Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

# Adaptive Acceleration Techniques for Ray Tracing

*Jakub Hendrich*

A dissertation submitted to
the Faculty of Electrical Engineering, Czech Technical University in Prague,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Ph.D. programme: Electrical Engineering and Information Technology
Branch of study: Information Science and Computer Engineering

*Supervisor: Jiří Bittner*

February 2023

**Thesis Supervisor:**

Doc. Ing. Jiří Bittner, Ph.D.
Department of Computer Graphics and Interaction
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo náměstí 13
121 35 Prague 2
Czech Republic

# Declaration

I hereby declare I have written this doctoral thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis. Moreover, I state that this thesis has neither been submitted nor accepted for any other degree.

In February 2023

............................................
Jakub Hendrich

# Abstract

Ray tracing as a rendering approach consists of two main parts: (1) finding the ray-scene intersections, and (2) evaluating the light contribution at the intersection. In contemporary production scenes, that contain up to billions of scene elements and require a complex shading workflow, both parts are extremely demanding for computing resources, most notably their memory footprint and time-to-image. Therefore, we need to accelerate and improve the algorithm on various levels.

In this thesis, I aim at three approaches to ray tracing acceleration that adapt to the characteristics of an input scene. All of them are based on the *bounding volume hierarchy* (BVH) data structure, which allows organizing the scene content so that the rays find their intersections fast.

The first method improves the topology of an existing BVH so that its ray traversal takes less time. It does so by rearranging the topology, leading to significantly higher quality BVH. It finds its application both for quickly built but low-quality BVHs or for maintaining good quality BVH in a dynamic scene.

The second method builds a data structure of 5D *shafts* on top of an existing BVH. The shafts are used for ray classification; the rays in a shaft then share a common prefix in their BVH traversal. This part of the traversal can be therefore skipped, saving much computational effort.

The third method uses the 5D shafts as containers for the aggregation of similar rays. Such rays have either similar origins and directions or reference the same texture. Once there are many rays in a container, we utilize their similarity to achieve the temporal coherence of their batched processing. Rays with texture coherence can be further used in *out-of-core* scenarios.

The methods were evaluated on a variety of scenes of different sizes (up to 4.3 billion instanced elements) and types (interior, exterior, objects). The proposed methods are able to achieve significant speedup ranging between 10 to 30 percent in build time and/or render time. As they target different parts of the ray tracing process, they may be combined together. This is especially prominent in the two methods that use BVHs amended with shafts, where the simultaneous operation creates a synergy between both approaches.

**Keywords:** ray tracing, object hierarchies, acceleration structures, ray classification, ray batching, ray reordering, ray coherence, cache hierarchy

# Abstrakt

Ray tracing jako metoda vykreslování se skládá ze dvou hlavních částí: (1) nalezení průsečíků paprsků se scénou a (2) vyhodnocení příspěvku světla v průsečíku. V současných produkčních scénách, které obsahují až miliardy elementů scény a vyžadují komplexní výpočet stínování, jsou obě části extrémně náročné na výpočetní zdroje, zejména na velikost paměti a čas potřebný k vytvoření obrazu. Proto je potřeba algoritmus na různých úrovních urychlit a zlepšit.

V této práci se zaměřuji na tři přístupy ke zrychlení ray tracingu, které se přizpůsobují charakteristikám vstupní scény. Všechny jsou založeny na datové struktuře *hierarchie obalových těles* (angl. BVH), která umožňuje organizovat obsah scény tak, aby paprsky rychle nacházely své průsečíky.

První metoda zlepšuje topologii existující BVH tak, že její průchod paprskem trvá kratší dobu. Děje se tak přeskupením její topologie, což vede k výrazně vyšší kvalitě BVH. Své uplatnění najde jak pro rychle postavené, ale nekvalitní BVH nebo pro udržení dobré kvality BVH v dynamické scéně.

Druhá metoda staví datovou strukturu 5D *shaftů* nad existující BVH. Shafty se používají pro klasifikaci paprsků; paprsky v shaftu pak sdílejí společnou úvodní část ve svém průchodu BVH. Tuto část průchodu lze proto přeskočit, což ušetří mnoho výpočetního úsilí.

Třetí metoda využívá 5D shafty jako zásobníky pro shromažďování podobných paprsků. Takové paprsky mají buď podobný počátek a směr, nebo odkazují na stejnou texturu. Jakmile je v zásobníku mnoho paprsků, využijeme jejich podobnosti k dosažení časové koherence jejich dávkového zpracování. Paprsky s koherencí textury lze dále využít v *out-of-core* scénářích.

Metody byly vyhodnoceny na scénách různých velikostí (až 4,3 miliardy instancovaných elementů) a typů (interiér, exteriér, objekty). Navrhované metody jsou schopny dosáhnout významného zrychlení v rozmezí 10 až 30 procent během stavby BVH nebo při samotném vykreslování. Protože se zaměřují na různé části procesu sledování paprsků, lze je kombinovat. To je zvláště výrazné u druhých dvou zmíněných metod, které používají BVH doplněnou o shafty, kde se jejich současné působení navzájem podporuje.

**Klíčová slova:** metoda sledování paprsku, hierarchie objektů, urychlovací struktury, klasifikace paprsků, dávkové zpracování paprsků, přeuspořádání paprsků, koherence paprsků, hierarchie cache

# Acknowledgments

I would like to express my sincere gratitude to my advisor Jiří Bittner for wisely guiding my research, for our fruitful discussions, and patience. Further, I thank my coauthors and brothers in arms, Daniel Meister and Adam Pospíšil, for working together and having fun; and my colleague Vlastimil Havran for many profound insights.

Without the support from Jiří Žára and all the other people from the management and staff of the DCGI, I would not be able to focus on my research. I also thank Paul Navrátil, João Barbosa, Bill Barth, and Greg Abram of TACC for warm acceptance and Jaroslav Křivánek for his attitude.

Special thanks go to my family, who always supported me, and my friends, namely those who encouraged me to pursue Ph.D.: Kristina, Petr, Renata, and Tereza. And I humbly thank Tereza and Veronika for their kindness, which helped me become a better person.

# Research Funding

This research was supported by the following grants:

- Czech Science Foundation (GA18-20374S)
- Grant Agency of the Czech Technical University in Prague (SGS16/237/OHK3/3T/13, SGS19/179/OHK3/3T/13, and SGS22/173/OHK3/3T/13)
- Research Center for Informatics (CZ.02.1.01/0.0/0.0/16_019/0000765)
- Texas Advanced Computing Center through Intel Graphics and Visualization Institutes of XeLLENCE (Intel GVI)

# Contents

# Chapter 1

# Introduction

The goal of photorealistic rendering is to synthesize an image of a scene using predictive methods of global illumination [68, 22]. Ray tracing as an underlying algorithm is a long-studied approach [5, 89, 31] that addresses the problem in an elegant, general way.

The complexity of a realistic scene and the global nature of light transport makes the task very difficult. Despite the constant development of hardware, most notably the advent and huge advances of manycore GPUs [65, 64], it is not viable to render in acceptable time without the support of various acceleration techniques. In fact, scene complexity keeps pace with the hardware and software progress, and rendering such scenes would be far from tractable with only the general approach.

The rendering process is usually severely constrained by the limits of usable computational resources (time, space, energy). In combination with the market demand for rendering more detailed and better-looking images, possibly also maintaining acceptable framerates in interactive rendering scenarios, the need for sophisticated approaches arises. As a result, it uncovers optimization opportunities that use various patterns intrinsic to the rendering process, the data content, low-level layout, and high-level data organization [3].

However, these recurring traits are usually not uniform in time and space; therefore, it is reasonable to look for a processing pipeline flexible enough to adapt to the variability. To exploit the patterns, we need algorithms that are able to follow the trends in the streaming input data, possibly also to reflect on their decisions' effects and thus fine-tune their performance.

The guidelines for designing such algorithms include *coherence*. With coherence in space or memory, we try to aggregate and process similar data at once, i.e., those close to each other; the outcome is efficient computation. Coherent data leverage their *locality* and form a compact working set with a good mapping to the hardware for efficient processing.

A related term, *temporal data locality*, denotes the organization of data processing so that a data element is accessed multiple times before the process moves to another working set. This natural approach is supported in hardware in the form of a CPU cache hierarchy – the closer the data needed for computation are, the faster it proceeds. It then pays off to carefully design algorithms to exploit the runtime architecture well, which always means keeping the process local for as long as possible.

The acceleration data structures that organize scene content include space partitioning schemes (BSP trees, kD-trees, uniform or non-uniform grids), object hierarchies (bounding volume hierarchies, or BVHs), or combinations of both. For grouping of similar rays, they can be enclosed in shafts that induce a partitioning on the 5D space of rays.

The general intent can be characterized as reducing computational demands or complexity on various levels. Specifically, we aim at the following goals:

1. Improve the BVH structure to accelerate the average ray traversal.
2. Decrease the number of ray traversal steps taken in a given BVH.
3. Increase the data coherence and reuse of quickly accessible data.

## 1.1 Overview of Contributions

We propose three algorithms, each of which targets a different aspect of the rendering process. The first algorithm improves the quality of a BVH by rearranging its structure; the other two methods build another geometric structure of shafts on top of a BVH to make ray traversal and general hardware utilization more efficient.

### 1.1.1 Parallel BVH Construction using Progressive Hierarchical Refinement

Different BVH construction methods can be compared with respect to the quality of the output, i.e., the average number of necessary intersection tests of ray traversal, and the build time for a given scene. To guide the BVH construction, heuristics such as SAH are used that estimate the future traversal performance. However, the higher quality of the final BVH usually comes at the cost of a longer build.

Our method focuses on fast rearranging the structure of an existing object hierarchy with the aim of improving the subsequent ray traversal performance. We build a high-quality BVH based on a coarse scene description, where a hierarchy cut is established and propagated toward the leaf nodes. A new strategy for refining the cut significantly reduces the workload of individual steps of BVH construction. Each refinement step produces a new node in the final BVH in a top-down manner. Additionally, we propose a new method for integrating spatial splits into the BVH construction algorithm. The initial (auxiliary) BVH is constructed using a very fast method such as LBVH based on Morton codes.

The trade-off between build time and quality of the output hierarchy can be manipulated easily and the method is also suitable for the rendering of dynamic scenes. We evaluated the method within the Embree ray tracing framework and show that it compares favorably with the Embree BVH builders regarding build time while maintaining comparable ray tracing speed (Fig. 1.1).

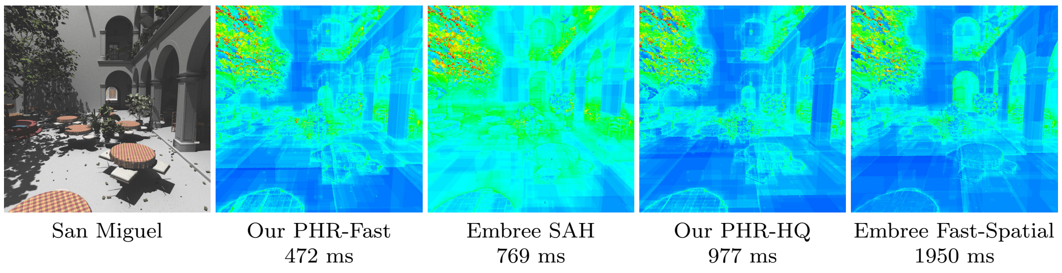


Figure 1.1: The San Miguel scene rendered in the Embree path tracer [87]. Visualization of the number of traversal steps for primary rays using BVHs from different builders (the red color corresponds to 100 traversal steps per ray). The bottom row shows the build times.

Our PHR-Fast method provides 1.6× lower build time than Embree SAH, while the PHR-HQ method has 2× lower build time than Embree Fast-Spatial. In both comparisons, the builders provide equivalent ray tracing performance.

### 1.1.2 Ray Classification for Accelerated BVH Traversal

Traversing a BVH takes up a substantial amount of the total rendering time. However, similar rays follow similar paths through the hierarchy that share many initial traversal steps. By extracting the ray similarity, it is possible to completely skip the shared parts of their traversal, which are normally always taken.

We propose an additional data structure that cuts off large parts of the hierarchical traversal. The idea is to classify rays within 5D space (origin, direction) where each class indexes only the necessary parts of the hierarchical scene representation provided by a bounding volume hierarchy. The precomputed short arrays of indices refer to subtrees inside the hierarchy and the traversal is initiated in them for a given ray class. These shortcuts save many steps compared to the traditional traversal descent from the tree root. This arrangement is compact enough to be cache-friendly, preventing the method from negating its traversal gains by excessive memory traffic.

The method is easy to use with existing renderers which we demonstrate by integrating it into the PBRT renderer. The proposed technique reduces the number of traversal steps by 42% on average, saving around 15% of time in finding ray-scene intersections on average. Fig. 1.2 shows the BVH traversal of an average ray, where the method skips 44% of nodes, both on the path from the BVH root and in the lateral branches.

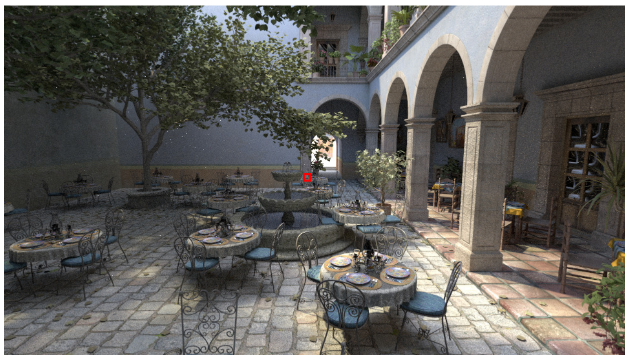
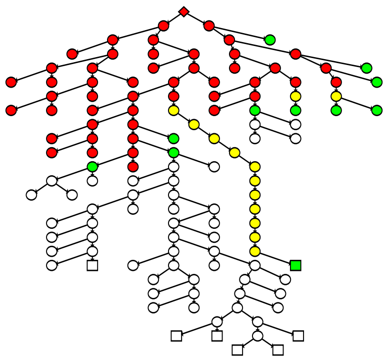



Figure 1.2: The San Miguel scene and a primary ray corresponding to the central pixel (left), the visualization of tracing the ray using the standard BVH traversal and our method (right). The red nodes are visited only by the standard traversal from the root. The green nodes are entry points of our traversal algorithm. The yellow nodes are visited by neither traversal method and only denote the path to candidate list elements. The white nodes are visited by both types of traversal.

### 1.1.3 Dynamic Ray Batching using Shafts

In a traditional ray tracing implementation, the light path connecting light sources with a sensor through any number of bounces is tracked segment by segment. This strategy is elegant and easy to implement; however, it suffers from its global nature, as the lightpath often visits many different regions of the scene. As a result, this approach generates excessive memory traffic by accessing huge amounts of data within a single lightpath propagation by accessing different parts of a BVH, geometry, or textures in subsequent steps.

To avoid unnecessary memory traffic, we can extract the ray coherence via ray classification into shafts (see Fig. 1.3) and defer their processing until we gather sufficiently large groups of coherent rays. In this context, we use the shaft as a basic geometry that contains whole rays; this allows us to store a particular ray only once in its lifetime unlike the previous approaches, which need to repeatedly propagate a ray through an auxiliary structure. The shafts are also organized in an adaptive hierarchy, that provides more detail where necessary while limiting the memory footprint elsewhere, thus saving construction and traversal effort. We show that such a hierarchy can be accessed almost as efficiently as a regular grid.

Being able to aggregate rays into shafts and process them jointly, the coherence in light propagation can be exploited to speed up the rendering. In some scenes, the method saves up to 30% of trace time, while the whole rendering can be faster by 20%.

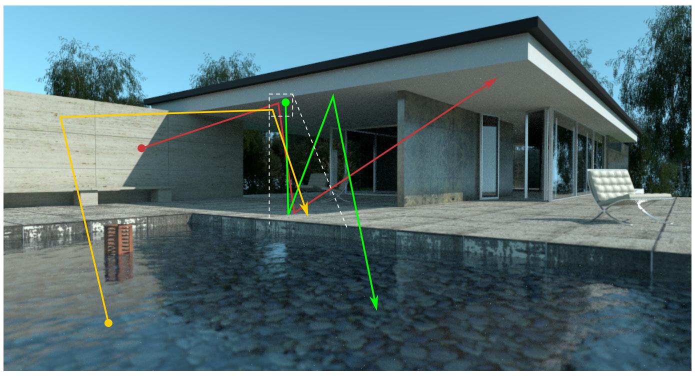


Figure 1.3: The Barcelona Pavilion scene with three independent light paths depicted by the colored line segments. All light paths meet in a single shaft (indicated by the dashed area) with one of their respective segments belonging to the shaft. The three similar rays may be of different types (secondary, shadow) and were generated in different bounces within their respective light paths.

## 1.2 Thesis Structure

The thesis is organized as follows. In Chapter 2, I present the state-of-the-art in the field of rendering related to this thesis's contributions. In Chapters 3, 4, and 5, respectively, the three new methods are described in depth and detail. The last Chapter 6 summarizes how the thesis goals were addressed and outlines future work.

Chapters 3 and 4 are based on papers presented at the Eurographics 2017 [41] and Eurographics Symposium on Rendering 2019 [42] conferences, respectively, and were published in Computer Graphics Forum. Chapter 5 presents recent research that has not been published yet.

# Chapter 2

# State of the Art

In this chapter, I summarize the previous work in the field related to this thesis, starting with introducing the bounding volume hierarchies. The second part of the survey covers BVH construction, the third part discusses ray space and shaft culling, and the last part presents BVH traversal, ray reordering, and coherence.

## 2.1 Bounding Volume Hierarchy

Since the first advancements in ray tracing [89], handling the amount of scenes elements has been of great attention. The linear complexity of naïvely intersecting large sets of rays with each of the scene elements is prohibitive. Rubin and Whitted [72] proposed the *bounding volume hierarchy* (BVH), a data structure that hierarchically organizes the scene elements, leading to an average complexity $O(\log n)$ with $n$ scene elements. The entire scene is represented by the root node of the hierarchy, which then branches into child nodes; the elements of the scene are referenced from the leaf nodes of the hierarchy.

The choice of the bounding volume type is arbitrary. However, in most use cases, we use the *axis aligned bounding boxes* (AABBs). They offer a good trade-off between how tightly is enclosed the AABB's content and the simplicity and performance of the ray-AABB intersection routines, which also allow for hardware implementations.

Recently, a survey on BVHs in the ray tracing context was presented by Meister et al. [59], which focuses both on the basic principles and state of the art methods.

## 2.2 BVH Construction

Bounding volume hierarchies have a long tradition in rendering and collision detection. Kay and Kajiya [49] designed one of the first BVH construction algorithms using spatial median splits. Goldsmith and Salmon [32] proposed the measure currently known as the *surface area heuristic* (SAH), which predicts the efficiency of the hierarchy during the BVH construction. The vast majority of currently used methods for BVH construction use a top-down approach based on SAH. A particularly popular method is the fast approximate SAH evaluation using binning proposed by Havran et al. [40] and Wald [82, 81].

A number of parallel methods for BVH construction have been proposed for both GPUs and multi-core CPUs. Lauterbach et al. [54] proposed a GPU algorithm that uses the Morton code-based primitive sorting (see Fig. 2.1). Hou et al. [43] proposed partial breadth-first search construction ordering to control the GPU memory consumption. Pantaleoni and Luebke [67] proposed the HLBVH algorithm that combines Morton sorting with SAH-based tree construction. Garanzha et al. [30] improved on this method by using SAH for the top part of the constructed BVH.

