Masaryk University
Faculty of Informatics

Building of SBVH on Graphical Hardware

Master's Thesis

Bc. Radek Stibora

Brno, Spring 2016

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Radek Stibora

Advisor: RNDr. Marek Vinkler, Ph.D.

Acknowledgement

I would like to thank my advisor, RNDr. Marek Vinkler, Ph.D, for his valuable insights and advice that eased making of this thesis. I would also like to express my gratitude for the support received by *MUNI/C/1477/2014 (Student Project Grant at MU - Category C - Rector's Programme)* and *MUNI/33/12/2014 (FI Dean's Programme)* projects.

Abstract

This thesis describes the design and implementation of the SBVH build algorithm using the parallel CUDA architecture on the GPU. It also compares this implementation with a serial implementation of the same build algorithm as well as with other acceleration structures.

The measurements performed across eight 3D scenes reveal that the CUDA variant of the SBVH build is substantially faster (on average 213×) than the serial variant while maintaining the same acceleration structure quality. Compared with other GPU acceleration structures, the GPU SBVH offers a better acceleration performance (on average 41% to 57%) for the cost of a longer build process (on average 4.0× to 4.4×).

Keywords

SBVH, BVH, K-d tree, GPU, CUDA, ray tracing

Contents

1 Introduction

To generate an image out of a polygonal 3D scene, one has to choose a tool for the job. Several rendering methods exist, differing in capabilities and computational complexity. If the task is to generate physically accurate images without limitations on materials and lighting, the ray tracing methods is an ideal tool.

Ray tracing in its principle produces result image simply by casting rays from a virtual camera to a 3D scene. Information about ray-scene intersections is used in later stages of rendering for determining the value of image pixels.

The ray tracing method is capable of producing high quality and physically accurate output images, but for such a high level of realism it requires high amount of rays to be traced. In the past, ray tracing was used chiefly for offline rendering, typically in the film industry. A single frame of a movie would take hours to render, even when rendered on a render farm.

With the advent of more powerful hardware, the ray tracing method is increasingly used for interactive image generation. Modern GPUs (Graphic Processing Unit) with their many-core architecture exploit embarrassingly parallel property[1] of the method and allows it to be computed especially fast. Once the ray-tracing method reaches real-time speeds, it may be employed in various new areas, most notably in video games.

One of the bottlenecks of the ray tracing method is the computation of ray-scene intersections. The amount of these intersection calculations can be reduced by utilizing an acceleration structure which partitions the 3D scene to sub-sections or sub-volumes. During the ray-scene intersection calculation this acceleration structure is traversed and whole parts of the scene, not single primitives, are dismissed in a single intersection test. The disadvantage of employing the acceleration structure to speed up the ray tracing method usually lies in the need to construct the structure before any ray can be traced. Additionally, the better the structure, the more complex and time consuming is

1. Computational problem is said to be embarrassingly parallel, if it can be separated into components that can be executed concurrently and such separation requires little or no effort [1, p. 14].

the build process and so aim is to strike balance between the structure build time and its quality.

The main aim of this thesis is to describe an implementation of a build algorithm constructing the SBVH (Split Bounding Volume Hierarchy) acceleration structure. The described algorithm performs build of the SBVH using the GPU and does so in a single function call to achieve high performance.

The thesis is structured as follows: the second chapter summarizes theoretical knowledge surrounding the SBVH and its build process. The third chapter describes the implementation of the SBVH acceleration structure into a ray tracing framework. The fourth chapter contains comparison of the GPU and the CPU variant of the SBVH build method as well as comparison of the GPU SBVH and other acceleration structures. The final chapter concludes the thesis and briefly summarizes the results.

2 Theory

2.1 Ray tracing method

The ray tracing method, as its name suggests, generates an output image by tracing rays of virtual light. These rays are traced from a virtual camera[1] through the each pixel (more than one ray can be traced through a single pixel) of the output image and into a 3D scene. When a ray intersects the geometry of the scene, the value of the image pixel the ray passed trough is calculated based on the material properties of the scene in the place of the intersection. Additional rays may be traced from the intersection point to simulate light propagation. The value of the pixel is then calculated using the information from all the rays originating in the given pixel.

The ray tracing method is capable of simulating real world effects such as light reflection, refraction and scattering and it is therefore used when a high physical accuracy of the output image is demanded. On the other hand, simpler image synthesis methods, such as scanline rendering, tend to be faster.

An important (especially in recent years) property of the ray tracing method is its very easy parallelization. Indeed, each thread can trace one or more rays independently from the other threads as the traced rays are independent from each other. Modern multi-core CPUs (Central Processing Unit) and many-core GPUs are therefore ideal for ray tracing calculations.

Acceleration of ray tracing method

The computational complexity of the ray tracing method depends mainly on the amount of traced rays which may be very large, especially when a high quality output is demanded.

---

1. In the ray tracing method, light rays travel from a camera to a light source; in the opposite direction than in the real world. The reason for this is effectiveness: a very low ratio of the light rays traced from the light source would hit the camera and therefore a very high amount would have to be traced to achieve at least mediocre output quality.

For each ray, the positions of intersection points with a 3D scene (if there are any) are necessary for further pixel value computations. In the naive implementation of the ray tracing method, the intersection with every geometry primitive in the scene is computed for each ray, which leads to very high performance demands[2].

An acceleration structure may be used to significantly reduce the number of ray-scene intersection tests. The acceleration structure partitions a 3D scene into partitions and assigns them a bounding volume. During the ray-scene intersection computation, the traced ray intersects these bounding volumes. When the ray misses the bounding volume of a partition, all the primitives belonging to that partition are marked as missed and need not be intersected by the ray any further. In this way the acceleration structure can substantially reduce the amount of ray-scene intersection tests and speed up ray the tracing method computation[3] (Figure 2.1).

Even though the acceleration structure nodes' bounding volumes can be of any shape, simple geometrical objects such as a sphere and a box are typically used as they allow for simple intersection tests. A special case of the box shaped bounding volume is an AABB (axis aligned bounding box). Its specialty is that it is always aligned to the coordinate axes of a 3D scene.

Acceleration structure can have a flat hierarchy or a tree-like hierarchy. An example of the flat acceleration structure is a grid, which partitions a 3D scene into partitions with evenly shaped AABBs. During the ray-scene intersection computation, the partitions that the ray intersects are visited and their primitives are intersected with the ray. The advantage of flat acceleration structures is their low memory overhead, but this advantage is balanced out by a typically lower performance.

Tree-like acceleration structures can be divided into two categories: those that partition space and those that partition scene primitives.

---

2. Rendering of a 3D scene containing 10^5 primitives to an output image having the resolution of 1920×1080 px with only a single ray per pixel would require roughly 2×10^{11} ray-scene intersection tests.

3. With the same setup as described in footnote 2 (output resolution of 1920×1080 px, one ray per pixel, 10^5 primitives), a hypothetical hierarchical acceleration structure with one leaf node per scene primitive and with the depth of $\lceil log_2 10^5 \rceil = 15$ would reduce the number of intersection tests to roughly 3×10^7.

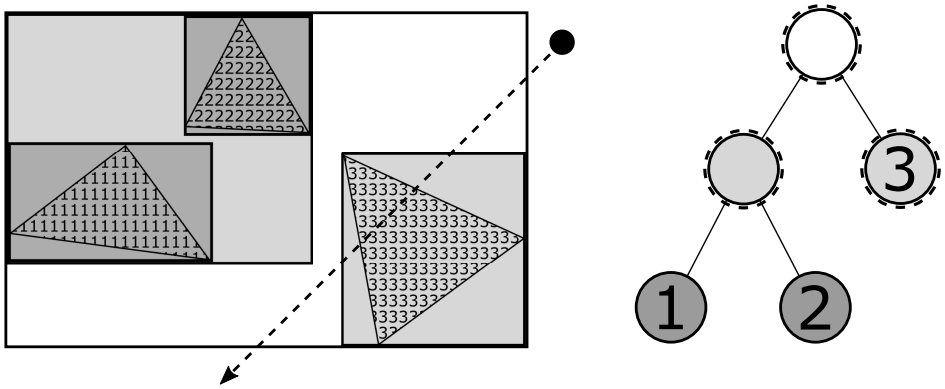

Figure 2.1: An example of the hierarchical acceleration structure BVH. **On the left:** a 3D scene partitioned by the BVH; the rectangles represent bounding volumes of the BVH nodes, the dashed line represents a traced ray. **On the right:** tree of the BVH; the number in the node represents the primitive in the node contained, the dashed circles represent nodes for which the intersection with the ray is computed.

The space partitioning structures adapt well to the 3D scenes with varying geometry densities but also tend to have a larger memory consumption because the scene primitives are saved in every single leaf node they intersect.

The primitive partitioning structures on the other hand save each of the scene primitives only once. Their drawback is the often large overlap of the nodes' bounding volumes. Such an overlap causes the traced ray to traverse an unnecessary high amount of branches of the acceleration structure tree and thus reduces the traversal speed [2].

2.2 SBVH acceleration structure

This section describes the SBVH, a hierarchical acceleration structure partitioning scene primitives. The BVH (Bounding Volume Hierarchy) is described as well, since it serves as a basis for the SBVH.

2.2.1 BVH

One of the commonly used ray tracing acceleration structures is the BVH introduced by Weghorst et al. [3]. This acceleration structure recursively partitions the primitives of a 3D scene to disjoint sets forming a binary tree[4]. An inner node of the BVH tree contains pointers to its two child nodes and an AABB[5] encapsulating the bounding volumes of those two child nodes. The leaf nodes store scene primitives (or, pointers to the scene primitives to save space) and an AABB encapsulating the bounding volumes of all the primitives belonging to that leaf node (Figure 2.1).

