Review and Comparative Study of Ray Traversal Algorithms on a Modern GPU Architecture

Artur Lira dos Santos
Voxar Labs - Informatics Center, UFPE
Av. Jornalista Anibal Fernandes, s/n
Cidade Universitária
Brazil 50740-560, Recife, Pernambuco
als3@cin.ufpe.br

Veronica Teichrieb
Voxar Labs - Informatics Center, UFPE
Av. Jornalista Anibal Fernandes, s/n
Cidade Universitária
Brazil 50740-560, Recife, Pernambuco
vt@cin.ufpe.br

Jorge Lindoso
Voxar Labs - Informatics Center, UFPE
Av. Jornalista Anibal Fernandes, s/n
Cidade Universitária
Brazil 50740-560, Recife, Pernambuco
jefl@cin.ufpe.br

ABSTRACT

In this paper we present a chronological review of five distinct data structures commonly found in literature and ray tracing systems: Bounding Volume Hierarchies (BVH), Octrees, Uniform Grids, KD-Trees, and Bounding Interval Hierarchies (BIH). This review is then followed by an extensive comparative study of six different ray traversal algorithms implemented on a modern Kepler CUDA GPU architecture, to point out pros and cons regarding performance and memory consumption of such structures. We show that a GPU KD-Tree ray traversal based on ropes achieved the best performance results. It surpasses the BVH, often used as primary structure on state-of-the-art ray tracers. A carefully well implemented ropes based KD-Tree CUDA traversal can improve performance on a 12-39% approximate range. This suggests that, for critic real time applications, the ropes based KD-Tree traversal is a more adequate option on GPU. However, this structure consumes at least 4x more memory space than BVHs and BIHs. This disadvantage can be a limiting factor on memory limited architectures.

Keywords

Ray tracing - Ray Traversal - Acceleration Structure - KD-Tree - Octree - BVH - BIH - Uniform Grid - CUDA

1 INTRODUCTION

Research on ray tracing has been done for more than four decades [App68]. This technique is commonly applied to solve the problem of visibility [Whi80], by searching for the nearest (thus visible) ray-object intersection of each ray emitted into the scene. An immediate search algorithm can be defined by testing intersections between the rays and all geometric primitives. However, this brute-force approach makes the search complexity to increase linearly with the number of objects present in the scene.

Since each of these ray-primitive intersections involves costly multiple floating point arithmetic operations, this exhaustive search algorithm becomes prohibitive for different application scenarios, such as photo-realistic image synthesis of complex geometric scenes and real time ray tracing. Along time, different data structures have been proposed capable of organize or group geometric primitives, in order to considerably reduce the amount of intersection tests necessary to find the nearest primitive.

Modern Graphics Processing Units (GPUs) are commonly used as a coprocessor for implementing highly parallel algorithms, such as ray tracing. GPU programming has become even more suitable for general purpose problems since the programming model was unified by the NVIDIA Compute Unified Device Architecture [NVI14] (CUDA), which delivers a many-core programmable solution containing parallel stream processors.

In this paper we review ray traversal algorithms for five different data structures: Bounding Volume Hieararchy (BVH); Octree; Uniform Grid; KD-Tree; and Bounding Interval Hierarchy(BIH). This work also exploits the CUDA architecture to implement six highly efficient ray traversal algorithms for these data structures. Finally, our main contribution is a comparative study of all of those traversals, showing their advantages and limitations on a modern GPU architecture. To the best of our knowledge, there isn't yet a complete comparison between all these main data structures for ray tracing on GPU in the literature.

This paper is organized as follows. Section 2 describes the basic concepts related to acceleration structures for ray tracing. Section 3 presents major previous work related to GPU ray traversals. Section 4 details the implementation of the ray traversal algorithms. The comparative analysis is highlighted in Section 5. Finally, Section 6 draws some conclusions and outlines future work.

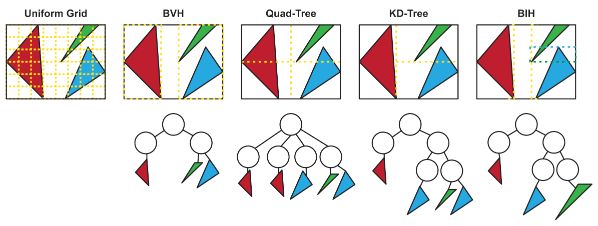


Figure 1: 2D examples of Acceleration Structures.

2 BACKGROUND

This section presents concepts related to data structures that are used throughout this paper.

2.1 Acceleration Structures

When a data structure for ray traversals is used on a ray tracer, only objects near or somehow related to the path of the ray are tested. The increase in performance is significant when compared to the brute-force approach, since for most cases these close-by, related objects comprise a small fraction of the scene. For affine 3D models, these structures can reduce average search complexity to a sub-linear level and consequently improve ray tracing performance by one or more orders of magnitude, being therefore commonly known as Acceleration Structures (AS). The following subsections briefly discuss main concepts of most AS found in literature. Figure 1 shows a simplified 2D representation of all evaluated structures.

BVH

The Bounding Volume Hierarchy (BVH) may be the oldest AS for ray tracing [Rub80], or at least some of its concepts, such as tight bounding volumes for early ray-object intersection termination [Whi80]. This structure partitions groups of objects into disjoint sets, as can be seen in Figure 1. It's usually implemented as a binary tree, on which each internal node stores a tight bounding volume that encloses all primitives it contains, and also pointers for its child nodes. A leaf node stores the bounding volume as well, but also the primitives this volume contains.

Traversing a BVH node is straightforward: the algorithm test intersections between the ray and the node's bounding volume. If it intersects, test the ray against the child nodes and traverse them or, if it's a leaf node, perform ray-primitive intersections. Otherwise, in case the ray doesn't intersect the node's bounds, the algorithm skips traversing that tree's branch.

Kay and Kajiya [Kay86] proposed a top-down BVH traversal algorithm, on which the ray is tested against both bounding volumes from the child nodes. The nearest child node from the ray's origin is visited while the farthest is stored in a priority queue like a heap, for a future traversal. However, the authors didn't show any performance gains than a simple stack. In fact, Shirley and Morley [Shi03] warned the negative impact of a priority queue overhead. Therefore, our BVH traversal implementation is stack-based, with optimizations from the GPU BVH open-source implementation of Aila et al. [Ail09][Ail12]. For details on our BVH implementation, see Section 4.

Octree

An Octree is a spatial 8-ary tree. Therefore, each internal node is split in eight subspaces, represented by its child nodes, which usually have the same volumes. A leaf node stores the primitives its subspace contains. Figure 1 shows an Octrees 2D representative, the Quad-Tree, since an Octree doesn't exist in 2D dimensions.