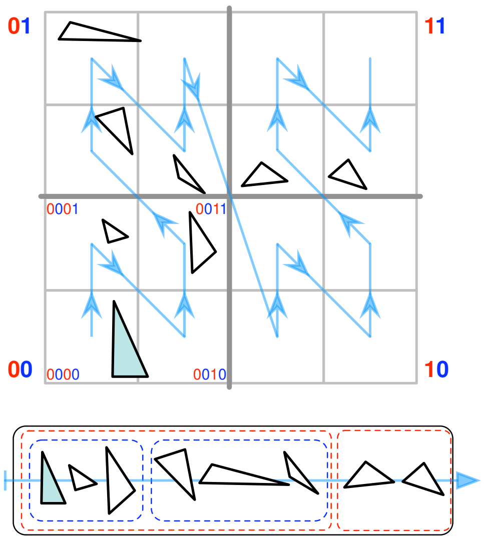


Figure 2.1: Morton ordering of scene triangles (courtesy of Lauterbach et al. [54] and Pantaleoni and Luebke [67]).

Karras [46] and Apetrei [4] proposed GPU-based methods for efficient parallel LBVH construction. These techniques achieve impressive performance but construct a BVH of a lower quality than the SAH-based builders.

Significant interest has been devoted to methods that construct high-quality BVHs albeit at an increased computational time compared to the fastest builders. Walter et al. [88] proposed to use bottom-up agglomerative clustering. Gu et al. [35] used parallel approximative agglomerative clustering for accelerating the bottom-up BVH construction.

Ganestam et al. [26] proposed the Bonsai method, which uses a two-level BVH construction scheme similar to HLBVH and subsequent tree pruning. In their follow-up work, they show how to integrate spatial splits [78] into the Bonsai algorithm [27] (see Fig. 2.2 for an example of spatial splits).

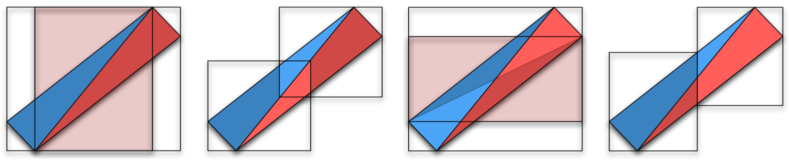


Figure 2.2: Possible splits of two difficult triangles into two bounding boxes with different amounts of mutual overlapping (courtesy of Stich et al. [78]).

Hunt et al. [45] proposed to construct a kD-tree from the scene graph hierarchy. Kensler [50], Bittner et al. [14], and Karras and Aila [47] optimize the BVH by performing topological modifications of the existing tree. These approaches allow for decreasing the expected cost of a BVH beyond the cost achieved by the traditional top-down approach.

Due to the widespread availability of SIMD instructions on today's CPU architectures, it is beneficial to construct multi-BVHs, i.e., hierarchies with a higher branching factor. Wald et al. [83] and Dammertz et al. [19] designed methods for constructing and traversing the multi-BVH, which are particularly important in combination with modern ray tracing frameworks such as Embree [87]. Further improvements of the multi-BVH can be achieved by adapting it to a particular ray distribution as suggested by Gu et al. [34].

## 2.3 Ray Space and Shaft Culling

In their classical paper, Arvo and Kirk [6] proposed the notion of 5D space of rays (three spatial and two directional dimensions) and its subdivision into beams which are assigned only a limited set of candidate objects. A ray is first classified as belonging to one of these beams, followed by intersecting only the relevant candidate objects. Similar ideas gained much attention in other contexts and under different names, where the beams were often referred to as shafts or frusta.

Haines and Wallace [36] came up with the idea of shaft culling. They decrease the time spent on determining the mutual visibility of two surfaces in the radiosity method by generating a list of possible occluders between these surfaces.

Bala et al. [7] use shaft-culling to (1) accelerate shading: to find occluders that may introduce discontinuities in radiance interpolation; (2) accelerate visibility: exploiting temporal coherence by reprojecting already existing radiance samples, assuming they have not become occluded in the current viewpoint.

When constructing local illumination environments, Fernandez et al. [24] determine the blocker list between two objects lazily and iteratively only by sampling the shaft connecting them. This approach may miss some blockers at first, but over time, with more samples taken, it converges to the correct state with the guarantee of maintaining the minimal set of blockers.

Schaufler et al. [74] construct the shafts not to query the space occlusion between two elements but rather behind an occluder with respect to the viewing point. They attempt to perform a fusion of multiple occluders into one, possibly incorporating occluded empty space as well.

Bittner [13] uses shafts to optimize the computation of from-region visibility.

For Brière and Poulin [17], shafts (here *sections*) are the building blocks of a ray-tree structure for quickly determining what has changed in a scene in order to rerender only the necessary parts of it.

Müller et al. [62] used a 5D tree structure to guide the Monte Carlo process into highly contributing path space regions. They use a binary tree spatially partitioning the 3D scene, which references 2D directional partitioning quadtrees from its leaf nodes (see Fig. 2.3).

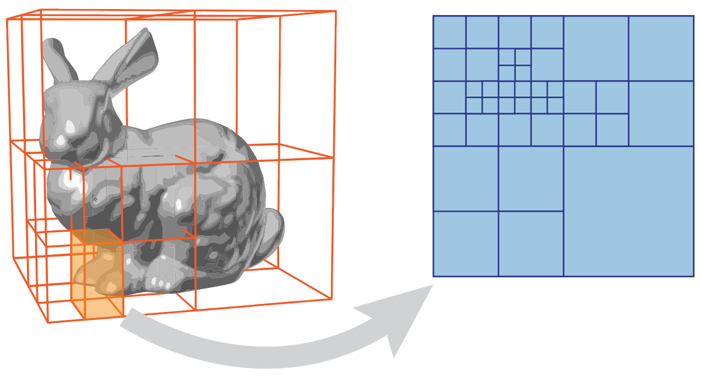


Figure 2.3: SD-tree, a spatial-directional subdivision scheme. Each leaf node of spatial subdivision (left) contains a directional quadtree (courtesy of Müller et al. [62]).

## 2.4 BVH Traversal, Ray Reordering, and Coherence

Havran and Bittner [39] introduced a technique for efficient traversal of rays within a given shaft: the initial parts of traversal trees common to all the rays can be skipped safely. A simpler version of this technique involves only the leaf nodes' traversal.

Havran et al. [38] offer additional means for traversing a hierarchical spatial structure, which they call the ropes (generalized to rope trees). Ropes introduce other paths within the tree to reach a leaf node that do not start from the root (i.e., climbing on a rope from one branch to another), successfully exploiting the coherence of rays.

Dmitriev et al. [20] group rays into pyramidal shafts, which are then traversed simultaneously using SIMD instructions. Contrary to the approach of Haines and Wallace, they use kD-trees and boundary/extremal rays instead of BVHs and plane testing.

Van der Zwaan et al. [93] construct pyramids for coherent rays with the same origin (e.g., primary rays or shadow rays to area light sources) instead of general shafts; then they classify the nodes of a spatial *bintree* as inside or outside a pyramid using a variant of the Cohen-Sutherland clipping algorithm.

Reshetov et al. [70] group the primary and shadow rays into a hierarchy of beams, effectively forming a frustum. This allows starting tracing packets of coherent rays simultaneously from a node deep within a kD-tree representing the whole scene. The depth of the starting node depends on the chosen level within the beam hierarchy – the narrower beam, the deeper the starting node could lie.

The idea of packet-based ray tracing was further refined by many subsequent CPU and GPU-based techniques [15, 63, 29]. Wald et al. [85] apply the frustum traversal to BVHs with the addition of applying it on deformable scenes.

Brière and Poulin [16] enclose light beams within the shafts. As a side effect, they are able to squeeze down the memory used by the shafts drastically while only slightly affecting the rendering times.

Schröder and Drucker [77] build a candidate list for a ray intersection as a union of lists contained in precomputed voxelization of the scene.

Keul et al. [52] precompute visibility in a scene using shaft culling, which is then stored in a line space structure. This data structure then terminates the traversal of a ray through a regular *n-tree* once it is clear that the ray cannot hit the currently traversed subnode content.

Exploiting various forms of coherence is an important and repeatedly studied topic in computer graphics. The particular focus usually depends on the target application. For offline methods exploiting spatial coherence of both the input data and the active working set is crucial, while for real-time methods the temporal coherence often becomes the key factor [75]. Even when focusing just on ray tracing-based methods, there is a plethora of techniques that aim to make use of some form of coherence either by organizing the scene in efficient data structures, designing the ray traversal to maximize the data locality, or performing efficient and more coherent shading.

Teller et al. [79] describe reordering for the radiosity method. They aim to localize the computations of energy propagation among the geometry clusters and study the interaction matrices with different types of orderings, which have a great impact on the overall system performance.

Pharr et al. [69] use a system of three types of grids to handle scenes in out-of-core context. The geometry grids encapsulate and partition single objects, while acceleration grids organize the primitives within the geometry grid cells. A scheduling grid is then used for ray propagation through the scene.

Budge et al. [18] construct a system that is composed of the classical path tracing part supported by a data management layer, that is capable of out-of-core scheduling in mixed CPU/GPU environment (see Fig. 2.4). The scheduler distinguishes several types of computing tasks that are characterized by size, priority, and preferred hardware processing unit. They claim that an application built on top of this generic system scales well with the number and capacity of involved hardware units, including a networking environment with many computing nodes.

In a similar context, Pantaleoni et al. [66] designed an efficient system for occlu-

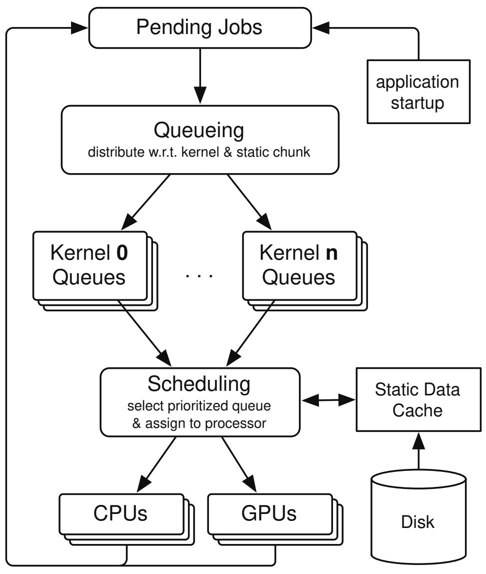


Figure 2.4: Overall architecture of a heterogeneous rendering system with out-of-core capabilities (courtesy of Budge et al. [18]).

sion computation and caching in massive out-of-core scenes.

Bikker [11] investigates the role of data locality in the path tracing context. Using a cache simulator, he shows how the ray traversal in BVH is affected by memory hierarchy with multiple cache levels, which is crucial for rendering performance. The proposed method involves queuing rays in an auxiliary octree structure that partitions the scene; within each octree cell, the geometry primitives are organized in a *mini-BVH*. The queues are periodically processed and new rays are put back into them. In this setup, a ray may have to traverse multiple octree nodes before it hits a surface or exits the scene, as in Fig. 2.5.

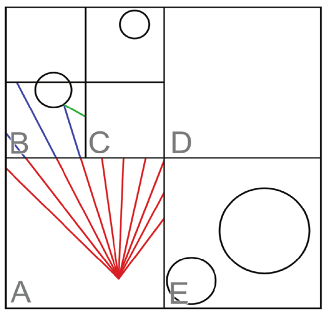


Figure 2.5: Advancing rays through a scheduling octree. Rays that do not hit geometry in a cell are propagated into their neighbor cell if any (courtesy of Bikker [11]).

Yoon and Manocha [91] studied methods for optimized memory layout of spatial data structures with the aim of maximizing the cache efficiency.

# Chapter 3

# Parallel BVH Construction using Progressive Hierarchical Refinement

Bounding volume hierarchies (BVHs) are one of the most efficient spatial data structures for organizing scene primitives. Most contemporary ray tracing implementations use BVHs for accelerating ray scene intersections. This makes the task of constructing high-quality BVH, i.e., a BVH leading to a low number of intersection evaluations thus high ray tracing speed, very important. This problem has been studied very intensively and currently a number of very efficient solutions exist that lead to high-quality BVHs [78, 81, 47, 35, 26]. Some techniques are even applicable for fully dynamic scenes as they can construct the BVH in real-time for moderately complex scenes. As a general rule of thumb, a higher quality BVH means longer build time.

The BVH can be constructed in $O(n \log n)$ time even for the relatively expensive full sweep method based on the surface area heuristic. Several methods were proposed that lead to BVHs with a lower cost (i.e., the expected number of intersections) than that of full sweep SAH [50, 14, 47, 35, 27]. While the SAH cost model correlates with the actual ray tracing performance, Aila et al. [2] identified the supremacy of top-down builders regarding the correlation of the cost and ray tracing performance. In other words, being able to quickly construct a BVH using a top-down algorithm, with its quality comparable to the state-of-the-art BVH optimization methods, boosts the overall rendering performance of ray tracing.

In this chapter, we revisit the idea of Hunt et al. [45], who proposed the build-from-hierarchy method using the scene graph structure to accelerate kD-tree construction. We apply the build-from-hierarchy in a different context: we build a BVH instead of a kD-tree and use a triangle soup instead of a scene graph as the input. Both points bring our method closer to the current state-of-the-art ray tracing frameworks and in turn to the actual usage in practice. It has not been clear whether build-from-hierarchy can compete with other methods when a scene graph (i.e., the input hierarchy) is not available. We show that the method is very competitive even with simple non-SIMD implementation and consequently we believe that our contribution will renew the interest in the build-from-hierarchy techniques in general.

Starting from a triangle soup, we first build an auxiliary BVH using a very fast BVH builder. This gives us quick access to the scene structure, i.e., a coarse description of the spatial distribution of the primitives. At the core of the method, we use a new adaptive method for accessing the auxiliary BVH during the construction of the final one. To further improve the quality of the constructed BVH for ray tracing, we propose a fast way of integrating spatial splits in the BVH construction process. We show that this approach is competitive with the fastest available BVH builders for multi-core architectures, such as AAC [35], Bonsai [26], or Fast-Binning [81]. An excerpt of a comparison with existing methods implemented in the Embree ray tracing framework [87] is shown in Figure 1.1.

The chapter is further structured as follows: Section 3.1 presents an overview of the proposed method, Section 3.2 presents the details of the proposed method, and Section 3.3 gives the results and their discussion.

## 3.1 Efficient Top-Down BVH Construction

### 3.1.1 Handling Large Data

Constructing an efficient BVH is an optimization problem equivalent to hierarchical clustering. The core issue for efficient BVH construction is the large amount of data that has to be organized at the beginning of the computation. A number of successful methods gain speed and efficiency by simplifying the initial phase of the computation through quick partitioning of the data into subsets, which are later processed using a more sophisticated algorithm. This is the case of the HLBVH [30], which uses Morton codes to cluster the data and thus reduces the search space for partitioning. Similarly, the Bonsai algorithm [26] uses simple spatial median partitioning to establish independent data clusters that are processed using more involved partitioning schemes. The AAC algorithm [35] uses a simple top-down subdivision scheme based on Morton codes, which allows for an efficient bottom-up clustering phase.

Another example of handling the initial scene complexity is the build-from-hierarchy method proposed by Hunt et al. [45], which was implemented in their innovative Razor ray tracing system. The method uses explicit knowledge of scene hierarchy to reduce the complexity of kD-tree construction significantly.

We follow the path visible in the above-discussed techniques with the aim of providing a highly scalable and efficient BVH construction algorithm. Similarly to HLBVH, Bonsai, or AAC, we use Morton code-based spatial sorting to establish the initial scene partitioning. Following Hunt et al. [45], we use the hierarchy resulting from the initial partitioning to construct the final BVH. We make substantial modifications to this phase of the algorithm that aim to better distribute the workload among different levels of the hierarchy and to efficiently perform spatial splits during the BVH construction. Despite the increased quality of the final BVH, we keep the second phase of the algorithm at $O(n)$ complexity by bounding the working set when creating each interior node, similarly to Hunt et al. [45].

### 3.1.2 Algorithm Overview

Our algorithm starts by constructing an auxiliary BVH that provides a scalable description of the scene structure. We use the LBVH algorithm [54] implemented on multi-core CPU architecture. The auxiliary BVH serves as a valuable hierarchical representation of the structure and distribution of objects. By establishing and progressively refining the cuts of the auxiliary BVH, we can efficiently construct the final BVH by a thorough analysis and partitioning of these cuts (see Figure 3.1).

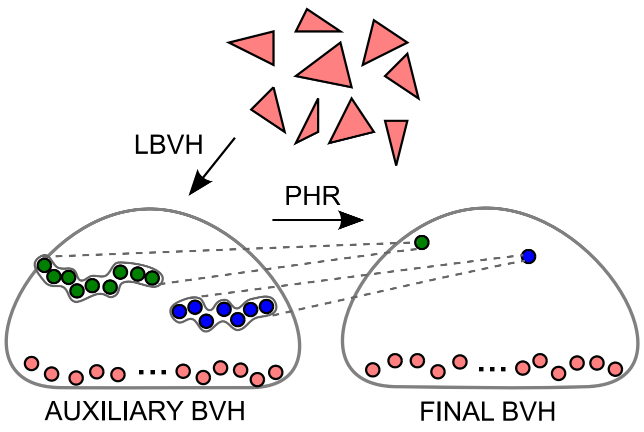


Figure 3.1: Overview of the method. Fast LBVH builder constructs the auxiliary BVH, which is used to build the final high-quality BVH using progressive hierarchical refinement (PHR).

The construction of the final BVH starts by finding the initial cut of the auxiliary BVH. This cut is formed as the smallest set of nodes that spans the BVH horizontally and the nodes have their surface smaller than a particular threshold. At each step of the algorithm, the nodes on this cut are examined and we search for the best partitioning of these nodes into two sets. We use full sweep SAH in all three axes to find the partitioning. As the size of the cut is small compared to the scene size, this search is very fast. Then the cut is partitioned using the best split found and two new cuts are formed. These cuts are refined according to a new threshold corresponding to the reached subdivision depth. Should a node on the cut be refined, we just replace the node in the cut with its children. The algorithm then continues by applying the method recursively in the new subtrees with the corresponding refined cuts. The illustration of the main steps performed on the auxiliary BVH cut, when building a node in the final BVH, is shown in Figure 3.2.

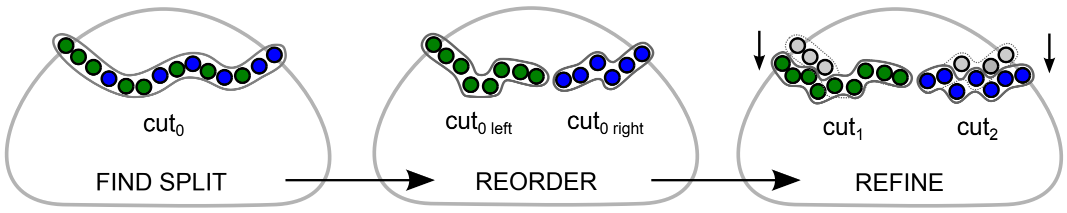


Figure 3.2: Illustration of the hierarchical cut refinement. (left) When computing the split, we determine the optimal subdivision of the current cut (cut$_0$) based on full sweep SAH. That results in two sets of nodes on the current cut which correspond to the nodes that will be contained in the left and right subtree of the subdivided cut (shown in green and blue). (center) The nodes on the cut are reordered and split into two disjoint cuts. (right) The new cuts (cut$_1$ and cut$_2$) are formed by hierarchical refinement based on their surface area.

## 3.2 Progressive Hierarchical Refinement

This section describes the proposed method in detail by presenting and discussing its individual components. For now, let us assume that we have an auxiliary BVH at our disposal. The details of constructing the auxiliary BVH will be given at the end of this section.