The BVH can be built in a top-down or a bottom-up fashion. The top-down build starts with a root node containing all the scene primitives, which is then recursively subdivided (the node's primitives are partitioned to the left and the right child node) until some given termination criteria are met (typically, the maximum number of primitives per leaf node). During the bottom-up construction, the BVH is built by the means of child nodes composition. Initially, for every scene primitive there is one leaf node. Bottom-up build terminates once all the nodes have one common parent node (the root node of the BVH).

During the ray-scene intersection test, for each ray the BVH tree is traversed starting in the root node. If the node's bounding box is hit by the ray, both child nodes are tested for an intersection and traversal recursively continues in the child nodes hit by the ray (none, one or both of them). If the ray hits a leaf node, every scene primitive belonging to that leaf is intersected by the ray and those that are hit are flagged so for the given ray.

2.2.2 BVH build techniques

Quality of the BVH is defined as its ability to accelerate the ray tracing method. When a naive build method is utilized, the constructed BVH typically performs worse during the ray traversal than if some of the more advanced build methods was used. The disadvantage of

4. The BVH is defined as a tree and its degree is not given. However, when utilized to speed up the ray tracing method, it is in the most cases binary tree.
5. Even though the shape of the BVH nodes' bounding volume is not defined, AABB is most often chosen in ray tracing scenarios.

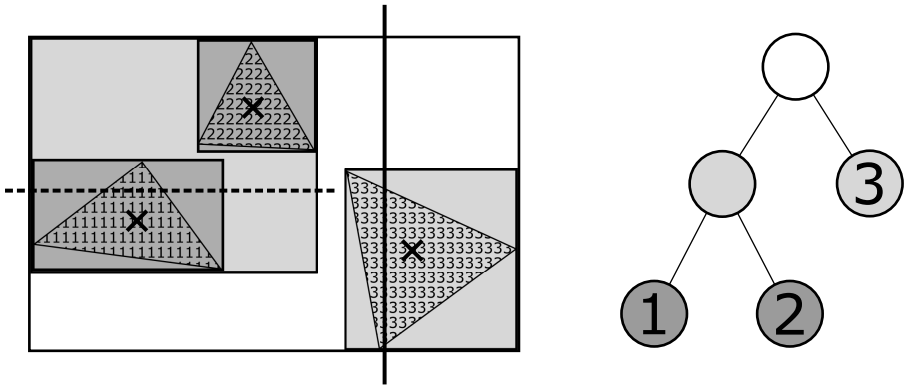

Figure 2.2: An illustration of split planes in the BVH construction process. A 3D scene composed of three primitives is partitioned by two split planes. On the first level by the split plane illustrated by the solid line and on the second (the left child of the root node) by the split plane illustrated by the dashed line. Only the primitives' centroids (illustrated as crosses) are considered in the split calculations.

these advanced techniques is their high computational complexity. Therefore, the aim is to strike balance between the BVH quality and its build time.

Split position

The partition of a BVH node can be seen as if the node was cut by a so-called split plane, infinitely large plane perpendicular to one of the coordinate axes. Primitives are partitioned to the left or to the right child node based on their position relative to the split plane (Figure 2.2).

The position of the split plane determines the quality of the split. Several techniques used to determine the position of the split plane exist. For example, the split plane can be placed in the middle of the node's AABB along one of its axes. Such simple techniques are fast to compute, but the BVH quality suffers.

In some scenarios, a split of a node by the split plane may not be possible: if all the primitives of the node have the same centroid position, all would be placed to either the left or the right child node and an empty BVH node would be created. Since such nodes are not

permitted in the BVH, measures need to be taken to prevent such a partition. To resolve the split, primitives may be partitioned evenly to the both child nodes without any regard to their position. Such a split reduces the quality of the BVH but it is rarely needed since the 3D scenes in most cases do not contain such specifically positioned primitives.

In the further text, a primitive may be said to be to the left or to the right of the split plane. The primitive is said to be to the left of the split plane if its centroid position along the axis to which the split plane is perpendicular is lower than the position of the split plane itself. Otherwise it is said to be to the right of the split plane.

SAH

One way to improve the BVH quality is to employ the SAH heuristic. This probabilistic heuristic defines the price of a split plane (the so-called SAH cost) given the bounding box of the node being split (B) and the bounding boxes (B_1, B_2) and the number of primitives ($|P_1|$, $|P_2|$) of the potential child nodes according to the formula

$$C = C_t + \frac{SA(B_1)}{SA(B)}|P_1|C_i + \frac{SA(B_2)}{SA(B)}|P_2|C_i, \tag{2.1}$$

where C is the SAH cost of the split, C_t is the cost of a single traversal step, C_i is the cost of a single primitive intersection test and $SA(V)$ is the surface area of the bounding box V. SAH assumes that rays are spread uniformly across the scene and do not hit any scene primitive before entering the node's bounding box [4].

Formula 2.1 approximates the SAH cost of a node by assuming that both the node's child nodes are leaves. The exact SAH cost calculation would require substitution of the terms $|P_i|C_i$ with the SAH costs of the corresponding subtree ($|P_i|C_i$ would be substituted by Formula 2.1 or 2.2), but such a calculation would be too complex during the build since all the combinations of all the split planes at all the tree levels would have to be computed. Nevertheless, this exact, total SAH cost also known as top-down SAH cost is valuable as a metric for already built structures.

Aim is to build the BVH for some given scene with as low top-down SAH cost as possible. This is typically achieved by always splitting the

nodes of the BVH with a split having the lowest SAH cost until the cost of the cheapest split is higher than the cost of a leaf node calculated according to the formula

$$C = C_i|P|, \tag{2.2}$$

where C is the SAH cost of the leaf node, C_i is the cost of a single primitive intersection test and $|P|$ is the number of primitives of the node.

2.2.3 Motivation for BVH improvements

In cases where the geometry is not ideally distributed throughout the scene, bounding boxes of BVH nodes may exhibit overlaps. These overlaps are undesirable since they are increasing the total SAH cost of the BVH by increasing the probability of a ray hitting both child nodes (term $\frac{SA(B_{1,2})}{SA(B)}$ of Formula 2.1). Several techniques reducing the bounding volume overlap exist.

Authors of [5] propose to compute AABBs for every single scene primitive and recursively split them along their longest axis until a threshold of surface area is met and then build the BVH from these subdivided bounding boxes. Drawback of this approach is the necessity of a user intervention to set the threshold and also a possible duplication of the same primitive in one leaf.

Authors of [6] propose similar technique. First, AABBs of the scene primitives are subdivided until the volume of the largest bounding box falls below a user set threshold. These subdivided AABBs are then used in an altered build method that removes the redundant primitive references.

Both these approaches introduce the bounding volume overlap in some scenarios, leading to an increased SAH cost. They also divide primitives to smaller than necessary pieces in some cases. To completely avoid the overlaps, authors of [2] propose to split not the single primitives, but entire sets of them instead. The primitives of a node being split that straddle[6] a split plane are partitioned into both the left

6. A primitive is said to straddle a split plane if the primitive or its bounding volume intersects the split plane.

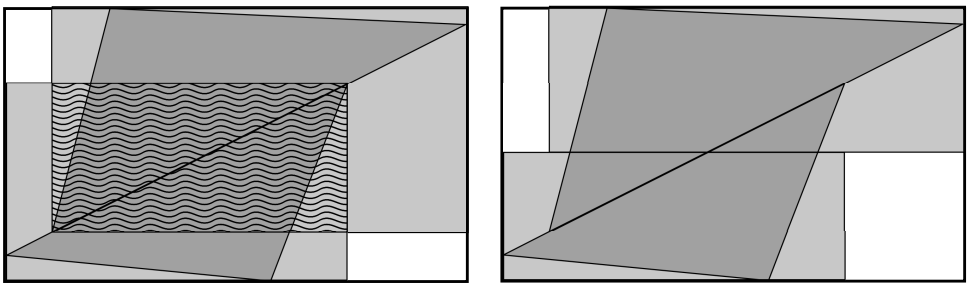

Figure 2.3: Illustration of the bounding volume overlap in a 3D scene with two primitives. **On the left:** each primitive (the dark triangles) is referenced in the BVH only once, but the bounding volumes of the BVH nodes (the light gray rectangles) may overlap (the rectangle with the wavy pattern). **On the right:** no overlap in the SBVH case. In this example, each primitive is referenced in the both leaf nodes.

and the right child node. The authors call this approach SBVH (Split BVH) and its GPU implementation is the topic of this thesis.

2.2.4 SBVH Build

Overview

The SBVH is built in the top-down fashion similarly to the BVH. When searching for the split of a node with the lowest SAH cost, two variants are considered. The first one, called object split, partitions the node in the same way as described in Subsection 2.2.2 using the SAH heuristic.

The second split variant considered is called spatial split and is described in further text. Best (having the lowest SAH cost) object and spatial split are compared and the better one is used to partition the node (unless some termination criteria are met and a leaf node is created instead).

Object split

Finding the best object split is straightforward. All the possible split planes can be tested since their number is bound by the number of primitives in the node. Indeed, the SAH cost of a split stays the same if the split plane moves only between the closest left and right primitive's

centroid, since the number of primitives and the bounding volume of the left and the right child node does not change; the exact position of the split plane is irrelevant in the SAH cost formula (Formula 2.1).

