Higher Performance Traversal and Construction of Tree-Based Raytracing Acceleration Structures

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur

vorgelegt von
Dipl.-Inform. Dominik Maximilian Wodniok
geboren in Darmstadt

Referenten der Arbeit: Prof. Dr.-Ing. Michael Goesele
Technische Universität Darmstadt

Prof. Dr.-Ing. Carsten Dachsbacher
Karlsruhe Institute of Technology

Tag der Einreichung: 23. Juli 2018
Tag der Disputation: 19. September 2018

Darmstädter Dissertation, 2018
D 17

Abstract

Ray tracing is an important computational primitive used in different algorithms including collision detection, line-of-sight computations, ray tracing-based sound propagation, and most prominently light transport algorithms. It computes the closest intersections for a given set of rays and geometry. The geometry is usually modeled with a set of geometric primitives such as triangles or quadrangles which define a scene. An efficient ray tracing implementation needs to rely on an acceleration structure to decouple ray tracing complexity from scene complexity as far as possible. The most common ray tracing acceleration structures are kd-trees and bounding volume hierarchies (BVHs) which have an $O(\log n)$ ray tracing complexity in the number of scene primitives. Both structures offer similar ray tracing performance in practice. This thesis presents theoretical insights and practical approaches for higher quality, improved graphics processing unit (GPU) ray tracing performance, and faster construction of BVHs and kd-trees, where the focus is on BVHs.

The chosen construction strategy for BVHs and kd-trees has a significant impact on final ray tracing performance. The most common measure for the quality of BVHs and kd-trees is the surface area metric (SAM). Using assumptions on the distribution of ray origins and directions the SAM gives an approximation for the cost of traversing an acceleration structure without having to trace a single ray. High quality construction algorithms aim at reducing the SAM cost. The most widespread high quality greedy plane-sweep algorithm applies the surface area heuristic (SAH) which is a simplification of the SAM.

Advances in research on quality metrics for BVHs have shown that greedy SAH-based plane-sweep builders often construct BVHs with superior traversal performance despite the fact that the resulting SAM costs are higher than those created by more sophisticated builders.

Motivated by this observation we examine different construction algorithms that use the SAM cost of temporarily constructed SAH-built BVHs to guide the construction to higher quality BVHs. An extensive evaluation reveals that the resulting BVHs indeed achieve significantly higher trace performance for primary and secondary diffuse rays compared to BVHs constructed with standard plane-sweeping. Compared to the Spatial-BVH, a kd-tree/BVH hybrid, we still achieve an acceptable increase in performance. We show that the proposed algorithm has subquadratic computational complexity in the number of primitives, which renders it usable in practical applications.

An alternative construction algorithm to the plane-sweep BVH builder is agglomerative clustering, which constructs BVHs in a bottom-up fashion. It clusters primitives with a SAM-inspired heuristic and gives mixed quality BVHs compared to standard plane-sweeping construction. While related work only focused on the construction speed of this algorithm we examine clustering heuristics, which aim at higher hierarchy quality. We propose a fully SAM-based clustering heuristic which on average produces better performing BVHs compared to original agglomerative clustering.

The definitions of SAM and SAH are based on assumptions on the distribution of ray origins and directions to define a conditional geometric probability for intersecting nodes in kd-trees and BVHs. We analyze the probability function definition and show that the assumptions allow for an alternative probability definition. Unlike the conventional probability, our definition accounts for directional variation in the likelihood of intersecting objects from different directions. While the new probability does not result in improved practical tracing performance, we are able to provide an interesting insight on the conventional probability. We show that the conventional probability function is directly linked to our examined probability function and can be interpreted as covertly accounting for directional variation.

The path tracing light transport algorithm can require tracing of billions of rays. Thus, it can pay off to construct high quality acceleration structures to reduce the ray tracing cost of each ray. At the same time, the arising number of trace operations offers a tremendous amount of data parallelism. With CPUs moving towards many-core architectures and GPUs becoming more general purpose architectures, path tracing can now be well parallelized on commodity hardware. While parallelization is trivial in theory, properties of real hardware make efficient parallelization difficult, especially when tracing so called incoherent rays. These rays cause execution flow divergence, which reduces efficiency of SIMD-based parallelism and memory read efficiency due to incoherent memory access. We investigate how different BVH and node memory layouts as well as storing the BVH in different memory areas impacts the ray tracing performance of a GPU path tracer. We also optimize the BVH layout using information gathered in a pre-processing pass by applying a number of different BVH reordering techniques. This results in increased ray tracing performance.

Our final contribution is in the field of fast high quality BVH and kd-tree construction. Increased quality usually comes at the cost of higher construction time. To reduce construction time several algorithms have been proposed to construct acceleration structures in parallel on GPUs. These are able to perform full rebuilds in realtime for moderate scene sizes if all data completely fits into GPU memory. The sheer amount of data arising from geometric detail used in production rendering makes construction on GPUs, however, infeasible due to GPU memory limitations. Existing out-of-core GPU approaches perform hybrid bottom-up top-down construction which suffers from reduced acceleration structure quality in the critical upper levels of the tree. We present an out-of-core multi-GPU approach for full top-down SAH-based BVH and kd-tree construction, which is designed to work on larger scenes than conventional approaches and yields high quality trees. The algorithm is evaluated for scenes consisting of up to 1 billion triangles and performance scales with an increasing number of GPUs.

Zusammenfassung

Raytracing (dt. Strahlennachverfolgung) ist ein wichtiger Berechnungsbaustein der in verschiedenen Algorithmen wie der Kollisionserkennung, Sichtverbindungsberechnungen, die Raytracing-basierte Simulation von Schallausbreitung, so wie am prominentesten in Lichttransport-Algorithmen verwendet wird. Zu einer gegeben Menge an Strahlen und einem geometrischen Objekt berechnet Raytracing die nächste Schnittpunkte der Strahlen mit dem Objekt. Das geometrische Objekt ist dabei meist aus einer Menge geometrischer Primitive wie Drei- oder Vierecke zusammengesetzt, die eine Szene definieren. Eine effiziente Raytracing-Implementierung ist auf eine Beschleunigungsstruktur angewiesen, um die Raytracing-Komplexität so weit wie möglich von der Komplexität der Szene zu entkoppeln. Die gängigsten Beschleunigungsstrukturen sind *k-d-Bäume* und *BVHs* (von engl. *bounding volume hierarchy*, dt. Hüllkörper-Hierarchie), die eine Raytracing-Komplexität von $O(\log n)$ in der Anzahl von Szenenprimitiven aufweisen. Beide Strukturen bieten in der Praxis eine ähnliche Raytracing Leistung. Diese Dissertation präsentiert theoretische Einsichten und praktische Ansätze für eine höhere Qualität, verbesserte Grafikprozessor (GPU) Raytracing Leistung, und schnellere Konstruktion von k-d-Bäumen und BVHs, wobei der Fokus auf letzterem liegt.

Die gewählte Konstruktionsstrategie für k-d-Bäume und BVHs hat einen signifikanten Einfluss auf die letzten Endes erzielbare Raytracing Leistung. Das gebräuchlichste Maß für die Qualität einer Baum-basierten Beschleunigungsstruktur ist die *Oberflächenmetrik*. Unter der Verwendung von Annahmen bezüglich der Verteilung von Strahlenursprüngen und -richtungen liefert die Oberflächenmetrik eine Approximation für den Berechnungsaufwand zur Durchquerung der Beschleunigungsstruktur, ohne einen Strahl zu verfolgen. Verfahren zur Konstruktion hochwertiger Beschleunigungsstrukturen zielen darauf ab, die Oberflächenmetrik zu minimieren. Der Standardalgorithmus zur hochwertigen Konstruktion ist der gierige Sweep-Algorithmus, der die *Oberflächenheuristik* verwendet, welche eine Vereinfachung der Oberflächenmetrik darstellt. Forschung zu Qualitätsmetriken für BVHs hat ergeben, dass gierige Oberflächenheuristik-basierte Sweep-Algorithmen BVHs erzeugen können, die eine überlegene Raytracing Leistung erzielen, obwohl die resultierenden Oberflächenmetrik Kosten höher sind als die Kosten von BVHs, die mit weiterentwickelteren Verfahren konstruiert wurden. Basierend auf dieser Beobachtung untersuchen wir verschiedene Konstruktionsverfahren, welche die Kosten interimistisch mit der Oberflächenheuristik konstruierter BVHs verwenden, um BVHs höherer Qualität zu erzeugen. Eine umfangreiche Evaluierung zeigt, dass die resultierenden BVHs verglichen mit dem Standardverfahren für primäre und diffuse sekundäre Strahlen eine signifikant höhere Raytracing Leistung erzielen. Verglichen mit der *räumlichen* BVH, einem k-d-Baum/BVH-Hybriden, erreichen wir immer noch einen akzeptablen Leistungszuwachs. Wir zeigen, dass der vorgeschlagene Algorithmus eine subquadratische Berechnungskomplexität in der Anzahl der Szenenprimitive besitzt, welche ihn in der Praxis nutzbar macht.

Ein alternativer Konstruktionsalgorithmus zum Sweep-Verfahren ist das agglomerierende Clustern, ein Bottom-up BVH Konstruktionsverfahren. Es clustert Primitive anhand

einer Oberflächenmetrik-inspirierten Heuristik und gibt gemischte Ergebnisse im Vergleich zum Standardverfahren. Während sich weitere Forschung an diesem Verfahren auf die Konstruktionsgeschwindigkeit konzentriert hat, untersuchen wir Clusterungsheuristiken für ein höhere BVH Qualität. Wir schlagen eine voll Oberflächenmetrik-basierte Clusterungsheuristik vor, die im Durchschnitt bessere Ergebnisse erzielt als herkömmliches agglomerierendes Clustern.

Die Definitionen der Oberflächenmetrik und -heuristik basieren auf Annahmen zur Verteilung von Strahlenursprüngen und -richtungen, mit deren Hilfe eine bedingte Wahrscheinlichkeit für das Schneiden von Baumknoten definiert wird. Wir analysieren die Definition der Wahrscheinlichkeitsfunktion und zeigen, dass die Annahmen eine alternative Definition zulassen. Im Gegensatz zur herkömmlichen Definition beziehen wir die richtungsabhängige Variation der Schneidewahrscheinlichkeit von Objekten in die Herleitung mit ein. Während die alternative Definition in der Praxis nicht zu einer Zunahme der Raytracing Leistung geführt hat, sind wir in der Lage, eine interessante Einsicht bezüglich der herkömmlichen Definition zu liefern. Wir zeigen, dass die herkömmliche Definition direkt mit unserer alternativen Definition verbunden ist und so interpretiert werden kann, dass sie auf nicht offensichtliche Weise richtungsabhängige Variation berücksichtigt.

Der Path Tracing (dt. Pfadverfolgung) Lichttransport-Algorithmus kann das Nachverfolgen von Milliarden von Strahlen erfordern. Dafür kann es sich auszahlen, eine Beschleunigunsstruktur hoher Qualität zu bauen, um die Raytracing Kosten pro Stahl zu reduzieren. Gleichzeitig bietet die anfallende Anzahl an Strahlen einen hohen Grad an Datenparallelität. Mit der Entwicklung von CPUs in Richtung Vielkernprozessoren und GPUs zu Mehrzweck-Architekturen, kann Path Tracing nun einfach auf handelsüblicher Hardware parallelisiert werden. Während die Parallelisierung theoretisch trivial ist, machen Eigenschaften der Hardware eine effiziente Umsetzung schwierig, insbesondere dann, wenn so genannte inkohärente Strahlen verfolgt werden. Diese Strahlen verursachen Abweichungen im Ausführungsfluss, welche zu einer reduzierten Effizienz von SIMD-basierter Parallelität und der Speicherauslesung durch unzusammenhängenden Speicherzugriff führen. Wir untersuchen, wie sich verschiedene BVH und Knoten Speicherlayouts, so wie das Ablegen der Knoten in verschiedenen Speicherregionen auf die Raytracing Leistung eines GPU-basierten Path Tracers auswirken. Des Weiteren optimieren wir das BVH Layout anhand von Informationen aus einem Vorverarbeitungsschritt durch Anwendung verschiedener Baum Umordnungstechniken und erzielen dadurch eine höhere Raytracing Leistung.

Unser letzter Beitrag ist im Gebiet der schnellen Konstruktion von BVHs und k-d-Bäumen hoher Qualität. Eine höhere Qualität bedingt für gewöhnlich eine höhere Konstruktionszeit. Um diese zu reduzieren wurden mehrere Algorithmen zur parallelen Beschleunigungsstruktur Konstruktion auf GPUs vorgeschlagen. Sie sind für moderate Szenengrößen in der Lage volle Wiederaufbauten in Echtzeit zu vollziehen, wenn alle Daten in den GPU-Speicher passen. Allein die Menge an Daten, die durch den geometrischen Detailgrad im Produktions-Rendering anfällt, macht diese Verfahren durch den limitierten Grafikspeicher unanwendbar. Existierende Out-of-Core-GPU-Ansätze führen eine hybride Bottom-up/Top-down Konstruktion durch, welche eine reduzierte Qualität in den wichtigen oberen Baumebenen aufweisen. Wir präsentieren einen Multi-GPU-fähigen Out-of-Core-Ansatz zur vollen Oberflächenheuristik-Sweep-basierten Top-down BVH und k-d-Baum Konstruktion, welcher auf größere Szenen als konventionelle Ansätze ausgelegt ist und eine höhere Baumqualität erreicht. Das Verfahren wird mit Szenen, die aus bis zu eine Milliarde Dreiecken bestehen, evaluiert und skaliert mit zunehmender Anzahl an GPUs.