The Octree was created to solve different problems [Red78][Jac80], such as z-culling and sub-model rotations. To the best of our knowledge, Fujimura et al. [Fuj83] were the first to propose the use of Octree to enhance ray tracing performance. Later, Glassner [Gla84] described his Octree building and traversal algorithms, comparing timing statistics that show speedups above 13x than the brute-force approach. It is interesting to notice the timing units in hours and minutes, for scenes that today run in milliseconds on a modern architecture.

Revelles et al. [Rev00] proposed a more efficient ray traversal algorithm for Octrees, based on how a ray crosses the children of an internal node, and using simpler arithmetic floating point computations. This algorithm will be discussed with more details in Section 4, since we implemented this approach for our comparative study on GPU. This choice relied on the fact that this Octree traversal algorithm is the fastest found in literature. Furthermore, it is important to point out that the Sparse Voxel Octree [Lai10], although its novelty and efficiency on GPUs, only supports voxels as primitives, and therefore isn't part of our comparative study.

Uniform Grid

An Uniform Grid is conceptually simple: the structure splits the whole scene space into equally sized axis-aligned bounding boxes, commonly known as cells, as can be seen in Figure 1. Each cell stores a list of primitives it may contain.

Fujimoto et al. [Fuj85] proposed the first Uniform Grid ray traversal algorithm, known as 3D Digital Differential Analyser (3D-DDA), an extension of Bresenham's algorithm, frequently used for line rasterization. However, they applied the 3D-DDA into traversing an Octree instead. In fact, each Octree subdivision can be interpreted as an Uniform Grid of 2x2x2 cells.

The Bresenham's algorithm defines the largest axis as driving axis, defining the step size for the other (passive) axis. The main difference is that the ray traversal algorithm needs to visit all the intersected cells, while the Bresenham's can miss some cells from the passive axis.

A 3D-DDA extension was proposed by Amanatides and Woo [Ama87]. Being a simple and efficient implementation, it's the most popular Uniform Grid traversal found in literature. Our Uniform Grid ray traversal is based on this approach. For implementation details of this algorithm, see Section 4.

KD-Tree

A KD-Tree [Ben75] is a particular type of a BSP tree [Fuc80]. In a KD-Tree, each internal node is split in two by an axis-aligned plane, defining disjoint bounding boxes represented by its child nodes, as can be seen in Figure 1. A leaf node stores a list of primitives its volume contains, similar to an Octree's leaf node. It is one of the most referenced AS in literature, with at least eight different ray-traversal algorithms. Figure 2 shows different traversal schemes of these algorithms.

Kaplan [Kap85] introduced the first known KD-Tree ray traversal algorithm, later referred as Sequential traversal [Hav00]. This traversal executes a cyclic top-down point search within the KD-Tree nodes, until the leaf that contains the current search point is found. After the leaf traversal, the point of interest is changed to be inside the next leaf node and the search algorithm goes back to the root node. Therefore, many internal nodes are repeatedly visited.

Jansen [Jan86] proposed a recursive traversal algorithm for KD-Tree. His method doesn't visit a node more than once, since the recursive calls handle the ordering of nodes to be visited. Aiming efficiency, Havran [Hav00] presented an iterative version, named later as KD-Standard traversal [Fol05]. The KD-Standard traversal uses a stack in order to store the child node located most far away. Each stacked node is visited later, when all other near child nodes have been visited. Therefore, the stack guarantees that the traversal occurs at the same order than the recursive approach.

MacDonald and Booth [Mac90] proposed the concept of neighbor-links between a leaf and its neighbors nodes for Octrees and BSP-Trees. A traversal algorithm can then use these links to directly access adjacent nodes, reducing the number of visited internal nodes. This technique follows the same concept of "ropes" [Hun79] in a Quad-Tree (see the dashed lines in Figure 2). In the strict sense, this structure isn't a tree anymore, since it has cyclic connections, a consequence of more than one possible path from some nodes to other. Havran et al. [Hav98] described how to build and traverse a KD-Tree with ropes. It's worth to mention the stackless nature of a traversal with ropes, since the neighbor-links of the leaves are enough to find the next tree branch to be traversed. On a GPU, a stackless algorithm reduces high latency memory bandwidth usage [Pop07]. Therefore, a ropes-based traversal algorithm is an important reference for modern GPU KD-Tree implementations, including our previous work [San12].

Havran et al. [Hav97] modified Jansen's traversal algorithm through statistical analysis of ray-node intersection cases, lowering traversal cost for more probable traversal situations. Moreover, this traversal leads to less numerical errors, and consequently less visual artifacts. Compared to KD-Standard traversal, their implementation on a Pentium architecture achieved up to 2x of speedup.

BIH

The Bounding Interval Hierarchy (BIH) [Wac06] is a binary tree that, similar to BVH, partitions a group of objects into disjoint subgroups, as shown in Figure 1. However, the BIH uses two axis-aligned planes instead of complete bounding boxes. These splitting planes are aligned to the same axis, being then represented by just two floating point values. One represents the end limit of the left node, while the other the start limit of the right node. Its main advantages are high efficiency of construction and very low memory usage compared to most of the others structures.

The BIH is a relatively recent data structure, with few related publications on GPU, with some positive results in related areas such the work of Kinkelin [Kin09] in Volume Raycasting, focussed on voxels from 3D image textures. To the best of our knowledge, our work is the first in literature to compare GPU ray tracing performance between BIH's ray traversal and other algorithms.

3 RELATED WORK

A CPU packet ray traversal algorithm [Bou07] groups the search computation of such rays in vectorized Single Instruction Multiple Data (SIMD) instructions. On the other hand, a GPU Single Instruction Multiple Threads (SIMT) architecture [NVI14] directly parallelizes a serial algorithm without the requirement of manual code adaptations into a vectorized one.

Several GPU ray traversal implementations appear in prior work. Purcell et al. [Pur02] showed how to implement a GPU ray tracing pipeline over a stream programming model. They used an Uniform Grid as AS, extending their ray tracer on stream programming.