This full SAH search of a split may be in some situations too computationally complex. If that is the case, the so called binning technique described further in the text that reduces the number of tested split plane positions may be employed instead.

Spatial Split

The binning technique is used[7]. Node's AABB is divided by equidistant planes (the candidate split planes) into bins and for every such a bin its AABB is stored. Each of the node's primitives that straddle the split plane is cut and for each its part a new AABB is constructed (up to one per bin, Figure 2.4). It is important to note that for each cut primitive the total surface area of the newly created AABBs is usually lower than the surface area of the original AABB, allowing for the lower SAH cost of the split.

The primitives' AABBs are used to grow bins' AABB. Once this process is finished, each bin has an AABB tightly enclosing all the primitives (or parts of the primitives cut by the split plane) intersecting the bin.

The SAH cost of each candidate split plane is calculated according to Formula 2.1, B_1 and B_2 is union of all the bins' AABBs to the left and to the right of the split plane respectively and $|P_1|$ and $|P_2|$ is the number of primitives to the left and to the right of the split plane respectively. The primitives straddling the split plane are therefore saved in both the child nodes.

Finally, the plane with the lowest SAH cost is selected as a spatial split for the given node.

Memory considerations

When cut by a split plane, both parts of the primitive are saved in the SBVH tree, each one in the respective subtree (the part of the primitive to the left of the split plane in the left subtree and vice versa). Splitting

7. The binning technique is in detail described in [7].

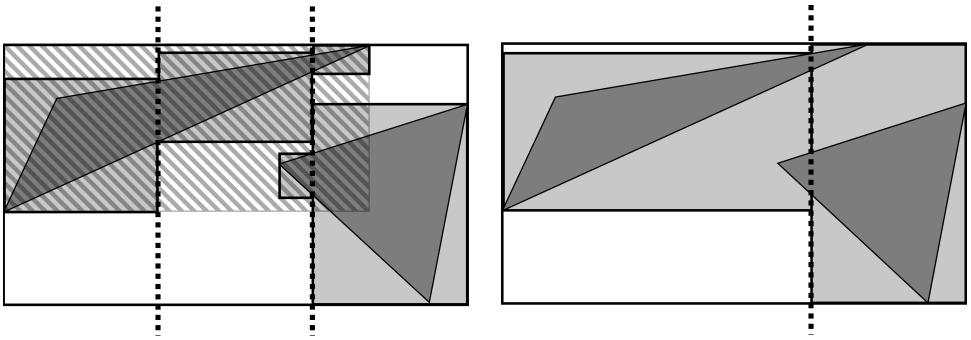

Figure 2.4: An illustration of the binning technique. **On the left:** binning of a BVH node with two split planes in one axis. AABBs are recomputed for each part of the cut primitives. The original AABB of the left primitive is illustrated in the striped pattern. **On the right:** the bounding boxes of the left and the right child node for the second split plane.

of the primitives therefore increases the memory consumption of the SBVH.

Since the BVH references each scene primitive only once, its memory consumption can be calculated before the build process begins and only one memory allocation is required. The SBVH, on the other hand, requires a dynamic memory allocation during the build process to accommodate for the split primitives.

Dynamic memory allocation in the context of GPUs is described in Section 3.3.

Traversal

Even though the BVH's property of referencing each scene primitive only once in the whole tree is violated, the same traversal algorithm can be used for the SBVH as for the BVH. The only difference is that a ray may traverse more than one path through the tree to reach the same primitive.

2.3 K-d tree acceleration structure

Overview

Another commonly used ray tracing acceleration structure is the K-d tree introduced by M. Kaplan [8]. Since it is similiar to the BVH in some aspects and more so to the SBVH, its is briefly introduced here and all the structures are compared in Chapter 4.

The K-d tree recursively partitions the space of a 3D scene to disjunct half-spaces[8]. Nodes of the K-d tree are cut by a split plane, which is always perpendicular to one of the three coordinate axes[9]. Similarly to the SBVH with its spatial split, The K-d tree often saves the same reference in multiple leaf nodes.

The K-d tree is a binary tree composed of inner nodes and leaf nodes. The inner node contains information about the split plane axis, the position of the plane on that axis and the two child nodes. The leaf node contains only information about the scene primitives belonging to that leaf.

The memory consumption of the K-d tree node is lower than that of the BVH, since no information about the node's bounding volume is stored.

K-d tree build

The build process of the K-d tree starts with a root node spanning all the scene primitives. The root node is then recursively cut by a split plane until some given termination criteria are met (for example, the maximum number of primitives per node or a maximum tree depth).

The process of finding the split plane axis and position is very important one as it determines the quality of the constructed K-d tree. Several build algorithms exists: In scenarios where the quality of the tree is not important, split planes can be chosen in a round robin fashion to ensure a fast build. On the other hand, one can utilize

8. Name of the K-d tree is a shorthand for K-dimensional tree, it can be utilized to partition space of any number of dimensions.
9. This is the difference between the K-d tree and the BSP (Binary Space Partitioning) data structure, which has its split planes arbitrarily rotated. The K-d tree is therefore a special case of the BSP.

techniques such as SAH (Formula 2.1) to build K-d trees of a very high quality.

An overview of various K-d tree build methods is in [9].

K-d tree traversal

The K-d tree traversal algorithm starts with a ray being traced intersecting the bounding box of the entire scene. This gives the so-called enter and exit distance - the distance from the ray origin along the ray direction to the nearest and furtherest point of the scene AABB. Between these two points an intersection with the scene geometry is sought by visiting the K-d tree leaves in order from the closest to the furthest.

The leaves that are intersected by the ray are found by means of point-location queries on the K-d tree. First such point is the one at the enter distance along the ray. The K-d tree is traversed from the root by visiting that child node whose half-space contains the sought point until a leaf node is reached. Then, all the primitives belonging to that leaf are intersected with the ray.

If no intersection is found, the point laying just outside of the AABB associated with the previously visited leaf node and along the ray direction is queried. The traversal end once the first intersection is found[10], or once the queried point is no longer within the scene AABB (distance along the ray is greater than the exit distance) [10].

10. If all the intersection points are demanded, the traversal continues even when the first intersection have been found.

3 Implementation

This chapter describes the implementation of the SBVH acceleration structure on the GPU. The framework and its extensions that made the SBVH implementation possible is described as well.

3.1 Framework

Framework NTrace have been chosen as a basis for the described SBVH build method. This framework is greatly extended version of the framework created by Timo Aila, Samuli Laine and Tero Karras to accompany their work on ray tracing traversal efficiency on GPUs. [11].

NTrace allows users to load 3D scenes and render them via the ray tracing method in real time as well as offline. Currently supported acceleration structures are the BVH and the K-d tree, each with various different build methods. Purpose of the NTrace framework is to serve as a tool for the development and testing of highly parallel implementations of algorithms for data structure build and trace using the GPU hardware [12].

NTrace is partly programmed in C++ programming language and partly in C for CUDA to enable real-time rendering and acceleration structure construction on the GPU. It is available on the address `https://github.com/marekvinkler/NTrace`.

3.1.1 CUDA Programming model

Overview

C for CUDA extends the C language with extensions that allows the programmer to write so-called kernels. The kernel is a program run on the GPU in a form of many parallel threads.

The threads in the CUDA programming model form a hierarchy. Individual threads are organized into blocks. The threads in a block synchronize trough the shared memory and thread fences. Thread blocks are further organized into a grid. The number of threads running the kernel depends on the dimensions of the grid and the blocks.

When a kernel is launched, thread blocks are distributed to the processors of the GPU called Streaming Multiprocessors (SMs) and begin processing the kernel program. The kernel computation is complete once all the blocks have finished.

Threads are scheduled and executed in groups of 32 called warps. All the threads in a warp execute one common instruction at a time. If some of the threads in the warp diverge due to a conditional branch into one or more groups with each having a different code to run, these groups are executed serially which effectively reduces the utilization of the SM [13].

Memory model

CUDA programming model offerers several different memory spaces for threads to access, each with its benefits and drawbacks. It is therefore important to use an appropriate memory space for a given task to utilize the GPU to its fullest.

Two distinct types of memory are present on the GPU. The so-called device memory, which is located on the GPU, but away from the SMs and on-chip memory, located directly on each of the SMs. The on-chip memory is faster and has a lower latency than the device memory, but its capacity is substantially lower.

Each thread can use a limited set of registers; very fast on-chip memory accessible on per-thread basis, used to store variables (unless the register spilling occurs as described below).

The global memory resides in the device memory. It has a high capacity, but imposes access rules for threads to achieve a maximum bandwidth. If the threads in a warp read or write aligned data of specific sizes, these operations are coalesced into one. On the other hand, if the coalescence rules are violated, the number of operations will increase and the resulting instruction throughput will be reduced.

The local memory is performance wise similar to the global memory, but unlike it, it is accessible on per-thread basis. It is typically used for large structures and arrays. Additionally, if a thread uses up all the available registers, additional variables are stored in the local memory which causes performance drops (this phenomenon is know as register spilling).

The shared memory allows for a fast thread communication within the thread blocks. Because it resides directly on the SMs (on-chip memory), it offers a higher bandwidth and lower latency than the device memory spaces. The shared memory is divided into memory banks. To expose its full potential, threads in a warp need to avoid accessing the same bank.

The constant memory (residing in the device memory) enables a fast read of constant data (the data that will not change during the kernel computation). The number of read requests is equal to the number of different memory addresses accessed by the threads in a warp. Therefore, if all the threads read the same constant variable, they will be served in a single read operation. Moreover, the constant memory is cached to further improve the performance.