Acknowledgements

I wish to express my gratitude to all those who played their part in the development of this dissertation and also to the many people who influenced my research.

First and foremost, I would like to thank my supervisor Prof. Dr.-Ing. Michael Goesele for giving me the opportunity to work in his research group. He always encouraged me during my years at GCC. I shall never forget his support in scientific and personal matters. Further, I should like to thank Prof. Dr.-Ing. Carsten Dachsbacher for kindly agreeing to review this thesis.

Dr. Timo Aila kindly provided the source code he and his colleagues wrote for computation of the EPO metric. This helped to identify an issue in their code when reproducing their research results. I also want to thank Mr. Aila for helpful correspondence.

Several 3D models used for evaluations in this thesis have been provided courtesy of different people and institutions. I would like to thank Johnathan Good for *Babylon*, Anat Grynberg and Greg Ward for *Conference*, the University of Utah for *Fairy*, Samuli Laine for *Hairball*, the University of North Carolina for *Powerplant*, Guillermo M. Leal Llaguno for *San Miguel*, Marko Dabrovic for *Sibenik*, UC Berkeley for *Soda*, Ryan Vance for *Bubs*, *kescha* for *Rungholt*, Epic Games for *Epic*, and Frank Meinl for *Crytek-Sponza*. *Boeing 777* has been provided courtesy of David Kasik and the Boeing Corporation. *Atlas* and *David* have been provided courtesy of *The Digital Michelangelo Project*.

This research has been supported by the ’Excellence Initiative’ of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt.

Many thanks also go to my former colleagues at GCC with whom I spent a considerable amount of my time. In particular I want to thank Sven Widmer, Martin Heß, Fabian Langguth, Mate Beljan, Nicolas Weber, Daniel Thuerck, André Schulz, Simon Fuhrmann, Jens Ackermann, Nils Möhrle ;), Daniel Thul, Stefan Guthe, Carsten Haubold, Patrick Seemann, Michael Wächter, and Nelli for interesting on and off topic discussions, technical support, proof-reading papers, taming LaTeX, and simply having a good time. Thanks also go to our secretary Ursula Paeckel for her support.

I should also thank my colleagues Christoph Lämmerhirt, Joanna Allison, Luiz Carlos da Rocha Júnior, and Steen Müller at GritWorld for attending my defense test talk and giving feedback.

I also want to thank my family and friends for their support. Marian Wieczorek eagerly volunteered to read parts of this thesis and provided valuable feedback. Special thanks go to my brother Remigius. Little did he know he pushed me on this track a long while ago.

Last but not least I thank my wife Andrea and children Lily and Zoe. My time at GCC surely had taken its toll on us, for little Lily once asked whether Daddy lived at work. You gave me the balance and serenity I needed for this chapter in my life. Your patience was invaluable.

To Andrea, Lily, and Zoe

Contents

Chapter 1

Introduction

Contents

Ray tracing or *ray casting* (which also has non-synonymous uses) is an important computational primitive used in different fields. It can be broken down to the process of computing the intersection of a ray with a virtual environment or object. The geometry of the object or environment is modeled with geometric primitives such as triangles or quadrangles.

Based on principles from geometric acoustics Krokstad et al. [1968] presented an early computational approach for ray tracing based sound propagation. Acoustic design software such as *EASE*[1] or *Odeon*[2] use ray tracing for *auralisation*. It simulates what a listener hears in a virtual environment taking into account the environment's geometry, surface properties and sound sources. This is e.g. important when designing the acoustics of auditoriums, concert halls, or churches. Game engines such as Unity[3] or the Unreal Engine[4] provide a ray tracing API for custom ray tracing queries. This e.g. allows to implement artificial intelligence of agents in virtual environments which require line-of-sight computations to be able to "see" with the same constraints like a human being, or to simulate projectile collision. There are also ray tracing-based collision detection and resolution algorithms for deformable bodies [Hermann et al. 2008, Lehericey et al. 2013].

Most prominently ray tracing is featured in light transport algorithms in computer graphics. Appel [1968] was the first to propose to trace rays to a virtual light source to create shadows in images of machine parts. A major step towards photorealism in computer graphics was the introduction of the *rendering equation* by Kajiya [1986]. He also proposed the *path tracing* algorithm as a solution to the rendering equation. Using very similar physics to auralisation it simulates what observers see taking into account the environment's geometry, surface properties and light sources. Figure 1.1 shows examples

---

[1] `http://ease.afmg.eu/`

[2] `http://www.odeon.dk/`

[3] `https://unity3d.com`

[4] `https://www.unrealengine.com`

Figure 1.1: Examples of scenes rendered with path tracing. The top two images were rendered using PBRT (`http://www.pbrt.org`). The bottom image shows a rendering of a multi-view stereo reconstruction of a sculpture by Ferdinand Seeboeck. Landscape scene courtesy of Jan-Walter Schliep, Burak Kahraman, and Timm Dapper. Kitchen scene courtesy of Jay-Artist under CC BY 3.0 (`https://creativecommons.org/licenses/by/3.0/`). Geometry changes were made. Elsbeth model courtesy of GCC, TU Darmstadt.

of images rendered with path tracing. Billions of rays had to be traced to generate those images. Sound propagation and light transport spend a substantial amount of time finding primitive intersections for millions to billions of rays with millions to billions of primitives.

An efficient ray tracing implementation uses an *acceleration structure*. The purpose of ray tracing acceleration structures is to reduce the number of primitives a ray must test to get a sublinear computational complexity in the number of scene primitives. The most common ray tracing acceleration structures are kd-trees and bounding volume hierarchies (BVHs). They allow to find an intersection in logarithmic time w.r.t. the number of primitives. The process of using acceleration structures to find an intersection point is called traversal.

1.1 Problem Statement

The focus of this dissertation is on two major intertwined problem fields of ray tracing acceleration structures: traversal performance and construction speed. Traversal performance depends on scene characteristics, the construction strategy, and characteristics of the input rays. In case of parallelized traversal ray characteristics have a strong influence on performance.

Depending on how well the chosen construction strategy adapts to the distribution of primitives, traversal performance can vary drastically. Strategies which aim at reducing the so called surface area metric (SAM) give best results (MacDonald and Booth [1989,1990]). SAM uses a result from geometric probability to compute an estimate for the cost of traversing an acceleration structure beforehand without tracing a single ray. Karras and Aila [2013] proposed the most efficient algorithm for constructing BVHs with minimum SAM cost. Unfortunately, this algorithm has a runtime complexity of $\Omega(2^n)$ and a space complexity of $O(n2^n)$ in the number of input primitives. Though this is sufficient for the specific problem of Karras and Aila [2013] it is highly unpractical for BVH construction in general. A scene consisting of just 64 primitives would already require at least 128 exabytes of auxiliary memory. According to Havran [2000], minimum-SAM cost construction of kd-trees is *NP*-hard.

Because of the unfeasibility of minimum-SAM cost, BVH and kd-tree construction research only focuses on approximative solutions. The standard SAM-based construction approach reduces the metric by applying the surface area heuristic (SAH), which is an approximation of the SAM. Unfortunately, construction is most expensive with SAH-based strategies. Minutes to hours are possible for huge scenes with millions to billions of primitives. This can be an issue for interactive ray tracing applications with dynamic content if frequent rebuilds of the acceleration structure are needed and a given time budget must not be exceeded. Aila et al. [2013] have shown that the accuracy of the SAM as a traversal performance predictor is scene dependent for BVHs. They introduced the end-point-overlap metric (EPO), which, in combination with SAM, produces more accurate predictions. As a consequence minimum-SAM construction is not sufficient for BVHs as EPO has to be optimized as well. So far no algorithm has been published which aims at reducing EPO.

The sheer amount of primitives can also be a problem for construction if they do not fit into memory. This can easily become a problem if one wants to leverage the massive parallelism of graphics processing units (GPUs) for construction due to their comparatively small graphics memory. Out-of-core construction techniques have to be applied in this case. Finding other construction strategies, which result in even better performing hierarchies are still a topic of ongoing research.

Another aspect of traversal performance is traversal itself. Theoretically, ray tracing is embarrassingly parallel as different rays can be traced independently from each other. On multi-core systems it can be implemented in a straightforward manner by letting each thread process its own batch of rays. Further parallelization can be achieved by taking advantage of *single-instruction-multiple-data* (SIMD) capabilities of multi-core architectures or the massive parallelism of many-core architectures such as GPUs. Efficient parallelization on SIMD architectures is, however, much harder when batches contain so called incoherent rays. These rays have widely varying origins and/or directions. Tracing incoherent rays requires traversing different paths through the acceleration structure, resulting in incoherent memory accesses since different nodes are traversed and different primitives are tested. Additionally, control flow diverges between rays which lowers SIMD efficiency. As incoherent rays form an absolute majority in sound propagation and light transport algorithms they pose a serious challenge.

1.2 Contributions

In this thesis we present several contributions on BVHs and kd-trees for ray tracing which partly have been published at conferences and in a journal where noted. Chapters based on publications have mostly been edited for layout and extended discussions. Where applicable text passages have been shortened or replaced by references into the background chapter. Our main contributions ordered from theoretical to practical are as follows:

- An alternative geometric probability function for surface area heuristic-based acceleration structure construction which includes *directional* variation of the probability of intersecting a ray with a convex body in its derivation. While computation of this probability is unpractical, we show that under certain assumptions the conventional geometric probability includes our proposed function, increasing the credibility of the conventional approach.

  Chapter 4

- A SAM-based bottom-up BVH construction algorithm, which improves on standard bottom-up agglomerative clustering-based construction. The SAM-based approach allows to naturally include the collapsing of subtrees into a leaf node as a clustering decision, instead of applying it as a post-process.

  Chapter 6

- An approach for reducing the forecasting error of the SAM-EPO ray tracing performance predictor from Aila et al. [2013] which also enables more accurate predictions for primary rays.

  [Wodniok and Goesele 2017], Chapter 5

- A BVH construction algorithm that produces BVHs with better average performance than state-of-the-art methods. Complexity analysis of our algorithm reveals subquadratic runtime in the number of primitives which renders it practical.

  [Wodniok and Goesele 2016, Wodniok and Goesele 2017], Chapter 5

- An analysis of GPU cache behavior when tracing incoherent rays in real-world scenarios. We investigate how different bounding volume hierarchy (BVH) and node memory layouts as well as storing the BVH in different memory areas impacts the ray tracing performance of a GPU path tracer. The BVH layout is optimized using information gathered in a pre-processing pass applying a number of different BVH reordering techniques.

  [Wodniok et al. 2013], Chapter 7

- An efficient out-of-core multi-GPU algorithm for BVH and kd-tree construction that allows the memory footprint of the output tree as well as the geometry exceed graphics memory. Data is assumed to fit into system memory, though. Construction applies SAH right from the beginning and does not rely on quality degrading pre-clustering of geometry.

  Chapter 8 [5]

1.3 Thesis Overview

The remainder of this thesis is structured as follows:

Chapter 2: Background
First, we give an introduction on ray tracing with path tracing, a global illumination algorithm, as an example application. Then, we proceed with ray tracing acceleration structures, which are necessary to make ray tracing efficient. We discuss bounding volume hierarchies and kd-trees, and their construction. We also discuss bounding volume types and argue why we use axis aligned bounding boxes. Further, we give a more in-depth introduction to the surface area metric and acceleration structure construction with the surface area heuristic. Both topics, in one way or the other, are important for the remaining chapters. We also shortly introduce the EPO metric, which plays an important role for Chapter 5 and Chapter 6.

Chapter 3: GPU Hardware Platform
The underlying GPU algorithms and techniques in Chapter 7 and Chapter 8 are implemented with the NVIDIA CUDA API for GPU programming. This chapter gives an introduction on the computation model behind CUDA and performance critical aspects of NVIDIA GPUs.

Chapter 4: On the Geometric Probability Function of the Surface Area Metric
This chapter elaborates more on the geometric probability, which is at the core of the surface area metric and surface area heuristic introduced in Chapter 2. We show that

---

[5] This chapter started as an unpublished paper in cooperation with Carsten Haubold, André Schulz, Nicolas Weber, Sven Widmer, and Michael Goesele. Sven Widmer and Carsten Haubold developed the first prototype of the underlying out-of-core multi-GPU middleware, which the author developed further. The middleware is not part of this dissertation. Work on out-of-core kd-tree construction started as Nicolas Weber's master's thesis [Weber 2013] under co-supervision of the author and Sven Widmer. The author completely reworked the kd-tree implementation and developed the BVH construction approach. For the evaluation André Schulz developed the implementation of the hybrid BVH-construction approach based on the author's BVH construction implementation. All experimental evaluation and discussion has been conducted by the author. The whole text of this chapter as well as of the unpublished paper has been written by the author.

the assumptions underlying this probability also allow for the definition of an alternative conditional probability, which includes directionally varying object intersection likelihood in its derivation. We further show how this alternative probability is connected to the conventionally used probability.