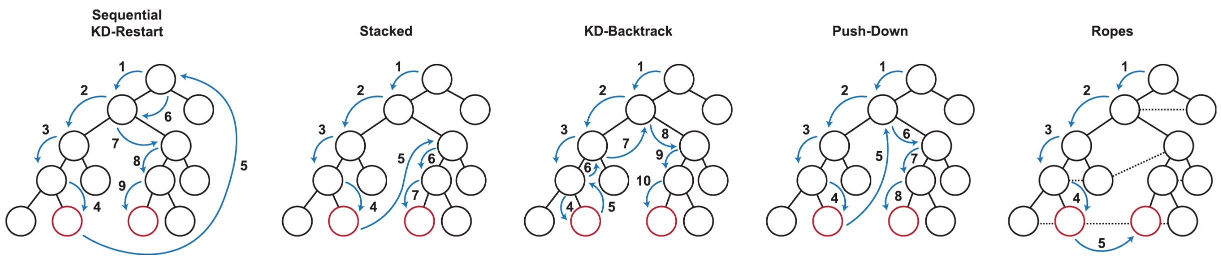


Figure 2: Traversal order example for different KD-Tree Traversal algorithms. The red circles represent two leaf nodes that, must be traversed in order. The main difference between KD-Tree traversals is how they go from the red circle to the other. The dashed lines represent the ropes.

Foley and Sugerman [Fol05] proposed new KD-Tree traversal algorithms in GPU as competitive replacements to the Uniform Grid traversal. They altered the KD-Standard traversal algorithm (see Section 2.1), creating two new ones, KD-Restart and KD-Backtrack. These techniques don't require a stack for the traversal, since the available graphics cards had very limited programmable stack memory at that time.

The KD-Restart algorithm starts over (restart) the traversal to the tree root every time it reaches a leaf node, in a similar manner as the Sequential traversal algorithm. This operation repeats until the ray leaves the scene volume or some primitive intersection is found. The cost of this search becomes higher than of the KD-Standard algorithm, since multiple internal nodes are revisited.

The KD-Backtrack algorithm uses pointers to parent nodes and their axis-aligned bounding boxes (AABB) data in order to perform a bottom-up backtrack, returning to the last visited node that has another child to be visited, not requiring a stack. However, KD-Backtrack may need about an order of magnitude of primary memory than the KD-Standard approach, since it has to store six extra floating point AABB data plus one parent pointer, per node [San12].

In order to reduce the number of revisited nodes, some modifications of the KD-Restart were proposed by Horn et al. [Hor07]. Their Push-Down algorithm changes the root search node to the last one where the ray hits only one of its children. Since the other child will never be visited, this internal node can safely become the new root search node. Then, when a restart event is triggered, the search goes back to this node instead of the root node of the tree.

Horn et al. [Hor07] also described the Short-Stack algorithm, a hybridization of the KD-Standard and KD-Restart traversals. Instead of a large, tree-depth sized stack, the Short-Stack uses a small circular array representing a short stack, with a length that considers the resources limits of the hardware architecture. On Short-Stack traversal, if the stack is non-empty, the next node to be visited is popped from the stack, similar to the KD-Standard traversal, and thus avoiding re-visitation of some nodes. Otherwise, a restart event is processed in the same way as KD-Restart. Finally, to the best of our knowledge, Horn's GPU ray tracing implementation is the first one that achieved interactive rates.

Popov et al. [Pop07] proposed a CUDA KD-Tree packet ray traversal based on the ropes stackless algorithm (see Section 2.1), showing interactive framerates. Concomitantly, Günther et al. [Gün07] presented a CUDA BVH packet ray tracer. They concluded that the BVH ray traversal achieved similar or better results than the KD-Tree approach. We show in our work that in most scenes this is not the case anymore.

In order to enhance the efficiency of ray traversals over the CUDA architecture, Aila et al. [Ail09][Ail12] proposed the use of persistent threads controlling a kernel to greatly reduce the amount of idle threads in a CUDA block. Our traversal kernels are entirely based on persistent threads, since they provide considerable increase of performance and can be used with any AS.

In our previous study [San09][San12], we compared the performance of eight KD-Tree ray traversal algorithms, all of them implemented in CUDA. We showed that the KD-Standard traversal algorithm can be efficiently implemented on GPU, even with a large stack, surpassing in performance other simplified algorithms, such as KD-Restart and Short-Stack. It seems that these adaptations made for old GPUs can do more harm than good on a modern hardware architecture. Moreover, we showed that, although the high performance of KD-Standard traversal, the Ropes algorithm, due to some optimizations, performed better for all scenes. We noticed that the amount of slow memory accesses of bounding boxes data could be reduced by half, fetching only three floating point values necessary to compute the exit parametric value of the ray-box intersection, differently from the original ray-box algorithm, that has to load the entire six floating point box data to compute the exit point. Therefore, our comparative study includes these two best KD-Tree ray traversal algorithms.

Zlatuška and Havran [Zla10] compared the performance of three AS implemented on GPU: Uniform Grid, BVH and KD-Tree. They showed that, for primary rays, a BVH traversal is generally faster than a stacked KD-Tree traversal. Their work is the closest one in literature to ours, showing similarities, but also divergent results to ours. Since it's an important reference, we discuss more aspects of their work in Section 4.

Laine [Lai10b] and Hapala et al. [Hap11] separately proposed different algorithms that allowed stackless traversals on BVHs. Like in our previous work on KD-Trees [San12], both papers show that the cost of the revisited nodes overweights the benefits of these algorithms on current GPU architectures. Laine suggests, however, that maybe in other and future architectures this approach could be beneficial by avoiding problems such as cache trashing.

4 ALGORITHM IMPLEMENTATIONS

Our ray tracing system (named as Real Time Ray Tracer or simply RT^2) builds the AS on CPU, while the ray tracing process is done entirely on GPU, through a CUDA [NVI14] programming model. RT^2 supports primary and secondary rays, following Whitted's [Whi80] simple model, and therefore supports visibility tests, reflections, refractions and hard shadows. We tested both recursive and iterative versions of Whitted's method. The RT^2 uses the latter, since it led to higher performance on CUDA. Moreover, RT^2 optionally supports diffuse soft shadow rays and a plethora of geometric primitive types, such as triangles, cylinders, spheres and parametric surfaces.

4.1 Ray Tracer CUDA Kernel

All of our ray tracing stages are integrated into a single CUDA kernel, since in our experiments, a multi-kernel implementation led to poor performance due to additional costs of memory writes and reads from a kernel to another, at least for real time ray tracing without a large amount of incoherent rays. Moreover, our kernel uses the concept of persistent threads [Ail09] to avoid idle threads inside a block. Instead of defining thousands of blocks to be processed, we create only the necessary to fully occupy the stream multiprocessors of the GPU. These persistent blocks have therefore fixed groups of warps that, through atomic add operations, dynamically schedule rays to be processed. Later, Aila et al. [Ail12] reported that the persistent threads technique wasn't more beneficial than default CUDA scheduler on Fermi, since this architecture has better hardware work distribution. However, in our tests, even on Fermi cards the persistent threads approach still provided up to 11% of performance improvement. The best configuration found was the smallest possible warp batch: 32 tasks per atomic operation.