The texture and surface memory spaces reside in the device memory. These memory spaces have its own cache optimized for spatial locality and in some cases it is beneficial to access the device memory not via the global memory space, but via the texture memory instead (for example when memory reads do not follow the global memory access pattern) [13].

3.2 Massively Parallel Hierarchical Scene Processing

Overview

To ensure the maximum build speed of the described SBVH build method, the implementation uses one of the extensions of the NTrace framework that allows to run the entire acceleration structure build process on the GPU in a single call. Once the build kernel is launched, no CPU intervention is needed until the build is finished. Such an approach improves the performance since the kernel launch overhead[1] as well as GPU memory latencies are greatly reduced.

Authors of [14] propose a novel method for massively parallel hierarchical scene processing on the GPU. This method is based on a sequential decomposition of the given hierarchical algorithm into small functional blocks and is managed entirely by the GPU [14].

1. As opposed to launching the kernel for each SBVH node separately.

Task processing algorithm

The proposed method is built on the concept of persistent warps. NVIDIA GPUs have its work distribution system optimized for homogeneous work units. In some scenarios, the units of work may vary greatly in complexity causing starvation issues in the work distribution [11]. The concept of persistent warps bypasses the work distribution system completely.

The build kernel (the kernel that performs the acceleration structure construction) is launched with as many warps as the GPU can handle to run concurrently. These then in loop seek for available work in the global work pool, process it, and then seek again until no work is available and the computation is completed.

The proposed method processes so-called tasks. The task is a computational job associated with a range of the scene primitives. It can have several phases, each phase being a logical algorithmic block of the task. A phase can be further decomposed to one or more steps. With the each step there are associated so-called work chunks, the smallest units of work in the method. A work chunk is always processed completely by a single warp.

The computation itself is managed by a task pool. Warps retrieve work chunks from the pool until it is empty which signals that the computation is finished. In the beginning of the computation, the task pool contains a single task spanning all the scene geometry. Once a task is completed, one or more child tasks may be added to the task pool [14].

A task may be dependent on some other tasks in the pool. If that is the case, all the tasks causing this dependency need to be finished first and only then the dependent task may be processed.

Task pool

The task pool is composed of two arrays: a header array and a data array. The elements of the header array are small in size (32bit integers) for a fast access and low memory footprint. The main purpose of the header array is to store information about the amount of work (the number of work chunks) each task offers. The data array holds

remaining information about the task, most notably: its data range, phase, step and dynamic memory pointers.

Warps retrieve work by looping through the header array. When the warp finds available work chunks, it atomically decrements the amount of work chunks available in the header array and begins their computation. It is often beneficial for warps to retrieve more than one work chunk since the operation of work retrieval is not trivial [14].

Once a warp processes retrieved work chunks, it checks whether it was the last one for the given task (all the work chunks of the task are completed) and if so, advances the state of the task. It may also create a new task by looping through the task header and filling in an empty cell as well as the corresponding cell in the data array.

3.3 Dynamic memory allocation on GPU

This section contains an overview of selected dynamic memory allocation algorithms for CUDA enabled GPUs.

Overview

As noted in Section 2.2.4, the SBVH build algorithm requires a dynamic memory allocation, since the size of the SBVH can not be easily determined beforehand. The described SBVH build method utilizes the dynamic memory allocators that are part of the NTrace framework.

Dynamic memory allocation on the GPU is problematic due to many-core architecture of the GPUs. A high amount of threads may want to allocate the memory at the very same time and cause a congestion (typically in a form of serialization of atomic operations). The task is to allocate the memory fast, without errors and as high amount of it as possible (while the allocation amount may highly as well).

Several GPU dynamic memory allocation algorithms for the CUDA enabled GPUs exist. The simplest example of such an algorithm is AtomicMalloc, which allocates the memory by the means of managing a pointer to the remaining free space. Whenever an allocation of memory of a given size is requested, it stores the current pointer, atomically increases it according to the request size and returns the

stored pointer. While AtomicMalloc is very fast, its drawback is the inability to deallocate memory.

The CUDA toolkit has a built-in (unpublished) algorithm for the dynamic memory allocation, but it has been shown to be rather slow [15].

More advanced algorithms exist (ScatterAlloc [16], FDGAlloc [17], Halloc [18]), but none is suitable for the general purpose memory allocation when the size and the frequency of allocation requests is not known beforehand. The authors of [15] propose a GPU memory allocation algorithm called CMalloc, which performs exceptionally well in these conditions.

CMalloc

Circular Malloc (CMalloc) pre-splits the memory space into chunks. These chunks are then organized into a singly linked list.

Each memory chunk has a header composed of a flag signaling whether the chunk is free or used and a pointer to the next memory chunk (*nextPtr*). Memory is pre-split to form a binary heap: the first chunk has size R, next two chunks have half the size of R, next four chunks one eight each and so on. R is chosen so that the biggest binary heap with a root node of size R fits into the memory pool of the GPU.

When a thread desires to allocate memory of size S, it loops through the chunk headers starting at the *memOffset*, which is a global variable shared across all the threads. When the thread arrives at a free chunk of size at least S, it atomically sets it as used and alters *memOffset* to point to the next chunk in the list.

Wasting of the free space caused by small allocation requests taking big memory chunks is restricted by *fragMult*, fragmentation multiplier. If the chunk being allocated is bigger than S times *fragMult*, it is split into two so that the first part has size S and the second one has size of the chunk minus S. *memOffset* is then set to the end of the first part.

When a thread deallocates some chunk, it first checks whether the next chunk in the list is free. If it is, it sets *nextPtr* of the chunk being deallocated to *nextPtr* of the next chunk thus effectively concatenating the two chunks. *memOffset* is set to the chunk next to the freed one.

3.4 GPU SBVH

This section describes a mapping of the SBVH build algorithm to the framework described in Section 3.2 and 3.3.

3.4.1 Mapping of SBVH build algorithm to framework

A core part of the framework for highly parallel hierarchical scene processing (Section 3.2) is a task spanning a set of the scene primitives. In the context of the described SBVH build method, the task can be viewed as an SBVH tree node. Each task holds information necessary for its processing, the most important include: the global memory offset pointing to the task's references (an input array), the two other offsets pointing to references assigned to the left and the right child (output arrays), its bounding box and information about the split of the node (a split plane axis and position, AABBs and reference counts of the left and the right child node).

The implemented SBVH build algorithm processes tasks which are composed of two phases. The first one (Subsection 3.4.2) bins scene primitives: for the given number of split planes in each coordinate axis, it computes the bounding boxes and the number of primitives for both the child nodes should the parent node be split by any of the split plane. Then, the best split plane and type of split is chosen.

The second phase (Subsection 3.4.3) performs the partition of the scene primitives according to the split plane chosen in the first phase. The node's references are copied to the left and to the right output array and if the termination criteria are met for either of the child nodes, leaf nodes (zero, one or two nodes) are created. For the non-leaf child nodes, new tasks spanning their references are inserted into the task pool.

Implementation details

The described SBVH build algorithm processes references as proposed in [2]. The reference is a structure composed of a pointer to some scene primitive and an AABB of that primitive.

If a reference (and therefore its primitive) is cut by a split plane, two new references are created with the freshly constructed AABBs

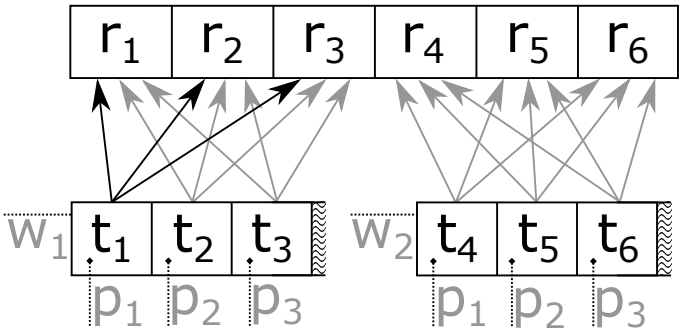

Figure 3.1: Two warps (w_1, w_2) bin six references. In this example, only three different split planes (p_1 to p_3) are considered. Therefore, only three threads in each warp participate.

encapsulating respective part of the primitive. Both the new references then have the same pointer as the original reference.

3.4.2 Binning phase

Each warp processing the binning phase bins some subset of the scene references belonging to the task (work chunks of the task). Based on the number of split planes in each coordinate axis, some threads of the warp may not participate in the binning. More specifically, as each participating thread bins references for a single split plane, the number of participating threads in the warp equals to the number of split planes and this number is always lower than or equal to 32 (number of threads in the warp) (Figure 3.1).

Throughout the phase computation, each participating thread needs to hold an AABB and the number of references of the left and the right bin for both the object split and the spatial split as they are computed at the same time.

Split plane

Each participating thread first calculates the position and axis of its split plane based on the thread index and the number of bins. The split planes are uniformly spread across the bounding box of the node being split. Also, the number of split planes is the same along each

coordinate axis. Since the warp is a group of 32 threads, the maximum number of split planes per axis is 10 and the number of bins is 11.

Binning

Threads loop through the references of the warp (its work chunks). Each reference is fetched from the global memory and then intersected with the split plane. If it lies completely to the left or to the right of the plane, the bounding box of the corresponding bin is enlarged so that it encapsulates the reference. In this case, the bounding box of both the split types is enlarged in exactly the same way.

If the reference straddles the split plane, the AABBs of the object split are adjusted in the similar fashion as if the reference didn't intersect the split plane at all: if the centroid of the reference lies to the left of the plane, the AABB of the left bin is enlarged and vice versa.