Chapter 5: Temporary Subtree SAH-based Bounding Volume Hierarchy Construction
BVHs constructed with the standard greedy top-down SAH-based construction algorithm have been shown to implicitly reduce the EPO metric. In this chapter we present a greedy top-down algorithm, which constructs temporary BVHs with the standard construction process for both sides of a partition candidate and uses the SAM cost of those temporary trees to compute a candidate cost. This implicitly guides construction towards partition candidates with lower EPO. An extensive evaluation reveals superior tree quality but also limits of the EPO metric for performance prediction.

Chapter 6: An SAM-Driven Approach to Agglomerative Clustering
Bottom-up BVH construction with so called agglomerative clustering uses a clustering metric which is loosely based on the SAM. While follow-up work only focused on improved construction speed this chapter presents a clustering metric which directly aims at reducing the SAM cost and also measurably improves the BVH quality of the original algorithm.

Chapter 7: Cache-Optimized BVH GPU Memory Layouts for Tracing Incoherent Rays
Tracing incoherent rays in the context of the massive parallelism of GPUs poses a challenge with respect to control flow divergence and incoherent memory access. This chapter analyzes the effects of incoherent memory access on GPU cache behavior which is caused by batches of incoherent rays from a GPU path tracer. We investigate how different bounding volume hierarchy (BVH) and node memory layouts as well as storing the BVH in different memory areas impacts the ray tracing performance. Further, we optimize the BVH layout using information gathered in a pre-processing pass applying a number of different BVH reordering techniques.

Chapter 8: Multi-GPU Out-of-Core Top-Down SAH-based kd-Tree and BVH Construction
Though out-of-core CPU and GPU ray tracing has been investigated, there has been less attention on efficient out-of-core acceleration structure construction. Out-of-core acceleration structures are typically assumed as given. The small amount of available related work sacrifices hierarchy quality in a geometry pre clustering step to partition a scene into handleable chunks. SAH-based construction is then applied on these chunks. In this chapter we investigate out-of-core multi-GPU acceleration structure construction that applies SAH right from the beginning and does not rely on quality degrading pre-clustering of geometry.

Chapter 9: Final Summary and Discussion
This chapter gives a summary and discussion of our contributions and techniques presented in Chapters 4 to 8.

Chapter 10: Future Work
Finally, we conclude this dissertation by identifying several open questions and challenges, which motivate future research.

Chapter 2

Background

Contents

In this chapter we give an in-depth introduction to ray tracing and ray tracing acceleration structures. We start by introducing ray tracing from a ray tracing-based global illumination perspective, one of the major applications of ray tracing. Then, we proceed with ray tracing acceleration structures, which are important to make ray tracing practical. Our focus is on traversal and construction of two tree-based acceleration structure: bounding volume hierarchies and kd-trees.

2.1 Ray Tracing in Computer Graphics

We define *ray tracing* (or alternatively *ray casting*) as the operation of finding the intersection of a ray r with a set of geometrical primitives $\mathcal{P}$ in $\mathbb{R}^3$. A ray $r : \mathbb{R} \to \mathbb{R}^3$ is defined as the function

$$r(t) = \mathbf{o} + t\mathbf{d} \tag{2.1}$$

with ray parameter $t \in \mathbb{R}$, origin vector $\mathbf{o} \in \mathbb{R}^3$, and normalized direction vector $\mathbf{d} \in \mathcal{S}^2$. Here, $\mathcal{S}^2$ is the 2-sphere

$$\mathcal{S}^2 = \left\{ \mathbf{d} \in \mathbb{R}^3 \mid \|\mathbf{d}\|_2 = 1 \right\}. \tag{2.2}$$

We are usually interested in finding the closest intersection point $\mathbf{x} \in \mathbb{R}^3$ for a ray and the primitives in $\mathcal{P}$. For convenience, we define the function $\mathbf{x} = \mathtt{intersect}(\mathbf{o}, \mathbf{d}, \mathcal{P})$ for this operation which has the set of primitives and a ray's origin and direction as arguments. Without loss of generality we assume primitives to be triangles though other primitives such as spheres or higher order surfaces are possible. A point on a triangle can be parametrized with barycentric coordinates as a convex combination of the triangle vertices $\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\}, \mathbf{v}_i \in \mathbb{R}^3$. This can be written in matrix notation as

$$\mathbf{p}(\lambda_0, \lambda_1, \lambda_2) = \begin{pmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 \end{pmatrix} \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \end{pmatrix}, \quad \lambda_i \in [0,1], \sum \lambda_i = 1 \tag{2.3}$$

with the triangle vertices as columns. Bearing in mind that barycentric coordinates always sum to one and setting $\lambda_1 = \mu$ and $\lambda_2 = \nu$ this can be written as

$$\mathbf{p}(\mu, \nu) = \begin{pmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 \end{pmatrix} \begin{pmatrix} 1 - \mu - \nu \\ \mu \\ \nu \end{pmatrix} = \mathbf{v} + \begin{pmatrix} \mathbf{e}_1 & \mathbf{e}_2 \end{pmatrix} \begin{pmatrix} \mu \\ \nu \end{pmatrix}, \tag{2.4}$$

where $\mathbf{v} = \mathbf{v}_0$, $\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0$ and $\mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0$. That is, the triangle can also be described with a base vertex $\mathbf{v}$ and edge vectors $\mathbf{e}_1$ and $\mathbf{e}_2$, which point from the base vertex to the other two vertices. To intersect a ray with a triangle we have to insert Equation 2.1 into Equation 2.4:

$$\mathbf{v} + \begin{pmatrix} \mathbf{e}_1 & \mathbf{e}_2 \end{pmatrix} \begin{pmatrix} \mu \\ \nu \end{pmatrix} = \mathbf{o} + t\mathbf{d}. \tag{2.5}$$

Rearranging gives:

$$\begin{pmatrix} -\mathbf{d} & \mathbf{e}_1 & \mathbf{e}_2 \end{pmatrix} \begin{pmatrix} t \\ \mu \\ \nu \end{pmatrix} = \mathbf{o} - \mathbf{v}. \tag{2.6}$$

To solve for the ray parameter and barycentric coordinates we simply have to invert the matrix $A = \begin{pmatrix} -\mathbf{d} & \mathbf{e}_1 & \mathbf{e}_2 \end{pmatrix}$ which consists of the triangle edges and the ray direction. The determinant of this matrix can be formulated as the triple scalar product

$$\det A = \mathbf{d} \cdot (\mathbf{e}_2 \times \mathbf{e}_1) \tag{2.7}$$

The geometric interpretation of this determinant is that there is a solution to the system if the ray is not parallel to the triangle. That is, the ray direction $\mathbf{d}$ is not orthogonal to the triangle normal $\mathbf{e}_2 \times \mathbf{e}_1$. Using Cramer's rule reformulated with cross products the inverse of A is

$$A^{-1} = \frac{1}{\det A} \begin{pmatrix} \mathbf{e}_1 \times \mathbf{e}_2 & \mathbf{d} \times \mathbf{e}_2 & \mathbf{e}_1 \times \mathbf{d} \end{pmatrix}^T. \tag{2.8}$$

Multiplying Equation 2.6 with A^{-1} we get the ray and triangle parameters:

$$\begin{pmatrix} t \\ \mu \\ \nu \end{pmatrix} = \frac{1}{\mathbf{d} \cdot (\mathbf{e}_2 \times \mathbf{e}_1)} \begin{pmatrix} \mathbf{e}_1 \times \mathbf{e}_2 & \mathbf{d} \times \mathbf{e}_2 & \mathbf{e}_1 \times \mathbf{d} \end{pmatrix}^T \cdot (\mathbf{o} - \mathbf{v}). \tag{2.9}$$

For the intersection to be valid it must be inside the triangle. That is μ and ν must be in $[0,1]$. For the third implicit barycentric coordinate to be one the sum of μ and ν must be at most one. This can already be checked during computation to reject an intersection early. The presented intersection test is essentially identical to the approach from Möller and Trumbore [1997].

2.1.1 Ray Tracing-Based Global Illumination

In the context of computer graphics, ray tracing is most popularly featured in global illumination algorithms, where the set of primitives $\mathcal{P}$ defines a *scene*. Global illumination algorithms create photo-realistic images by simulating the interaction of light with the environment. We give a brief introduction to *path tracing* as an example for a prominent and simple ray tracing-based global illumination algorithm which showcases two important basic ray tracing operations. Other more sophisticated and more complicated ray tracing-based global illumination algorithms such as bidirectional path tracing (Veach and Guibas [1995] and Lafortune and Willems [1993]), photon mapping (Jensen [1996]), and Metropolis light transport (Veach and Guibas [1997]) exist which can better handle certain light situations and/or have faster convergence. For an in depth introduction to these algorithms and global illumination we refer to Pharr et al. [2016] or Dutre et al. [2006].

To create an image with path tracing a virtual observer has to be placed in the scene. In case of a virtual camera or person the image corresponds to the camera's sensor or person's retina. Possibly taking into account its internal system of lenses, this observer creates so called *primary rays* which are associated with pixels of the image. A primary surface intersection point $\mathbf{x}$ is computed for each primary ray. For this, we are interested in finding the closest intersection point of all surfaces the ray intersects. *Closest point determination* is the first of two important basic ray tracing operations. What the observer sees at $\mathbf{x}$ is determined by the *light transport equation* or *rendering equation*, which is at the core of all global illumination algorithms. The rendering equation has been introduced by Kajiya [1986]. For a point $\mathbf{x} \in \mathbb{R}^3$ on a surface and an outgoing direction ω_o at this point it is defined as the sum of two terms:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + L_r(\mathbf{x}, \omega_o). \tag{2.10}$$

It describes that the outgoing light L_o (more specifically *radiance*) leaving surface point $\mathbf{x}$ in direction ω_o depends on the directly *emitted* light L_e at $\mathbf{x}$ and the fraction of incoming light L_r at $\mathbf{x}$ that is *reflected* into direction ω_o. L_r is the integral

$$L_r(\mathbf{x}, \omega_o) = \int_{\omega_i \in \Omega^+} f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)\, \omega_i \cdot \mathbf{n}\, d\omega_i. \tag{2.11}$$

It integrates all incoming light L_i from all directions ω_i in the positive hemisphere $\Omega^+ = \{\omega \in \mathcal{S}^2 \mid \omega \cdot \mathbf{n} \geq 0\}$ defined by the surface normal $\mathbf{n}$ at $\mathbf{x}$ that is reflected into direction ω_o. At surface point $\mathbf{x}$ the bidirectional reflectance distribution function (BRDF) f_r describes the ratio of differential irradiance from a differential incident light cone around ω_i to exitant differential radiance in direction ω_o for all pairs $(\omega_i, \omega_o) \in \Omega^+ \times \Omega^+$ [Nicodemus 1965]. Thus, it is responsible for the material appearance at a point. The incoming light $L_i(\mathbf{x}, \omega_i)$ itself can be the outgoing emitted and/or reflected light $L_o(\mathbf{x}', \omega_o' = -\omega_i)$ from another surface point $\mathbf{x}' = \texttt{intersect}(\mathbf{x}, \omega_i, \mathcal{P})$ in direction ω_i from $\mathbf{x}$ which again can be found with ray tracing. In general the rendering equation cannot be solved analytically and numerical methods have to be applied. Figure 2.1 further depicts the geometric constellation of L_r.