The primary rays are grouped in tiles of 16x32 pixels. To improve access locality, these tiles are ordered using a z-order Morton code on screen space. When a CUDA warp schedules a batch of 32 rays, the first job is to identify which tile, sub-tile and pixel each task belongs to. Then, the camera's frustum and pixel's coordinates are used to define the origin and direction of each primary ray. After ray initializations, the kernel enters the iterative Whitted-style ray tracing loop. The first step on this loop is the ray traversal of the scene. The intersection result is then stored and the ray's origin is shifted to the point of intersection. They'll be used on the next stage, shading. Our shader supports hard and soft shadows, texture fetches using ray differentials for sample anti-aliasing, Phong shading, and normal maps. The shader then compute de final color contribution of the current ray, adding it to the global pixel color vector.

At the end of shading step, the ray direction is changed based on possible reflections and refractions caused by the surface's material properties. If there's no reflection or refraction, or the loop reached a maximum depth, the ray tracing loop is interrupted. Finally, the global pixel color vector is written on a CUDA surface attached to an OpenGL 2D texture used to exhibit the image result.

4.2 Ray Traversal Implementations on GPU

The following subsections briefly discuss how the algorithms used in this comparative analysis have been implemented.

BVH

Our BVH traversal starts with a standard intersection between the ray and the scene's bounding box. If there's no intersection, there's no traversal and we return a null intersection. Otherwise, the minimum and maximum parametric values of the bounding box intersection are stored. They represent the entry and exit points of the root node traversal. Then, the algorithm creates an iterative loop that does the job of a recursive traversal. As already mentioned, recursive calls are being avoided in due to significant overhead cost than a straight iterative stacked version, especially on current GPU architectures. Traversing a BVH internal node consists in intersection tests between the ray and the child nodes. A child node is visited only if the ray intersects its bounds and the nearest entry point of this child is near than the closest ray-primitive intersection point found at the time. This avoids visiting nodes that can't possibly have closer ray-primitive intersections. When both child nodes are intersected, the one with the farthest entry point is pushed on the stack and the other is visited.

This node traversal stage occurs in another inner loop, which stops only when we reach a leaf. In this case, ray primitive intersections are performed. After reaching a leaf node, a node is popped from stack to be visited. If the stack is empty, then the traversal has finished. It's worth mentioning that we implemented two different versions of this iterative BVH traversal algorithm. The first one was using a node that stores its bounding box and a stack that keeps three variables per element: the node index and the two parametric intervals it has. However, we experienced an approximately 20% of performance improvement using the approach described in the open source code of the work of Aila [Ail09], and therefore our BVH traversal implementation for this comparative study is based on his method.

Aila stores the children bounding boxes on the parent node. In our tests, this memory layout was more efficient, since when internal node is visited, the children bounds always have to be loaded from memory. Then, putting these data in a contiguous memory address has benefits for the L1 and L2 cache usage. Moreover, Aila's stack stores only the node index, computing or having on the fly the minimum and maximum intervals of the current node. This is also important, since the stack is stored on cached, but slow Local Memory CUDA space. We also used the optimized ray bounding box intersection with intrinsic CUDA PTX instructions vmin and vmax proposed by Aila et al. [Ail12].

Octree

Our Octree ray traversal implementation is based on the work of Revelles et al. [Rev00]. They proposed a top-down traversal algorithm based on index computation of the child nodes to define the traversal order. However, the original algorithm is recursive, with 8-branches factor. We implemented it using template based pseudo-recursion. We used his optimization details, which consists in computing the parametric values of the ray intersection against the three subdivision planes using only floating point add and multiplication by half operations, since the Octree nodes are always subdivided in half dimensions. Moreover, our implementation also take advantage of computations made in previous nodes, avoiding redundant operations.

Uniform Grid

As mentioned in Section 2.1, our Uniform Grid traversal implementation is based on the 3D-DDA extension proposed by Amanatides and Woo [Ama87]. They don't define a special driving axis like the original differential analyser, meaning that all axes can be now considered as a driving axis.

The algorithm first tests if the ray intersects the bounding box of the scene, for early termination. In case it intersects, we use the entry point to define the first intersected voxel/cell. Some constant 3D values, such as interval variations for each axis, are precomputed. Then, a main loop is defined that incrementally visit one cell at the time, deciding which axis will be used to shift to a neighbor cell. This incremental loop stops only if an intersection is found or the current visited cell is the one that contains the exit point of the ray intersection with the scene's bounding box. It's important to inform that we tested access of axis data using indirect runtime access (like point3D[axis]) versus a direct branched if-else structure. The indirect access led to lower performance, since this indirection forces the variable to go to a memory space where indirect access are possible, that in that case was from registers to slow local memory space.

The traversal of this structure led to high register pressure, consuming almost a dozen more registers than other structures, due to storage of the precomputed 3D values. To reduce this problem, we followed the suggestion of Zlatuška and Havran [Zla10] to store some of these values on constant memory space. Indeed, we experienced a reduction of three registers using this approach, not enough to improve occupancy, but it showed slightly faster results.

KD-Tree

For this comparative analysis, we implemented two KD-Tree traversal algorithms: the KD-Standard and Ropes, mentioned in Section 2.1. Although their differences, both use the same compact node size (8 bytes) and starts with a ray intersection against the scene's bounding box, for early termination, or to define the entry and exit point in case of intersection.

The KD-Standard algorithm uses a stack to store the farthest node when both child nodes are intersected. Our version of this traversal is implemented with a main loop of the traversal that contains another repetition block that handles the internal node traversal, in a very similar manner than the BVH inner loop traversal. However, while the BVH traversal tests against two bounding boxes, the KD-Standard traversal only tests against a splitting plane. This plane is represented by an axis and a single floating point value. The inner loop only exits when a leaf node is found. Then, on a leaf, the algorithm tests the ray against the primitives. The traversal is finished if one intersection is found, or the stack is empty. Our stack stores the node index and the maximum parametric interval of the node, but not the minimum, since it can be retrieved on the fly from the maximum of the last visited node. As it was separately pointed out by Zlatuška and Havran [Zla10], and our previous work [San09][San12], this layout enhances performance, since the stack has to be implemented on slow local memory.