In the spatial split case, both the left and the right AABB is enlarged, but not with the original AABB of the reference. Instead, two new AABBs are created tightly enclosing the two parts of the primitive cut by the split plane and those are then used to grow the respective bin's AABB (Figure 2.4, the difference between the object split and the spatial split is illustrated in Figure 3.2). The primitive itself has to be fetched from the global memory first, since references do not contain vertex positions necessary for the AABB construction.

The split data (an AABB and the number of primitives for the two bins of each split plane) are stored in the global memory. Once a thread processes all the references assigned to it, it atomically grows the bins and increases the primitive counts of the given split plane. This operation indeed needs to be performed atomically, since more than one warp may be binning the task's references.

Best split plane selection

Even though the warps can cooperate on the binning (each one processing the subset of the task's primitives), only a single warp chooses the best split plane and split type. Based on the information gathered in the preceding computation, each participating thread calculates the SAH cost for a single split plane. Cost of both the spatial split and the object split is computed and the one with lower SAH cost is used in

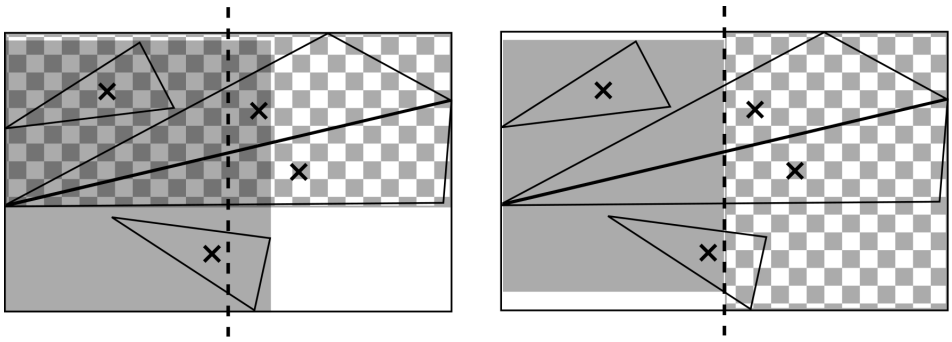

Figure 3.2: The differences between the object split (on the left) and the spatial split (on the right). The solid rectangle represents the AABB of the left child, the checkered rectangle the AABB of the right child. In the object split scenario, the whole primitives are split to the child nodes based on the positions of the primitives' centroids. On the other hand, only parts of the primitives in the relevant bin are placed in the bin in the spatial split case.

further computation. Finally, by the means of warp reduction, form all the candidate split planes the best one (with the lowest SAH cost) is selected for the entire task.

Build termination

The SAH cost of the split is compared with the cost of the node as if it was a leaf node. If the split cost is bigger, the partition phase is skipped completely and a leaf node is created (the leaf node construction is described in Section 3.4.3). Otherwise, memory is allocated for both the child nodes.

The aforementioned SAH cost comparison takes place only if the number of primitives of a node falls below some given threshold, otherwise the node is split and partitioned without any cost comparison.

Object median split

On some rare occasions, the primitives may be binned so that they all lay to the left or to the right of the split plane, resulting in an empty node (Figure 3.3). Such nodes are not permitted in the BVH/SBVH as

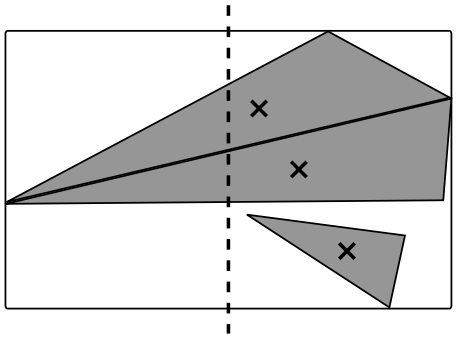

Figure 3.3: BVH node split by a single split plane. Centroids of triangles are marked with crosses. Object split of the node would result in an empty node (centroids of all the primitives are to the right of the split plane).

described in Subsection 2.2.2. To overcome this problem, two measures are taken. First, if the problematic split is of object type, the spatial split with the same split plane can be used instead.

If the AABB of the node being split is very small, the spatial split may create an empty node as well due to numerical inaccuracies. If that is the case, an object median split of the node is performed: the child nodes are assigned the same AABB as the parent node has and during the partition phase, the references are evenly and without any regard to their position in the 3D space distributed to the two child nodes.

The quality of the object median split is very poor (due to the parent AABB copying and the naive reference partition), but since it is rarely present in the tree (and if it is, it is near the bottom of the tree), its impact on the SBVH quality is usually not noticeable.

3.4.3 Partitioning phase

Partitioning

First, the position of the task's references relative to the split plane (left of, right of, straddling the split plane) is computed once more, since, to save the memory, only the child node's AABBs and reference counts were saved during the binning phase. The references therefore

need to be fetched from the global memory once more and in the case of the straddling references the corresponding primitives need to be fetched as well.

Each thread in the warp is assigned a number X ranging from zero to L, where L is the number of references of the warp classificated as being left of the split plane or straddling it. This number is obtained by means of exclusive warp scan. X is then used together with a global atomic counter (since the warps may cooperate) to determine the position of the reference in the left output array. The AABB of the straddling references need to be cut and recomputed as described in Subsection 3.4.2 before the references are saved to the left output array.

The same procedure is then repeated for the references classificated as being right of the split plane and the appropriate subset of the straddling references. These references are written to the right output array.

Object median split

If the object median split is chosen as a mean to split a node, the task's references are marked as being left of or right of the split plane based on their position in the input array: if the reference's index in the input array is even, it is marked as being left of the split plane and vice versa.

The partition then continues in the same way as if the split was of object or spatial type.

Build termination

Once the references are partitioned and copied to the output arrays, a single warp checks whether the computation will continue, or will be terminated.

Termination criteria are computed for the child nodes. If the number of references of the node is below some given threshold or if the depth of the node is above some given threshold, a leaf node is created.

Each reference belonging to the leaf is saved into two different output arrays. The triangle index stored in the reference is copied into the triangle index output array and triangles themselves are first

transformed to the format suited for intersection testing as described in [19] and then saved into the triangle output array.

A new task is created for each non-leaf child node (one or two tasks). The newly created task spans the child node's references and its phase is set to binning. Its input array is one of the two output arrays of the parent node. Once these tasks are saved in the task pool, warps may start processing them. Since the references of the task are copied from the input array to the output arrays, the input array is deallocated.

4 Results

This chapter summarizes the comparison of the GPU SBVH with various other acceleration structures: CPU SBVH, CPU K-d tree, GPU BVH and GPU K-d tree.

4.1 Comparison Metrics

Quality of acceleration structure

The quality of an acceleration structure is the structure's ability to accelerate the ray tracing method. One way to measure the quality is to calculate the structure's top-down SAH cost (as defined in Subsection 2.2.2).

The SBVH and the BVH can be directly compared using the top-down SAH cost as both have an identical node structure. The comparison of the SBVH/BVH and the K-d tree using the top-down SAH cost is possible as well, since the concept of a node with a bounding box and the number of primitives in the subtree of the node is present in the both acceleration structures. But generally, a lower top-down SAH cost of one kind of acceleration structure does not necessarily mean that it will accelerate the ray tracing method better than structure of other kind with a higher cost. Moreover, to a small extent, this applies to scenarios where acceleration structures of the same kind are compared as well.

The direct comparison of acceleration structures of various kinds may be performed by measuring the number of traced rays per second. Unlike the SAH cost comparison, this metric shows the real performance. Moreover, it correlates with the render time: the time necessary to render the output image is similar to the time necessary to trace rays for all the pixels. The difference is that only the time of the actual ray tracing is included and time necessary for shading and output display is omitted.

Three distributions of rays may be traced in the NTrace framework: primary, ambient occlusion and diffusion[1]. The number of traced rays

1. *Primary rays* are rays traced from a camera to the 3D scene. *Ambient occlusion rays* are rays traced from the impact points of the primary rays into the 3d scene in

was measured and averaged across two to three cameras for the each scene, since the metric is view-dependent. Even though detailed measurements were performed for all the ray distributions, only primary and diffusion ray measurements are listed in detail to reduce the number of tables in the thesis.

Acceleration structure build time

A build time is the time necessary to construct an acceleration structure for the given scene. Since the structure needs to be constructed before any ray tracing can take place, it is desirable to decrease the time as much as possible.

The implemented SBVH and already existing GPU build methods build the acceleration structure so that its nodes are stored in a linear array with the parent nodes holding offsets to the child nodes. CPU build methods construct the acceleration structure so that the nodes are stored in the dynamic memory and parent nodes hold pointers to its child nodes.

NTrace framework requires the acceleration structure to be in the former format (a linear array) and so the conversion of the CPU acceleration structure to the linear format is necessary before the rendering process begins. The time the conversion process takes is included in the CPU SBVH and CPU K-d tree build time. Nevertheless, the measured conversion time was always at most 1% of the build time and therefore it does not affect the total time much.

3D scenes used in measurements

The above mentioned metrics were measured across a set of 3D scenes (Table 4.1, Figure 4.1). Each scene in this set represents some specific 3D model category and the entire set covers a broad range of all the different 3D scene types.

---

random directions and with a limited length. These rays simulate light transport by estimating how much of the hemisphere above each impact point is occluded [20, p. 926]. *Diffusion rays* are similar to the ambient occlusion, but their range is not limited and they alter color component of the image at the position of their origin.