Kajiya [1986] proposed the *path tracing* algorithm as a possible solution, which is based on Monte Carlo integration. Given some multivariate function $f(\mathbf{x})$ defined on

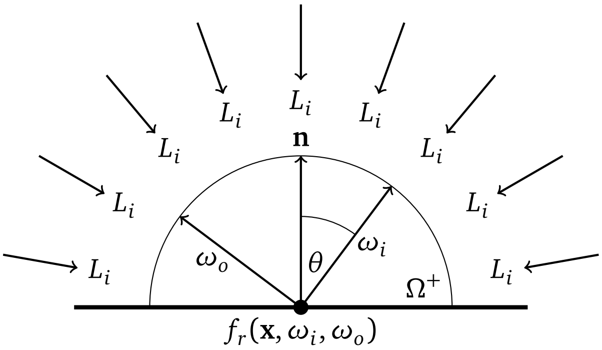


Figure 2.1: Depiction of the integral in the rendering equation. For the outgoing direction ω_o and all directions ω_i in the positive hemisphere Ω^+ defined by the surface normal $\mathbf{n}$ it integrates the product of incoming light L_i, the BRDF f_r at $\mathbf{x}$, and $\cos\theta_i = \omega_i \cdot \mathbf{n}$.

some domain $\mathcal{X}$, Monte Carlo integration approximates the integral

$$I = \int_{\mathcal{X}} f(\mathbf{x}) d\mathbf{x} \tag{2.12}$$

with a random variable $\hat{I}$ defined as

$$I \approx \hat{I} = \frac{1}{N} \sum_{i=1}^{N} \frac{f(\mathbf{x}_i)}{p(\mathbf{x}_i)}, \tag{2.13}$$

which draws $N \in \mathbb{N}$ samples $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ with some probability density function $p(\mathbf{x})$. The expected value of $\hat{I}$ is the solution to the original integral itself:

$$\mathtt{E}\left[\hat{I}\right] = I. \tag{2.14}$$

To approximate the rendering equation with Monte Carlo integration we have to evaluate the integrand with randomly generated direction samples. This results in the approximation

$$\begin{aligned} L_o(\mathbf{x}, \omega_o) &\approx L_e(\mathbf{x}, \omega_o) + \frac{1}{N} \sum_{j=1}^{N} \frac{f_r(\mathbf{x}, \omega_i^j, \omega_o)}{p(\omega_i^j)} \left(\omega_i^j \cdot \mathbf{n}\right) L_i(\mathbf{x}, \omega_i^j) \\ &= L_e(\mathbf{x}, \omega_o) + \frac{1}{N} \sum_{j=1}^{N} r(\mathbf{x}, \omega_i^j, \omega_o) L_i(\mathbf{x}, \omega_i^j), \end{aligned} \tag{2.15}$$

where we introduced the function $r(\mathbf{x}, \omega_i, \omega_o) = \frac{f_r(\mathbf{x}, \omega_i, \omega_o)}{p(\omega_i)} (\omega_i \cdot \mathbf{n})$ for brevity. Evaluating the rendering equation with one sample gives

$$L_o(\mathbf{x}, \omega_o) \approx L_e(\mathbf{x}, \omega_o) + r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i). \tag{2.16}$$

As already mentioned the incoming light L_i itself depends on light, which is emitted and/or reflected from another surface point $\mathbf{x}' = \mathtt{intersect}(\mathbf{x}, \omega_i, \mathcal{P})$. Expanding L_i with a one sample approximation gives

$$\begin{aligned} L_o(\mathbf{x}, \omega_o) \approx & L_e(\mathbf{x}, \omega_o) + r(\mathbf{x}, \omega_i, \omega_o) \left(L_e(\mathbf{x}', -\omega_i) + r(\mathbf{x}', \omega_i', -\omega_i) L_i(\mathbf{x}', \omega_i')\right) \\ = & L_e(\mathbf{x}, \omega_o) + \\ & r(\mathbf{x}, \omega_i, \omega_o) L_e(\mathbf{x}', -\omega_i) + \\ & r(\mathbf{x}, \omega_i, \omega_o) r(\mathbf{x}', \omega_i', -\omega_i) L_i(\mathbf{x}', \omega_i'). \end{aligned} \tag{2.17}$$

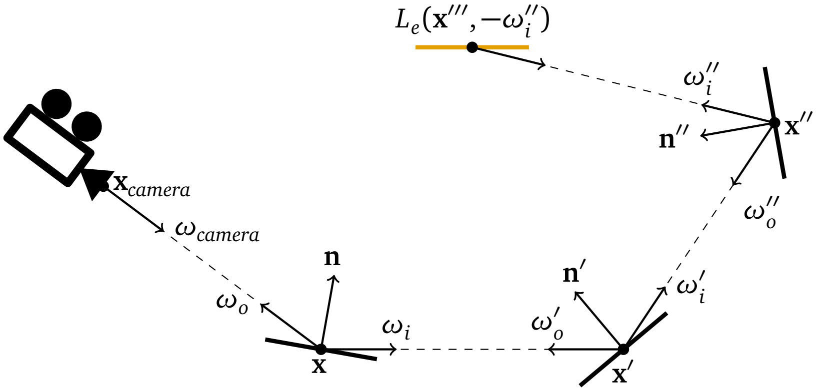


Figure 2.2: Depiction of path construction in the path tracing algorithm. Starting with a sampled camera ray with origin $\mathbf{x}_{camera}$ and direction ω_{camera} ray tracing is used to find the intersection points $(\mathbf{x}, \mathbf{x}', \mathbf{x}'', \mathbf{x}''')$. The incoming light directions at each intersection point are randomly sampled according to a probability density function, which depends on local properties at the intersection point.

This approximation gathers light emitted from $\mathbf{x}$ and the light from some other point $\mathbf{x}'$ which is reflected at $\mathbf{x}$. Recursively repeating this one-sample expansion of L_i (see Figure 2.2) constructs a series of raytraced intersection points $(\mathbf{x}, \mathbf{x}', \mathbf{x}'', \mathbf{x}''', \dots)$, which form a path, hence the name path tracing. Simply terminating construction of a path after a pre-determined number of path vertices introduces an unrecoverable error into the estimate. Proper path termination probabilistically terminates a path using *Russian roulette*. With this technique the expected value of Equation 2.15 is still the solution to the rendering equation, but at the cost of possibly higher variance of the random variable.

So far the path construction process finds a light source by accidently intersecting an emitting surface. To more effectively generate paths to light sources incoming directions can also be sampled by directly sampling a point on a light source for each path vertex. This is illustrated in Figure 2.3. As can be seen from the figure the sampled point on the light source might be occluded with respect to a path vertex. Thus, a so called *line-of-sight-*, *visibility-*, or *occlusion*-test has to be performed which is the second of two important basic ray tracing operations. It simply checks if there is *any* intersection up to a maximum ray parameter t_{max}. In case of the light sample t_{max} is the distance from the path vertex to the light sample position. The associated query rays are called *visibility*, *shadow*, or *occlusion* rays. Closest point and line-of-sight computation are summarized in Figure 2.4.

2.1.2 Ray Tracing Acceleration Structures

Usually we want to find intersections for a set of rays $\mathcal{R}$. The set of primary rays in path tracing from the previous section is an example for this. A naïve ray tracing algorithm finds the closest intersection of a ray with the set $\mathcal{P}$ of primitives in $O(|\mathcal{P}|)$ time by iteratively intersecting all primitives. For the whole set $\mathcal{R}$ this complexity is $O(|\mathcal{R}||\mathcal{P}|)$. If either the number of rays $|\mathcal{R}|$ to test or the number of primitives $|\mathcal{P}|$ is very small the naïve algorithm is acceptable. In practice $|\mathcal{R}|$ *and* $|\mathcal{P}|$ can be very large at the same time. Creating a path traced image with a minimum Ultra HD[1] resolution of 3840×2160 already requires tracing

---

[1] `http://www.itu.int/rec/R-REC-BT.2020-2-201510-I/en`

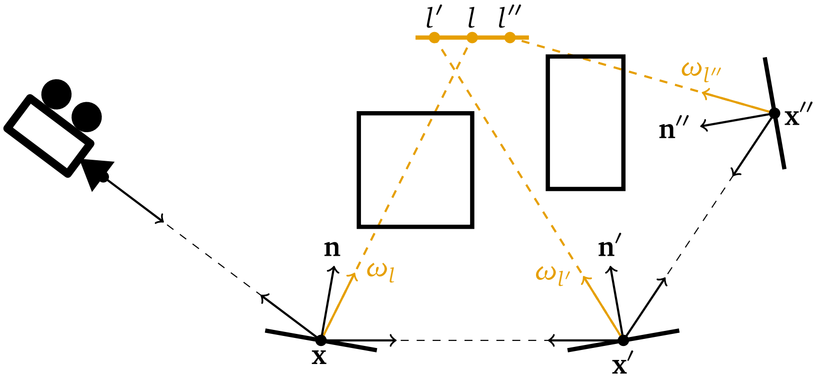


Figure 2.3: Depiction of light sampling in path tracing with an occluding object. To more efficiently find paths to light sources for every intersection point a light sample is created (yellow dots). For every pair of a path vertex and a light sample a light direction can be computed to evaluate the BRDF at the path vertex. As there might by occluding surfaces between the path vertex and the light sample visibility rays have to be traced.

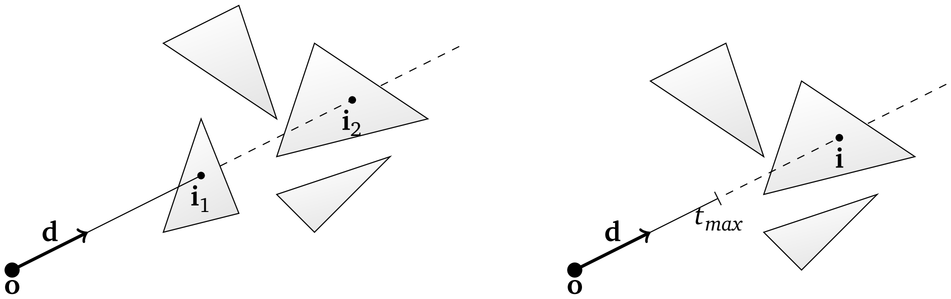


Figure 2.4: Ray tracing with a ray and a set of triangle primitives. The ray has origin $\mathbf{o}$ and direction $\mathbf{d}$. In the left image the ray has two intersection points, $\mathbf{i}_1$ and $\mathbf{i}_2$. Closest point determination returns $\mathbf{i}_1$ as $\mathbf{i}_2$ is farther away. Line-of-sight computation simply returns *that* a ray has at least one intersection. Though the occlusion ray in the right image intersects a triangle at $\mathbf{i}$ the computation returns that there is no intersection, as the ray parameter for $\mathbf{i}$ is larger than the upper bound t_{max} on the ray parameter.

of $8,294,400$ primary rays, assuming one sample per pixel. Several times more rays are needed when constructing paths for each primary ray and using pixel supersampling for anti-aliasing. Thus, with higher numbers of samples per pixel the number of rays $|R|$ can be in the order of trillions. The number of primitives $|\mathcal{P}|$ of CGI shots in movies can also be in the order of billions (e.g. Pantaleoni et al. [2010]).

One way to handle the $O(|\mathcal{R}||\mathcal{P}|)$ cost are variance reduction techniques for ray tracing-based global illumination, which allow to reduce the required number of samples, and thus rays to trace. We refer to Pharr et al. [2016] for an introduction to variance reduction techniques such as stratified or importance sampling in the context of ray tracing. This way the computational cost is still linear in the number of rays and primitives. In the context of this thesis we focus on so called *ray tracing acceleration structures*, which aim at reducing the $O(|\mathcal{P}|)$ complexity for a single ray. The main principle behind these structures is to safely reject or ignore whole groups of primitives, which cannot be intersected by a given ray. Acceleration structures can be classified into *grid-based* and *tree-based* structures. Tree-based structures can again be classified into *object*- or *space*-partitioning structures. The main contributions of this thesis are based on the two most widespread tree-based structures: *bounding volume hierarchies (BVH)* and *kd-trees*. We proceed with a thorough introduction of these structures and afterwards briefly elaborate on alternative acceleration structures.

2.2 Bounding Volume Hierarchies

The bounding volume hierarchy (BVH) has been introduced by Rubin and Whitted [1980]. It is a so called object partitioning structure as it partitions the set $\mathcal{P}$ of scene primitives. It is defined as a tree with inner nodes $\mathcal{I}$ and leaf nodes $\mathcal{L}$. Only leaves store references to primitives. Every node stores a bounding volume which fully contains all primitives in its subtree. This gives a hierarchy of bounding volumes (hence the name of the structure). Conceptually, when a ray does not intersect a node's bounding volume the contents of the whole subtree of the node can be ignored. Arbitrary and also varying branching factors are possible. Implementations usually use a fixed branching factor. In the context of SIMD ray tracing Ernst and Greiner [2008], Wald et al. [2008], and Dammertz et al. [2008] simultaneously proposed *multi-branching BVHs* (MBVH) where the branching factor corresponds to the SIMD width. Resulting branching factors can range from 4 (SSE), 8 (AVX), and 16 (AVX512) for Intel's SIMD hardware implementations on CPUs [Intel 2017] to 32 and higher for graphics processing units (GPUs). Our focus is on binary BVHs for two reasons. Construction-wise MBVHs are a special case of binary BVHs as they are constructed from the latter by collapsing nodes. Secondly, in the context of ray tracing with NVIDIA GPUs Aila and Laine [2009] observed that branching factors higher than 2 showed no clear performance benefit and even started to be detrimental to performance for factors higher than 4. They noted that with a GPU ray traversal implementation similar to the approach from Wald et al. [2008] (who used 16) higher branching factors might be beneficial. An efficient GPU adaptation was not possible at that time, though.

Usually BVHs are full trees. That is, all nodes have either 2 (inner nodes) or 0 (leaves) children. Full trees have the property that $|\mathcal{I}| = |\mathcal{L}| - 1$ holds (see e.g. Mehta and Sahni [2004]). Thus the size $|\mathcal{N}|$ of the set of all nodes $\mathcal{N} = \mathcal{I} \cup \mathcal{L}$ can be expressed with respect to $|\mathcal{L}|$:

$$\begin{aligned} |\mathcal{N}| &= |\mathcal{I}| + |\mathcal{L}| \\ &= |\mathcal{L}| - 1 + |\mathcal{L}| \\ &= 2|\mathcal{L}| - 1. \end{aligned} \tag{2.18}$$

Every primitive in $\mathcal{P}$ is referenced exactly once by some leaf $l \in \mathcal{L}$. A leaf node contains at least one primitive reference. Thus, for a scene with $|\mathcal{P}|$ primitives the corresponding BVH has at most $|\mathcal{P}|$ leaf nodes. Combined with Equation 2.18 the upper bound of the number of BVH nodes for a scene with $|\mathcal{P}|$ primitives is:

$$|\mathcal{N}| \leq 2|\mathcal{P}| - 1. \tag{2.19}$$

The bounding volumes of nodes can be simple objects such as spheres or boxes. A more in depth discussion of bounding volume types is provided in Section 2.2.2. Bounding volumes of leaf nodes are large enough that they fully contain all referenced primitives.