### 3.2.1 Finding the Initial Cut

Our method starts with identifying the nodes of the auxiliary BVH that will form a cut of the hierarchy. This cut contains nodes that are just below a given threshold on their surface area. We will discuss how to determine this threshold in Section 3.2.3. The threshold is set in a way that it provides an initial cut with its size below a given hard bound (e.g., 2048 nodes); which is, in general, several orders of magnitude lower than the total number of scene primitives.

The algorithm for finding the cut uses a priority queue and a simple test that checks whether the visited node has a surface area larger than a threshold. If this is the case, its children are put into the priority queue. If this is not the case or the node is a leaf it is appended to the cut. When there are no more nodes to visit or the cut together with the unprocessed nodes has reached the maximum cut size, we terminate the cut search.

### 3.2.2 Splitting the Cut

The core of the method is splitting the current BVH cut into two disjoint sets. As the cut is small, we can use a relatively expensive evaluation of the best split without a significant performance penalty. Thus, we use full-sweep SAH in all three axes to subdivide the current cut. We sort the cut along all three axes based on node centroids. Then for all axes, we perform a sweep that computes the cost of the node subsets on the left and right of the sweep plane. For a given position $i$ of the sweep plane, the cost is given as

$$C(i) = S_L(i)n_L(i) + S_R(i)n_R(i), \tag{3.1}$$

where $S_L(i)$ and $S_R(i)$ are the surface areas of the bounding boxes of left and right subsets and $n_L(i)$ and $n_R(i)$ are the numbers of nodes on the left and on the right from the sweep plane, respectively. Then we select the axis and the position of the sweep plane that yield the minimum cost.

There is a minor difference from the traditional SAH cost evaluation: we use the numbers of nodes $n_L(i)$ and $n_R(i)$ instead of the numbers of triangles in the subtree [45]. We discovered that tracking the numbers of nodes provides slightly better results for most tested scenes (in a range of a few percent). We expect that this is because the numbers of nodes better reflect the complexity of hierarchically organizing the corresponding scene part than the raw triangle numbers. Using the numbers of triangles yields a cost corresponding to the node being a leaf, which is obviously not the case for nodes higher in the hierarchy. The node counts, on the contrary, are somewhere between the actual triangle counts and the final (but yet unknown) costs of the subtrees being constructed. Moreover, the numbers of nodes $n_R(i)$ and $n_L(i)$ for current $i$ are readily available values, whereas the triangle counts rely on storing them as additional information for each node of the auxiliary BVH.

### 3.2.3 Refining the Cut

By subdividing the BVH cut into two parts, the number of nodes in each of the two new cuts is reduced. Repeating this step several times would lead to a cut size of 1, and thus we would lose all the information about the scene structure that the cut can provide. Hence we refine the cut to maintain its properties. Hunt et al. [45] suggested keeping a constant size of the cut that is specified by the user (they used the cut size 500 for the results in the paper). While this is a valid approach that maintains the linear complexity of the algorithm [45], this technique becomes expensive, especially for medium to high BVH depths, where the algorithm has to cope with a large number of cuts with sizes comparable to the number of primitives in their subtree.

Instead, we propose an adaptive way of refining the cut that aims to progressively reduce the cut size, and thus to better balance the computational effort among different levels of the BVH. This approach is similar to adapting the number of clusters handled by the AAC algorithm [35] in their bottom-up BVH construction.

Our method is based on thresholding the surface area of nodes forming the cut while taking into account the current depth. The cut is refined as follows: for each node forming the cut, we compare its surface area with the threshold. If the surface area of the node is greater than the threshold, the node is replaced by its children. Otherwise, the node is kept in the cut. When refining the cut, we intentionally descend just one level in the tree. This simplifies the implementation as no stack is needed to refine the cut.

We propose to use an adaptive threshold that is based on the depth of the currently computed node in the BVH. The threshold is computed as

$$t_d = \frac{S}{2^{\alpha d + \delta}}, \tag{3.2}$$

where $S$ is the surface area of the scene bounding box, $d$ is the current depth, $\alpha$ and $\delta$ are parameters of the method. The $\delta$ parameter determines the size of the initial cut for $d = 0$. The $\alpha$ parameter describes how quickly will the cut size shrink with increasing depth. The setting of these parameters depends on the desired trade-off between the build time and the trace speed. We used two settings of these parameters: $\alpha = 0.5$ and $\delta = 6$ for fast builds (PHR-Fast) and $\alpha = 0.55$ and $\delta = 9$ for high-quality builds (PHR-HQ).

Note that the formula is inspired by the regular subdivision of the scene into non-overlapping voxels. In this case, the size of the voxel in depth $d$ is given as $S_d \approx \frac{S}{2^{\frac{2}{3}d}}$. The bounding boxes of the input hierarchy do not follow this ideal subdivision case, but keeping the working set of nodes in the cut with areas staying close to this function proved beneficial for the performance vs. quality trade-off of the algorithm.

An example of the lengths of the cut obtained using the proposed adaptive threshold function is given in Figure 3.3. As we show and discuss in Section 3.3, the adaptive threshold leads to consistently better results than a predefined cut size originally proposed by Hunt et al. [45].

### 3.2.4 Spatial Splits

Spatial splits proposed by Stich et al. [78] can improve the BVH quality significantly. However, their evaluation is relatively expensive. We make use of the availability of the limited-size BVH cut and propose a novel technique for the evaluation of spatial splits.

When splitting a cut, we also evaluate its result when applying spatial splits on the cut nodes. This contrasts with the traditional spatial splits evaluation which is performed directly on geometric primitives (triangles).

The actual spatial split evaluation is done using an algorithm similar to the kD-tree construction. We sort the boundaries of the nodes' bounding boxes and perform a plane sweep while evaluating the split cost. Note, however, that in this case, the classification of the node into the left and right subtree is not based on the centroid of the node but on the two extents of the node bounding box in the given axis. A node is classified as lying in both left and right subtrees if the sweep plane intersects its bounding box. For each cost evaluation, we clip the left and right bounding

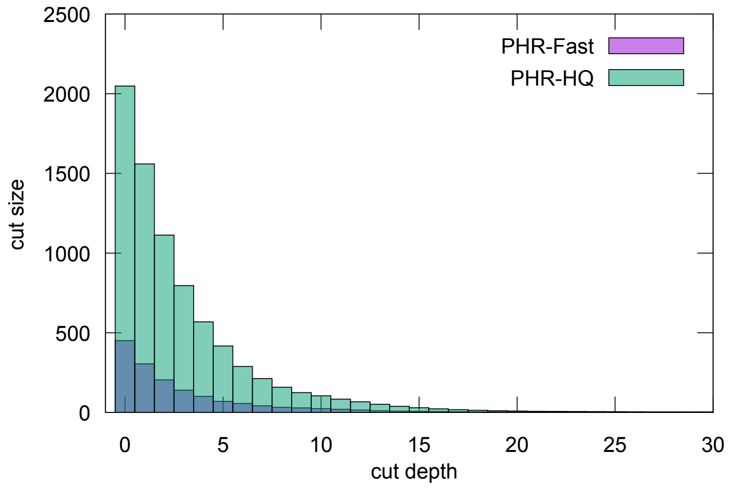


Figure 3.3: The average cut size at different BVH depths for the San Miguel scene. PHR-HQ (green) works on substantially larger cuts than PHR-Fast (purple). The adaptive threshold function implies a larger cut size at the top of the tree, which has a crucial impact on the overall tree quality.

boxes by the sweep plane which potentially reduces the respective areas ($S_L$ and $S_R$). Thus, we obtain

$$C_S(i) = S_L^c(i) n_L^*(i) + S_R^c(i) n_R^*(i), \tag{3.3}$$

where $S_L^c(i)$ and $S_R^c(i)$ are surface areas of clipped bounding volumes of nodes intersecting the left and right volumes defined by the sweep plane, $n_L^*(i)$ and $n_R^*(i)$ are the numbers of nodes intersecting these left and right volumes.

If the spatial split cost $C_S$ is lower than the cost $C$, we perform a spatial split. We actually do not subdivide any bounding boxes or primitives themselves. Instead, we just clip the two bounding boxes for the left and right subsets using the split plane. The clipped bounding box is always passed over with the current cut to apply clipping also in the deeper levels of the tree. More precisely, when the cost function is evaluated deeper in the tree, the bounding boxes of the nodes being processed are always clipped by the clipping bounding box for the given cut that was passed from the parent nodes.

If spatial splits are used, it can happen that some nodes on the refined cut do not intersect the clipped bounding box passed over with the current cut. These nodes are simply skipped, i.e., they are not placed into the refined cut. An illustration of a spatial split applied on the current cut is shown in Figure 3.4.

To avoid performing spatial splits at lower parts of the tree where their benefit is usually low, we use a simple rule proposed by Stich et al. [78]. Spatial splits are performed only when the ratio of the surface area of the currently constructed node and the bounding box of the scene is above a given threshold (e.g., 1e-3).

The proposed spatial splits method is very simple and requires minimal modifications of the other parts of the code. The results show that this method, although very fast, improves the trace performance by up to 20%.

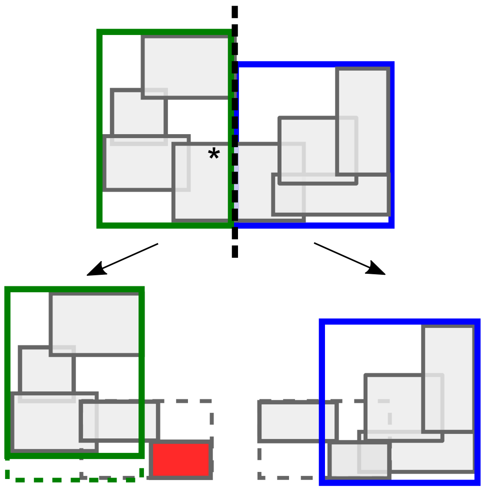


Figure 3.4: Illustration of a spatial split. A spatial split is selected for nodes forming the current BVH cut. Two subsets of nodes are formed for the left and right subtree, and the bounding volumes of these subsets are clipped by the splitting plane. The node denoted by the * symbol is split by the split plane and therefore it is placed in both subsets. Left and right subsets are refined to form new cuts. Particularly, the node * is replaced by its children (see the images at the bottom). Note that for the left subset, one of the child nodes (shown in red) does not intersect the currently clipped bounding volume, and thus it is discarded from the cut in this branch. The clipped bounding box can thus be shrunk to reflect this.

### 3.2.5 Multi-BVH

The trace performance of the BVH can be improved by increasing the branching factor of interior nodes, which results in a shallower BVH [83, 19]. This is particularly important for SIMD-optimized ray tracers. We construct the multi-BVH by postponing the formation of interior nodes in the final BVH until $n$ children are available or no child interior node exists. If the current number of child nodes is lower than $n$, the algorithm processes the largest available interior child node first [83].

### 3.2.6 Parallelization

The parallelization of the method is relatively straightforward. We use a shared work queue that contains entries representing unconstructed parts of the tree. Each entry contains an index of a node in the new BVH and an array representing the corresponding cut in the auxiliary BVH. A thread pops an entry from this queue, finds a subdivision of the cut and if the node does not become a leaf, it stores the right child in the shared queue so that other threads can fetch this entry and process it. The left branch is processed immediately by the same thread unless it is a leaf node. Note that to prevent large synchronization overhead, we only store the nodes in the shared work queue up to a given depth (e.g., 12) or when we find out that some threads are idle (using a counter of active threads).

### 3.2.7 Constructing the Auxiliary BVH

We use parallel sorting based on Morton codes and subsequent binary search to identify the nodes of the auxiliary BVH. We first compute Morton codes for all triangles. For that, each thread is assigned a continuous span of $\lceil n/t \rceil$ triangles, where $n$ is the number of scene triangles and $t$ is the number of threads. For parallel sorting, we use approximate parallel bucket sort. In each thread, we insert the primitives into $k$ buckets. We used $k = 2^{12}$ that corresponds to sorting according to the highest 12 bits of the Morton code. Then the buckets are merged in parallel by evaluating $\lceil k/t \rceil$ buckets per thread. The auxiliary BVH is constructed using a shared work queue. Each thread fetches a node from the queue and finds the intervals corresponding to its children by identifying an entry with a change in the highest bit of the Morton code within the current interval [54]. Finally, bounding boxes are updated by a parallel recursive procedure using synchronized access to the node's data. We use an atomic counter for each node: the thread that visits the node as the second node will update the node's bounding box using the bounding boxes of its children.

## 3.3 Results

We implemented the proposed method in C++ using standard language constructs. In the current implementation, we did not exploit SIMD instructions. We performed a series of tests comparing the build times, BVH costs, and the ray tracing performance of our method and several reference methods on nine test scenes. As a reference, we used our implementations of the method of Hunt et al. [45] (denoted Hunt-50, Hunt-500) and the AAC method of Gu et al. [35] (AAC). We also evaluated the methods available in the latest version of Embree ray tracer, i.e., Embree 2.11 (Embree SAH, Embree Fast-Spatial, Embree Morton). The Hunt-50 and Hunt-500 methods used a fixed length cut of the length of 50 and 500, respectively. The cut of a given length is established by partial sort and subsequent refinement of the largest nodes, which proved more efficient than using a priority queue. We did not use the fast-scan and lazy-build also proposed by Hunt et al. [45] as these are specific to kD-tree construction and the Razor system. For the AAC method, we used parameters corresponding to AAC-Fast settings [35]. For all methods, we used a 4-ary multi-BVH [83, 19].

The trace times were evaluated using a simple path tracer provided as a tutorial in Embree. The times are computed as average times for three different representative views of each scene using 1024×1024 resolution and 1 sample per pixel. The measurements were performed using 16 working threads on a PC equipped with 2×Intel Xeon E5-2620. The measured results are summarized in Table 3.1. Below we discuss the results from different points of view.

### 3.3.1 Construction Speed

The table shows that the lowest build times among the reference methods are achieved by the Embree Morton method, usually followed by Hunt-50, AAC, Embree SAH, Hunt-500, and Embree Fast-Spatial. Our PHR-Fast method provides build times usually between Embree Morton and Hunt-50; thus, it is the second fastest builder tested. The PHR-HQ method has build times mostly between Embree SAH and Hunt-500. The PHR-HQ build times are about twice lower as the build times of Embree Fast-Spatial.

On scenes with a simple structure (e.g., Buddha), the PHR-Fast and PHR-HQ methods perform comparably to Hunt-50 and Hunt-500, respectively. On larger scenes, PHR-Fast and PHR-HQ achieve about 20% lower build times than Hunt-50 and Hunt-500 while leading to slightly higher quality BVHs.

### 3.3.2 BVH Quality and Ray Tracing Performance

The overall highest BVH quality in terms of trace times was achieved by the Embree Fast-Spatial method. The PHR-HQ method closely follows in half of the test scenes. In two tested scenes (Hairball and Soda Hall), PHR-HQ actually provided marginally better trace performance. In four test scenes, PHR-HQ provides about 10%-20% lower trace performance than Embree Fast-Spatial.

The PHR-Fast method provides trace performance usually between Embree Morton and Embree SAH while being closer to Embree SAH in terms of trace speed. The build times on the other hand are closer to Embree Morton, which makes the PHR-Fast method a good candidate for interactive applications.

We provide a graphical comparison of build time vs. trace time for all tested methods in Figure 3.5. The figure also sketches the *Pareto front* known from multi-objective optimization [73] that indicates the methods which will perform superior to the others for some combination of build time vs. trace time, i.e., the number of rays traced. According to the measurements, the Pareto front is formed by Embree Morton, PHR-Fast, Embree SAH, PHR-HQ, and Embree Fast-Spatial methods.

### 3.3.3 Influence of Spatial Splits

Spatial splits provide trace speedup particularly for large architectural scenes with primitives of different sizes. The influence of spatial splits can be seen by relating to the Hunt-50 and Hunt-500 methods, which do not employ spatial splits. The trace

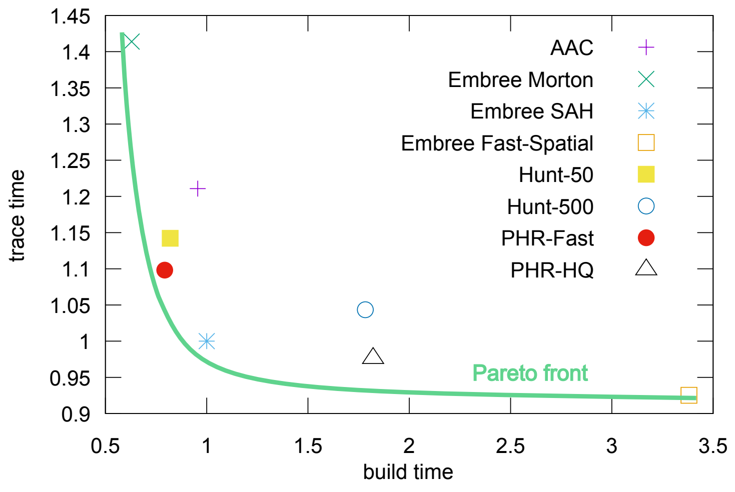


Figure 3.5: The relative performance of different methods with respect to Embree SAH as the reference method and averaged over all test scenes.

speedup due to our fast spatial splits method is in the range of 2% and 20%. Spatial splits had the largest positive impact in the San Miguel scene.

Our node-level spatial splits could also be combined with triangle presplitting [47]. We have not yet evaluated this possibility, but we consider it an interesting topic for future work.

### 3.3.4 Parameters

Prior to determining the settings for PHR-Fast and PHR-HQ, we conducted a series of measurements to evaluate the dependence of the method on the $\alpha$ and $\delta$ parameters. We tested values of $\alpha$ slightly below $\frac{2}{3}$ (an explanation of this threshold is given at the end of Section 3.2.3). We determined the range of measured $\delta$ values by balancing the amount of work done at each subdivision step with the potential of finding a precise enough split while focusing on the splits near the root. In particular, we tested all the combinations of $\alpha$ and $\delta$ from the sets $\{0.45, 0.5, 0.55, 0.65\}$ and $\{5, 6, 7, 8, 9, 10\}$, respectively. From these measurements, we selected the two characteristic settings (PHR-Fast, PHR-HQ) that roughly defined the Pareto front for all scenes. In a selected subset of scenes, the representative parameter settings can be slightly different, and thus better performance/quality ratios could be achieved. For example, in San Miguel and Power Plant scenes, the optimal values are $\alpha = 0.5$, $\delta = 7$ for the high-quality scenario (saving about 35% build time while actually decreasing the trace time by about 3%). In Conference and Hairball scenes, it is $\alpha = 0.5$, $\delta = 5$ for the fast build scenario (saving 9% and 33% build time, respectively, while keeping the same trace time).

### 3.3.5 Using PHR with Other Builders

We have tested the proposed method with other BVH builders for constructing the initial BVH (AAC and Binning-SAH); the results were generally worse concerning Pareto optimality. For example, using AAC allowed us to build a BVH with a slightly lower cost (a few percent) for the same parameter settings of PHR; using different PHR parameters, a similar quality BVH could be constructed faster.

### 3.3.6 Discussion and Limitations

The results indicate that the method is configurable in a large range of build time vs. BVH quality trade-off. On one side, this is a positive feature; on the other hand, this makes it difficult to find optimal parameters ($\alpha$ and $\delta$) for the PHR method. We used two representative settings, but we observed that in a number of scenes setting different values to these parameters would provide better build times or trace performance while not significantly altering the other value.

The PHR-Fast method seems a good fit for interactive applications. In fact, we could avoid constructing the auxiliary BVH from scratch and reuse it over multiple frames by simple refitting to further reduce the build time of PHR-Fast. This strategy was already suggested by Hunt et al. [45], but its verification in practice remains an open topic.

The current implementation of the method uses a moderately optimized C++ code, but we did not yet exploit SIMD constructs. We expect that by exploiting SIMD we could further lower the build times while keeping the same BVH quality. On a similar note, the sweep SAH algorithm used at the core of our method could be replaced by SIMD-optimized binning SAH.