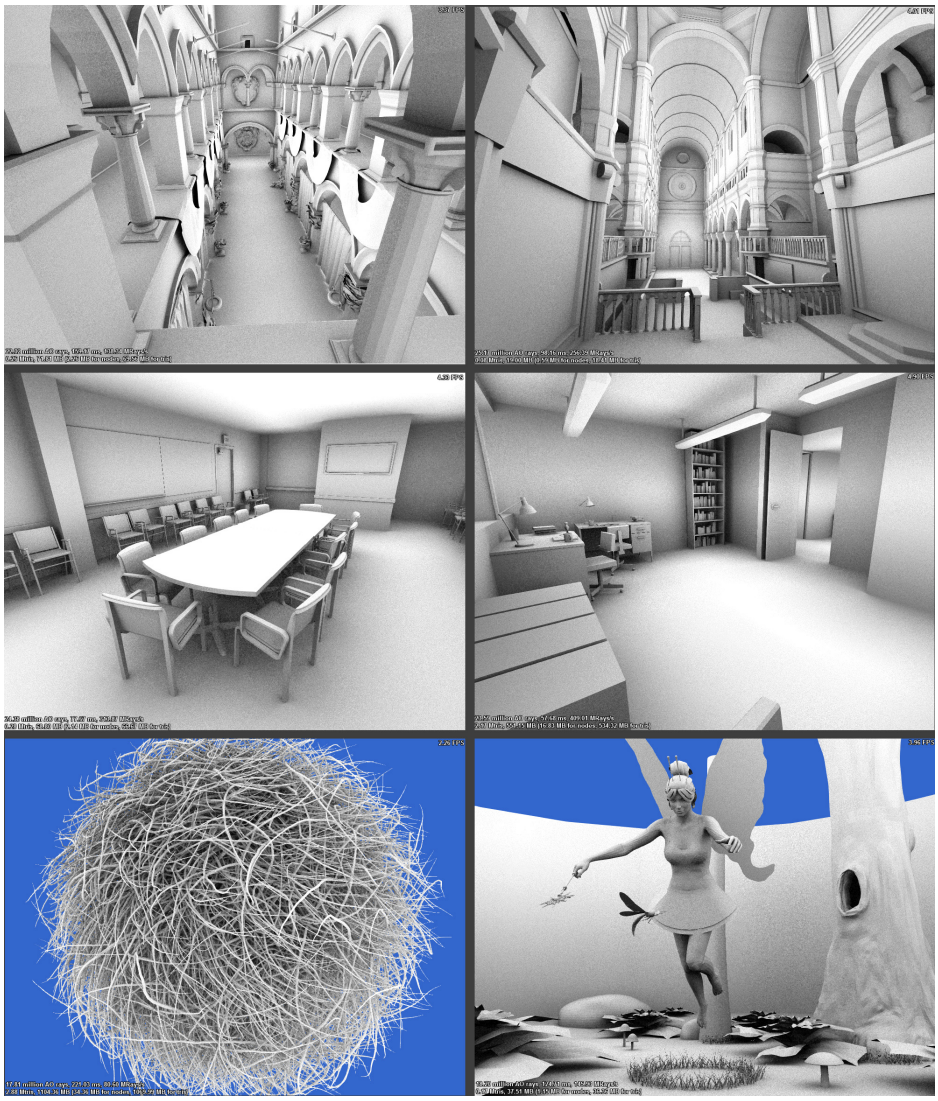

Figure 4.1: *Sponza*, *Sibenik*, *Conference*, *Sodahall*, *Hairball* and *Fairyforest* 3D scenes.

Name	Triangles	Short description
Sibenik	80k	Non-uniform scene with aligned geometry
Fairy Forest	97k	Partly uniform scene without aligned geometry
Sponza	145k	Non-uniform scene with aligned geometry
Rotated Sponza	145k	*Sponza* scene rotated 45° along each coordinate axis
Conference	282k	Non-uniform scene with aligned geometry
Blade	1765k	3D scan, very uniform scene
Soda Hall	2169k	Non-uniform scene with aligned geometry
Hairball	2880k	Ball-shaped thin polygon strands, fairly uniform

Table 4.1: A large spectrum of 3D scenes was selected to cover many possible scene types. Two scenes are worth further note: *Rotated Sponza* is the same scene as *Sponza*, but rotated 45 degrees around each of the coordinate axis. *Hairball* is a model of thin polygonal strands formed to shape a ball. Both these scenes are hard for the BVH to partition without heavy overlap.

A very important attribute of a 3D scene that affects the acceleration structure build process is the uniformity of the scene model[2]. If a scene is uniform, acceleration structure can be usually built with the simplest methods available without compromising the structure quality. In the SBVH/BVH case, the more the scene is uniform, the closer quality-wise are the BVH and the SBVH.

Another important factor is the orientation and alignment of the geometry. Flat, axis aligned geometry is typically easier to partition by the means of axis aligned bounding boxes than arbitrarily rotated primitives. Both the BVH and the K-d tree node may have a flat AABB which has a low surface area and thus helps reducing the overall SAH cost of the structure.

The absolute scale of a 3D scene needs to be considered as well. If the scene is too small or too big, numerical errors may occur during the acceleration structure build as well as during its traversal.

2. The uniformity of a 3D scene is given by the distribution of its primitives. Uniform scenes are composed of similar primitives evenly distributed throughout the scene.

Measurements setup

All the measurements were performed using the NVIDIA GeForce GTX 980 GPU and the Intel Xeon e5-2620 v2 CPU.

Output images were rendered in resolution of 1024×768 px and 64 secondary rays[3] were traced per single primary ray.

The termination criteria were set the same for all the acceleration structures if possible: the maximum tree depth was set to 50, the maximum primitive count to always crate a leaf node to 16 and both C_i and C_t constants of the SAH cost formula (Formula 2.1 and 2.2) to 1.

4.2 Comparison of GPU SBVH with CPU acceleration structures

The CPU variant of the SBVH build method is a single-threaded algorithm which builds the SBVH exactly as it is described in [2]. The implemented GPU SBVH builds a very similar, but not exactly the same structure, mainly due to performance constrains and differences in the architecture between the CPU and the GPU.

CPU K-d tree is a the highly optimized algorithm described in [9]. It builds the K-d tree with the time complexity of $O(n \log n)$ while maintaining a high acceleration structure quality.

4.2.1 Build time comparison

The build time of the GPU and the CPU variant of the SBVH differs greatly (Table 4.2). The GPU variant is on average 213× faster than the CPU variant. Moreover, the speedup is fairly consistent, ranging from 140× to 422×.

Noteworthy is the comparison of build times on *Sponza* and *Rotated Sponza* scenes: GPU SBVH achieves almost the same times on the both scenes, but the CPU methods suffer from the rotated geometry. All the methods build deeper and larger tree in the case of rotated variant of the *Sponza* scene, but the GPU SBVH handles this workload increase well thanks to the relatively small primitive count

3. Secondary rays are the rays shot from the place the primary ray hit. They are generated when ambient occlusion or diffusion rays are traced.

Scene	GPU SBVH	CPU SBVH	CPU K-d tree	SBVH	K-d tree
Sibenik	0.032 s	6.450 s	0.656 s	197×	20×
Fairy Forest	0.063 s	9.335 s	2.287 s	146×	35×
Rotated Sponza	0.107 s	26.810 s	5.638 s	249×	52×
Sponza	0.098 s	13.809 s	3.640 s	140×	37×
Conference	0.110 s	17.951 s	3.022 s	162×	27×
Blade	0.467 s	98.199 s	14.353 s	209×	30×
Soda Hall	0.681 s	121.472 s	22.492 s	178×	32×
Hairball	1.529 s	645.462 s	128.832 s	422×	84×

Table 4.2: Build time comparison of the GPU SBVH with the CPU SBVH and the CPU K-d tree (time is listed in seconds, lower values are better). The fourth and fifth column contain the speedup of the GPU SBVH buid over the CPU SBVH and the CPU K-d tree respectively.

of the *Sponza/Rotated Sponza* scenes and the high parallelism of the algorithm.

The difference of the build time of the CPU K-d tree and GPU SBVH (Table 4.2) is not as big as in the CPU SBVH case mentioned above. On average, GPU SBVH is built 40× faster than the CPU K-d tree. Once again, the speedup is consistent over the set of test scenes.

4.2.2 SAH cost comparison

The SAH cost measurements suggest that the GPU variant builds the SBVH of the same quality as the CPU variant (Table 4.3). Indeed, the average difference of the SAH cost between the two variants is 2.1% in favor of the GPU variant.

As mentioned in Section 4.1, the SAH cost comparison of the K-d tree and the BVH/SBVH needs not reflect real performance differences as the traversal algorithms of the structures differ.

The SAH cost of the CPU K-d tree is on average 16% lower than that of the GPU SBVH. The K-d tree benefits from an aligned geometry (most notably the *Sodahall* and the *Sibenik* scenes), since it allows for the simple empty space cutoff. On the other hand, the unaligned geometry of the *Rotated Sponza* is problematic for the CPU K-d tree and causes it to perform worse than the GPU SBVH.

Scene	GPU SBVH	CPU SBVH	CPU K-d tree	SBVH	K-d tree
Sibenik	44.5	44.4	32.5	−0.2%	−26.9%
Fairy Forest	58.3	58.5	54.6	0.3%	−6.2%
Rotated Sponza	98.0	104.3	103.0	6.4%	5.1%
Sponza	80.3	87.6	64.0	9.1%	−20.2%
Conference	68.0	69.3	53.0	1.9%	−21.9%
Blade	136.6	134.0	113.8	−1.9%	−16.6%
Soda Hall	123.2	124.0	87.1	0.6%	−29.2%
Hairball	607.7	619.0	495.1	1.9%	−18.5%

Table 4.3: The SAH cost comparison of the GPU SBVH, CPU SBVH and the CPU K-d tree (lower values are better). The fifth and the sixth column contain the increase (negative values decrease) of the CPU SBVH and the CPU K-d tree SAH cost respectively over the GPU SBVH SAH cost (values are listed in percents).