The Ropes algorithm also has two loops, on which the innermost one traverses internal nodes, through a simple node search using an entry point to decide to go to the left or to the right node. Since a point can't be inside both child nodes, it doesn't have to store the farthest nodes. Later in the traversal, the ropes links will provide a more direct access to the farthest nodes. This is an efficient GPU inner loop traversal, since the inner loop just checks if the entry point floating point value of the splitting axis is less or equals to the splitting plane to decide if go left or right. Furthermore, the entry point is obtained before the inner loop, by computing only three GPU fast fused-multiply-add (fmadd) operations. If the node is a leaf, the inner loop is skipped and then occurs the ray-primitives intersections. Before going back to the inner loop, the algorithm retrieves the next node to be visited by using the index of the face the ray exits the leaf node. Thus, is necessary a ray-box intersection to define the exit face. On our previous work [San12], we proposed an efficient method of intersection that reads only three floating points instead of six of the bounding box data, and yet can obtain the complete exit point information. This way, half of global memory loads of bounding boxes were avoided, allowing greater performance than older ropes based algorithms.

It's possible to use the ropes to start a ray traversal directly in a leaf instead at the root node [Hav00]. We use this concept for rays that have their origin inside the scene, such as reflected, refracted and shadow rays. Moreover, primary rays originated from cameras inside the scene can also be traversed slightly faster. Before executing the ray tracer kernel, we do a fast single point search location for the leaf that contains the camera's position. Then, the ray tracing kernel uses this leaf node to start the traversals of primary rays. In our tests, we achieved up to 6% of performance improvement by starting traversals at a leaf node.

BIH

Our BIH ray traversal implementation follows the work of Wächter and Keller [Wac06]. It's a hybrid traversal that mixes concepts of the KD-Standard and BVH traversal algorithms. The inner loop for traversing internal nodes is very similar to the one on the KD-Standard traversal, using a stack to push farther child nodes. However, like the BVH traversal, it's possible that a ray misses both child nodes. In this case, a stacked node is popped to be visited, or the traversal finishes if the stack is empty. Ray-primitive intersections occur in leaf nodes. Differently from KD-Tree traversal, it's not possible to finish the traversal as soon as a leaf with an intersected primitive is found, due to possible overlapping cases between child nodes. However, like the BVH traversal, nodes that start after the nearest primitive intersection found are not visited.

5 RESULTS

In this section we present the quantitative results of our comparative analysis. In our tests we used distinct scenes, most of them available on public repositories: a Land Rover car (77k triangles); Utah Fairy Forest[Uta14] (174k triangles) , Dubrovnik's Crytek Sponza (262k triangles); and Guillermo's San Miguel Model (7.8 millions of triangles). We chose the Land Rover and Fairy Forest scenes for their low polygon count and/or uniform geometry distribution. On the other hand, the Sponza was selected for its non-uniformity. The San Miguel was chosen for its high poly count. The results were collected using an Intel Core i7 3.06 GHz CPU with 8GB of RAM, running Microsoft Windows 8 Professional 64-bit, and Visual Studio 2012 with CUDA 6 Toolkit. We used the fastest single CUDA GPU available at the time: a Zotac GeForce GTX 780 Ti Amp!, a Kepler device with Capability 3.5. We activated the L1 cache for Global memory accesses, since on Kepler L1 disabled for Global memory is the default state.

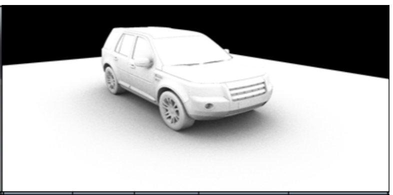
We contrast some important information, such as execution time for Whitted-style ray tracing vs random ambient occlusion rays, timing in ms and Mrays/s, number of average traversal steps and intersections per ray. We also show memory consumption and CPU build time for all acceleration structures.

Our BVH and KD-Tree SAH builders feature build time optimizations based on [Wal06] work, which reduced build time by 3x-4x. Our BVH with spatial splits (SBVH) is based on [Sti09] work.

As shown in Figure 3, the KD-Tree Ropes and BVH (SBVH) traversal algorithms achieved the best performance results for all tested scenes. The reason for such performance enhancement is directly related to a lower number of traversal steps and/or intersections per ray. Regarding rendering times, Ropes has as advantage its ray-AABB intersection optimizations and lower traversal cost for secondary rays, as discussed in Section 4.2. These optimizations combined significantly reduce the amount of global memory accesses when the L1 and/or L2 cache miss. Unfortunately, the Ropes approach demands more memory when compared to other traversal algorithms. For instance, on the Fairy Forest scene, a Ropes based KD-Tree uses 4.76x and 8.71x more memory than KD-Standard tree and BVH (SBVH), respectively. However, it's worth mentioning that even on current low end devices with 1 GB of memory is possible to ray trace with Ropes on most practical scenes, and even on large ones, with two millions of primitives.

The BVH with Spatial (SBVH) splits on construction achieves closer performance to the KD-Tree Ropes traversal, since it virtually splits triangles to offer higher performance, and consequently might have high memory cost, like the ropes. It's worth mentioning that, in our tests, on Fermi architecture, the BVH SBVH slightly surpasses KD-Tree Ropes performance. This is not the case on Kepler devices, on which the Ropes algorithm outperformed any other approach.

Land Rover
Triangle Count: 77,2k
Lights (Hard Shadows): 4
Max Ray Bounces: 10
AO Samples : 16 / Primary Ray
Resolution: 1920x1080
Walkthrough Animation: 100 frames