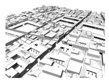
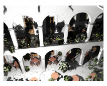
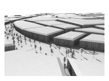
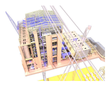
| | build time [ms] | SAH cost [-] | frame time [ms] | trace speed [MRps] | build time [ms] | SAH cost [-] | frame time [ms] | trace speed [MRps] | build time [ms] | SAH cost [-] | frame time [ms] | trace speed [MRps] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Conference #triangles 331k | | | | Happy Buddha #triangles 1087k | | | | Soda Hall #triangles 2169k | |
| Embree Morton | 18 | 23 | 663 | 27 | 70 | 33 | 60 | 30 | 111 | 39 | 62 | 34 |
| Hunt-50 | 20 | 18 | 549 | 32 | 99 | 28 | 57 | 31 | 143 | 33 | 55 | 39 |
| AAC | 21 | 15 | 499 | 36 | 109 | 31 | 60 | 30 | 177 | 31 | 56 | 38 |
| Embree SAH | 31 | 15 | 508 | 35 | 101 | 26 | 54 | 33 | 184 | 29 | 50 | 43 |
| Hunt-500 | 42 | 15 | 508 | 35 | 202 | 27 | 56 | 32 | 322 | 30 | 51 | 42 |
| Embree Fast-Spatial | 81 | 14 | 521 | 34 | 377 | 26 | 54 | 33 | 586 | 26 | 48 | 45 |
| PHR-Fast | 23 | 15 | 530 | 34 | 88 | 29 | 58 | 31 | 134 | 27 | 49 | 44 |
| PHR-HQ | 39 | 15 | 509 | 35 | 222 | 26 | 54 | 33 | 259 | 25 | 46 | 46 |
| | | | Hairball #triangles 2880k | | | | Crown #triangles 4868k | |  | | Pompeii #triangles 5632k | |
| Embree Morton | 165 | 216 | 1027 | 7 | 247 | 12 | 313 | 15 | 289 | 67 | 224 | 14 |
| Hunt-50 | 246 | 171 | 735 | 9 | 331 | 11 | 292 | 16 | 391 | 36 | 156 | 20 |
| AAC | 278 | 190 | 864 | 8 | 380 | 12 | 316 | 15 | 458 | 44 | 177 | 18 |
| Embree SAH | 225 | 164 | 695 | 10 | 411 | 10 | 253 | 18 | 453 | 30 | 138 | 23 |
| Hunt-500 | 518 | 160 | 692 | 10 | 761 | 11 | 269 | 17 | 873 | 31 | 141 | 22 |
| Embree Fast-Spatial | 1607 | 157 | 715 | 10 | 1163 | 10 | 243 | 19 | 1442 | 20 | 108 | 29 |
| PHR-Fast | 341 | 166 | 724 | 10 | 262 | 12 | 304 | 15 | 321 | 35 | 156 | 20 |
| PHR-HQ | 1064 | 159 | 680 | 10 | 589 | 11 | 273 | 17 | 631 | 28 | 133 | 23 |
| |  | | San Miguel #triangles 7880k | |  | | Vienna #triangles 8637k | |  | | Powerplant #triangles 12759k | |
| Embree Morton | 430 | 39 | 1563 | 8 | 506 | 46 | 203 | 17 | 620 | 20 | 227 | 15 |
| Hunt-50 | 512 | 30 | 1244 | 10 | 707 | 25 | 150 | 23 | 674 | 17 | 184 | 18 |
| AAC | 606 | 26 | 1221 | 10 | 713 | 29 | 168 | 20 | 1090 | 16 | 196 | 16 |
| Embree SAH | 769 | 25 | 1060 | 12 | 722 | 19 | 120 | 28 | 1166 | 13 | 146 | 22 |
| Hunt-500 | 1112 | 27 | 1136 | 11 | 1491 | 20 | 133 | 26 | 1560 | 14 | 155 | 21 |
| Embree Fast-Spatial | 1950 | 22 | 879 | 14 | 1687 | 15 | 115 | 29 | 3312 | 10 | 114 | 28 |
| PHR-Fast | 472 | 23 | 1050 | 12 | 534 | 24 | 155 | 22 | 652 | 15 | 165 | 20 |
| PHR-HQ | 977 | 21 | 914 | 14 | 1089 | 17 | 123 | 28 | 1388 | 12 | 140 | 23 |

Table 3.1: Performance comparison of the tested methods. The reported numbers are averaged over three different viewpoints for each scene. For computing the SAH cost, we used $c_T = 1$ and $c_I = 1$. The leaf cost is computed using the Embree cost model, i.e., triangle quartets are considered a single primitive.

Chapter 4

# Ray Classification for Accelerated BVH Traversal

Bounding volume hierarchies (BVHs) provide an efficient means of organizing the spatial distribution of a 3D scene. In the context of ray tracing, they are used to determine quickly which parts of the scene are possibly hit by a ray. Many efforts have pushed the performance of BVHs forward [78, 81, 47, 35, 26], which offer fast build/update/query operations, low memory demands, and ease of implementation. Some techniques are tailored for specific scenarios, such as rendering dynamic scenes [84, 55, 85, 67].

Most efficient BVH methods use the Surface Area Heuristic (SAH) [32] during the BVH construction to determine efficient hierarchical object partitioning. This heuristic is however only a static guidance that does not take into account the nature of subsequent ray queries. The question is how to leverage the known ray's properties for faster traversal through a BVH.

Depending on the scene structure, the expected ray distribution, and other possible influences taken into account, the hierarchy is about $\log n$ deep on average, which is also proportional to the number of traversal steps needed to descend from the tree root (representing the whole scene) to the leaves, where the scene geometry is referenced. To find the intersection, the traversal typically visits multiple leaf nodes and thus it branches into several subpaths. There are millions of elements in contemporary scenes, resulting in trees where the distance from the root to the leaves is in the order of several tens of nodes. Combined with millions of rays cast, this takes a considerable amount of time spent by the traversal, so any optimization here has the potential of making a significant impact on the total rendering time.

In this chapter, we use the concept of *frustum shafts* as a tool to quickly determine which parts of the scene geometry can be potentially intersected by a given ray. While a BVH organizes scene regions hierarchically into AABBs, shafts are topologically similar to rays, so the volume of a shaft contains those scene parts which are relevant for the corresponding rays. For each frustum shaft, we precompute a short list of BVH nodes that are used as entry points for the ray traversal. We describe the necessary criteria for computing these lists so that the ray traversal skips large parts of the root–leaf traversal sequences, thus saving a lot of traversal steps (see Figure 1.2).

Our method builds on the ray classification idea that has a long history [6]. However, until now the technique has been considered impractical for contemporary rendering techniques. We propose a new algorithm that connects ray classification (direct lookup) with contemporary BVHs (hierarchical traversal), demonstrating that an efficient combination is possible.

Our method takes advantage of the classical ideas of ray classification [6, 62], shaft culling [36, 70], and scene structuring using BVHs [32, 85, 26]. This combination offers substantial performance gain in ray traversal while providing control over the memory consumption as well as seamless integration of the method into existing ray tracing implementations.

## 4.1 Efficient BVH Traversal

In this section, we discuss in detail how the standard traversal of a BVH progresses down the tree, and identify its weak spots. Then we briefly introduce the idea of culling the scene geometry with shafts, which is able to cut the traversal costs substantially.

### 4.1.1 Standard Traversal

Having a spatial index built above the geometry of a scene, we need to traverse it to determine the nearest intersection for primary or secondary rays, or any intersection for shadow rays, which test visibility between a surface and a light source. This is usually done by descending the tree. We start with pushing the root node id onto the stack; the topmost stack entry is then popped repeatedly to be processed further. If it represents a leaf node, the referenced geometry is checked for an intersection; for inner nodes, the bounding boxes of its children are checked for an intersection. All the children of an inner node that have been hit by the ray (if any) are pushed back onto the stack, usually after depth-sorting them along the ray for efficiency. The running closest intersection with geometry is maintained for primary and secondary rays throughout the traversal, effectively pruning the search tree even more.

In scenes with large and complex geometry, the path length from the root to the geometry elements amounts to tens of traversal steps. The actual traversal is even more complex as it visits many lateral branches, which in most cases do not advance the results of the search. This is because the hierarchical index is a general spatial structure with no information about the properties of the rays being traced in the scene; it does not take into account the restricted part of the scene only relevant to a given ray.

### 4.1.2 Algorithm Overview

Our method assumes there is already a BVH built which organizes the scene geometry. We construct a set of convex *frustum shafts*, which extend from a scene voxel in a (narrow) interval of directions. The voxels are elements of a 3D regular grid and the direction intervals are constructed on a regular 2D grid subdividing the space of directions. For each shaft, we identify the parts of the BVH which are intersected by its volume and represent these subtrees as a short array of indices to them (a *candidate list*). The sorted list associated with a shaft thus represents a combination of the hierarchical nature of the BVH and the directionality of the shaft, as shown in Figure 4.1.

The tracing of a ray starts with identifying the shaft which contains the ray. The traversal algorithm is then seeded with the node ids in the associated candidate list. With these ids pushed onto the stack, the traversal proceeds as usual. As the list contains only a very limited subset of the scene geometry, the traversal process does not have to take unnecessary steps through the usual initial parts. Most importantly, it does not have to roam through many lateral subtrees, thus saving significant effort both in terms of traversal steps and memory traffic.

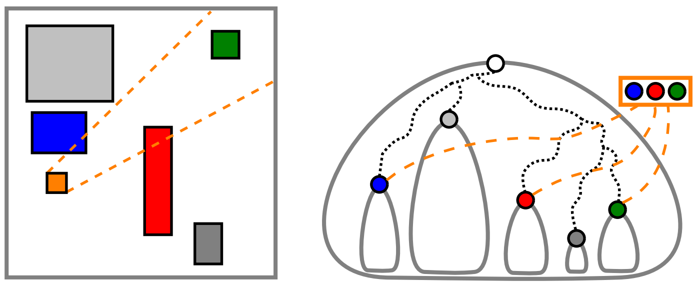


Figure 4.1: A simple scene with one selected frustum shaft (orange box and dashed lines) and several intersected bounding volumes (left) and the corresponding situation in the associated candidate list (orange array of node ids) referring to base BVH (right). The shaft contains only a small subset of the scene geometry, which can be represented by a few nodes inside the BVH. The traversal of nodes above them (dotted lines) and in many parallel branches not intersecting the shaft (grey subtrees) can be skipped altogether.

The light transport in the scene is usually distributed very unevenly and there are many shafts that do not contain a significant number of rays. The construction can be therefore guided by the result of an optional preprocessing step, in which we sample the ray distribution in the scene. Only those shafts which hold the most rays will have their candidate lists built in the end.

## 4.2 Shaft Culling in BVH Traversal

Here we describe and discuss the proposed method in detail, starting with the underlying data structures and their setup followed by their usage during the ray tracing phase.

### 4.2.1 Collection of Shafts

We subdivide the scene AABB into a regular 3D grid, where every single voxel forms a base for individual frustum shafts and groups together the origins of contained rays (see Figure 4.2). For each voxel, we subdivide the ray directions into the six major intervals of directions corresponding to standard cube mapping. Each face of the directional cube map is further subdivided into a regular 2D grid.

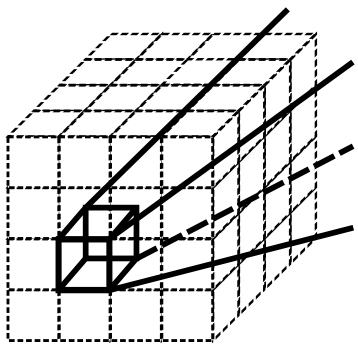


Figure 4.2: Spatial subdivision using a regular grid with a sample frustum shaft originating at the selected voxel. In most scenes, the grid resolution differs among the three axes in order to form cube-like voxels.

There are different ways to set up spatial and directional resolutions. It turns out a single spatial resolution in all three axes does not often fit well, especially in scenes with largely non-cubic proportions; so we allow all three axes to be divided into a different number of equal intervals. On the other hand, for most scenes there are no substantially dominant directions of rays (or at least they are not known in advance), so we use uniform resolution for directional subdivision.

To easily determine the spatial grid resolution, a convenient option is to derive it automatically from the allowed memory budget for the shaft collection (given either in absolute amount or relative to the memory occupied by the scene geometry). We force the voxels to be as close to the cube shape as possible, which aligns well with the SAH preference for cube-like nodes. These soft constraints are limiting the useful values for resolution from both sides: the shafts should not be too wide, in which case they do not cut off much of the scene geometry and thus do not save much traversal costs. They should not be too thin either as this induces high memory consumption, caused by both increased redundancy of content among adjacent shafts and the vast number of shaft structures themselves.

The collection consists of two parts: the list area, which is a concatenation of all candidate lists, and the shaft lookup table, which maps each shaft's id to the corresponding offset of the associated candidate list in the list area. The node ids are just 4-byte indices to the main BVH node array. An example of a BVH with a shaft collection is shown in Figure 4.3.

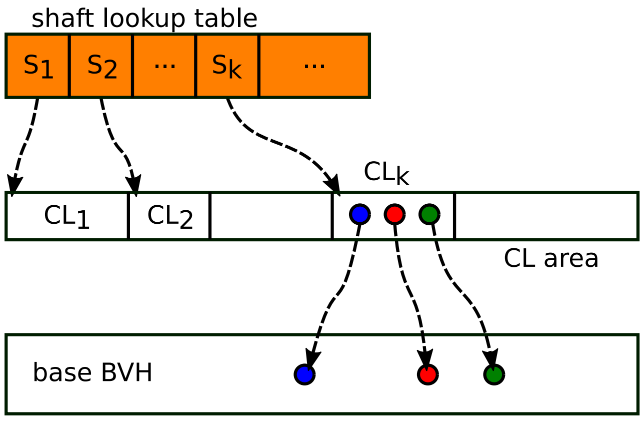


Figure 4.3: Schematic layout of a BVH and a shaft collection. The shaft lookup table maps shaft ids to the offset of their associated candidate lists in the candidate list area. The lists contain direct indices to the nodes of the base BVH.

The shaft id is computed as a combination of the shaft's voxel $x/y/z$ coordinates and the $u/v$ directional coordinates. This id then serves as a key to the shaft lookup table.

### 4.2.2 Shaft Geometry

A shaft's geometrical definition is fully described by its 3D grid base voxel together with culling planes enclosing the subinterval of directions, very much like Arvo and Kirk [6] proposed. The voxel is surrounded by the culling planes, which leave the frustum shaft open on its other end.

At most seven culling planes are used to define the shaft: four of them enclose the directional range of the shaft, and up to three planes contain the outer faces of the shaft's voxel. The shaft geometry is utilized only during the candidate list construction, so we do not store it with the data structure past that step.

### 4.2.3 Candidate List

The shaft's candidate list provides complete coverage of the scene geometry enclosed by the shaft. Its elements are ids of the nodes of the original BVH which intersect the shaft. In Algorithm 1, we show in pseudocode the construction of a candidate list.

```
1: stack.push(bvh.root)
2: while !stack.empty() do
3:    node ← stack.top()
4:    stack.pop()
5:    if shaft.overlaps(node) then
6:       if node.isInner() then
7:          p ← shaft.getHitProbability(node, sampleRays)
8:          if p ≥ minHitProbability then
9:             if shaft.centerRay.dir[node.splitAxis] > 0 then
10:               stack.push(node.child)
11:               stack.push(node.child+1)
12:            else
13:               stack.push(node.child+1)
14:               stack.push(node.child)
15:         else
16:            node ← shaft.cullNodeSubtree(node)
17:            if node.valid() then
18:               shaft.candidateList.append(node)
19:      else {leaf}
20:         shaft.candidateList.append(node)
```

**Algorithm 1:** Construct shaft candidate list for a binary BVH.

A crucial component of our method is the *getHitProbability*() function. This function tests the intersection of a small number (20) of rays within the shaft with the node's AABB to estimate the probability of traversing this node in the rendering phase. These rays are generated randomly using a uniform distribution within the given shaft and they are cast using the standard BVH to establish the intersections with the scene. The hit probability estimation uses these rays to check if the intersections with the node's AABB lie before the intersections of the sample rays with the scene. The estimation thus also considers occlusion, which prevents creating candidate list entries in occluded areas of the scene. The hit probability serves as a measure that tells us whether the node is worth opening and being replaced with its children. If the node passes this check, its children are pushed back onto the stack with respect to the orientation of the shaft's center ray, i.e., depth-sorted to increase the probability of early ray termination later in the rendering phase.

For nodes that do not pass the check of minimum hit probability, in *cullNode-Subtree*() we try to find a single descendant node that still contains all relevant geometry. This is done by opening the current node and checking the number of its children intersected by the shaft; if we find a single descendant with this property, it replaces the ancestor node in the candidate list. Occasionally, all the descendant nodes are culled, effectively preventing this entire subtree from being part of the candidate list.

In some cases, the algorithm produces more nodes than we allow as the maximum size of candidate lists. To address this issue, we progressively increase the *minHitProbability* threshold and call the algorithm again, which tends to generate fewer nodes in the candidate list.

It can be shown that to probabilistically reduce the number of traversal steps, the minimum hit probability threshold has to be set to $\frac{k-1}{k}$ for $k$-ary BVH. If this threshold was lower, the algorithm would descend deeper into the tree and produce longer candidate lists. As a result, we could potentially end up traversing even more nodes than with the standard traversal. Conversely, if the threshold was larger, we would not utilize the potential of the method, staying too close to the BVH root.

Having a candidate list and the BVH subtrees referenced from the nodes in the list, we effectively form a set of sub-BVHs of substantially lower height than the base BVH (see Figure 1.2). These smaller BVHs can be interpreted as presorted view-dependent subsets of scene geometry with respect to a group of coherent rays within a shaft.

### 4.2.4 Memory Optimizations

The upper bound on the build time complexity of a shaft collection is $O(s^3 d^2 k \log n)$ while the space complexity is $O(s^3 d^2 k)$; here $s$ represents a single spatial resolution in all three axes, $d$ a directional resolution in both coordinates, $k$ the maximum length of the candidate list, and $n$ the number of scene primitives.

We can use an optional reduction scheme, which limits the computational and storage costs very efficiently. The idea relies on ray sampling when only the most ray-occupied shafts are allowed to build their candidate lists. The shaft occupancy criterion can be given by a fraction of rays or shafts we want to use for accelerated traversal; it operates on shaft statistics data gathered in the sampling phase and sorted from the most used shafts to the less important ones. The ray sampling is a rendering phase executed in a lower image resolution and/or with fewer samples per pixel. The light transport or any potential information gathered by these sample rays can be utilized during rendering to amortize the cost of sampling.

A cheaper alternative to ray sampling is to construct the candidate lists only for shafts originating in voxels occupied by scene geometry. This approach handles all secondary rays and also shadow rays cast from surfaces toward light sources. However, it excludes the primary rays from processing (unless the camera is located in a nonempty voxel).

Often the candidate lists of adjacent shafts (those with nearby base voxels and similar ray directions) are identical. Therefore, during the construction, we index the candidate lists in a hashmap which gives us a quick answer to whether there already is an identical candidate list in the collection. If so, we do not store the new copy again, but rather let the respective shaft refer to the original instance. This very cheap check saves between 10-80% of memory in the candidate list area in our scenes.

### 4.2.5 Traversal

The traversal of a ray starts by classifying the ray and looking up the corresponding candidate list. If we successfully found the list, all its nodes become the new entry points of the traversal, being pushed onto the traversal stack instead of the BVH root. Otherwise, we continue in the usual manner by following the BVH root.

This is the only modification of a standard traversal kernel that needs to be made: except for the different start-up, we traverse the BVH in the usual fashion. Thus, we remove large parts of the traversal path. It might be necessary to increase the traversal stack capacity, as the stack has to be able to hold the entire candidate list plus some extra space for the traversal of the individual subtrees referenced from the list.