4.2.3 Traversal speed comparison

The comparison of the number of traced rays per second (Table 4.4 and 4.5) shows that the GPU SBVH is on average 11%, 4% and −0.5% (for primary, ambient and diffusion rays respectively) faster than the CPU SBVH.

When compared with the CPU K-d tree, GPU SBVH is 33%, 26% and 19% faster on average.

4.2.4 Summary

The implemented GPU SBVH achieves a similar performance (on average 2.1% lower SAH cost and on average 5% higher number of traced rays per second) as the CPU variant while its construction is considerably (on average 213×) faster.

The GPU SBVH tree has on average 20% more nodes and is 16% deeper than the CPU SBVH one. On the other hand, the average number of triangles per leaf node is almost the same (the difference is below 0.4%). Such a difference is caused by small alterations of the SBVH build algorithm.

When compared with the CPU K-d tree, the build process of the GPU SBVH is on average 40× faster while the number of traced rays per second is 15% higher. The performance speedup (during the rendering) varies highly since the uniformity and the alignment

Scene	GPU SBVH	CPU SBVH	CPU K-d tree	SBVH	K-d tree
Sibenik	375.5	380.5	416.3	−1.3%	−10.0%
Fairy Forest	237.5	272.0	142.2	−12.6%	66.9%
Rotated Sponza	147.4	109.8	126.4	34.3%	16.6%
Sponza	242.9	190.5	248.3	27.5%	−2.1%
Conference	347.8	324.4	231.5	7.2%	50.2%
Blade	371.4	365.8	328.7	1.5%	12.9%
Soda Hall	504.9	433.7	518.8	16.4%	−2.6%
Hairball	61.5	53.2	26.3	15.5%	133.5%

Table 4.4: Comparison of traced primary rays per second (listed in millions of rays per second, higher values are better). The fifth and the sixth column contain speedup of the GPU SBVH over the CPU SBVH and the CPU K-d tree respectively.

Scene	GPU SBVH	CPU SBVH	CPU K-d tree	SBVH	K-d tree
Sibenik	223.7	244.1	252.7	−8.3%	−11.4%
Fairy Forest	128.0	139.5	76.5	−8.2%	67.2%
Rotated Sponza	98.9	86.6	78.3	14.2%	24.8%
Sponza	149.6	131.3	136.7	13.9%	9.4%
Conference	223.8	227.1	161.1	−1.4%	38.8%
Blade	121.3	132.8	126.9	−8.6%	−4.4%
Soda Hall	317.6	310.9	348.7	2.1%	−8.9%
Hairball	37.0	40.5	26.5	−8.5%	39.4%

Table 4.5: Comparison of traced diffusion rays per second (listed in millions of rays per second, higher values are better). The fifth and the sixth column contain speedup of the GPU SBVH over the CPU SBVH and the CPU K-d tree respectively.

of the scene geometry considerably influences the CPU K-d tree performance.

4.3 Comparison of GPU acceleration structures

This section contains the comparison of the implemented GPU SBVH with the GPU BVH and the GPU K-d tree. All the methods utilize the framework for massively parallel hierarchical scene processing described in Section 3.2.

The build process is similar for all three structures: all are constructed in a top-down fashion and all utilize the binning technique during the partition phase. In addition, both the GPU SBVH and the GPU K-d tree require dynamic memory allocation.

Since the build process of the structures is so similar, the results reflect mainly the differences in the acceleration structures themselves; especially their fit for different 3D scene types and sizes (in terms of a primitive count) and the efficiency of their traversal.

4.3.1 Build time comparison

The construction of the GPU SBVH is on average 4× slower (while maximum being 5.9× and minimum 3.2×) than the construction of the GPU BVH on the given scene (Table 4.6).

The build time increase of the GPU SBVH over the GPU BVH is fairly consistent with the one exception of the *Hairball* scene. Even though the number of its primitives is only 31% larger than that of the *Soda Hall* scene, the GPU SBVH build time is 114% slower. This build time increase is caused by the GPU SBVH building twice as big tree as in the case of the *Soda Hall* scene due to extensive reference splitting caused by a high primitive overlap. The GPU BVH on the other hand experiences slowdown proportional to the primitive count increase.

The comparison of the GPU K-d tree and the GPU SBVH reveals that the GPU SBVH build is on average 2.9× slower (the difference ranges from 0.76× to 4.4×). The *Hairball* scene is especially is problematic for the GPU K-d tree, even more so than for the GPU SBVH. The GPU K-d tree built for this scene has six times more nodes than in the case of the *Soda Hall*.

Scene	SBVH	BVH	K-d tree	BVH	K-d tree
Sibenik	0.0326 s	0.0096 s	0.0089 s	3.73×	3.66×
Fairy Forest	0.0639 s	0.0165 s	0.0186 s	3.86×	3.42×
Rotated Sponza	0.1073 s	0.0242 s	0.0515 s	4.42×	2.08×
Sponza	0.0982 s	0.0258 s	0.0257 s	3.80×	3.80×
Conference	0.1107 s	0.0262 s	0.0252 s	4.22×	4.39×
Blade	0.4676 s	0.1472 s	0.1832 s	3.17×	2.55×
Soda Hall	0.6818 s	0.2098 s	0.2402 s	3.24×	2.83×
Hairball	1.5290 s	0.2582 s	1.9930 s	5.92×	0.76×

Table 4.6: A GPU acceleration structures build time comparison (times are listed in seconds, lower is better). The fifth and the sixth column list the increase of the construction time of the SBVH when compared with the GPU BVH and the GPU K-d tree respectively.

4.3.2 SAH cost comparison

The SAH cost of the GPU SBVH (Table 4.7) is in almost all cases the lowest of the three acceleration structures: on average the GPU SBVH cost is 13.4% lower than the cost of the second best structure, GPU BVH (the difference ranges from 46.8% to −1.8%).

The GPU BVH performs poorly especially on the *Rotated Sponza* scene. Since the rotated geometry is hard to partition, the nodes of the GPU BVH have a very high overlap, which increases the SAH cost significantly.

The SAH cost of the GPU K-d tree is in all cases higher than that of the GPU BVH or the GPU SBVH. It is important to note that since the traversal algorithm of the K-d tree is not the same as the BVH one, The SAH cost comparison of the K-d tree and the BVH/SBVH does not represent exact performance differences of the acceleration structures during the image rendering.

4.3.3 Traversal speed comparison

The GPU SBVH performs the best of the three GPU acceleration structures in almost all scene and ray-type combinations (Table 4.8 and 4.9).

Scene	SBVH	BVH	K-d tree	BVH	K-d tree
Sibenik	44.5	55.0	59.0	23.4%	32.5%
Fairy Forest	58.3	57.3	88.2	−1.8%	51.2%
Rotated Sponza	98.0	184.0	348.5	87.7%	255.3%
Sponza	80.3	88.8	104.7	10.6%	30.4%
Conference	68.0	74.5	82.8	9.5%	21.7%
Blade	136.6	136.7	216.2	0.1%	58.2%
Soda Hall	123.2	141.7	144.5	15.0%	17.3%
Hairball	607.7	688.7	1238.6	13.3%	103.0%

Table 4.7: A SAH cost comparison of the GPU acceleration structures. The fifth and the sixth column contain the SAH cost increase of the GPU BVH and the GPU K-d tree over the GPU SBVH.

Compared with the GPU BVH, the GPU SBVH performs on average 57%, 45% and 41% (primary, ambient occlusion, diffusion rays) better.

The GPU BVH slightly outperforms the GPU SBVH on the *Fairy-forest* and *Blade* scenes. Since the geometry of these scenes is very uniform, any spatial splits duplicating the scene' primitives will only add a complexity to the traversal of the SBVH tree (making it larger) while not improving the SAH cost of the structure.

Compared to the GPU K-d tree, GPU SBVH performs on average 147%, 162% and 179% (primary, ambient occlusion, diffusion rays) better, most likely due to worse fit of the K-d tree traversal algorithm for the GPU architecture.

Summary

The GPU SBVH build time is on average 4× to 4.4× slower that that of other GPU acceleration structures. This is balanced out by the better performance during the image rendering: 41% to 57% more rays per second are traced when the GPU SBVH is used instead of the GPU BVH. The speedup is even greater when compared to the GPU K-d tree (Figure 4.2).

Additional measurements using the CUDA profiler show that the CUDA SBVH performs 50% more instructions than the CUDA BVH. That is understandable since the CUDA SBVH computes the spatial

Scene	SBVH	BVH	K-d tree	BVH	K-d tree
Sibenik	375.5	258.8	243.9	45.0%	53.9%
Fairy Forest	237.5	251.8	125.0	−5.6%	89.9%
Rotated Sponza	147.4	50.0	29.4	194.5%	401.2%
Sponza	242.9	139.3	173.3	74.3%	40.1%
Conference	347.8	288.8	139.6	20.4%	149.2%
Blade	371.4	372.6	131.0	−0.3%	183.5%
Soda Hall	504.9	339.9	345.3	48.5%	46.2%
Hairball	61.5	36.0	12.8	70.9%	380.9%

Table 4.8: Comparison of traced primary rays per second (listed in millions of rays per second, higher values are better). The fifth and the sixth column contain speedup of the GPU SBVH over the GPU BVH and the GPU K-d tree respectively.