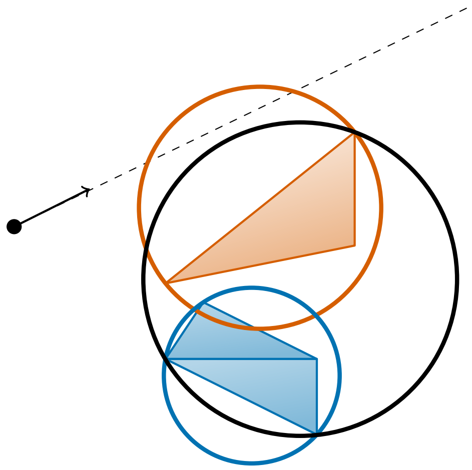


Figure 2.5: Example for a constellation where the spherical bounds of a parent node (black) do not fully contain the bounds but the primitives of its children (red and blue). The depicted ray does not intersect the parent bounds. This already rules out the possibility of intersecting any primitive in the parents subtree, though the ray intersects the bounds of the red child.

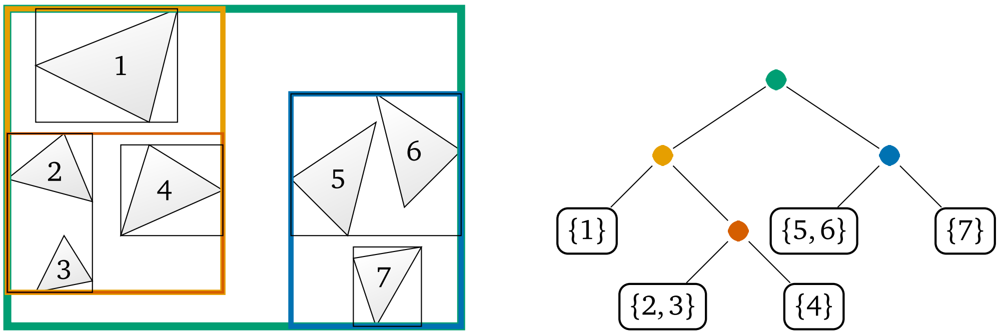


Figure 2.6: Example BVH for a scene with seven triangles. Axis aligned bounding boxes are used as the bounding volumes for nodes.

Bounding volumes of inner nodes also have to at least fully contain all primitives referenced in their subtrees. Depending on the chosen bounding volume type it can happen that an inner node's bounding volume does not fully contain the bounds of its children. By itself this is not an issue. If a ray misses the bounds of an inner node but intersects the bounds of one or more children, the subtree still can be safely skipped as the parent bounds test already ruled out any primitive intersection in the subtree. This situation is depicted in Figure 2.5. As we will discuss in Section 2.5 a downside of such parent bounds is, that the surface area metric cost for a BVH with such bounds cannot easily be evaluated. The bounding volume of a node can potentially overlap with the bounding volumes of other non-ancestral BVH nodes. Low node bounds overlap is beneficial for ray tracing performance as will be explained in the next section. An example BVH with axis aligned bounding boxes as bounding volumes is depicted in Figure 2.6.

2.2.1 Traversal

Pseudocode for BVH traversal is provided in Algorithm 1. Traversal for a query ray $r(t)$ keeps track of an upper ray parameter limit t_{max} and the currently processed node $n_{current} \in \mathcal{N}$. Intersections with bounding volumes or primitives which have a ray parameter larger than t_{max} are rejected. t_{max} is initialized with infinity while $n_{current}$ is set to the root. Additionally, an auxiliary *traversal stack* is needed which can temporarily store nodes for later processing. The stack is empty in the beginning. If $n_{current}$ is an inner node the bounds of its children nodes are tested for intersection with r. If both children are intersected the farther one is pushed onto the traversal stack and traversal continues with the closer child. If only one child is intersected traversal simply continues with this child. If no child has been intersected the next $n_{current}$ is popped from the stack. If $n_{current}$ is a leaf node the closest intersection with the contained primitives is computed. In case of no primitive intersection again the next $n_{current}$ is popped from the stack. In case of an intersection traversal cannot terminate if the traversal stack is not empty; as bounding volumes of nodes can overlap with other nodes there might be a closer intersection in one of the subtrees of the nodes on the traversal stack. Thus, at least all nodes on the stack still have to be processed and traversal has to continue by popping the next $n_{current}$ from the stack. But at least on primitive intersection we can "shorten" the query ray by setting t_{max} to the current closest intersection. This way subtrees of nodes on the traversal stack will likely be culled. Benthin et al. [2012] proposed to additionally store the distance to a pushed node when pushing nodes on the traversal stack. This allows to rapidly discard popped stack entries by comparing the popped node distance to the current t_{max}.

Any time the traversal stack is empty when it is queried traversal is terminated and the intersection is returned if one has been found. What intersection information exactly is returned depends on the application.

By always processing the closer child first when both children have been intersected we get an approximate front-to-back traversal. When children of inner nodes on average split the list of contained primitives in half, an intersection can be found in $O(\log_2 |\mathcal{P}|)$. With suboptimal bounding volumes or degenerated hierarchies almost all nodes might have to be visited. As according to Equation 2.19 the number of nodes is bound by $|\mathcal{P}|$ this can result in a complexity of $O(|\mathcal{P}|)$, which is the complexity of naïve ray tracing.

Traversal for occlusion rays is only slightly different from closest-hit traversal. Traversal terminates immediately if any intersection is found in the parameter bounds interval $[0, t_{max}]$. Thus, no front-to-back traversal is needed.

2.2.2 Bounding Volumes

Theoretically arbitrary geometrical solids can be used as bounding volumes. Several aspects can be important when choosing bounding volume types.

A first aspect is fast ray/bounding volume intersection. Arvo and Kirk [1989] argue that convexity of bounding volumes is a prerequisite for fast intersection computation as it guarantees that there are at most two intersection points.

A second aspect is *tightness of the fit* or *bounding efficiency*. The most compact convex bounding volume for an object is its convex hull, which has the smallest area *and* the smallest volume of all convex bounding volumes at the same time. In the context of collision detection applications bounding efficiency is the ratio of the *volume* of the bounded object to the volume of the bounding volume (Vogiannou et al. [2010]). As we will see

Algorithm 1: Pseudocode for BVH traversal.

```
input  : ray                                   // ray we have to intersect with the scene
input  : root                                  // root node of the scene BVH
output: iIdx                                   // index of the closest intersected primitive
iIdx ← InvalidIdx                              // Set intersection index to invalid index
push ( stack, NIL)                             // Init stack with invalid node
$t_{max} \leftarrow \infty$                    // Init ray parameter limit
$n_{current}$ ← root                           // Start traversal at root
while $n_{current} \neq$ NIL do                // Traversal loop
    if is_inner_node ( $n_{current}$ ) then    // Handle inner nodes
        ( leftBounds, rightBounds ) ← children_bounds ( $n_{current}$ )    // Get children bounds
        // Compute ray intersection parameter intervals
        tIntervals ← get_intersection_intervals ( ray, leftBounds, rightBounds )
        if intersected_both_children (tIntervals, $t_{max}$) then
            // Get farther away child and push it on the stack
            $c_{far}$ ← get_far_child ( $n_{current}$, tIntervals )
            push ( stack, $c_{far}$)
            // Next node is the closer child
            $n_{current}$ ← get_near_child ( $n_{current}$, tIntervals )
        else if intersected_one_child (tIntervals, $t_{max}$) then
            // Just proceed with the lone intersected child
            $n_{current}$ ← get_intersected_child ( $n_{current}$, tIntervals )
        else
            // No intersection. Get the next node from the stack.
            $n_{current}$ ← pop ( stack )
        end
    else
        // Find closest leaf primitive intersection and update $t_{max}$ and iIdx if possible
        intersect_leaf_primitives ( $n_{current}$, ray, $t_{max}$, iIdx)
        // Get next node
        $n_{current}$ ← pop ( stack )
    end
end
```

in Section 2.5 the probability of a ray intersecting a convex volume is approximately proportional to the volume's surface area when a uniform ray distribution of infinitely far away originating rays is assumed. The surface area of the convex hull of an object is a tight lower bound on the surface area of convex bounding volumes. Thus, for ray tracing we can define bounding efficiency $\eta(B, O)$ of a convex bounding volume B for a bounded object O as

$$\eta(B,O) = \frac{\mathtt{Area}(\mathtt{ConvexHull}(O))}{\mathtt{Area}(B)}. \tag{2.20}$$

Any convex bounding volume other than the convex hull has a larger surface area and will thus have a bounding efficiency of less than one. The reciprocal of $\eta(B, O)$ describes how many times more likely it is to intersect B compared to intersecting the convex hull of O.

Additional important aspects can be the number of parameters needed for parametrization and fast computation of the bounding volume from a set of input primitives. Finally, it might be worth considering how difficult it is to compute bounds for a pair of bounds of the same type. This is relevant when bounds have to be propagated bottom-up in a hierarchy. We continue with a discussion of several bounding volume types and conclude with a bounding efficiency comparison.

Convex Hull According to the definition of η in Equation 2.20 the convex hull is the convex bounding volume with highest bounding efficiency. The Quickhull algorithm from Barber et al. [1996] allows to compute the convex hull for a set of n points in $O(n \log n)$. The resulting convex hull is a polygonal mesh where the polygons can have a variable number of edges. Thus a varying number of parameters is needed to describe the convex hull. Assuming that all polygons are turned into a set of triangles and using the *Euler characteristic* of convex polyhedra discovered by Euler [1758] the number of convex hull triangles can be shown to be at most $2n-4$. Intersecting those triangles for larger n is too expensive and unfortunately needs itself an acceleration structure in case of convex hulls with many faces.

Sphere The sphere is the simplest bounding volume in terms of intersection and parametrization. It can be parametrized with the sphere center $\mathbf{c} \in \mathbb{R}^3$ and sphere radius $r \in \mathbb{R}_0^+$. Intersection with a ray $r(t)$ is done by inserting into the implicit sphere equation

$$\|r(t) - \mathbf{c}\|_2^2 - r^2 = 0. \tag{2.21}$$

Solving the resulting quadratic equation for t gives

$$t_{1,2} = -\mathbf{d} \cdot (\mathbf{o} - \mathbf{c}) \pm \sqrt{(\mathbf{d} \cdot (\mathbf{o} - \mathbf{c}))^2 - (\mathbf{o} - \mathbf{c})^2 + r^2}. \tag{2.22}$$

If the discriminant is negative there is no intersection. In practice spheres have lower bounding efficiency compared to other bounding volume types. It also turns out that computation of a tight sphere is difficult. Gärtner [1999] proposed the fastest exact method, which requires about 30 milliseconds for computing a tight sphere for about 10^6 points on our setup (Intel i7 7700K, 4.2 GHz). Ritter [1990] proposed a widespread, simple, and about three orders of magnitude faster approximative $O(n)$ algorithm. According to Larsson [2008] spheres computed with Ritter's method can have an about 50% larger surface area than optimal. Larsson [2008] himself proposed an approximative algorithm which is slightly slower than Ritter's method but more often produces close to optimal spheres which empirically have a 16% larger surface area in the worst case. Computing a bounding sphere from two bounding spheres is straight forward, but the resulting sphere has a lower bounding efficiency than when directly computing a bounding sphere for objects contained in the original spheres. This problem is depicted in Figure 2.7.

Axis Aligned Bounding Box An axis aligned bounding box (AABB) is a cuboid which restricts faces to be parallel to the coordinate system axes planes. This allows to parametrize an AABB with a lower bound vector $\mathbf{a} \in \mathbb{R}^3$ and an upper bound vector $\mathbf{b} \in \mathbb{R}^3$ with $a_d \leq b_d, d \in \{x, y, z\}$, which define three bounding intervals $[a_d, b_d]$.

Intersection computation is more expensive than sphere intersection. Though the first AABB intersection test has implicitly been introduced by Rubin and Whitted [1980] we follow the more efficient approach from Kay and Kajiya [1986]. Each bounding interval

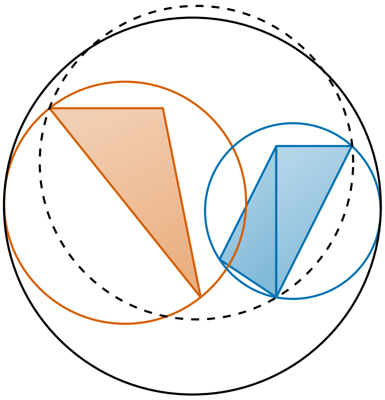


Figure 2.7: Tight bounding spheres for two sets of primitives (red and blue). The black sphere, which contains the red and blue sphere, is not tight with respect to the union of the two primitive sets. While the more bounding efficient dashed sphere tightly contains the union of the two point sets, it does not fully contain the bounding spheres of each point set. It still could be used as the bounds for a hypothetical parent node.