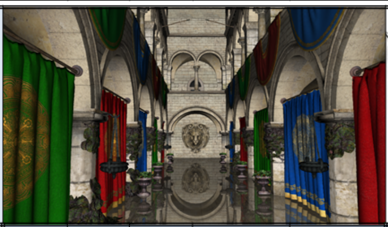


	Size [MB]	Leaves [x10^3]	SAH Cost [Trav./Int.]	Max. Depth	Build Time [CPU ms]	Render Time [ms]	MRays/s.	Perf. Reference	Steps/Ray Avg.(Max.)	Inter./Ray Avg.(Max.)	AO Render Time [ms]	MRays/s.	Perf. Reference	Steps/Ray Avg.(Max.)	Inter./Ray Avg.(Max.)
1º KD Ropes (SAH)	17,0	108	1,79/1	46	590	13,0	698,9	100%	11.20(147)	5.03(180)	52,8	476,4	100%	6.61(147)	3.61(242)
2º BVH (SBVH)	1,9	22	2,90/1	21	2171	16,8	540,2	77%	16.21(134)	6.08(123)	64,2	392,3	82%	13.40(149)	4.89(182)
3º BVH (SAH)	1,6	18	2,90/1	22	127	17,4	521,3	75%	16.99(129)	6.25(141)	69,2	364,0	76%	13.93(139)	4.98(186)
4º KD Standard (SAH)	2,9	106	1,81/1	46	532	17,5	516,8	74%	26.18(217)	5.06(180)	85,1	295,9	62%	20.58(216)	3.63(229)
5º KD Ropes (Mid. Cut)	8,4	62	1,79/1	62	144	16,3	556,2	80%	12.86(121)	12.26(292)	87,9	286,5	60%	7.51(121)	8.51(435)
6º BVH (Mid. Cut)	1,8	21	2,90/1	29	23	20,99	431,1	62%	23.74(161)	5.08(140)	92,1	273,2	57%	18.74(192)	3.94(183)
7º KD Standard (Mid. Cut)	2,3	61	1,81/1	62	120	22,5	403,0	58%	37.07(216)	12.32(292)	133,2	188,9	40%	27.83(250)	8.55(400)
8º BIH	0,7	13	-	34	16	34,6	261,9	37%	39.95(304)	23.96(487)	258,2	97,5	20%	30.11(333)	17.74(491)
9º Uniform Grid	1,5	Dimensions: 64x24x128			6	41,2	219,5	31%	37.46(131)	62.91(1501)	360,9	69,7	15%	27.54(131)	46.45(3319)
10º Octree	3,2	35	25,00/1	12	330	52,74	171,6	25%	17.06(120)	44.62(949)	502,1	50,1	11%	14.81(259)	36.10(2028)

Fairy Forest
Triangle Count: 174k
Lights (Hard Shadows): 4
Max Ray Bounces: 10
AO Samples : 16 / Primary Ray
Resolution: 1920x1080
Walkthrough Animation: 100 frames

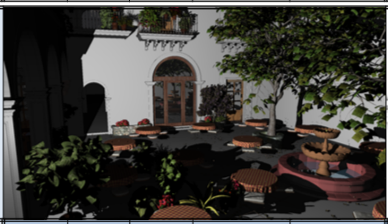
	Size [MB]	Leaves [x10^3]	SAH Cost [Trav./Int.]	Max. Depth	Build Time [CPU ms]	Render Time [ms]	MRays/s.	Perf. Reference	Steps/Ray Avg.(Max.)	Inter./Ray Avg.(Max.)	Render Time [ms]	MRays/s.	Perf. Reference	Steps/Ray Avg.(Max.)	Inter./Ray Avg.(Max.)
1º KD Ropes (SAH)	34,0	202	1,79/1	49	1435	15,2	545,1	100%	19.49(179)	11.89(380)	86,6	392,4	100%	12.44(209)	10.39(460)
2º BVH (SBVH)	3,9	43	2,90/1	26	3375	19,1	434,6	80%	27.88(223)	10.66(202)	98,5	344,8	88%	22.47(259)	9.50(236)
3º BVH (SAH)	3,4	38	2,90/1	25	344	18,9	437,9	80%	26.71(229)	11.07(200)	100,1	339,6	87%	22.47(273)	9.80(256)
4º BVH (Mid. Cut)	3,8	43	2,90/1	39	54,7	22,03	375,9	69%	33.06(253)	11.75(207)	123,4	275,3	70%	26.84(360)	10.71(291)
5º KD Ropes (Mid. Cut)	20,9	152	1,79/1	73	363	19,0	436,7	80%	17.52(168)	26.04(599)	131,7	258,0	66%	11.67(182)	22.16(708)
6º KD Standard (SAH)	5,9	199	1,81/1	49	1302	20,6	402,1	74%	47.33(354)	12.07(380)	133,7	254,1	65%	36.48(447)	10.57(491)
7º KD Standard (Mid. Cut)	5,6	150	1,81/1	73	299	25,0	331,4	61%	48.34(407)	26.27(599)	188,3	180,4	46%	38.47(426)	22.41(729)
8º BIH	1,4	28	-	45	43	36,5	226,9	42%	52.70(400)	37.22(579)	327,0	103,9	26%	41.23(537)	29.93(921)
9º Octree	6,9	75	25,00/1	12	841	52,53	157,6	29%	19.67(165)	68.58(1021)	526,9	64,5	16%	17.09(256)	59.26(1825)
10º Uniform Grid	32,4	Dimensions: 256x64x256			54	104,9	79,0	14%	118.42(240)	93.93(15844)	1207,7	28,1	7%	57.78(211)	83.11(16636)

Crytek Sponza
Triangle Count: 262k
Lights (Hard Shadows): 1
Max Ray Bounces: 10
AO Samples : 16 / Primary Ray
Resolution: 1920x1080
Walkthrough Data: 100 frames



	Size [MB]	Leaves [x10^3]	SAH Cost [Trav./Int.]	Max. Depth	Build Time [CPU ms]	Render Time [ms]	MRays/s.	Perf. Reference	Steps/Ray Avg.(Max.)	Inter./Ray Avg.(Max.)	Render Time [ms]	MRays/s.	Perf. Reference	Steps/Ray Avg.(Max.)	Inter./Ray Avg.(Max.)
1º KD Ropes (SAH)	55,3	349	1,79/1	46	2044	14,0	274,1	100%	26.72(128)	10.27(153)	37,7	900,8	100%	8.32(109)	7.52(154)
2º KD Ropes (Mid. Cut)	28,1	207	1,79/1	66	529	14,5	264,2	96%	22.14(119)	24.64(283)	62,0	548,5	61%	6.83(93)	16.27(283)
3º BVH (SBVH)	68,5	77	2,90/1	25	8449,4	15,8	243,6	89%	42.73(147)	11.86(142)	64,64	525,6	58%	25.42(156)	7.77(193)
4º KD Standard (SAH)	9,4	343	1,81/1	46	1854	16,5	232,8	85%	62.33(244)	10.39(155)	65,2	521,4	58%	33.83(209)	7.68(169)
5º KD Standard (Mid. Cut)	8,0	204	1,81/1	66	439	17,5	219,6	80%	69.20(253)	24.76(283)	93,1	364,8	40%	34.26(230)	16.43(288)
6º BVH (SAH)	5,4	60	2,90/1	27	541	19,4	198,2	72%	63.56(188)	21.12(139)	104,7	324,4	36%	38.20(167)	16.22(162)
7º Uniform Grid	56,5	Dimensions: 275x128x256			85	28,4	135,3	49%	97.94(263)	50.22(1648)	130,5	260,3	29%	15.84(257)	34.94(1696)
8º Octree	9,6	104	25,00/1	12	1183	31,08	123,6	45%	25.77(89)	67.66(579)	185,5	183,2	20%	12.45(68)	37.52(404)
9º BVH (Mid. Cut)	5,8	66	2,90/1	37	82,9	51,05	75,3	27%	99.68(276)	193.97(348)	497,4	68,3	8%	73.05(276)	66.13(374)
10º BIH	2,2	43	-	50	75	57,4	66,9	24%	236.68(706)	129.34(538)	586,3	58,0	6%	15.02(706)	66.11(520)