## 4.3 Results

We implemented the proposed method in standard C++11 with thread parallelism integrated into the open-source PBRT-v3 renderer [68]. We performed a series of tests on ten scenes, comparing the ratio of traversed steps, the ray tracing performance, the build times, and the memory footprint of our method, having the standard BVH traversal as a reference. The measurements were performed on a PC equipped with Intel Xeon E3-1245 with 8 cores @ 3.5 GHz and 16 GB RAM, both the shaft collection build and ray tracing phases used 8 working threads. The source code of the algorithm can be downloaded from the project website: `http://dcgi.fel.cvut.cz/projects/rc+bvh`.

The BVHs were constructed using Binning SAH method [82] with 32 bins and 4 triangles per leaf; we used them as static bases for generating different configurations of shaft collections. For direct comparison with PBRT, we built binary BVHs; we also measured the performance on quaternary hierarchies, for which we had to slightly adjust the renderer's traversal code. We chose 31 nodes as the maximum size of candidate lists. This value proved to be a good balance to describe the shaft content with enough detail yet not overly memory-demanding.

### 4.3.1 Ray Tracing Performance

To evaluate the performance of ray tracing aided by shafts, we adjusted the BVH traversal code in the PBRT renderer to execute our lookup and stack population code. The evaluation uses the default rendering setup in PBRT, i.e., path tracing with max. 5 bounces and Russian roulette, accelerated by binary BVHs. This setup incorporates a good amount of incoherent load for most scenes we tested. Each scene was traced from three different camera locations and orientations using $1920 \times 1080$ image resolution with 8 samples per pixel. We tested three different variants of the method:

- **Complete** – Candidate lists of all shafts were built.

- **OccupiedVoxels** – Candidate lists were built only for voxels occupied by scene geometry.
- **ViewDependent** – Ray sampling phase was executed to determine the shafts most used by rays. The candidate lists were computed for 10% of the most populated shafts.

For all variants, we used three different spatial resolutions (100k, 200k, and 500k voxels), combined with two different directional resolutions (2 and 4).

A comprehensive overview of the results for the *OccupiedVoxels* variant and resolution of 200k×4 is summarized in Table 4.1. In the ray tracing phase, the number of traversed steps ranges between 37% and 81% related to the standard traversal with rendering times measured between 81% and 97%. The major reason for this difference between traversed steps and rendering times ratios is that the rendering process also includes geometry intersection and shading on top of the ray traversal itself. These rendering phases are not targeted by our method and remain roughly constant.

The results evaluated on quaternary BVHs exhibit on average 5% lower savings of traversal steps than binary BVHs. Considering there are three times fewer interior nodes in the quaternary BVH than in the binary variant, the achieved savings are actually higher than we initially expected. PBRT does not implement SIMD traversal, therefore our rendering times for quaternary BVHs are generally slightly higher than those for binary BVHs. Using the SIMD traversal would speed up both the reference quaternary BVH method as well as the method accelerated by shafts.

A graphical overview of the speedups for the *FirstHit* and *AnyHit* routines as well as the total rendering speedup is shown in Figure 4.4. In most cases, the complete build yields the best speedups, at the expense of larger memory consumption and longer build time.

The relation of the speedups averaged over all tested scenes to the average relative memory consumption of all evaluated variants is shown in Figure 4.5. The flat shape of the graphs suggests it does not make much sense to increase the memory budget beyond using approx. the same amount as for the geometry. The speedup for shadow rays is significantly higher than for the other types of rays. The method saves about 15% of time in finding ray-scene intersections, dominated by the higher ratio of *FirstHit* calls. The *OccupiedVoxels* variant saves substantial memory but suffers from not handling the primary rays. The *ViewDependent* variant leads to the lowest memory consumption, but due to some uncovered rays, the speedups are also slightly lower than for the *Complete* variant.

To gain more insight into how many elements there are in the candidate lists, we gathered statistics for two scenes (Conference and San Miguel) that largely differ in the number of geometry primitives and whether they represent the interior or exterior. It turns out that despite these differences, the distributions are very similar, resembling the binomial distribution (see Figure 4.6). In the Conference scene, most candidate lists are able to contain all nodes in the first pass of Algorithm 1, the

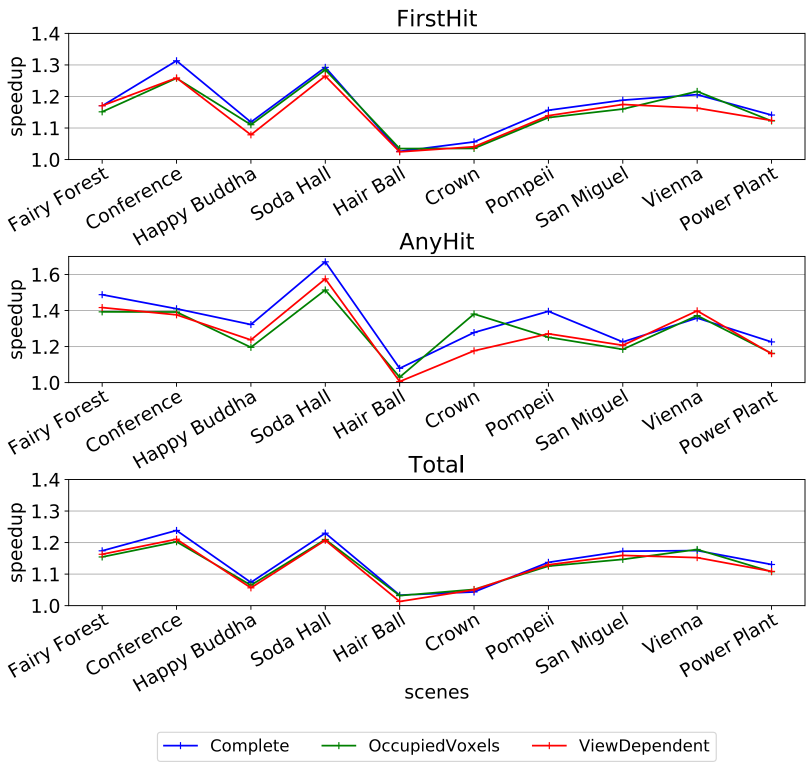


Figure 4.4: The performance of all variants of our method on different scenes for FirstHit and AnyHit routines and the total rendering time. The speedups are calculated relative to the performance of the standard traversal without shafts using a binary BVH.

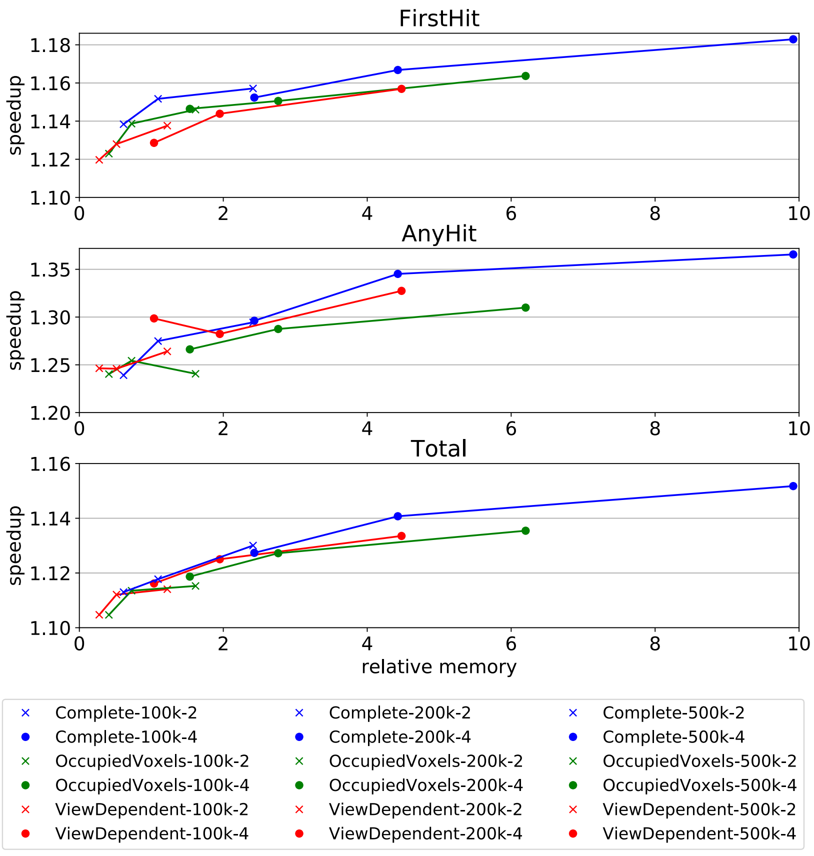


Figure 4.5: Average speedups in relation to the memory demands of our method itself, related to the size needed for storing the geometry of the scenes. To better show the influence of increasing the spatial resolution, the measurements with the same resolution of the directional subdivision for a given method are connected.

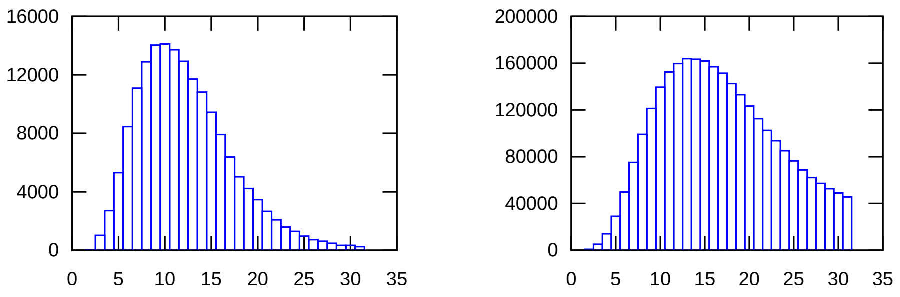


Figure 4.6: Distribution of candidate list lengths in the Conference scene (left) and in the San Miguel scene (right), both having the maximum length equal to 31 indices. In both cases, the most frequent number of elements in candidate lists lies between 10 and 15.

| | Fairy Forest #triangles 174k | | | | Conference #triangles 331k | | | | Happy Buddha #triangles 1087k | | | | Soda Hall #triangles 2169k | | | | Hair Ball #triangles 2850k | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | build size [MB] | build time [s] | traversals per ray [- / %] | render time [s / %] | build size [MB] | build time [s] | traversals per ray [- / %] | render time [s / %] | build size [MB] | build time [s] | traversals per ray [- / %] | render time [s / %] | build size [MB] | build time [s] | traversals per ray [- / %] | render time [s / %] | build size [MB] | build time [s] | traversals per ray [- / %] | render time [s / %] |
| Binary BVH | | | | | | | | | | | | | | | | | | | | |
| *no shafts* | 11 | 0.2 | 44.8 / 100% | 522 / 100% | 22 | 0.4 | 45.7 / 100% | 1006 / 100% | 71 | 1.3 | 23.5 / 100% | 90 / 100% | 139 | 2.3 | 62.0 / 100% | 739 / 100% | 190 | 3.7 | 75.3 / 100% | 358 / 100% |
| *with shafts* | 122 | 19.9 | 21.3 / 47% | 451 / 86% | 127 | 17.6 | 16.9 / 37% | 837 / 83% | 223 | 23.4 | 16.0 / 68% | 83 / 93% | 325 | 29.0 | 27.2 / 44% | 596 / 81% | 851 | 161.5 | 57.7 / 77% | 345 / 96% |
| Quaternary BVH | | | | | | | | | | | | | | | | | | | | |
| *no shafts* | 10 | 0.2 | 43.3 / 100% | 551 / 100% | 20 | 0.4 | 41.0 / 100% | 1049 / 100% | 65 | 1.3 | 23.1 / 100% | 93 / 100% | 128 | 2.4 | 59.0 / 100% | 800 / 100% | 173 | 3.8 | 71.5 / 100% | 360 / 100% |
| *with shafts* | 125 | 21.2 | 22.2 / 51% | 469 / 85% | 130 | 17.2 | 17.5 / 43% | 869 / 83% | 236 | 30.6 | 16.8 / 73% | 87 / 93% | 335 | 42.6 | 32.8 / 56% | 680 / 85% | 963 | 412.2 | 58.1 / 81% | 350 / 97% |

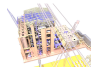
| | Crown #triangles 4868k | | | | Pompeii #triangles 5632k | | | | San Miguel #triangles 7842k | | | | Vienna #triangles 8637k | | | |  Power Plant #triangles 12759k | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | build size [MB] | build time [s] | traversals per ray [- / %] | render time [s / %] | build size [MB] | build time [s] | traversals per ray [- / %] | render time [s / %] | build size [MB] | build time [s] | traversals per ray [- / %] | render time [s / %] | build size [MB] | build time [s] | traversals per ray [- / %] | render time [s / %] | build size [MB] | build time [s] | traversals per ray [- / %] | render time [s / %] |
| Binary BVH | | | | | | | | | | | | | | | | | | | | |
| *no shafts* | 314 | 7.1 | 48.0 / 100% | 219 / 100% | 366 | 6.9 | 79.7 / 100% | 503 / 100% | 514 | 10.1 | 124.0 / 100% | 1898 / 100% | 581 | 10.6 | 59.9 / 100% | 372 / 100% | 809 | 15.3 | 76.8 / 100% | 399 / 100% |
| *with shafts* | 406 | 13.5 | 36.2 / 76% | 208 / 95% | 790 | 76.4 | 50.0 / 63% | 445 / 89% | 664 | 40.5 | 70.8 / 57% | 1640 / 86% | 786 | 46.5 | 31.6 / 53% | 311 / 84% | 911 | 23.9 | 49.9 / 65% | 364 / 91% |
| Quaternary BVH | | | | | | | | | | | | | | | | | | | | |
| *no shafts* | 288 | 7.2 | 46.7 / 100% | 229 / 100% | 335 | 6.9 | 72.7 / 100% | 518 / 100% | 470 | 10.2 | 120.5 / 100% | 1974 / 100% | 528 | 10.8 | 57.3 / 100% | 391 / 100% | 744 | 15.4 | 71.7 / 100% | 426 / 100% |
| *with shafts* | 385 | 16.1 | 37.2 / 80% | 220 / 96% | 809 | 94.4 | 50.4 / 69% | 472 / 91% | 635 | 48.6 | 75.9 / 63% | 1715 / 87% | 757 | 51.4 | 33.1 / 58% | 329 / 84% | 849 | 27.2 | 51.0 / 71% | 391 / 92% |

Table 4.1: Performance comparison of the tested method for the OccupiedVoxels variant with a spatial resolution of 200k and direction resolution of 4. The build times, apart from BVH build itself, refer to the view-independent construction of candidate lists; for static scenes they can be amortized over many rendered frames. The total render times comprise ray generation, ray traversal, intersection evaluation, and shading.

number of the most populated CLs approaching zero. On the other hand, in San Miguel with a much higher number of primitives, we have to iterate the algorithm with an increased threshold in order to squeeze the candidate lists into the allowed size, leaving many candidate lists with high node counts.

To evaluate the behavior with an increased ratio of incoherent rays, we ran an additional test with a closed interior scene (Conference), higher max. bounces (10), and disabled Russian roulette. The speedups remained practically unchanged.

### 4.3.2 Construction Overhead

For high-quality rendering with many samples per pixel and/or many camera frames, the shaft construction times are easily compensated during rendering. This can be seen from most results shown in Table 4.1, although we used a relatively low number of samples per pixel (8). Still, the reduction of the construction overhead may be of concern. The ViewDependent variant uses only the most used shafts and it provides about three times faster construction at the expense of a slightly more complicated implementation (note that the memory footprint drops accordingly, too). In a production renderer, it would also be possible to further optimize the construction using lower-level optimizations, such as SIMD instructions.

# Chapter 5

# Dynamic Ray Batching using Shafts

Ray tracing is an elegant approach to solving light propagation through a virtual scene. However, the actual computational process is usually highly complex in many aspects: the potentially high number of various objects and data structures involved, irregular geometry distribution in the scene, and the non-trivial interrelations among all programmatic and hardware components involved in the computation.

The efficient solution to the problem is closely related to the hardware architecture that executes the computation, which involves a cache hierarchy. While it is mostly not an issue to handle large quantities of data alone, the way to do it efficiently involves identification and minimizing the *working set* used for solving the problem. This is hard in ray tracing as the rays diverge, especially in later bounces of the respective light paths.

The optimization approaches include coherent ray traversal through BVH or ray reordering, which aim to increase the coherence of rays processed either sequentially or in batches [60, 58, 1]. Another approach is ray streaming [33, 8], which employs breadth-first traversal and separates rays to active and inactive subsets, or traversing and pruning a ray-space hierarchy [71]. Another way to increase coherence is to reorder rays and defer their processing until many of them can be traced together through a limited region of the scene [69, 11]. The rays are stored in nodes of a scheduling data structure; however, they may have to traverse multiple nodes of this data structure before they eventually hit a geometry primitive, making the incurred overhead significant.

Traditionally, ray batching was supported by 3D space partitioning data structures such as multiple regular grids [69], or octrees [11]. In this chapter, we employ 5D ray classification using frustum shafts [42] described in Chapter 4 for batching. These geometrical structures contain whole rays, so a ray is stored in the scheduling structure and fetched from it only once before it is traversed through the BVH and intersects with the geometry. We also present a hierarchical alternative to the regular grid of shafts that can put more detail where the shaft-referenced geometry is abundant while being sparser elsewhere. On a reference implementation based on PBRT, we show that such a system exploits the ray coherence present in the scene, thus substantially decreasing the working set size and rendering time.

Concisely, we aim at the following contributions:

- A new technique for adaptive ray classification using a shaft hierarchy.
- Dynamic formation of ray batches during the rendering process.
- Efficient ray batch scheduling to minimize the active working set.

The chapter is further structured as follows: Section 5.1 presents an overview of the proposed method, Section 5.3 describes the method in detail, and Section 5.4 gives the results and their discussion.

## 5.1 Working Set During Rendering

Two of the largest data structures in the ray tracing phase are (1) scene geometry, organized in an acceleration structure (BVH in our case), and (2) rays, which together form *light paths* and need to be traced through the scene.

We target high-quality rendering of large scenes where rendering one frame takes significant time. Therefore we assume the scene geometry to be static, while rays are intermediate read/write geometry structures used for sampling the light propagation through the scene.

The traditional recursive ray tracing [89] is very elegant in handling the process: light paths are being constructed segment by segment, thus connecting the camera and light source(s) in one single, uninterrupted sequence of steps. However, due to the global nature of light propagation, each light path segment (ray) visits a different part of the scene; this makes the working set very large, putting the underlying processing system under heavy stress. In this section, we show how the process can be rearranged so that the rendering system uses as small a working set as possible at most points in time.

### 5.1.1 Tracing the Light Paths

In an ordinary ray tracer [89], the light sampling process starts with a primary ray going from the camera into the scene. Upon finding an intersection with the scene, light sources are sampled using shadow rays, and a new ray is *bounced* (reflected or refracted) further into the scene. Based on terminating criteria, this light path's processing is finished after several bounces, and the light sources' contributions are accumulated to the frame buffer according to the primary ray.

This depth-first process is repeated for each frame pixel that generates a primary ray, possibly with multiple samples per pixel. The frame pixels are usually processed in a tiling scheme, i.e., primary rays are generated in small compact squares in the image plane, which brings limited coherence into the process and makes it viable for parallel processing. Soon after the first bounce, the light paths begin to diverge from each other, leading to large incoherence in tracing the individual ray segments one after another.

As a form of breadth-first approach, wavefront path tracing [53] is a step toward better coherence as it does not require tracing an entire light path in one pass. It keeps track of many light paths under construction at once, of which the same-bounce segments are traced together. An inherent limitation of wavefront path tracing is that it can only exploit coherence among rays from the same wave. To mitigate to a certain degree the quick coherence drop that deteriorates with the number of bounces, the rays in the current wave can be sorted [58] before the BVH traversal.