$[a_d, b_d]$ can be interpreted as an axis aligned slab, which is bounded by two bounding planes with plane equations $-x_d + a_d = 0$ and $x_d - b_d = 0$. Inserting a ray into these equations we get the two ray intersection parameters $t_1^d = (a_d - o_d)/d_d$ and $t_2^d = (b_d - o_d)/d_d$ per slab. Sorting both parameters per slab we get the slab entry point parameter $t_{min}^d = \inf\{t_1^d, t_2^d\}$ and slab exit point parameter $t_{max}^d = \sup\{t_1^d, t_2^d\}$. The AABB entry point parameter t_{min} is the maximum of the slab entry parameters while the exit point parameter t_{max} is the minimum of the slab exit parameters. That is, $t_{min} = \sup\{t_{min}^x, t_{min}^y, t_{min}^z\}$ and $t_{max} = \inf\{t_{max}^x, t_{max}^y, t_{max}^z\}$. If t_{max} is smaller than t_{min} the ray missed the AABB.

For a set $\left\{\mathbf{p}^1, \ldots, \mathbf{p}^n\right\}, \mathbf{p}^i \in \mathbb{R}^3$ of n points a tight AABB can be computed in $O(n)$. We simply have to find the minimum and maximum component value of the points for each dimension to determine the bounding intervals:

$$[a_d, b_d] = \left[\inf\left\{p_d^1, \ldots, p_d^n\right\}, \sup\left\{p_d^1, \ldots, p_d^n\right\}\right]. \tag{2.23}$$

Computing the tight AABB for a pair (B_1, B_2) of AABBs simply requires to find the component wise minima of the lower bounds and maxima of the upper bounds:

$$[a_d, b_d] = \left[\inf\left\{a_d^{B_1}, a_d^{B_2}\right\}, \sup\left\{b_d^{B_1}, b_d^{B_2}\right\}\right]. \tag{2.24}$$

As opposed to spheres the resulting AABB is also tight with respect to points contained in B_1 and B_2, if the later are also tight. Denoting the sets of point cloud coordinates in dimension $d \in \{x, y, z\}$ of the point clouds from B_1 and B_2 with $\mathcal{P}_d^{B_1}$ and $\mathcal{P}_d^{B_2}$, respectively, we can show this by inserting Equation 2.23 in Equation 2.24:

$$\begin{aligned}
[a_d, b_d] &= \left[\inf\left\{a_d^{B_1}, a_d^{B_2}\right\}, \sup\left\{b_d^{B_1}, b_d^{B_2}\right\}\right] \\
&= \left[\inf\left\{\inf \mathcal{P}_d^{B_1}, \inf \mathcal{P}_d^{B_2}\right\}, \sup\left\{\sup \mathcal{P}_d^{B_1}, \sup \mathcal{P}_d^{B_2}\right\}\right] \\
&= \left[\inf\left\{\mathcal{P}_d^{B_1} \cup \mathcal{P}_d^{B_2}\right\}, \sup\left\{\mathcal{P}_d^{B_1} \cup \mathcal{P}_d^{B_2}\right\}\right]
\end{aligned} \tag{2.25}$$

Thus, the tight AABB for B_1 and B_2 is also tight w.r.t. the union of the point clouds of B_1 and B_2.

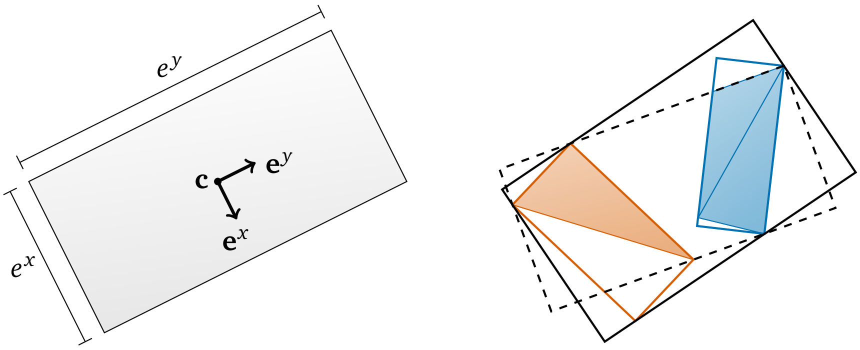


Figure 2.8: *Left:* Parametrization of an OBB in 2D with center **c**, orientation vectors $\mathbf{e}^x$ and $\mathbf{e}^x y$, and corresponding box extents e^x and e^y. *Right:* Tight oriented bounding boxes for two sets of primitives (red and blue). The black box, which contains the red and blue box, is not tight with respect to the union of the two primitive point. While the more bounding efficient dashed box tightly contains the union of the two primitive sets, it does not fully contain the bounding boxes of each primitive set. It still can be used as the bounds for a hypothetical parent node

Oriented Bounding Box Oriented bounding boxes (OBB) have higher bounding efficiency than AABBs as they have more degrees of freedom. The parametrization chosen by Gottschalk et al. [1996] consists of the box center $\mathbf{c} \in \mathbb{R}^3$, an orthonormal basis for the box orientation with basis vectors $\{\mathbf{e}^x, \mathbf{e}^y, \mathbf{e}^z\}, \mathbf{e}^d \in \mathcal{S}^2$, and corresponding box extents $\{e^x, e^y, e^z\}, e^i \in \mathbb{R}_+$. The parametrization is depicted in Figure 2.8.

Gottschalk et al. [1996] showed how to compute an optimal fitting OBB from the convex hull of input points. Therefore, the analytical mean and covariance matrix of the convex hull's continuous surface points is computed. The normalized eigenvectors of the covariance matrix define the orthonormal basis of the OBB. The extents and the center can be computed by projecting the input points onto the basis vectors. For objects with rotational symmetry such as cylinders, cones or spheres there is no unique solution.

OBB intersection is reduced to AABB intersection by first transforming the query ray into the local coordinate system of the OBB using the center and orientation of the box. In the local coordinate system the OBB is an AABB which is centered at the origin. Thus, like for AABBs the slab test from Kay and Kajiya [1986] can be used. As the box is centered the bounding interval for dimension d is $[-\frac{e^d}{2}, \frac{e^d}{2}]$. Because of the initial ray transformation OBB intersection is more expensive than AABB intersection.

Computing a tight OBB from two OBBs requires to go through the construction process with point clouds but using the union of the corners of the boxes as input. Like for spheres the resulting box can have a lower bounding efficiency than when directly computing a tight box for objects contained in the original boxes. This issue is also depicted in Figure 2.8.

k-DOP In three dimensional space a k-DOP is a *discrete oriented polyhedron*, where the orientations of the polyhedron faces are restricted to k distinct directions $\mathbf{d}^i \in \mathcal{S}^2, i \in \{1, \ldots, k\}$. They were introduced to ray tracing by Kay and Kajiya [1986]. k-DOPs are implicitly assumed to be convex. As such a k-DOP can be interpreted as the intersection of up to $2k$ half-spaces. Each direction $\mathbf{d}^i$ is associated with a bounding interval $[a_{\mathbf{d}^i}, b_{\mathbf{d}^i}]$

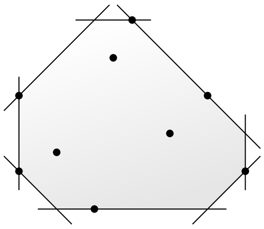


Figure 2.9: Tight 4-DOP for a set of points in two dimensions. The set of directions is $\{(1,0),(1,1),(0,1),(-1,1)\}$. The interior (gray) is defined by the intersection of the half-spaces (lines).

which is computed from the projection of an object onto $\mathbf{d}^i$. For a set $\mathcal{P} = \left\{\mathbf{p}^1, \ldots, \mathbf{p}^n\right\}, \mathbf{p}^i \in \mathbb{R}^3$ of n points the set of projections $\mathcal{P}_\perp^{\mathbf{d}}$ for a direction $\mathbf{d}$ is $\mathcal{P}_\perp^{\mathbf{d}} = \{\mathbf{p} \cdot \mathbf{d} \mid \mathbf{p} \in \mathcal{P}\}$. Thus, the bounding interval for direction $\mathbf{d}^i$ is

$$[a_{\mathbf{d}^i}, b_{\mathbf{d}^i}] = [\inf \mathcal{P}_\perp^{\mathbf{d}^i}, \sup \mathcal{P}_\perp^{\mathbf{d}^i}]. \tag{2.26}$$

An example k-DOP in two dimensions is depicted in Figure 2.9. AABBs can be interpreted as 3-DOPs, where the directions are restricted to the coordinate system axes. We can see that Equation 2.23 for AABBs is a special case of Equation 2.26. Constructing a k-DOP from two other k-DOPs B_1 and B_2 with the same set of directions is done by simply applying Equation 2.24 from AABBs. Like for AABBs, the resulting k-DOP is also automatically tight w.r.t. the content of B_1 and B_2, when the latter are also tight. Intersection computation is completely analogous to AABB intersection, just with k oriented slabs instead of the fixed three slabs. The plane intersection tests are more expensive in general as the directions can be more general and have up to three non-zero components.

Unlike for the other discussed bounding volumes it is difficult to compute the surface area of a k-DOP, as the surface is only implicitly defined by the intersection of half-spaces. Barber et al. [1996] showed how to compute an explicit mesh for the half-space intersection. Figure 2.10 describes how to compute the surface area of a k-DOP with this approach. As can be seen the whole process is quite expensive considering that we are only interested in the surface area. This makes BVH construction strategies based on bounding volume surface areas quiet slow when using k-DOPs.

2.2.3 Bounding Efficiency Comparison

For a comparison of the bounding efficiency of the discussed bounding volumes we used all 1814 models of the *Princeton Shape Benchmark* from Shilane et al. [2004]. This model collection has a wide diversity of real life objects. Examples are humans, insects, birds, mammals, cars, planes, ships, furniture, tools, or buildings. As a k-DOP we used the 13-DOP from Klosowski et al. [1998]. The baseline for their 13-DOP is an AABB equivalent 3-DOP. They added the four space diagonals $(\pm1, 1, \pm1)$ and the six quadrant diagonals $(1, \pm1, 0)$, $(1, 0, \pm1)$, and $(0, \pm1, 1)$. The original benchmark data is essentially axis aligned. This is the optimal case for AABBs. Thus, for fairness we also evaluated average efficiency for a modified version of the data, where all models are exemplary rotated around the $(1,1,1)^T$ axis by 60 degree. The average bounding efficiency for unrotated and rotated models is depicted in Figure 2.11.

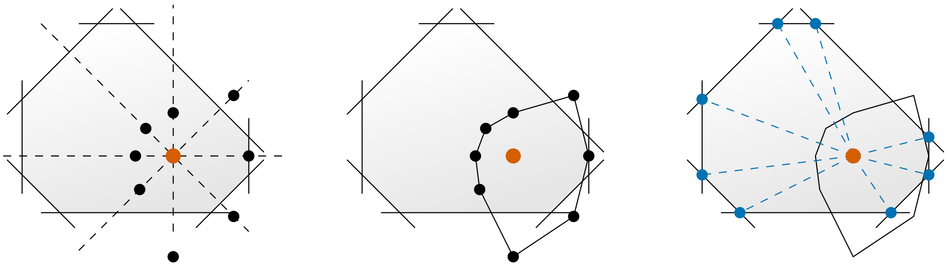


Figure 2.10: Exemplary computation of the surface area of a 4-DOP by computing an explicit mesh for the intersection (gray) of half-spaces (lines) as described by Barber et al. [1996]. The set of directions is $\{(1,0),(1,1),(0,1),(-1,1)\}$. *Left:* In a first step the k-DOP is transformed to the origin by choosing an arbitrary interior point (red). Then, the half-space planes are transformed into dual three dimensional points (black) by dividing the plane normals by the plane offsets. *Middle:* Then the convex hull of the dual points is computed. *Right:* From the plane equations of the faces of the convex hull another set of dual points (blue) is created by dividing the plane normals by the plane offsets. This new set of points corresponds to the intersection points of the original half-spaces. The connectivity of the intersection points can be extracted from the convex hull of the first set of dual points. The surface area of the k-DOP can then be computed from the explicit mesh.

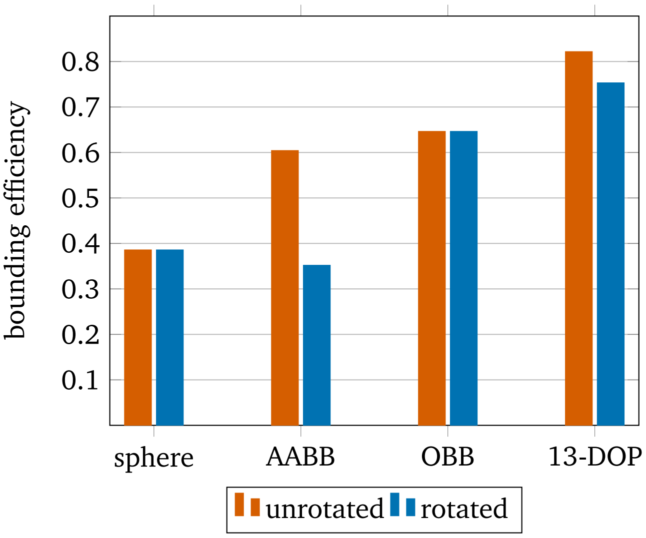


Figure 2.11: Average bounding efficiency (see Equation 2.20) of spheres, axis aligned bounding boxes (AABB), oriented bounding boxes (OBB), and 13-DOPs computed for all 1814 models of the *Princeton Shape Benchmark*. Models are either left as is (unrotated), or rotated around the $(1,1,1)^T$ axis by 60 degree (rotated).