Scene	SBVH	BVH	K-d tree	BVH	K-d tree
Sibenik	223.7	178.7	141.1	25.1%	58.5%
Fairy Forest	128.0	125.2	63.7	2.2%	100.8%
Rotated Sponza	98.9	41.6	19.3	137.4%	412.1%
Sponza	149.6	100.3	90.1	49.1%	66.1%
Conference	223.8	182.5	92.1	22.5%	142.8%
Blade	121.3	123.6	57.5	−1.8%	111.0%
Soda Hall	317.6	254.8	155.0	24.6%	104.9%
Hairball	37.0	24.1	8.7	53.6%	322.0%

Table 4.9: Comparison of traced diffusion rays per second (listed in millions of rays per second, higher values are better). The fifth and the sixth column contain speedup of the GPU SBVH over the GPU BVH and the GPU K-d tree respectively.

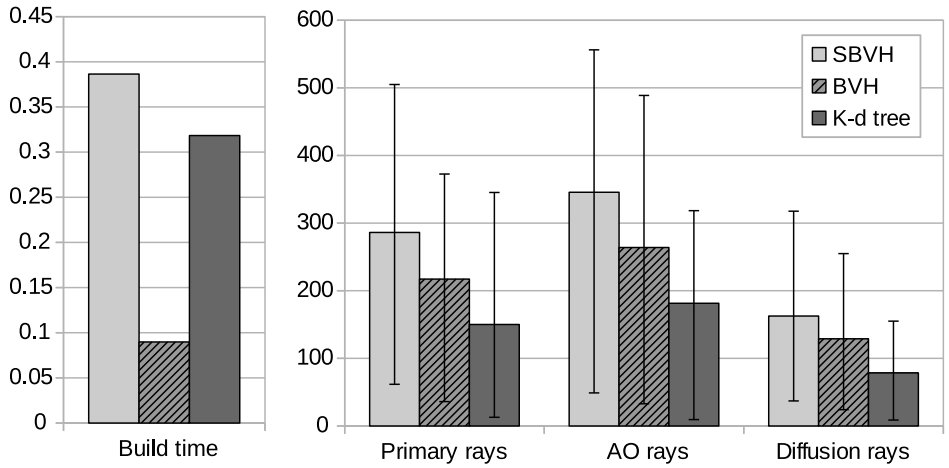

Figure 4.2: An overview of the average build time (the chart on the left, listed in seconds) and the average number of traced rays per second (the chart on the right, maximum and minimum as error bars, values listed in millions of rays per second) for the GPU acceleration structures.

split in addition to the object split present in the CUDA BVH. Moreover, the dynamic memory allocation utilizes computational resources as well.

4.4 GPU allocator performance

The GPU build time measurements (Subsection 4.2.1, 4.3.1) were performed with the CMalloc as the GPU dynamic memory allocator (Section 3.3) for the GPU SBVH as well as for the GPU K-d tree since it proved to be robust and fast. In this section, other GPU dynamic memory allocators (AtomicMalloc, CudaMalloc, ScatterAlloc) are compared in terms of the GPU SBVH build speed.

AtomicMalloc proved to be the fastest GPU allocation algorithm, the build time was on average 8% shorter than in the case of CMalloc. As mentioned in Section 3.3, drawback of AtomicMalloc is its inability to deallocate the memory and thus it is not suitable for scenarios where the GPU memory is limited or large 3D scenes are rendered.

CudaMalloc is part of the CUDA toolkit. Build times of the GPU SBVH when CudaMalloc was utilized were on average 10% longer than in the case of the CMalloc.

ScatterAlloc proved to be the slowest allocator in the measurements, on average 25% slower than CMalloc.

The choice of the GPU dynamic memory allocation algorithm has impact over the GPU SBVH build performance. Therefore, when fine-tuning the build performance, it is important to choose the most suitable one for the given GPU capabilities and the size of the 3D scene.

For small scenes, AtomicMalloc is ideal as it is very fast. When the GPU memory is limited, CMalloc achieves similar performance as AtomicMalloc and allows for a memory deallocation and thus build on larger scenes.

5 Conclusion

The aim of this thesis was to design and implement a GPU variant of the SBVH build method.

The implemented GPU SBVH build method is substantially faster (on average 213×) than its CPU counterpart while maintaining the same quality of the constructed structure. Since the performance of the GPUs in parallel applications (such as the described SBVH build) grows faster than the power of the CPUs, the build speedup is expected to be even greater in the future.

The GPU SBVH build is on average 4.2× slower compared to the GPU BVH and GPU K-d tree already present in the framework. On the other hand, it performs on average 50% better than the GPU BVH and 160% better than the GPU K-d tree during the image rendering.

Usability of the GPU SBVH therefore depends on the build and traversal speed requirements. If the maximum performance during the ray tracing is demanded while maintaining relatively fast build, GPU SBVH is the acceleration structure of choice. On the other hand, when the build time is required to be the lowest possible, the GPU BVH may be more suitable. The algorithm and its implementation described in this thesis is thus well suited for applications such as real-time rendering of large static scenes.

Bibliography

[1] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123973375.

[2] Martin Stich, Heiko Friedrich, and Andreas Dietrich. "Spatial Splits in Bounding Volume Hierarchies". In: *Proc. High-Performance Graphics 2009*. 2009.

[3] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. "Improved Computational Methods for Ray Tracing". In: *ACM Trans. Graph.* 3.1 (Jan. 1984), pp. 52–69. ISSN: 0730-0301. DOI: 10.1145/357332.357335. URL: http://doi.acm.org/10.1145/357332.357335.

[4] Jeffrey Goldsmith and John Salmon. "Automatic Creation of Object Hierarchies for Ray Tracing". In: 7.5 (May 1987), pp. 14–20. ISSN: 0272-1716 (print), 1558-1756 (electronic).

[5] Manfred Ernst and Gunther Greiner. "Early Split Clipping for Bounding Volume Hierarchies". In: *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*. RT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 73–78. ISBN: 978-1-4244-1629-5. DOI: 10.1109/RT.2007.4342593. URL: http://dx.doi.org/10.1109/RT.2007.4342593.

[6] Holger Dammertz and Alexander Keller. "Edge Volume Heuristic - Robust Triangle Subdivision for Improved BVH Performance". In: *Proc. 2008 IEEE/EG Symposium on Interactive Ray Tracing*. 2008, pp. 155–158.

[7] Ingo Wald. "On Fast Construction of SAH-based Bounding Volume Hierarchies". In: *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*. RT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 33–40. ISBN: 978-1-4244-1629-5. DOI: 10.1109/RT.2007.4342588. URL: http://dx.doi.org/10.1109/RT.2007.4342588.

[8] Michael R. Kaplan. "Space-tracing: A constant time ray-tracer". In: *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes*. Addison Wesley, 1985, pp. 149–158.

[9] Ingo Wald and Vlastimil Havran. "On building fast kd-trees for ray tracing, and on doing that in O(N log N)". In: *Proceedings*

of IEEE Symposium on Interactive Ray Tracing 2006. Sept. 2006, pp. 61–69.

[10] Michal Hapala and Vlastimil Havran. "Review: Kd-tree Traversal Algorithms for Ray Tracing". In: *Computer Graphics Forum* 30.1 (2011), pp. 199–213. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2010.01844.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2010.01844.x.

[11] Timo Aila and Samuli Laine. "Understanding the Efficiency of Ray Traversal on GPUs". In: *Proc. High-Performance Graphics 2009*. 2009.

[12] *NTrace GPU ray tracing framework*. URL: http://https://github.com/marekvinkler/NTrace (visited on 04/13/2016).

[13] *NVIDIA CUDA C Programming Guide 7.5*. nVidia Corporation, 2015. ISBN: N/A.

[14] Marek Vinkler et al. "Massively Parallel Hierarchical Scene Processing with Applications in Rendering". In: *Computer Graphics Forum* 32.8 (2013), pp. 13–25. ISSN: 1467-8659. DOI: 10.1111/cgf.12140. URL: http://dx.doi.org/10.1111/cgf.12140.

[15] Marek Vinkler and Vlastimil Havran. "Register Efficient Dynamic Memory Allocator for GPUs". In: *Computer Graphics Forum* 34.8 (2015), pp. 143–154. ISSN: 1467-8659. DOI: 10.1111/cgf.12666. URL: http://dx.doi.org/10.1111/cgf.12666.

[16] Markus Steinberger et al. "ScatterAlloc: Massively parallel dynamic memory allocation for the GPU". In: *Innovative Parallel Computing (InPar), 2012*. May 2012, pp. 1–10. DOI: 10.1109/InPar.2012.6339604.

[17] Sven Widmer et al. "Fast Dynamic Memory Allocator for Massively Parallel Architectures". In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. GPGPU-6. Houston, Texas, USA: ACM, 2013, pp. 120–126. ISBN: 978-1-4503-2017-7. DOI: 10.1145/2458523.2458535. URL: http://doi.acm.org/10.1145/2458523.2458535.

[18] Andrew V. Adinetz and Dirk Pleiter. *Halloc: A high-throughput dynamic memory allocator for GPGPU architectures*. URL: https://github.com/canonizer/halloc (visited on 05/23/2016).

[19] Sven Woop. "A ray tracing hardware architecture for dynamic scenes". PhD thesis. Universität des Saarlandes, 2004.

[20] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123750792, 9780123750792.