### 5.1.2 Algorithm Overview

Our approach builds on ray sorting and batching. The light path is broken into elementary rays handled separately in time. Rays processed together in one batch are not restricted to belonging to the same bounce of their respective light paths, not even to be of the same type (i.e., secondary or shadow rays). The general system architecture of our method is shown in Figure 5.1.

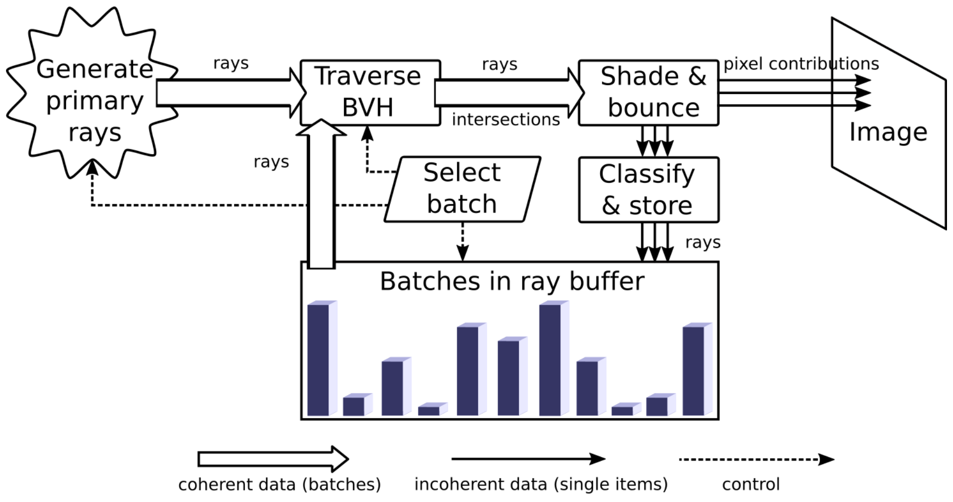


Figure 5.1: Block diagram of the ray batching system. The main difference from an ordinary path tracer is in processing coherent ray batches (thick arrows), whereas the path tracer handles rays of a light path in a depth-first manner, one by one. The coherent processing is enabled by a ray buffer that accumulates rays until large coherent groups emerge. A batch selection oracle also aims to assign ray batches to worker threads in a coherent manner. Unlike previous works, the bounced (originally incoherent) rays are buffered only once per ray's lifetime.

The overhead induced by a batching structure is not negligible and can impede the system performance. With the design goal of reducing this overhead, we decided to use frustum shafts [42] for ray batching because of their close geometrical relation to rays. In this setup, a ray is classified and stored for later processing only once before it gets fully processed, as it does not leave its enclosing shaft.

However, there is a significant overlap in the scene geometry covered by different shafts, which we call the *shaft coherence.* The method takes this type of coherence into account and sequences the ray batches carefully so they exploit the similarity of content among shafts.

## 5.2 Hierarchical Ray Classification

We present the motivation for a shaft hierarchy, which is a viable alternative to the regular grid of shafts, and describe its construction and querying in detail.

While our previous method [42] uses a regular grid to subdivide the space, we explored an adaptive scheme where shafts consider the uneven distribution of elements in the scene. Similarly to the *teapot in stadium* problem for spatial subdivision of the scene, with regular 5D space subdivision, each shaft contains a very different amount of scene geometry. Eventually, this leads to inefficient, unbalanced BVH traversal. An alternative approach is to adapt the volume of a shaft to the local geometry density.

### 5.2.1 Shaft Hierarchy Construction

We propose an adaptive structure of a hierarchy of shafts. In an iterative refinement process, we subdivide the six elementary shafts covering directions of cube map faces until a termination criterion is met.

The shaft hierarchy construction is governed by a priority queue that maintains the current set of shaft hierarchy leaves ordered by relative importance. The queue is seeded with the six elementary shafts, subdivided and refined until at least one of the stopping conditions occurs: the subdivision reaches the maximum depth, the queue has been fully processed, or a memory budget on the number of shafts has been reached.

We experimented with different ordering keys and finally obtained the best results when we set the importance to be proportional to the 5D volume of the shaft multiplied by its candidate list length. This approach effectively minimizes the expected candidate list length for uniformly distributed random rays.

The candidate lists are short arrays of nodes that fully cover the region of scene geometry and the BVH encompassed by a shaft [42]. When constructing the candidate lists, we cut the time costs substantially by considering BVH instance references as opaque BVH nodes. This saves much of the shaft build time at the cost of candidate lists of only marginally lower quality, which does not have an apparent effect on the render times.

The spatial subdivision yields eight new shafts (as the octree spatial subdivision would do), and the directional subdivision yields four new shafts (as the quadtree subdivision of the cube map faces). In our basic implementation, the subdivision scheme is rigid. At each level, we choose the type of subdivision (spatial vs. directional) according to hardcoded rules (subdividing the first four levels only spatially, then alternating with directional subdivision). Surprisingly, this scheme turns out to be the best for all test cases and allows us to classify rays faster than it would be possible with a fully adaptive hierarchy. A schematic subdivision and the corresponding hierarchy are depicted in Fig. 5.2.

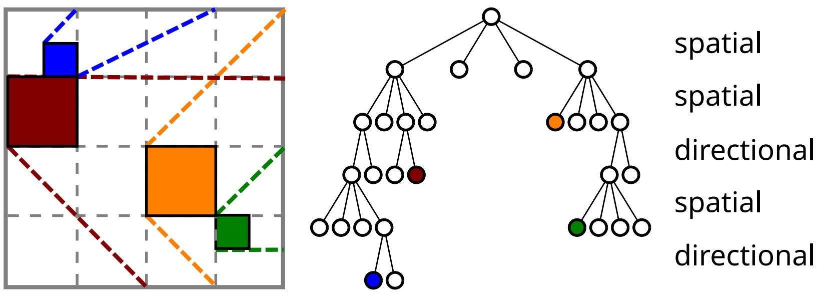


Figure 5.2: A simplified 2D scene partially covered by a hierarchy of shafts. On the left, four shafts with differently sized base voxels and ranges of directions demonstrate the ability to adapt to the amount of geometry in a shaft (the actual scene geometry not shown for clarity). The shafts are stored in nodes of a hierarchy, with the topmost node representing the shaft that spans the whole scene and the descendants referring to the alternating spatial and directional subdivision (right).

### 5.2.2 Shaft Hierarchy Traversal

Each ray within a scene is classified into exactly one shaft by quantizing the ray coordinates. Thus, all shafts form a partitioning on the 5-dimensional space, covering the three spatial ($x$, $y$, $z$) and two directional ($u$, $v$) coordinates of rays. We use the individual quantized bits as a key for the hierarchy traversal: starting in the root node with the full set of quantized bits, we take the most significant bits in order to decide into what child node to descend. The key is then formed as an interleaved sequence of the ray coordinates bit representations.

With uniform grid shafts, we only need $O(1)$ computation to classify a ray into a shaft. Conversely, the non-uniform shaft hierarchy incurs an average overhead of roughly $O(\log k)$ steps of hierarchy traversal per ray (the hierarchy having $k$ leaf nodes), which also involves incoherent memory access. To mitigate this cost, we implemented a small, fast round-robin shaft cache that answers most classification queries, thus bypassing the costly hierarchy traversal.

The cache implementation uses AVX-512 SIMD intrinsics. The traversal key is compared simultaneously with 64 keys packed together in the shaft cache, starting with their highest byte and narrowing down the set of possible matches. If the key is not found in the cache, we fall back to the hierarchical ray classification. The key with the corresponding shaft index then replaces the oldest cache entry.

This way, we are able to answer the cache query in a short sequence of instructions while touching only a few CPU cache lines per query. The hit ratio of the shaft cache varies between 70% to 80% on average for a 0.75 kB cache, which is enough to make the cost of the shaft hierarchy traversal tolerable.

## 5.3 Ray Batching with Shafts

In this section, we show different aspects of the rendering process dynamics. First, we outline the general operation of the batching process. Then we show how the framework can also be used for deferred batched shading, possibly in an out-of-core scenario. At the end, we give a detailed view of the coherent sequencing of ray batches.

### 5.3.1 Process Overview

To defer the processing of a newly spawned ray until many similar rays are gathered, we store it in a huge linear array. However, the sequence of generated rays is usually highly incoherent. We also process each of the rays at a different time in the future; this prevents us from storing the rays consecutively with respect to their containing shaft. Instead, the rays of the shaft are stored in the array separately, linked together via an index to the next ray in the array, and the shaft holds the head index of the list.

The ray buffer holds the head index of the list of free rays, i.e., currently unallocated items in the array. The worker threads also maintain their private linked pools of free rays to avoid expensive allocation from the shared supply. This scheme is flexible yet efficient enough for the seemingly random light bounces in the scene.

The overall rendering progresses in the image plane one tile with $16 \times 16$ pixels after another, in scanline order. Unlike the standard depth-first path tracing, we do not immediately trace the light paths beyond their respective primary rays: we only traverse, shade, and bounce the primary rays first. The newly generated rays are classified into shafts and stored in the ray buffer.

The *fill ratio* of the ray buffer (the ratio of stored rays count and the buffer capacity) is controlled by three constraints:

- overall coherence, which is higher with higher ray count,
- robustness, which ensures that the buffer will not overflow, and
- batching efficiency, which is higher with processing larger batches of coherent rays.

The ray buffer shows a periodic behavior under these constraints, where the fill ratio oscillates between a lower and an upper limit.

We start with an empty buffer and primary rays generation and tracing. As the primary rays bounce, new rays are stored in the buffer. Once the ray buffer attains the upper limit, we switch from primary rays generation to ray buffer processing. The batch selection oracle picks a shaft largely populated with rays called *pivot*, which represents a contiguous *cluster* of adjacent shafts, and maps individual cluster shafts to worker threads (see Fig. 5.3).

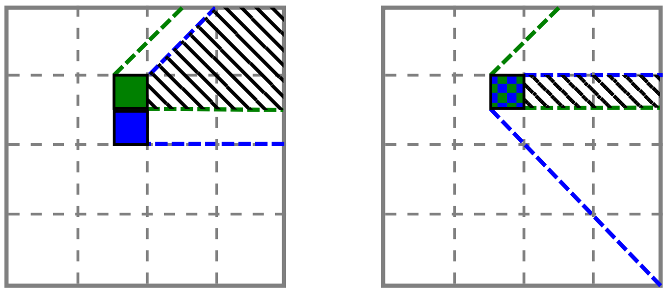


Figure 5.3: Examples of shaft coherence. Two shafts with adjacent base voxels and the same range of directions (left) and two shafts based at the same voxel but differing in their directions (right) share the scene regions marked with hatches. In both cases, the scheduling oracle aims to assign the neighboring shafts for successive processing.

Over time, the fill ratio decreases under the lower limit. At this moment, the oracle switches the worker threads to generate and bounce new primary rays in the system again. This cycle is repeated as long as some primary rays are left; eventually, the system enters the tail phase until all remaining rays in the buffer have been processed.

To ensure that the process is both efficient and robust, we found experimentally that the lower and upper limits on the ray buffer fill ratio should be set around 80% and 90% of the buffer capacity, respectively. Depending on the type of scene (interior vs. exterior) and the rendering configuration (most importantly, the maximum number of bounces per light path), the dynamics of the process varies significantly. Not only do the primary rays spawn multiple times more new rays, the same holds for subsequent bounces; hence care has to be taken not to overflow the buffer with bounced rays. If it happens, the ray buffer can be reallocated to accommodate for more rays, and the upper limit then has to be lowered so that the process does not run into such a situation too often.

### 5.3.2 Deferred and Out-of-core Batched Shading

With the batching system working with the 5D ray classification, another opportunity arises to extract coherence from the process. The texture memory is typically accessed in an almost random order in the shading phase of rendering. As with the geometry incoherence of the recursive light path traversal, it decreases the rendering performance significantly.

Our framework also allows for batching at this rendering stage (see Fig. 5.4). Once a ray has been traversed and intersected with geometry, the texture at the hitpoint is known, and its identifier is used as a classification key. Instead of immediately fetching a texel value and launching the shading code, we use the key to store the ray in the ray buffer again. In this case, rays with the same texture identifier are grouped in a dedicated container (a *texture bucket*) that aggregates rays similarly to geometric shafts.

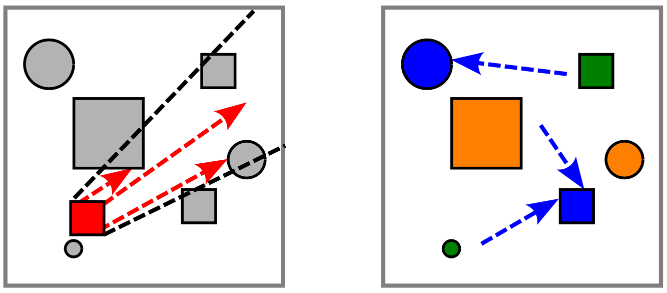


Figure 5.4: Examples of geometrical (left) and texture (right) ray coherence. Rays within one shaft intersect the same limited amount of geometry for coherent traversal, while rays that hit the same texture anywhere in the scene allow for batched shading.

The textures in a scene can be tagged with a timestamp of the last access, which induces a time ordering on the set of all textures. We utilize this attribute for an out-of-core texture swapping scheme, where the in-memory texture cache is not large enough to contain all textures at once. A cache eviction policy aligned with the texture scheduling strategy allows us to achieve order-of-magnitude faster rendering than with the recursive path tracer under the same memory restrictions.

### 5.3.3 Batch Selection and Processing

The batch selection oracle aims to pick cluster pivots with many rays so that both high coherence and ray buffer size are maintained close to the optimal levels during processing. As the adjacent shafts (i.e., shafts with similar indices) tend to share much of the pivot's properties (see Fig. 5.3), including only slight differences in referenced geometry, they are processed in a sequence until the shaft cluster is considered exploited. Then again, the oracle independently picks another highly populated pivot and schedules shafts from its related cluster for processing.

While the frustum shafts ensure that a ray only needs to be classified and stored once, this scheme introduces a new problem: adjacent shafts share much of the scene geometry and should be processed tightly after one another. With coherence in mind, we need to walk through the 5D shaft space guided by their similarity.

We implemented a straightforward yet efficient way of sequencing shafts: we construct a global space-filling curve that spans the entire 5D shaft space and connects adjacent shafts, see Figure 5.5. This strategy aims to maximize BVH traversal coherence between successive shafts in terms of the BVH nodes that need to be loaded when rays of a new shaft are being traced.

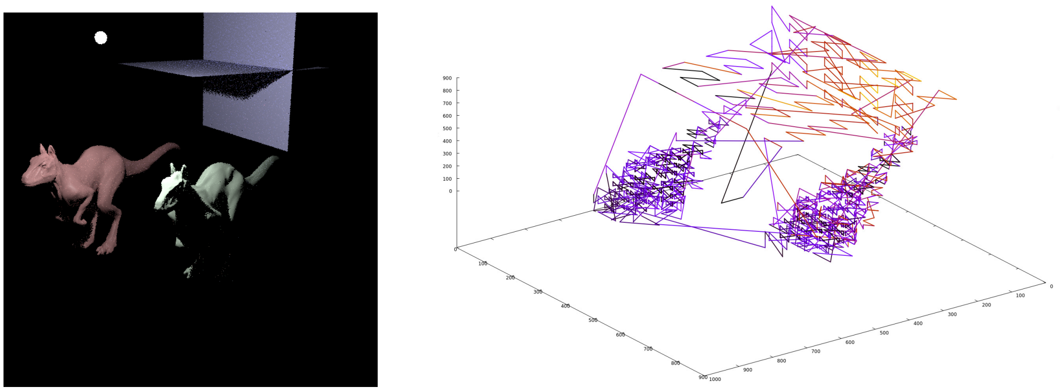


Figure 5.5: The global reduced Morton curve connects adjacent shafts for their coherent scheduling in the Killeroo scene. Segments of the curve that reference empty base voxels are unnecessary and removed from further processing. The color coding of the curve segments indicates the expected cost of BVH traversal using the neighboring shafts in a coherent cluster for successive batch processing (the dark segments denote the cheapest transitions). For brevity, the visualization omits the segments between shafts with the same base voxel but differing in the range of directions.

For easy selection of the currently most populated clusters, we maintain a ray count-ordered linked list of shafts and use the most populated shafts located at the list head as pivots. The list itself is only lazily ordered: every time a shaft receives a new ray, it checks whether it has become the currently largest shaft; if so, this shaft is moved to the head of the list. This way, the ordering overhead is kept negligible. The list is not ordered perfectly, but still well enough to heuristically control shaft sequencing.

With deferred shading, the shaft selection becomes slightly more complicated as we need to select the next ray batch from these two separate sets. In our implementation, we use a fixed threshold on the ratio of rays in texture buckets; once there are enough rays in a texture bucket, the oracle switches to deferred shading of the bucket's rays.

## 5.4 Results

We integrated our method into the publicly available PBRT renderer and provide our implementation for further use. The method was evaluated on a workstation running on Intel Core i7-9800X CPU with eight cores @ 3.8 GHz and 64 GB RAM, using a single thread per core (no hyperthreading).

To evaluate the method, selected scenes from the publicly accessible scenes for pbrt-v3 and Bitterli's Rendering Resources [12] repositories were used. For the main comparison, the only parts that changed are the `Integrator` and `Accelerator` sections, where the standard `path`/`bvh` combination was replaced with batching path tracer and frustum shaft accelerator; we also adjusted the values of samples per pixel so that the render times were similar among different scenes.

### 5.4.1 Ray Tracing Performance

On four scenes of different sizes and complexity, we show the important metrics of the method: the amount of memory used for the acceleration structures, their respective build time, and the render times of different variants of the method. The render times are further broken down to the bare trace times, which is the core indicator of our method.

The scene sizes are characterized by three values: the number of unique primitives (proportional to memory footprint), the total number of primitives that include the instanced ones (scene size), and the number of individual instances in the scene. The build times consist of two values: the first refers to the BVH construction, and the other denotes the optional shaft set build; in all cases, the construction effort can be amortized by using only moderately high samples per pixel values during rendering. The render and trace times are provided for seven different method configurations, which combine the following features: *PT* denotes the baseline path tracing, while *ST* (shaft traversal), *RB* (ray batching), and *grid*/*hierarchy* denote the techniques used by our method. The main results can be seen in Table 5.1.

In most setups, the trace times are lower than those of the standard path tracer; they seldom exceed it, and only by a small margin. Much more often, the trace time is significantly lower, up to one-third of the baseline time can be saved. Almost always is the PT+RB variant faster than PT+ST, but the best is their combination, PT+ST+RB, which combines the individual benefits of both shaft uses (shaft traversal and ray batching using shafts).

Perhaps the most important factor that affects the overall acceleration is the size of the scene and the size of the instanced geometry in terms of the number of primitives. For example, a larger scene (San Miguel) that contains instances with a small number of primitives exhibits the most noticeable speedup, which indicates the method is best suited for large scenes with dominant main BVH, i.e., the top-level acceleration data structure. To confirm this claim, we conducted another series of tests, where the scenes were reduced to include only a single large BVH. The results in Table 5.2 indicate that such setup is indeed the most favorable for the method.