The rotation invariant bounding efficiency of OBBs and spheres is immediately visible. But the efficiency of OBBs is more than 50% higher than the efficiency of spheres. While the 13-DOP shows some rotational variance it clearly has the highest bounding efficiency with roughly $75\% - 82\%$. AABBs show highest variance in bounding efficiency. For unrotated data AABBs come close to the bounding efficiency of OBBs. In the rotated case they slightly fall behind spheres. Still we advocate usage of AABBs as they combine more positive aspects than the other bounding volumes. Computation of tight bounds is cheapest and simplest for AABBs. Computing an AABB for two AABBs is very cheap and the result is guaranteed to be tight w.r.t. the primitives contained in the original bounds. While the latter is also true for 13-DOPs computation of the combined 13-DOP is four times more expensive, as an AABB is essentially a 3-DOP. In Section 2.5.3 and Section 2.5.4 we will see that surface area heuristic-based BVH construction requires incremental construction of tight bounds for a growing set of primitives. To compute a tight fitting OBB it is required to iterate over all primitives in the current set each time a primitive is added which makes construction quite expensive. Further, the intersection test of a 13-DOP requires 26 plane intersections of which 20 are not axis aligned. This is several times more expensive than the intersection of the 6 axis-aligned planes of the AABB and has to be contrasted with the 1.35 and 2.15 times higher bounding efficiency of the 13-DOP in the unrotated and rotated case, respectively. Finally, computation of the surface area of AABBs is straightforward while for the 13-DOP a surface mesh has to be computed first.

2.2.4 Number of BVHs for a Scene

Before we introduce basic BVH construction strategies we elaborate on the total number of different BVHs which can be constructed for a given set of primitives. Swapping the order of children of an inner node has no effect on traversal when a ray always enters the closer child first. Thus, two BVHs are only different if they cannot be transformed into each other by a series of children swaps. According to Karras and Aila [2013] the number of different BVHs with $n \in \mathbb{N}$ leaves is

$$\#BVHs \text{ with } n \text{ leaves} = (2n-3)!!, \tag{2.27}$$

where $x!!$ is the double factorial function. As $2n-3$ is always an odd number the double factorial in this case is $x!! = x(x-2)(x-4)\cdot\ldots\cdot 3\cdot 1$. When there is only one primitive in every leaf this is also the number of different BVHs for a set of primitives. As Karras and Aila [2013] neither provided a derivation nor a source for this result, and we ourselves were unable to find a source we provide our own derivation.

Our approach is based on an incremental BVH construction procedure which iteratively adds leaves with primitives to a BVH. We assume that leaves contain one primitive. Leaves are either added at an inner node or at a leaf of an existing BVH. Both cases are conceptually identical. The targeted node is removed and attached to a new parent node along with the leaf to add. Then the parent node is inserted at the original position of the target node. In case of an inner node the whole subtree is moved. Whether the new leaf is added to the new parent node as the left or right child is irrelevant as both results are equivalent in the context of BVHs. Figure 2.12 depicts this operation for a leaf and an inner node as the target nodes. The key observation is that for a BVH with n nodes there are n different possibilities to add an additional leave. The other way around with this procedure a BVH with n leaves is constructed from a BVH with $n-1$ leaves. According

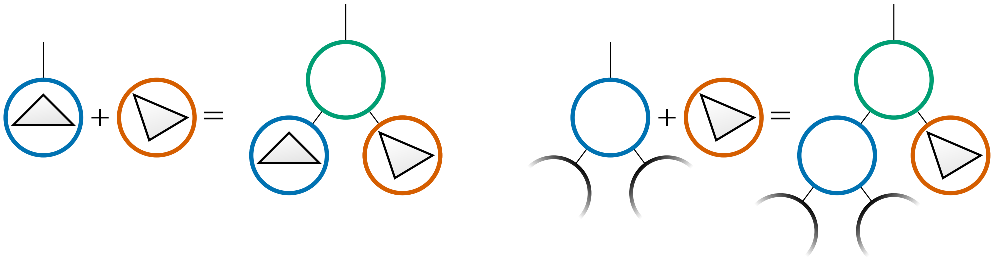


Figure 2.12: Depiction of adding a leaf node (red) to a BVH at a target node (blue) for the derivation of the number of different BVHs for a given number of primitives. The left image shows adding a node at a leaf, while the right image shows adding a leaf at an inner node. The targeted node is removed and attached to a new parent node (green) along with the leaf to add. Then the parent node is inserted at the original position of the target node.

n	1	2	3	4	5	6	7	8	9	10
$B(n)$	1	1	3	15	105	945	10,395	135,135	2,027,025	34,459,425

Table 2.1: Number of different BVHs for a scene with n primitives assuming one primitive per leaf. This number can be computed as $B(n) = (2n-3)!!$ where $x!!$ is the double factorial.

to Equation 2.19 a BVH with $n-1$ leaves has $2(n-1)-1 = 2n-3$ nodes. Thus, there were $2n-3$ possibilities to construct a BVH with n leaves from a BVH with $n-1$ leaves. Denoting the number of BVHs with n leaves $B(n)$ this gives us $B(n) = (2n-3)B(n-1)$. A BVH with $n-2$ leaves has $2(n-2)-1 = 2n-5$ nodes. Thus, there were $2n-5$ possibilities to construct a BVH with $n-1$ leaves from a BVH with $n-2$ leaves which gives $B(n-1) = (2n-5)B(n-2)$ and consequently $B(n) = (2n-3)(2n-5)B(n-2)$. Each BVH with one leaf less has two nodes less. Repeating this expansion for $B(n-2)$, $B(n-3)$ and so on we arrive at the special case $B(1)$. There is only one possibility to construct a BVH with one leaf. Thus, we have $B(1) = 1$. Putting everything together gives again rise to the double factorial $B(n) = (2n-3)(2n-5)\cdot\ldots\cdot 3\cdot 1 = (2n-3)!!$ as the number of different BVHs with n leaves as stated by Karras and Aila [2013]. Table 2.1 shows the number of different BVHs for very small numbers of primitives. We can see that there is already a combinatorial explosion for such unrealistically small scenes.

2.2.5 Basic Construction Strategies

In the early days of ray tracing, BVHs were often constructed by hand as described in the initial work on BVHs from Rubin and Whitted [1980] and Weghorst et al. [1984]. The simplest construction strategy for automatic hierarchy generation is the *spatial median split* introduced by Reddy and Rubin [1978]. First, we compute tight AABBs for all input primitives and then compute the scene AABB, which tightly contains all primitives. The primitives with their AABBs and the scene bounds are the initial current primitive working set and current working bounds. For the current bounds a splitting plane which is orthogonal to the coordinate axis with the largest bounds extent is created. The position of the plane is the center of the largest extent, the spatial median. The next step is to

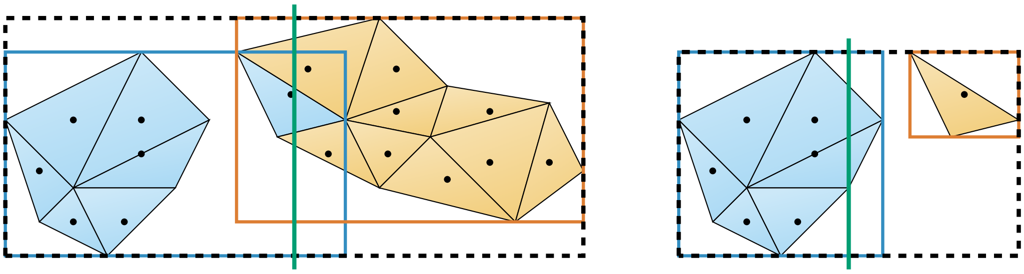


Figure 2.13: Example scene with 16 primitives for two iterations (left and right image) of BVH construction with the spatial median split strategy. This strategy places a splitting plane (green line) at the spatial median of the largest extent of the node bounds (dashed box). Depending on the side of the splitting plane the primitive AABB centroid coordinates lie on primitives are put to the left (blue) or to the right (orange). As the splitting plane is placed completely independent of the centroids the resulting BVHs can be highly unbalanced.

distribute the current primitives to both sides of the plane to form a partition. This is done depending on which side of the plane the AABB midpoint lies. Then, tight bounds are computed for each side of the partition. The current node is turned into an inner node with both sides of the partition as its children and construction recursively proceeds on both sides. The recursion ends when some pre-specified number of current primitives are left or a maximum allowed depth has been reached, and a leaf is created for the current primitives. Figure 2.13 depicts BVH construction with this strategy.

If the list of primitives is roughly split in half in each step the complexity of this algorithm is $O(n \log n)$ for n input primitives. Lauterbach et al. [2009] presented a fast construction solution for GPUs which also works for CPUs that assigns an m-bit *Morton code* to each primitive. The list of codes can be sorted with *Radix sort* in $O(mn)$. The hierarchy can be extracted from the sorted codes in linear time which results in $O(mn)$ overall complexity. Karras [2012] proposed a highly parallel implementation for the hierarchy extraction step. While the spatial median split construction is simple and fast the tree does not really adapt to the geometry as the splitting plane positioning does not take the geometry into account. In Section 2.5 we will see that geometry adaptation is important for traversal performance. The best case input for this strategy is uniformly distributed geometry. For non-uniform distributions the tree can partially degenerate into a list. Vinkler et al. [2017] proposed to extend Morton codes with primitive size information and then perform construction with the methods from Lauterbach et al. [2009] or Karras [2012]. This allowed to increase BVH quality of the fast $O(mn)$ construction. Quality is still lower than the state-of-the-art.

Next we describe the *object median split* construction strategy which has been introduced by Heckbert [1982] and applied to BVHs by Kay and Kajiya [1986]. The object median split algorithm is similar to the spatial median split algorithm. The main difference is the splitting strategy. It first sorts the primitives w.r.t. the primitive bounds center coordinate in the dimension of the largest extent. Then this sorted list is simply split into two equal halves. As a result trees constructed with this strategy are always balanced. Figure 2.14 depicts BVH construction with this strategy. Because of the required sorting step, the complexity of this algorithm is $O(n \log^2 n)$ for n input primitives. Adopting concepts from Wald and Havran [2006] for the high quality construction of kd-trees, which

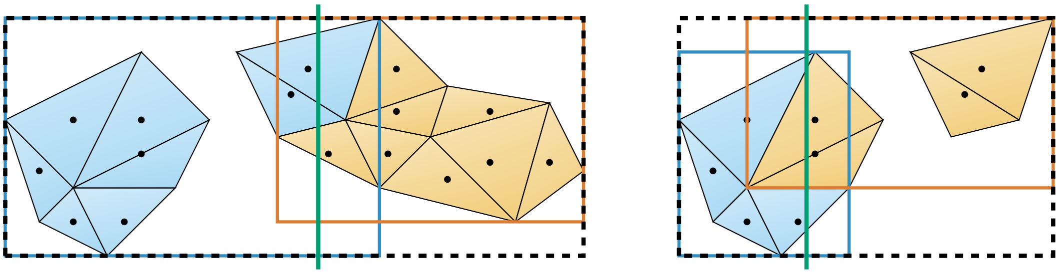


Figure 2.14: Example scene with 16 primitives for two iterations (left and right image) of BVH construction with the object median split strategy. This strategy focuses solely on splitting a set of primitives into two halves w.r.t. the primitive AABB centroid coordinates in the dimension with the largest extent. The green line depicts a symbolic splitting plane which is positioned such that the set is partitioned into two halves. As resulting BVHs are balanced by construction they have minimal height.

are introduced in the next section, an alternative faster implementation is possible. It first sorts the primitives in each dimension in separate arrays before construction. During construction these arrays are kept sorted in each recursion step, which can be done in linear time. This improves the complexity to $O(n \log n)$ at the cost of higher memory consumption. While the construction is still simple and reasonably fast the tree again does not adapt to the geometry. For uniform geometry distributions this strategy has no real benefit over the spatial median split as the resulting hierarchies are essentially the same. For non-uniformly distributed geometry though this approach can expose higher trace performance as their balancedness causes them to be of minimum height. This does not make balancedness a magic bullet. The only reason this is an advantage is that spatial median splits expose an unthoughtful unbalancedness which unnecessarily increases height. As we will see in Section 2.5 state-of-the-art construction with the surface area heuristic deliberately introduces unbalancedness in the tree where it is considered beneficial.