San Miguel
Triangle Count: 7880k
Lights (Hard Shadows): 4
Max Ray Bounces: 10
AO Samples : 16 / Primary Ray
Resolution: 1920x1080
Walkthrough Animation: 100 frames



	Size [MB]	Leaves [x10^3]	SAH Cost [Trav./Int.]	Max. Depth	Build Time [CPU ms]	Render Time [ms]	MRays/s.	Perf. Reference	Steps/Ray Avg.(Max.)	Inter./Ray Avg.(Max.)	Render Time [ms]	MRays/s.	Perf. Reference	Steps/Ray Avg.(Max.)	Inter./Ray Avg.(Max.)
1º KD Ropes (SAH)	1418	8751	1,79/1	66	61927	12,6	318,7	100%	41.70(319)	13.30(330)	56,4	601,5	100%	11.62(319)	9.64(330)
2º BVH (SBVH)	167,7	1847	2,90/1	34	129066,5	15,8	255,0	80%	59.35(600)	13.86(670)	84,42	401,7	67%	34.74(1197)	10.38(1685)
3º KD Standard (SAH)	253,8	8618	1,81/1	66	60527	16,9	238,1	75%	95.53(650)	13.45(330)	95,9	353,7	59%	43.01(650)	9.78(330)
4º KD Ropes (Mid. Cut)	960,3	6897	1,79/1	79	16740	17,2	233,2	73%	38.71(281)	32.12(500)	109,4	309,9	52%	10.17(281)	22.39(497)
5º BVH (SAH)	147,6	1612	2,90/1	457	24983	18,1	222,1	70%	69.41(416)	14.11(225)	111,7	303,7	50%	41.74(416)	9.87(236)
6º KD Standard (Mid. Cut)	253,1	6812	1,81/1	79	13475	22,4	179,4	56%	102.57(649)	32.38(500)	155,6	218,0	36%	44.04(649)	22.55(529)
7º Octree	276,3	3069	25,00/1	12	37632	42,2	95,2	30%	33.18(184)	76.20(1150)	320,9	105,7	18%	16.81(184)	48.12(1478)
8º Uniform Grid	314,4	Dimensions: 384x384x384			1148	55,7	72,1	23%	142(488)	130,5(17193)	605,1	56,0	9%	24,18(488)	100,3(21769)
9º BIH	69,4	1375	-	63	2803	72,2	55,6	17%	334.82(1365)	104.47(926)	706,2	48,0	8%	42.43(1365)	60.59(926)
10º BVH (Mid. Cut)	162,6	1821	2,90/1	53	2791,0	>8000	-	<1%	-	-	>8000	-	<1%	-	-

Figure 3: Measurements for all Acceleration Structures on four different scenes. Color images (on left) show Whitted-style ray tracing, with reflections, refractions, hard shadows, and a shader that includes complete Phong's specular computations, normal mapping, mip mapping, summed-area tables and ray differentials for texture antialiasing. Grayscale images (on right) show ambient occlusion results from shooting random rays at the scene.

It's important to notice that a BVH gains some advantage when the scene has a geometric uniform distribution. This is closely related to the costs of overlapping of child nodes: evenly distributed models offers lower probability of overlap. Furthermore, the BVH suffers with overlapping, creating large, overlapped leaf nodes. In these cases, even the SBVH approach wasn't capable to outperform the Ropes. On the Sponza Scene, the SBVH used more memory than Ropes, and still couldn't reach its performance. Even worse, the build time was 3x slower than the already slow KD-Tree SAH builder. On the other hand, the traditional BVH with SAH construction has low memory usage, but consequently it has considerably lower timings.

The Octree, Uniform Grid, and BIH traversals had less than 50% of MRays/s performance of KD-Tree and BVH. All of them suffer from high number of ray-primitives intersections. However, the Uniform Grid and BIH have the advantage of faster construction, at least on CPU. Moreover, the latter stands out regarding memory usage, around four times less when compared to other structures. Unfortunately, the Octree, in general, seems to have no significant advantage besides its simplicity of construction.

6 CONCLUSION AND FUTURE WORK

This work presented a compilation of different acceleration structures applied to the development of ray tracers, focusing real time rendering. A considerable number of traversal approaches were compared regarding performance and memory consumption of such structures. We show that, for most cases, a GPU KD-Tree ray traversal achieved better performance results, specially the one based on ropes. It surpassed even the BVH, often used as primary structure on state-of-the-art ray tracers, but probably due to it's high performance on construction algorithms. A carefully well implemented ropes based KD-Tree CUDA traversal can improve performance on a 12-39% approximate range. This suggests that, for critic real time applications, the ropes based KD-Tree traversal is a more adequate option on GPU. However, this structure consumes more memory space than others. This can be a limiting factor for rendering highly complex geometric scenes or on memory limited architectures. In this case, structures such as BVH and BIH are more suitable options.

As future work, we intend to extend our comparative study of ray traversals to GPU construction time, to define the best acceleration structures for rendering dynamic scenes.

7 ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their constructive comments, and the CNPq for partially funding this research.

8 REFERENCES

[Ail09] Aila, T., and Laine, S. Understanding the Efficiency of Ray Traversal on GPUs. Conference on HPG 2009, ACM, pp.145–149, 2009.

[Ail12] Aila, T., Laine, S., and Karras, T. Understanding the Efficiency of Ray Traversal on GPUs - Kepler and Fermi Addendum. In NVIDIA Technical Report, June 2012.

[Ama87] Amanatides, J., and Woo, A. A Fast Voxel Traversal Algorithm for Ray Tracing. In Eurographics '87, pp.3-10, August 1987.

[App68] Appel, A. Some techniques for shading machine renderings of solids. In AFIPS '68 (Spring): April 30-May 2, Spring Joint Computer Conference, ACM, pp.37-45, 1968.

[Ben75] Bentley, J.L. Multidimensional Binary Search Trees Used for Associative Searching. Communications of the ACM, vol.18(9), pp.509-517, 1975.

