, D. Coming, D. Norpchen, and K. Joy. Accelerated building and ray tracing of restricted BSP trees. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 167 –174, August 2008.

[5] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart. Parallel SAH k-D tree construction. In *High Performance Graphics*, pages 77–86. EG, 2010.

[6] T. Foley and J. Sugerman. KD-tree acceleration structures for a GPU raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM.

[7] A. S. Glassner. *Space subdivision for fast ray tracing*, pages 160–167. Computer Science Press, Inc., New York, NY, USA, 1988.

[8] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987.

[9] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, Aug. 2007.

[10] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Electrical Engineering, Czech Technical University, Prague, 2000.

[11] V. Havran and J. Bittner. Stackless ray traversal for kD-trees with sparse boxes. *Computer Graphics & Geometry*, 9(3):16–30, December 2007.

[12] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-D tree GPU raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM.

[13] D. M. Hughes and I. S. Lim. Kd-Jump: a path-preserving stackless traversal for faster isosurface raytracing on GPUs. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1555–1562, 2009.

[14] W. Hunt, W. Mark, and G. Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 81 –88, sept. 2006.

[15] P. Ivson, L. Duarte, and W. Celes. GPU-accelerated uniform grid construction for ray tracing dynamic scenes. Master's thesis, Departamento de Informatica, Pontificia Universidade Catolica, Rio de Janeiro, 2009.

[16] J. Kalojanov and P. Slusallek. A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 23–28, New York, NY, USA, 2009. ACM.

[17] R. P. Kammaje and B. Mora. A study of restricted BSP trees for ray tracing. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 55–62, Washington, DC, USA, 2007. IEEE Computer Society.

[18] K. S. Klimaszewski and T. W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Comput. Graph. Appl.*, 17:42–51, January 1997.

[19] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Comput. Graph. Forum*, 28(2):375–384, 2009.

[20] S. Lefebvre and H. Hoppe. Perfect spatial hashing. *ACM Trans. Graph.*, 25:579–588, July 2006.

[21] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994.

[22] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24:1176–1185, July 2005.

[23] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[24] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. *Comput. Graph. Forum*, 26(3):395–404, 2007.

[25] M. Stich, H. Friedrich, and A. Dietrich. Spatial splits in bounding volume hierarchies. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 7–13, New York, NY, USA, 2009. ACM.

[26] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[27] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1):6, 2007.

[28] 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 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69, 2006.

[29] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph.*, 25(3):485–493, 2006.

[30] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the art in ray tracing animated scenes. In D. Schmalstieg and J. Bittner, editors, *STAR Proceedings of Eurographics 2007*, pages 89–116. The Eurographics Association, Sept. 2007.

[31] S.-E. Yoon, S. Curtis, and D. Manocha. Ray tracing dynamic scenes using selective restructuring. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, page 55, New York, NY, USA, 2007. ACM.

[32] K. Zhou, M. Gong, X. Huang, and B. Guo. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics: to appear*, 2010.

[33] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time KD-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008.