For more detailed insight, in Figure 5.6 we show how the method scales with the number of threads. With low concurrency, our method is able to arrange the memory traffic well, and the speedup over the standard path tracer is relatively high

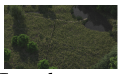
| *complete scenes* | Curly Hair | San Miguel | Ecosys |  Landscape |
|---|---|---|---|---|
| *size* | 26.3M/26.3M/1 | 2.66M/10.7M/160k | 1.18M/19M/4.82k | 26M/4.33G/30k |
| | | *Build time* | | |
| *BVH+grid* | 47.0(+80.5) | 3.8(+1.3) | 1.5(+12.9) | 36.7(+357.7) |
| *BVH+hierarchy* | 47.3(+65.6) | 3.8(+4.4) | 1.5(+14.4) | 37.2(+328.3) |
| | | *Render time* | | |
| *PT* | 253.1 (100.0%) | 225.2 (100.0%) | 225.8 (100.0%) | 202.9 (100.0%) |
| *PT+ST+grid* | 253.7 (100.3%) | 203.4 (90.3%) | 213.3 (94.5%) | 212.1 (104.5%) |
| *PT+ST+hierarchy* | 257.0 (101.5%) | 208.4 (92.6%) | 222.4 (98.5%) | 224.1 (110.4%) |
| *PT+RB+grid* | 231.5 (91.5%) | 198.4 (88.1%) | 204.8 (90.7%) | 222.0 (109.4%) |
| *PT+RB+hierarchy* | 232.8 (92.0%) | 205.3 (91.2%) | 214.4 (94.9%) | 227.5 (112.1%) |
| *PT+ST+RB+grid* | 229.5 (90.7%) | 181.9 (80.8%) | 194.1 (86.0%) | 225.0 (110.9%) |
| *PT+ST+RB+hierarchy* | 232.5 (91.9%) | 185.3 (82.3%) | 206.7 (91.5%) | 244.3 (120.4%) |
| | | *Trace time* | | |
| *PT* | 233.1 (100.0%) | 152.7 (100.0%) | 178.4 (100.0%) | 179.0 (100.0%) |
| *PT+ST+grid* | 233.5 (100.2%) | 130.8 (85.7%) | 166.0 (93.1%) | 188.0 (105.0%) |
| *PT+ST+hierarchy* | 236.8 (101.6%) | 135.8 (88.9%) | 174.9 (98.0%) | 199.8 (111.7%) |
| *PT+RB+grid* | 208.9 (89.6%) | 121.4 (79.5%) | 158.0 (88.6%) | 195.8 (109.4%) |
| *PT+RB+hierarchy* | 210.6 (90.4%) | 126.3 (82.7%) | 166.6 (93.4%) | 201.1 (112.4%) |
| *PT+ST+RB+grid* | 207.0 (88.8%) | 105.5 (69.1%) | 147.5 (82.7%) | 199.1 (111.3%) |
| *PT+ST+RB+hierarchy* | 210.4 (90.3%) | 106.2 (69.6%) | 159.1 (89.2%) | 217.9 (121.7%) |

Table 5.1: Path tracing (PT) performance comparison of various combinations of ray batching (RB) and shaft traversal (ST) on four scenes. The scene sizes refer to the number of unique (memory items) vs. instantiated (scene) primitives, and the number of instances, respectively. The build time consists of BVH construction and optional shaft construction; for all scenes, we used the same budget of about 4.8M grid shafts or 1M hierarchical shafts, respectively, and a 1 GB ray buffer. The render times consist of all ray lifetime stages: generation, BVH traversal, geometry intersection, and shading. The trace times show in detail how the increased ray coherence accelerates the BVH traversal.

(up to 1.28×). However, as the number of threads increases, the memory subsystem becomes again a bottleneck and the performance of the two methods begins to level.

The rendering time structure is shown in detail in Figure 5.7. The scenes vary in several aspects, such as the number of unique vs. instanced primitives and the complexity of the shading phase. In San Miguel, for example, the number of primitives within instanced BVHs is relatively low with respect to the large top-level BVH; on the other hand, shading takes a significant amount of time. This suggests that while the method aims at coherent BVH traversal, it may also have a positive influence on the coherent access to the textures and materials as a side effect. The huge Landscape scene, though, suffers from substantial ray classification in the ubiquitous and large instanced geometry. The classification here is used only for shaft traversal and

| *single-BVH scenes* | | | | |
|---|---|---|---|---|
| | San Miguel (main) | Betula pendula | Abies concolor | Landscape (main) |
| *size* | 1.62M | 4.17M | 1.41M | 6.54M |
| | | *Build time* | | |
| *BVH+grid* | 2.6(+1.3) | 6.2(+17.1) | 2.0(+32.0) | 9.9(+1.3) |
| *BVH+hierarchy* | 2.5(+5.5) | 6.2(+20.6) | 2.0(+23.7) | 9.8(+3.9) |
| | | *Render time* | | |
| *PT* | 399.8 (100.0%) | 367.5 (100.0%) | 339.7 (100.0%) | 393.0 (100.0%) |
| *PT+ST+grid* | 353.5 (88.4%) | 366.1 (99.6%) | 326.1 (96.0%) | 385.2 (98.0%) |
| *PT+ST+hierarchy* | 365.1 (91.3%) | 379.2 (103.2%) | 342.6 (100.9%) | 406.3 (103.4%) |
| *PT+RB+grid* | 347.9 (87.0%) | 340.7 (92.7%) | 309.5 (91.1%) | 400.2 (101.8%) |
| *PT+RB+hierarchy* | 360.1 (90.1%) | 334.6 (91.1%) | 311.1 (91.6%) | 417.1 (106.1%) |
| *PT+ST+RB+grid* | 313.3 (78.4%) | 332.7 (90.5%) | 294.2 (86.6%) | 385.4 (98.0%) |
| *PT+ST+RB+hierarchy* | 321.8 (80.5%) | 329.4 (89.7%) | 300.1 (88.3%) | 403.3 (102.6%) |
| | | *Trace time* | | |
| *PT* | 258.3 (100.0%) | 266.9 (100.0%) | 256.9 (100.0%) | 168.2 (100.0%) |
| *PT+ST+grid* | 212.6 (82.3%) | 265.2 (99.3%) | 243.0 (94.6%) | 157.8 (93.8%) |
| *PT+ST+hierarchy* | 223.7 (86.6%) | 278.7 (104.4%) | 258.9 (100.8%) | 178.5 (106.1%) |
| *PT+RB+grid* | 197.8 (76.6%) | 228.3 (85.5%) | 220.4 (85.8%) | 164.9 (98.1%) |
| *PT+RB+hierarchy* | 205.6 (79.6%) | 223.2 (83.6%) | 222.3 (86.5%) | 176.6 (105.0%) |
| *PT+ST+RB+grid* | 164.0 (63.5%) | 220.1 (82.5%) | 205.4 (79.9%) | 151.2 (89.9%) |
| *PT+ST+RB+hierarchy* | 166.0 (64.3%) | 218.0 (81.7%) | 211.7 (82.4%) | 162.4 (96.6%) |

Table 5.2: Method performance on large single BVHs extracted from the San Miguel, Barcelona Pavilion, and Landscape scenes. In this setup with no instancing, the method gives the best results: most test cases show trace times around 80% of the standard path tracing, reaching almost 60% for San Miguel. On the other hand, in Landscape, we observe mixed results, perhaps due to the unusually flat shape of the scene.

its cost is not justified by the batched coherent access, which only takes place in the main BVH.

### 5.4.2 Discussion and Limitations

Our current results show good behavior with a moderate thread count. However, when utilizing the full computing potential, the memory traffic again becomes a bottleneck. We conducted experiments with ray distributions that were dumped to disk during a real rendering session. Using an offline script, we artificially presorted the ray population by several keys that interleaved the ray's origin and direction [58], without honoring the ordering among bounces within a single light path, and fed a modified renderer with the permutation. Not even this favorable setup showed significant speedup in comparison with the real batching renderer performance.

We believe this outcome stems from two possible reasons. One, the renderer used has come to its limits and will not accelerate without implementing fundamental

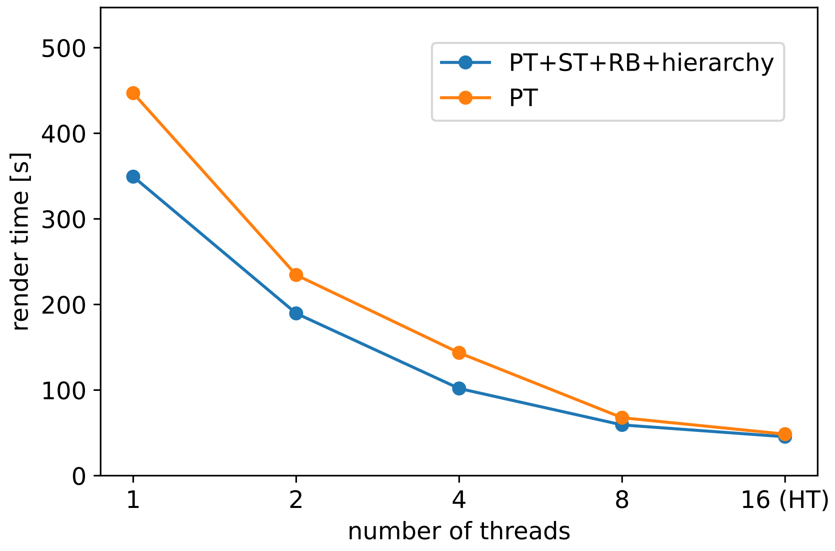


Figure 5.6: Comparison of scalability of our method (PT+ST+RB+hierarchy) and standard path tracing (PT). With a moderate number of threads, we achieve noticeable speedup thanks to the lower memory demands. When approaching higher concurrency up to full hyper-threading (HT), the memory throughput eventually gets saturated but our method is still slightly faster than the baseline path tracer.

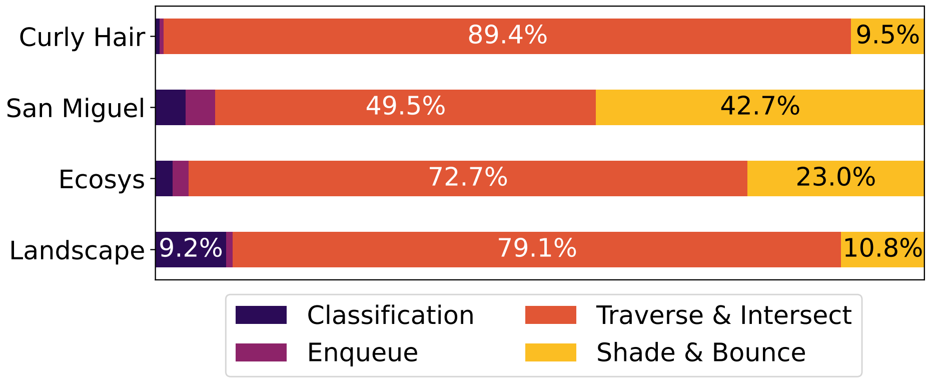


Figure 5.7: Render time breakdown for different scenes. For scenes with many textures or materials, such as San Miguel, the shading phase limits the gain of coherent traversal. In scenes with many instances, the ray classification phase takes a significant amount of time. The trace speedup is then limited by the amount of incoherent traversal in the instances.

changes in the codebase. Two, the system itself is not capable of sustaining higher memory traffic than we already achieved. A more in-depth analysis with detailed profiling and implementing the method in a cutting-edge ray tracing system like Embree [87] can shed more light on the problem.

Currently, the method uses only the main BVH for batching using shafts. In scenes with large instance BVHs, however, the potential for coherence extraction is not fully used. The solution is not obvious as the instance BVH traversal result (the closest intersection with the ray) may be later invalidated by discovering an even closer intersection within the main BVH or another instance BVH. Such traversal would have to be prepared to be interrupted and resumed, i.e., to be able to save and restore its traversal state.

# Chapter 6

# Conclusion and Future Work

I have presented three contributions to the field of accelerating the ray tracing method. Each of them improves the rendering process from a different perspective. This chapter summarizes the achieved results outlined in Chapter 1 and briefly discusses possible directions for future research.

## 6.1 Summary

**Improving the BVH quality** We proposed a novel scalable algorithm for BVH construction on multi-core CPU architectures. The algorithm employs a two-phase process: first, it constructs an auxiliary BVH using the LBVH algorithm, followed by constructing the final BVH using progressively refined cuts of the auxiliary BVH. The progressive refinement of the cut size is driven by adapting surface area thresholds based on the current depth of the constructed node. We provided a simple way of integrating spatial splits in the BVH construction process.

The results show that the method yields superior build performance compared to the high-quality builders implemented in the Embree framework while closely matching their ray tracing performance. Compared with the strategy of Hunt et al. [45] adapted to BVH construction, our method brings about 20% build time improvement while also providing a few percent improvements in trace performance.

**Decreasing the number of ray traversal steps** We described a simple and flexible algorithm for accelerating BVH traversal. The method builds an additional data structure on top of an existing BVH and is able to cut off unnecessary parts of traversal based on the spatial and directional classification of individual rays. Groups of similar rays are enclosed in frustum shafts, each containing only a limited subset of BVH nodes and scene geometry. This subset is represented by a candidate list, thus forming a forest of sub-BVHs of substantially lower traversal costs than the base BVH.

Although previously deemed impractical, we showed that the ray classification can support contemporary BVHs traversal successfully, and we highlighted the important technical details that make it work. On the PBRT renderer, we showed that the method could be plugged easily into an existing framework. Our experiments showed that it saves a large portion of traversal steps, about 42% on average, and a significant amount of rendering time. We also provided a basic view of how the algorithm behaves with quaternary hierarchies, showing results similar to native binary variants.

**Increasing the data coherence and reuse of quickly accessible data** We have described a system that efficiently schedules rays for coherent batched processing. We store rays in a ray buffer to traverse them through BVH later when they have formed large coherent batches. In contrast with previous efforts, we do this just once per ray lifetime; this is crucial for the system's efficacy, as the overhead of batching is not negligible. The coherent batch selection oracle addresses the problem of shared parts of a scene among multiple ray-space shafts. We also introduced a new form of shaft collection, the hierarchy of shafts, that adapts to the scene geometry distribution and is a viable alternative to the regular grid of shafts.

The system can easily be used in an out-of-core scenario as well as in-core. In our implementation, we also use the batching system to group rays just before they enter the shading phase, which often takes more time than BVH traversal in contemporary production scenes. By sorting the rays by the texture or material id of the intersected geometry, we significantly improve the coherent access to these large data and the respective code and reuse them many times before they get swapped in memory.

The shafts can be used to improve data coherence and shorten the ray traversal simultaneously. These two approaches support each other well as both reduce the amount of memory traffic in the rendering process, saving up to 30% of trace time, while the whole rendering can be faster by 20%.

## 6.2 Future Work

**Parallel BVH Construction using Progressive Hierarchical Refinement** We see great potential in integrating static and dynamic content using progressive hierarchical refinement to merge a high-quality static hierarchy and dynamically constructed auxiliary BVH. This technique could have immediate applications in video game technology.

The proposed method scales in a large range of build time vs. quality. However, finding optimal parameters for a given scene and target frame rate in interactive applications remains an open problem. This area has been only seldom addressed algorithmically [61, 37], but recent advances in neural rendering [80, 23] or differentiable rendering [48, 56] bring promising ideas.

**Ray Classification for Accelerated BVH Traversal** We provided a basic view of how the algorithm behaves with quaternary hierarchies. A more in-depth analysis of the method's behavior with wide BVHs and optimized traversal kernels would open the field for the currently more employed SIMD-friendly BVHs.

We construct the candidate lists with secondary rays in mind in our implementation. A modified version specialized for any-hit usage would consider the different objective of tracing shadow rays.

Another interesting topic is the combination of our method with the path guiding approach [62], which also uses a subdivision of ray space that the two algorithms could easily share.

**Dynamic Ray Batching using Shafts** Ray batching in our current implementation only occurs at the main BVH level. In scenes with large instance BVHs, their potential for coherent batched access is also an important topic.

Coherent access to textures in our method is crucial for accelerated out-of-core rendering, where at least some parts of data have to be swapped repeatedly in the main memory. The low coherence in the system at the end of the out-of-core rendering is significantly more prominent than with the in-core variant and would be worth investigating.

# References

[1] Timo Aila and Tero Karras. “Architecture Considerations for Tracing Incoherent Rays”. In: *Proceedings of the Conference on High Performance Graphics.* 2010, pp. 113–122.

[2] Timo Aila, Tero Karras, and Samuli Laine. “On Quality Metrics of Bounding Volume Hierarchies”. In: *Proceedings of High Performance Graphics.* 2013, pp. 101–108.

[3] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering.* CRC Press, 2008.

[4] Ciprian Apetrei. “Fast and Simple Agglomerative LBVH Construction”. In: *Computer Graphics and Visual Computing (CGVC).* 2014.

[5] Arthur Appel. “Some Techniques for Shading Machine Renderings of Solids”. In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference.* 1968, pp. 37–45.

[6] James Arvo and David Kirk. “Fast Ray Tracing by Ray Classification”. In: *Computer Graphics (SIGGRAPH ’87 Proc.)* 21.4 (July 1987), pp. 55–64.

[7] Kavita Bala, Julie Dorsey, and Seth Teller. “Radiance Interpolants for Accelerated Bounded-Error Ray Tracing”. In: *ACM Trans. Graph* 18.3 (1999), pp. 213–256.

[8] Rasmus Barringer and Tomas Akenine-Möller. “Dynamic Ray Stream Traversal”. In: *ACM Transactions on Graphics (TOG)* 33.4 (2014), pp. 1–9.

[9] Carsten Benthin, Radoslaw Drabinski, Lorenzo Tessari, and Addis Dittebrandt. “PLOC++: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited”. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 5.3 (2022), pp. 1–13.

[10] Carsten Benthin, Sven Woop, Ingo Wald, and Attila T. Áfra. “Improved Two-Level BVHs using Partial Re-Braiding”. In: *Proceedings of High Performance Graphics.* 2017, pp. 1–8.

[11] Jacco Bikker. “Improving Data Locality for Efficient In-Core Path Tracing”. In: *Comput. Graph. Forum* 31.6 (2012), pp. 1936–1947.

[12] Benedikt Bitterli. *Rendering Resources.* URL: `https://benedikt-bitterli.me/resources/`.

[13] Jiří Bittner. "Hierarchical Techniques for Visibility Computations". In: *Prague: Department of Computer Science and Engineering, Czech Technical University* (2002).

[14] Jiří Bittner, Michal Hapala, and Vlastimil Havran. "Fast Insertion-Based Optimization of Bounding Volume Hierarchies". In: *Computer Graphics Forum* 32.1 (2013), pp. 85–100.

[15] Solomon Boulos, Dave Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. "Packet-Based Whitted and Distribution Ray Tracing". In: *Proceedings of Graphics Interface 2007.* 2007, pp. 177–184.

[16] Normand Brière and Pierre Poulin. "Adaptive Representation of Specular Light". In: *Computer Graphics Forum* 20.2 (2001), pp. 149–162.

[17] Normand Brière and Pierre Poulin. "Hierarchical View-Dependent Structures for Interactive Scene Manipulation". In: *SIGGRAPH 96 Conference Proceedings.* ACM SIGGRAPH. Aug. 1996, pp. 83–90.

[18] Brian Budge, Tony Bernardin, Jeff A. Stuart, Shubhabrata Sengupta, Kenneth I. Joy, and John D. Owens. "Out-of-core Data Management for Path Tracing on Hybrid Resources". In: *Computer Graphics Forum.* Vol. 28. 2. Wiley Online Library. 2009, pp. 385–396.

[19] H. Dammertz, J. Hanika, and A. Keller. "Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays". In: *Computer Graphics Forum* 27.9 (2008), pp. 1225–1233.

[20] Kirill Dmitriev, Vlastimil Havran, and Hans-Peter Seidel. *Faster Ray Tracing with SIMD Shaft Culling.* Research Report. Max-Planck-Institut für Informatik, Dec. 2004.

[21] Michael Doyle and Karthik Vaidyanathan. *Apparatus and Method for Reduced Precision Bounding Volume Hierarchy Construction.* US Patent 11,321,910. May 2022.