[Bou07] Boulos, S., Edwards, D., Lacewell, J.D., Kniss, J., and Kautz, J., Shirley, P., and Wald, I. Packet-based Whitted and Distribution Ray Tracing. Graphics Interface 2007, ACM, pp.177-184, 2007.

[Fol05] Foley, T., and Sugerman, J. Kd-tree acceleration structures for a GPU Raytracer. ACM SIGGRAPH/Eurographics Conference on Graphics Hardware, ACM Press, pp.15-22, 2005.

[Fuc80] Fuchs, H., Kedem, Z.M., and Naylor, B.F. On Visible Surface Generation by a Priori Tree Structures. 7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '80), ACM, pp.124-133, 1980.

[Fuj83] Fujimura, K., Toriya, H., Yamaguchi, K., and Kunii T.L. An Enhanced Oct-tree Data Structure and Operations for Solid Modeling. NASA Symposium of Computer-Aided Geometry Modeling, pp.279-287, April 1983.

[Fuj85] Fujimoto, A., and Iwata, K. Accelerated ray tracing. Computer Graphics Visual Technology and Art (Computer Graphics '85), pp.41-65, 1985.

[Gla84] Glassner, A. Space Subdivison for Fast Ray Tracing. IEEE Computer Graphics & Applications, Vol.4, pp.15-22, 1984.

[Gün07] Günther, J., Popov, S., Seidel, H.-P., and Slusallek, P. Realtime ray tracing on GPU with BVH-based packet traversal. IEEE/Eurographics Symposium on Interactive Ray Tracing, pp.113–118, 2007.

[Hap11] Hapala, M., Davidovic, T., Wald, I., Havran, V. and Slusallek, P. Efficient Stack-less BVH Traversal Algorithm for Ray Tracing. SCCG 2011 conference, pp.7-12, April 2011.

[Hav97] Havran, V., Kopal, T., Bittner, J., and Žára, J. Fast robust BSP tree traversal algorithm for ray tracing. Journal of Graphics Tools, vol.2(4), pp.15-23, 1997.

[Hav98] Havran, V., Bittner, J., and Žára, J. Ray Tracing with Rope Trees. Conference SCCG 98, pp.130-139, April 1998.

[Hav00] Havran, V. Heuristic Ray Shooting Algorithms. Ph.D. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, November 2000.

[Hor07] Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P. Interactive k-d tree GPU raytracing. Symposium on Interactive 3D graphics and games (I3D '07), ACM Press, pp.167-174, 2007.

[Hun79] Hunter, G.M., and Steiglitz K. Operations on images using quad trees. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.1(2), pp.145-53, 1979.

[Jac80] Jackins, C.L., and Tanimoto, S.L. Oct-trees and their use in representing three-dimensional objects. Computer Graphics and Image Processing, pp.249-270, November 1980.

[Jan86] Jansen, F. Data structures for ray tracing. In Data Structures for Raster Graphics, Springer-Verlag, pp.57-73, 1986.

[Kap85] Kaplan, M. Space-tracing: a constant time ray-tracer. In SIGGRAPH '85: 12th annual conference on Computer graphics and interactive techniques, 1985.

[Kay86] Kay, T.L., and Kajiya, J.T. Ray tracing complex scenes. ACM Siggraph Computer Graphics, 20:269–278, 1986.

[Kin09] Kinkelin, M. GPU Volume Raycasting using Bounding Interval Hierarchies. Research Student Project, The Institute of Computer Graphics and Algorithms Computer Graphics Group, 2009.

[Lai10] Laine, S., and Karras, T. Efficient sparse voxel octrees. ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (I3D '10). ACM, New York, pp.55-63, 2010.

[Lai10b] Laine, S. Restart Trail for Stackless BVH Traversal. High-Performance Graphics, ACM, 2010.

[Mac90] MacDonald, D.J., and Booth, K.S. Heuristics for ray tracing using space subdivision. The Visual Computer: International Journal of Computer Graphics, vol.6(3), pp.153-166, 1990.

[NVI14] NVIDIA Company. Compute Unified Device Architecture Programming Guide. In CUDA Toolkit 6, 2014.

[Pop07] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. Stackless KD-tree traversal for high performance GPU ray tracing. Computer Graphics Forum, Eurographics, vol.26(3), pp.415–424, 2007.

[Pur02] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. Ray tracing on programmable graphics hardware. ACM SIGGRAPH 2002, ACM, pp.703-712, July 2002.

[Red78] Reddy, D.R., and Rubin, S. Representation of three-dimensional objects. Technical Report. Computer Science Department, Carnegie-Mellon University, Pittsburgh, April 1978.

[Rev00] Revelles, J., Ureña, C., and Lastra, M. An Efficient Parametric Algorithm for Octree Traversal. Journal of WSCG, pp.212-219, 2000.

[Rub80] Rubin, S.M., and Whitted, T. A 3-dimensional Representation for Fast Rendering of Complex Scenes. In SIGGRAPH '80 7th annual conference on Computer graphics and interactive techniques, ACM Press, v.14(3), pp.110-116, 1980.

[San09] Santos, A., Teixeira, J.M., Farias, T., Teichrieb, V., and Kelner, J. KD-Tree traversal implementations for ray tracing on massive multiprocessors: A comparative study. International Journal of Parallel Programming, Springer US, vol.40, pp.331-352, 2012.

[San12] Santos, A., Teixeira, J.M., Farias, T., Teichrieb, V., and Kelner, J. Understanding the Efficiency of kD-tree Ray-Traversal Techniques over a GPGPU Architecture. International Journal of Parallel Programming, Springer US, vol.40, pp.331-352, 2012.

[Shi03] Shirley, P., and Morley, R.K. Realistic Ray Tracing, AK Peters, Second Edition, 2003.

[Uta14] The Utah 3D Animation Repository, 2014. http://www.sci.utah.edu/ wald/animrep/

[Sti09] Stich M., Friedrich H. and Dietrich A. Spatial Splits in Bounding Volume Hierarchies. Conference on High Performance Graphics, 2009.

[Wac06] Wächter, C., and Keller, A. Instant ray tracing: The bounding interval hierarchy. 17th Eurographics Conference on Rendering Techniques, Eurographics Assoc., pp.139-149, 2006.

[Wal06] Wald I. and Havran V. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). IEEE Symposium on interactive ray tracing, 2006.

[Whi80] Whitted, T. An Improved Illumination Model for Shaded Display. Communications of ACM, ACM, v.23(6), pp.343-349, June 1980.

[Zla10] Zlatuška, M., and Havran, V. Ray Tracing on a GPU with CUDA - Comparative Study of Three Algorithms. WSCG '2010, pp.69-76, 2010.