[22] Philip Dutré, Philippe Bekaert, and Kavita Bala. *Advanced Global Illumination, 2nd Edition.* Natick, USA: A K Peters, 2006.

[23] S.M. Ali Eslami, Danilo Jimenez Rezende, Frederic Besse, Fabio Viola, Ari S. Morcos, Marta Garnelo, Avraham Ruderman, Andrei A. Rusu, Ivo Danihelka, Karol Gregor, et al. "Neural Scene Representation and Rendering". In: *Science* 360.6394 (2018), pp. 1204–1210.

[24] Sebastian Fernandez, Kavita Bala, Moreno A. Piccolotto, and Donald P. Greenberg. *Interactive Direct Lighting in Dynamic Scenes.* Technical report PCG-00-2. Program of Computer Graphics, Cornell University, Jan. 2000.

[25] Petr Frantál. "Metody pro vrhání paprsku s hierarchiemi obálek na CPU". MA thesis. České vysoké učení technické v Praze, 2019.

[26] Per Ganestam, Rasmus Barringer, Michael Doggett, and Tomas Akenine-Möller. "Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees". In: *Journal of Computer Graphics Techniques (JCGT)* 4.3 (2015), pp. 23–42.

[27] Per Ganestam and Michael Doggett. "SAH Guided Spatial Split Partitioning for Fast BVH Construction". In: *Computer Graphics Forum* 35.2 (2016), pp. 285–293.

[28] Tianhan Gao and Ying Li. "Real-Time Ray Tracing Algorithm for Dynamic Scene". In: *Proceedings of the 13th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2019)*. Springer. 2020, pp. 125–131.

[29] Kirill Garanzha and Charles Loop. "Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing". In: *Computer Graphics Forum*. Vol. 29. 2. Wiley Online Library. 2010, pp. 289–298.

[30] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. "Simpler and Faster HLBVH with Work Queues". In: *Proceedings of High Performance Graphics*. 2011, pp. 59–64.

[31] Andrew S. Glassner. "Space Subdivision For Fast Ray Tracing". In: *IEEE Computer Graphics and Applications* 4.10 (Oct. 1984), pp. 15–24.

[32] Jeffrey Goldsmith and John Salmon. "Automatic Creation of Object Hierarchies for Ray Tracing". In: *IEEE Computer Graphics and Applications* 7.5 (1987), pp. 14–20.

[33] Christiaan P. Gribble and Karthik Ramani. "Coherent Ray Tracing via Stream Filtering". In: *2008 IEEE Symposium on Interactive Ray Tracing*. IEEE. 2008, pp. 59–66.

[34] Yan Gu, Yong He, and Guy E. Blelloch. "Ray Specialized Contraction on Bounding Volume Hierarchies". In: *Computer Graphics Forum* 34.7 (2015), pp. 309–318.

[35] Yan Gu, Yong He, Kayvon Fatahalian, and Guy E. Blelloch. "Efficient BVH Construction via Approximate Agglomerative Clustering". In: *Proceedings of High Perform. Graphics*. 2013, pp. 81–88.

[36] Eric A. Haines and John R. Wallace. "Shaft Culling for Efficient Ray-Cast Radiosity". In: *Photorealistic Rendering in Computer Graphics: Proceedings of the Second Eurographics Workshop on Rendering*. Springer. 1994, pp. 122–138.

[37] Michal Hapala, Ondřej Karlík, and Vlastimil Havran. *When It Makes Sense to Use Uniform Grids for Ray Tracing*. May 2012.

[38] V. Havran, J. Bittner, and J. Žára. "Ray Tracing with Rope Trees". In: *Proceedings of 13th Spring Conference on Computer Graphics*. Budmerice, 1998, pp. 130–139.

[39] Vlastimil Havran and Jiří Bittner. “LCTS: Ray Shooting using Longest Common Traversal Sequences”. In: *Comput. Graph. Forum* 19.3 (2000), pp. 59–70.

[40] Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel. “On the Fast Construction of Spatial Data Structures for Ray Tracing”. In: *Proceedings of Symposium on Interactive Ray Tracing.* 2006, pp. 71–80.

[41] Jakub Hendrich, Daniel Meister, and Jiří Bittner. “Parallel BVH Construction using Progressive Hierarchical Refinement”. In: *Computer Graphics Forum.* Vol. 36. 2. Wiley Online Library. 2017, pp. 487–494.

[42] Jakub Hendrich, Adam Pospíšil, Daniel Meister, and Jiří Bittner. “Ray Classification for Accelerated BVH Traversal”. In: *Computer Graphics Forum.* Vol. 38. 4. Wiley Online Library. 2019, pp. 49–56.

[43] Qiming Hou, Xin Sun, Kun Zhou, C. Lauterbach, and D. Manocha. “Memory-Scalable GPU Spatial Hierarchy Construction”. In: *IEEE Trans. on Visualization and Comp. Graphics* 17.4 (2011), pp. 466–474.

[44] Yingsong Hu, Weijian Wang, Dan Li, Qingzhi Zeng, and Yunfei Hu. “Parallel BVH Construction using Locally Density Clustering”. In: *IEEE Access* 7 (2019), pp. 105827–105839.

[45] Warren Hunt, William R. Mark, and Don Fussell. “Fast and Lazy Build of Acceleration Structures from Scene Hierarchies”. In: *Proceedings of Symposium on Interactive Ray Tracing.* 2007, pp. 47–54.

[46] Tero Karras. “Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees”. In: *Proceedings of High Performance Graphics.* 2012, pp. 33–37.

[47] Tero Karras and Timo Aila. “Fast Parallel Construction of High-Quality Bounding Volume Hierarchies”. In: *Proceedings of High Performance Graphics.* 2013, pp. 89–100.

[48] Hiroharu Kato, Deniz Beker, Mihai Morariu, Takahiro Ando, Toru Matsuoka, Wadim Kehl, and Adrien Gaidon. “Differentiable Rendering: A Survey”. In: *arXiv preprint arXiv:2006.12057* (2020).

[49] Timothy L. Kay and James T. Kajiya. “Ray Tracing Complex Scenes”. In: *Computer Graphics (SIGGRAPH ’86 Proceedings)* 20.4 (1986), pp. 269–278.

[50] Andrew Kensler. “Tree Rotations for Improving Bounding Volume Hierarchies”. In: *Proceedings of Symposium on Interactive Ray Tracing.* 2008, pp. 73–76.

[51] Kevin Keul, Tilman Koß, F.L. Schröder, and Stefan Müller. “Combining Two-Level Data Structures and Line Space Precomputations to Accelerate Indirect Illumination.” In: *VISIGRAPP (1: GRAPP).* 2019, pp. 228–235.

[52] Kevin Keul, Stefan Müller, and Paul Lemke. “Accelerating Spatial Data Structures in Ray Tracing Through Precomputed Line Space Visibility”. In: *Computer Science Research Notes* (2016).

[53] Samuli Laine, Tero Karras, and Timo Aila. “Megakernels Considered Harmful: Wavefront Path Tracing on GPUs”. In: *Proceedings of the 5th High-Performance Graphics Conference*. 2013, pp. 137–143.

[54] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. “Fast BVH Construction on GPUs”. In: *Comput. Graph. Forum* 28.2 (May 7, 2009), pp. 375–384.

[55] Christian Lauterbach, Sung-Eui Yoon, Dinesh Manocha, and David Tuft. “RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs”. In: *IEEE Symposium on Interactive Ray Tracing*. Sept. 2006, pp. 39–46.

[56] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. “Differentiable Monte Carlo Ray Tracing Through Edge Sampling”. In: *ACM Transactions on Graphics (TOG)* 37.6 (2018), pp. 1–11.

[57] Daniel Meister. “Bounding Volume Hierarchies for High-Performance Ray Tracing”. PhD thesis. Czech Technical University, 2018.

[58] Daniel Meister, Jakub Bokšanský, Michael Guthe, and Jiří Bittner. “On Ray Reordering Techniques for Faster GPU Ray Tracing”. In: *Symposium on Interactive 3D Graphics and Games*. 2020, pp. 1–9.

[59] Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jiří Bittner. “A Survey on Bounding Volume Hierarchies for Ray Tracing”. In: *Computer Graphics Forum*. Vol. 40. 2. Wiley Online Library. 2021, pp. 683–712.

[60] Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. “Cache-Oblivious Ray Reordering”. In: *ACM Transactions on Graphics (TOG)* 29.3 (2010), pp. 1–10.

[61] Gordon Müller and Dieter W. Fellner. “Hybrid Scene Structuring with Application to Ray Tracing”. In: *Proceedings of the International Conference on Visual Computing (ICVC’99)*. 1999, pp. 19–26.

[62] Thomas Müller, Markus Gross, and Jan Novák. “Practical Path Guiding for Efficient Light-Transport Simulation”. In: *Computer Graphics Forum*. Vol. 36. 4. Wiley Online Library. 2017, pp. 91–100.

[63] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. “Large Ray Packets for Real-Time Whitted Ray Tracing”. In: *2008 IEEE Symposium on Interactive Ray Tracing*. IEEE. 2008, pp. 41–48.

[64] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. “GPU Computing”. In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[65] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. “A Survey of General-Purpose Computation on Graphics Hardware”. In: *Computer Graphics Forum*. Vol. 26. 1. Wiley Online Library. 2007, pp. 80–113.

[66] Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. “Pantaray: Fast Ray-traced Occlusion Caching of Massive Scenes”. In: *ACM Transactions on Graphics (TOG)* 29.4 (2010), pp. 1–10.

[67] Jacopo Pantaleoni and David Luebke. “HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry”. In: *Proceedings of High Performance Graphics*. 2010, pp. 87–95.

[68] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2016. URL: `https://pbrt.org/`.

[69] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. “Rendering Complex Scenes with Memory-Coherent Ray Tracing”. In: *SIGGRAPH 97 Conference Proceedings*. Aug. 1997, pp. 101–108.

[70] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. “Multi-Level Ray Tracing Algorithm”. In: *ACM Transactions on Graphics* 24.3 (2005), pp. 1176–1185.

[71] David Roger, Ulf Assarsson, and Nicolas Holzschuch. “Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU”. In: *Symposium on Rendering, Rendering Techniques*. 2007, pp. 99–110.

[72] Steven M. Rubin and Turner Whitted. “A 3-Dimensional Representation for Fast Rendering of Complex Scenes”. In: *SIGGRAPH ’80 Proceedings*. Vol. 14. 3. July 1980, pp. 110–116.

[73] S. Ruzika and M. M. Wiecek. “Approximation Methods in Multiobjective Programming”. In: *Journal of Optimization Theory and Applications* 126.3 (2005), pp. 473–501.

[74] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X. Sillion. “Conservative Volumetric Visibility with Occluder Fusion”. In: *Computer Graphics (Proceedings of SIGGRAPH 2000)*. 2000, pp. 229–238.

[75] Daniel Scherzer, Lei Yang, Oliver Mattausch, Diego Nehab, Pedro V. Sander, Michael Wimmer, and Elmar Eisemann. “Temporal Coherence Methods in Real-Time Rendering”. In: *Computer Graphics Forum*. Vol. 31. 8. Wiley Online Library. 2012, pp. 2378–2408.

[76] Robert Schmidtke and Kenny Erleben. “Chunked Bounding Volume Hierarchies for Fast Digital Prototyping using Volumetric Meshes”. In: *IEEE transactions on visualization and computer graphics* 24.12 (2017), pp. 3044–3057.

[77] Peter Schröder and Steven M. Drucker. “A Data Parallel Algorithm for Raytracing of Heterogeneous Databases”. In: *Proceedings of Computer Graphics Interface* (May 1992), pp. 167–175.

[78] Martin Stich, Heiko Friedrich, and Andreas Dietrich. “Spatial Splits in Bounding Volume Hierarchies”. In: *Proceedings of High Performance Graphics*. New Orleans, Louisiana, 2009, pp. 7–13.

[79] Seth Teller, Celeste Fowler, Thomas Funkhouser, and Pat Hanrahan. “Partitioning and Ordering Large Radiosity Computations”. In: *Proceedings of SIGGRAPH ’94*. July 1994, pp. 443–450.

[80] Ayush Tewari, Justus Thies, Ben Mildenhall, Pratul Srinivasan, Edgar Tretschk, Wang Yifan, Christoph Lassner, Vincent Sitzmann, Ricardo Martin-Brualla, Stephen Lombardi, et al. “Advances in Neural Rendering”. In: *Computer Graphics Forum*. Vol. 41. 2. Wiley Online Library. 2022, pp. 703–735.

[81] Ingo Wald. “Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.1 (2012), pp. 47–57.

[82] Ingo Wald. “On Fast Construction of SAH-based Bounding Volume Hierarchies”. In: *2007 IEEE Symposium on Interactive Ray Tracing*. IEEE. 2007, pp. 33–40.

[83] Ingo Wald, Carsten Benthin, and Solomon Boulos. “Getting Rid of Packets – Efficient SIMD Single-Ray Traversal using Multi-Branching BVHs”. In: *Proceedings of Symposium on Interactive Ray Tracing*. IEEE. 2008, pp. 49–57.

[84] Ingo Wald, Carsten Benthin, and Philipp Slusallek. “Distributed Interactive Ray Tracing of Dynamic Scenes”. In: *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. 2003, pp. 77–86.

[85] Ingo Wald, Solomon Boulos, and Peter Shirley. “Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies”. In: *ACM Trans. Graph.* 26.1 (Jan. 2007).

[86] Ingo Wald and Steven G. Parker. “Data Parallel Path Tracing with Object Hierarchies”. In: *Proceedings of High Performance Graphics* (2022).

[87] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. “Embree: A Kernel Framework for Efficient CPU Ray Tracing”. In: *ACM Transactions on Graphics* 33.4 (2014), 143:1–143:8.

[88] Bruce Walter, Kavita Bala, Milind Kulkarni, and Keshav Pingali. “Fast Agglomerative Clustering for Rendering”. In: *Proceedings of Symposium on Interactive Ray Tracing*. 2008, pp. 81–86.

[89] Turner Whitted. “An Improved Illumination Model for Shaded Display”. In: *Computer Graphics (Special SIGGRAPH ’79 Issue)*. Vol. 13. 3. Aug. 1979, pp. 1–14.

[90] Lei Xiao, Gang Mei, Salvatore Cuomo, and Nengxiong Xu. “Comparative Investigation of GPU-Accelerated Triangle-Triangle Intersection Algorithms for Collision Detection”. In: *Multimedia Tools and Applications* (2022), pp. 1–16.

[91] Sung-Eui Yoon and Dinesh Manocha. “Cache-Efficient Layouts of Bounding Volume Hierarchies”. In: *Computer Graphics Forum*. Vol. 25. 3. Wiley Online Library. 2006, pp. 507–516.

[92] T.S. Zoet and Jacco Bikker. “Accelerating Ray Tracing with Origin Offsets”. In: *Journal of WSCG*. Vol. 28. 1-2. 2020, pp. 1–8.

[93] Maurice van der Zwaan, Erik Reinhard, and Frederik W. Jansen. “Pyramid Clipping for Efficient Ray Traversal”. In: *Rendering Techniques '95*. 1995, pp. 1–10.

# Appendix A

# Author's Publications

The following research papers were first-authored by the author of this thesis and are related to the thesis.

## Journals with Impact Factor

- Jakub Hendrich, Daniel Meister, and Jiří Bittner. "Parallel BVH Construction using Progressive Hierarchical Refinement". In: *Computer Graphics Forum*. Vol. 36. 2. Wiley Online Library. 2017, pp. 487–494 [41]

  Cited in:

  – Carsten Benthin, Radoslaw Drabinski, Lorenzo Tessari, and Addis Dittebrandt. "PLOC++: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited". In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 5.3 (2022), pp. 1–13 [9]

  – Carsten Benthin, Sven Woop, Ingo Wald, and Attila T. Áfra. "Improved Two-Level BVHs using Partial Re-Braiding". In: *Proceedings of High Performance Graphics*. 2017, pp. 1–8 [10]

  – Michael Doyle and Karthik Vaidyanathan. *Apparatus and Method for Reduced Precision Bounding Volume Hierarchy Construction*. US Patent 11,321,910. May 2022 [21]

  – Petr Frantál. "Metody pro vrhání paprsku s hierarchiemi obálek na CPU". MA thesis. České vysoké učení technické v Praze, 2019 [25]

  – Tianhan Gao and Ying Li. "Real-Time Ray Tracing Algorithm for Dynamic Scene". In: *Proceedings of the 13th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2019)*. Springer. 2020, pp. 125–131 [28]

  – Yingsong Hu, Weijian Wang, Dan Li, Qingzhi Zeng, and Yunfei Hu. "Parallel BVH Construction using Locally Density Clustering". In: *IEEE Access* 7 (2019), pp. 105827–105839 [44]

  - Kevin Keul, Tilman Koß, F.L. Schröder, and Stefan Müller. "Combining Two-Level Data Structures and Line Space Precomputations to Accelerate Indirect Illumination." In: *VISIGRAPP (1: GRAPP).* 2019, pp. 228–235 [51]
  - Daniel Meister. "Bounding Volume Hierarchies for High-Performance Ray Tracing". PhD thesis. Czech Technical University, 2018 [57]
  - Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jiří Bittner. "A Survey on Bounding Volume Hierarchies for Ray Tracing". In: *Computer Graphics Forum.* Vol. 40. 2. Wiley Online Library. 2021, pp. 683–712 [59]
  - Robert Schmidtke and Kenny Erleben. "Chunked Bounding Volume Hierarchies for Fast Digital Prototyping using Volumetric Meshes". In: *IEEE transactions on visualization and computer graphics* 24.12 (2017), pp. 3044–3057 [76]
  - Ingo Wald and Steven G. Parker. "Data Parallel Path Tracing with Object Hierarchies". In: *Proceedings of High Performance Graphics* (2022) [86]
  - Lei Xiao, Gang Mei, Salvatore Cuomo, and Nengxiong Xu. "Comparative Investigation of GPU-Accelerated Triangle-Triangle Intersection Algorithms for Collision Detection". In: *Multimedia Tools and Applications* (2022), pp. 1–16 [90]

- Jakub Hendrich, Adam Pospíšil, Daniel Meister, and Jiří Bittner. "Ray Classification for Accelerated BVH Traversal". In: *Computer Graphics Forum.* Vol. 38. 4. Wiley Online Library. 2019, pp. 49–56 [42]

  Cited in:

  - Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jiří Bittner. "A Survey on Bounding Volume Hierarchies for Ray Tracing". In: *Computer Graphics Forum.* Vol. 40. 2. Wiley Online Library. 2021, pp. 683–712 [59]
  - T.S. Zoet and Jacco Bikker. "Accelerating Ray Tracing with Origin Offsets". In: *Journal of WSCG.* vol. 28. 1-2. 2020, pp. 1–8 [92]

## Unpublished Research

- Jakub Hendrich, Adam Pospíšil, and Jiří Bittner. *Dynamic ray batching using shafts*

# Appendix B

# Authorship Contribution Statement

**Parallel BVH Construction using Progressive Hierarchical Refinement** (Hendrich 70 %, Meister 15 %, Bittner 15 %) I implemented supporting code and a part of the method, performed most of the experiments and evaluation, created some figures, wrote part of the paper, and presented the paper.

**Ray Classification for Accelerated BVH Traversal** (Hendrich 60 %, Pospíšil 20 %, Meister 10 %, Bittner 10 %) I implemented the majority of the method, performed part of the experiments and evaluation, created most figures, and wrote and presented the paper.

**Dynamic Ray Batching using Shafts** (Hendrich 70 %, Pospíšil 20 %, Bittner 10 %) I came up with the initial idea, implemented the majority of the method, performed part of the experiments and evaluation, created most figures, and wrote the paper.