2.3 kd-Trees

In contrast to BVHs, which partition sets of objects, a kd-tree is a binary tree, which partitions space using split planes. kd-trees restrict split planes to be parallel to the coordinate axes planes. The more general *binary space partitioning* tree (BSP) allows arbitrarily oriented split planes. Each inner kd-tree node stores a split plane, which partitions space into two half-spaces. Primitives which straddle the split plane are split themselves. Instead of actually splitting primitives only references to primitives are kept which are duplicated on a split. In $\mathbb{R}^3$ a kd-tree split plane can be defined by a distance $d \in \mathbb{R}$ and the coordinate axis $a \in \{x, y, z\}$ the plane intersects and is perpendicular to. In the context of ray tracing a kd-tree also stores a tight AABB for the complete scene geometry. As the split planes are axes parallel this gives every node in the tree an implicit AABB, which can be computed by intersecting the scene AABB with all half-spaces of the ancestors of the node. In case of a BSP tree its arbitrarily oriented split planes result in general convex polyhedra as the implicit bounding volumes of nodes. The union of all node bounds in the same tree level is the scene bounds. The pairwise intersection of node bounds on the same level is disjoint. As a result there is no overlap in a kd-tree.

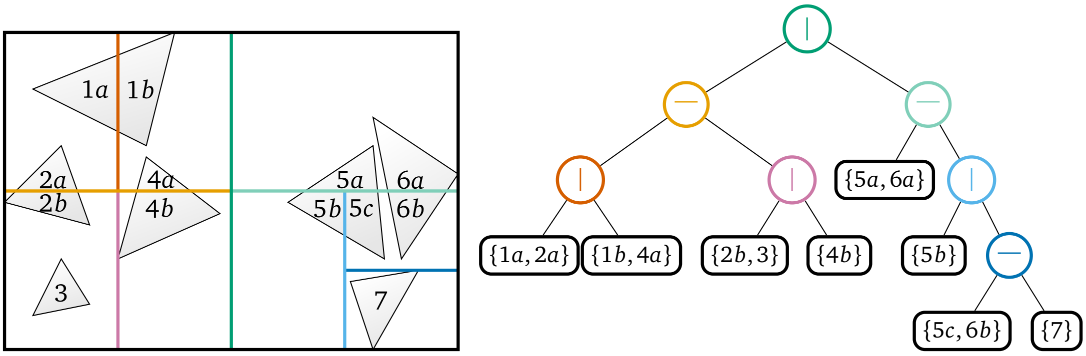


Figure 2.15: Example kd-tree for a scene with seven input triangles. A spatial median construction strategy is applied, where the maximum allowed number of primitives in a leaf is set to two. The constructed kd-tree references about two times as many primitives as there are in the input.

The basic construction strategies discussed in Section 2.2.5 can also be applied to kd-trees. Implementation of the object median strategy is a bit more involved because of the primitive splitting. After primitives have been sorted the list cannot be simply split in half. Instead, a split plane has to be swept through the primitives to find the plane which, including primitive duplicates, partitions the set into ideally two halves. Observing that the number of primitives on both sides of a split plane changes at primitive starts and ends this can be implemented in $O(n)$ by counting so called primitive *enter-* and *exit-events* as described in Wald and Havran [2006]. An example kd-tree constructed with the spatial median split is depicted in Figure 2.15. Due to primitive splitting there is no upper bound on the number of kd-tree nodes depending on the number of input primitives. In practice, this requires to specify a maximum depth to ensure construction algorithm termination.

2.3.1 Traversal

kd-tree traversal is similar to BVH traversal. A major difference is that bounding volumes are defined implicitly by the series of half spaces down the tree. Only the scene bounding box is explicitly defined. Another major difference to BVHs is that traversal can immediately terminate as soon as an intersection has been found. The reason is that there is no overlap with other non-ancestral nodes as described earlier. Thus, the intersection is guaranteed to be the closest. This property of kd-trees is called *early ray termination*.

We outline the traversal process according to the description from Pharr et al. [2016]. Pseudocode for kd-tree traversal is provided in Algorithm 2. First, the ray is intersected with the tight axis aligned scene bounding box to obtain initial ray parameter bounds $[t_{min}, t_{max}]$. If the ray misses the bounds there can be no intersection and traversal terminates. The currently processed node $n_{current}$ is initialized with the kd-tree root. If $n_{current}$ is an inner node we have to identify which children are intersected by the ray. For this we have to intersect the split plane with the ray. The ray parameter for the intersection with the plane is

$$t_{plane} = \frac{d - o_a}{d_a} \tag{2.28}$$

Algorithm 2: Pseudocode for kd-tree traversal.

```
input  : ray                                  // ray we have to intersect with the scene
input  : root                                 // root node of the scene BVH
input  : sceneBounds                          // tight AABB of the scene
output: iIdx                                  // index of the closest intersected primitive
iIdx ← InvalidIdx                             // Set intersection index to invalid index
($t_{min}$, $t_{max}$) ← intersect_bounds (ray,sceneBounds)      // Init ray parameter interval
if $t_{min} \geq t_{max}$ then                 // Return if we miss the scene bounds
    return
end
push ( stack, (NIL,[$\infty$,$-\infty$]) )     // Init stack with invalid node and parameter interval
$n_{current}$ ← root                          // Start traversal at root
while $n_{current} \neq$ NIL do               // Traversal loop
    if is_inner_node ( $n_{current}$ ) then   // Handle inner nodes
        plane ← get_split_plane ( $n_{current}$ )      // Get split plane
        // Compute ray/plane intersection parameter
        $t_{plane}$ ← intersect_plane ( ray, plane )
        // Order children to get a near and far child
        ($c_{near}$, $c_{far}$) ← identify_near_and_far_child (ray,$n_{current}$)
        // Check which children have been intersected
        if $t_{max} \leq t_{plane} \vee t_{plane} \leq 0$ then // only intersected near child
            $n_{current}$ ← $c_{near}$
        else if $t_{min} \geq t_{plane}$ then// only intersected far child
            $n_{current}$ ← $c_{far}$
        else // intersected both children
            $n_{current}$ ← $c_{near}$
            // Push far child with corresponding bounds interval on the stack
            push ( stack, ($c_{far}$, [$t_{plane}$, $t_{max}$]))
            // Shorten ray
            $t_{max}$ ← $t_{plane}$
        end
    else
        // Find closest leaf primitive intersection
        iIdx ← intersect_leaf_primitives ( $n_{current}$, ray, $t_{min}$, $t_{max}$)
        if iIdx $\neq$ InvalidIdx then // terminate on primitive intersection
            return
        end
        // Get next node and corresponding parameter interval
        ($n_{current}$, [$t_{min}$, $t_{max}$]) ← pop ( stack )
    end
end
```

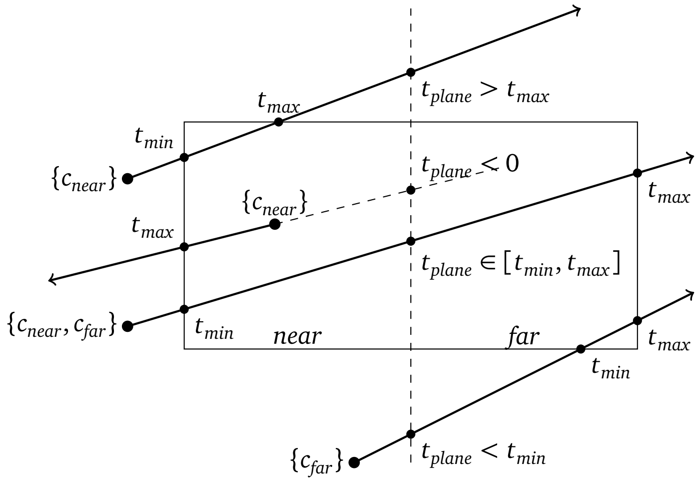


Figure 2.16: Depiction of the four situations when determining the next node during kd-tree traversal. Every ray shows the set of relevant child nodes c_{near} and/or c_{far} depending on the plane intersection parameter t_{plane} and its relation to the parent bounds intersection parameters t_{min} and t_{max}. (Based on Pharr et al. [2010], Figure 4.18)

with $d \in \mathbb{R}$ being the plane distance and $a \in \{x, y, z\}$ being the orthogonal plane axis. This gives us the ray parameter intervals $[t_{min}, t_{plane}]$ and $[t_{plane}, t_{max}]$ of the intersection with the implicit children AABBs. The child which corresponds to the side of the split plane containing the ray origin is called the near child c_{near} and the other one is the far child c_{far}. Analyzing the sign of the ray direction component d_a in dimension a and t_{plane} the intervals can be attributed to c_{near} and c_{far}. In case a child parameter interval has invalid limits, that is the upper bound is lower than the lower bound, the child is not intersected. As the union of the disjoint implicit children bounds is exactly the implicit bounds of the parent node the ray is guaranteed to intersect at least one child since the ray intersected the parent. If t_{plane} is larger than t_{max} the ray only intersects c_{near}. In case t_{plane} is smaller than zero the ray points away from the split plane and again only intersects c_{near}. If t_{plane} is smaller than t_{min} the ray only intersects c_{far}. The only remaining case is where $t_{plane} \in [t_{min}, t_{max}]$ holds and both children are intersected. Figure 2.16 shows the four different situations. In the case where both children are intersected the far child is put on the stack along with its parameter interval $[t_{plane}, t_{max}]$. Before traversal continues with the near child the current parameter interval is shortened to its interval $[t_{min}, t_{plane}]$.

If $n_{current}$ is a leaf the closest primitive intersection is determined and traversal can terminate immediately on intersection as explained at the beginning of this section. If a leaf contains duplicate primitive references it can happen that the closest intersection is outside of the implicit leaf bounds. The intersection has to be rejected in this case. In case there was no intersection the next node along with its parameter interval is popped from the stack.

2.4 Other Acceleration Structures

For the sake of completeness, before we proceed with BVH and kd-tree construction with the surface area metric in the next chapter we close this chapter with a brief discussion

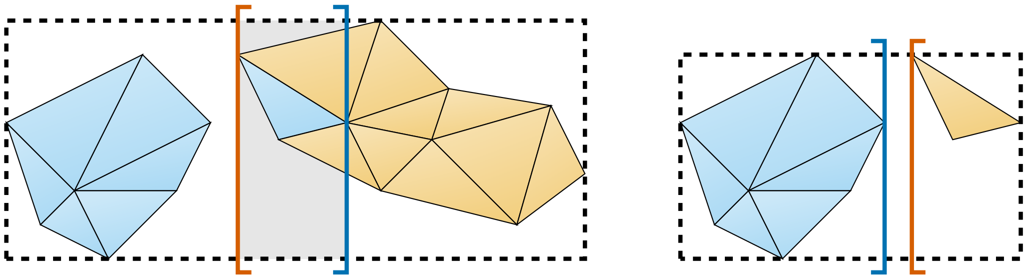


Figure 2.17: Example scene with 16 primitives for two iterations (left and right image) of an skd-tree partition with the spatial median split construction strategy. The left partition has overlapping (gray area) implicit bounds, which are also larger than with a kd-tree. The right partition has smaller implicit bounds, but still larger bounds than a BVH with AABBs as bounding volumes.

of skd-trees/bounding interval hierarchies, graphs, and uniform grids. The first two are directly related to BVHs and kd-trees and can benefit from our contributions on hierarchy quality, out-of-core construction, and memory layouts.

Skd-Trees / Bounding Interval Hierarchies The *spatial kd-tree* (skd-tree) has been originally introduced by Ooi et al. [1987] in the context of spatial databases to store non-points objects (objects with an extent) without producing duplicates. Havran et al. [2006] and Wächter and Keller [2006] concurrently introduced skd-trees to ray tracing where the latter coined the term *bounding interval hierarchy* as they were unaware of the work from Ooi et al. [1987]. Skd-trees can be interpreted as relaxed kd-trees, where the implicit bounds of the children are allowed to overlap. Instead of a split plane it stores an upper bound for the left child and a lower bound for the right child in the split dimension. Another interpretation is to view skd-trees as restricted BVHs with AABBs as children bounding volumes, where five bounding interval limits are predetermined by the parent bounds. Figure 2.17 shows an skd-tree example. The main benefit of skd-trees is their lower memory footprint compared to kd-trees and BVHs. Compared to a BVH with its twelve bounding interval limits for each pair of children AABBs the skd-tree only has to store two limits. As an skd-tree is an object partitioning structure it has the same bounded number of nodes as a BVH (see Equation 2.19). Empirically the number of skd-tree nodes is much lower than for a kd-tree resulting in a lower memory footprint even though a kd-tree node is slightly smaller (see Wächter and Keller [2006]).

The disadvantage of an skd-tree is its traversal performance. Wächter and Keller [2006] reported roughly the same and up to three times lower trace performance than kd-trees constructed with an unspecified construction strategy. Havran et al. [2006] observed a generally lower trace performance, which was even about one order of magnitude lower for some scenes. This can be partially explained by the possibly larger implicit bounds compared to kd-trees due to the allowed overlap (see Figure 2.17). Also, traversal is more involved. Two planes have to be intersected and more logic is needed to determine which children have been intersected with the additional case that no child is intersected. In contrast to kd-trees, skd-trees have no early ray termination mechanism because of the possible node overlap.

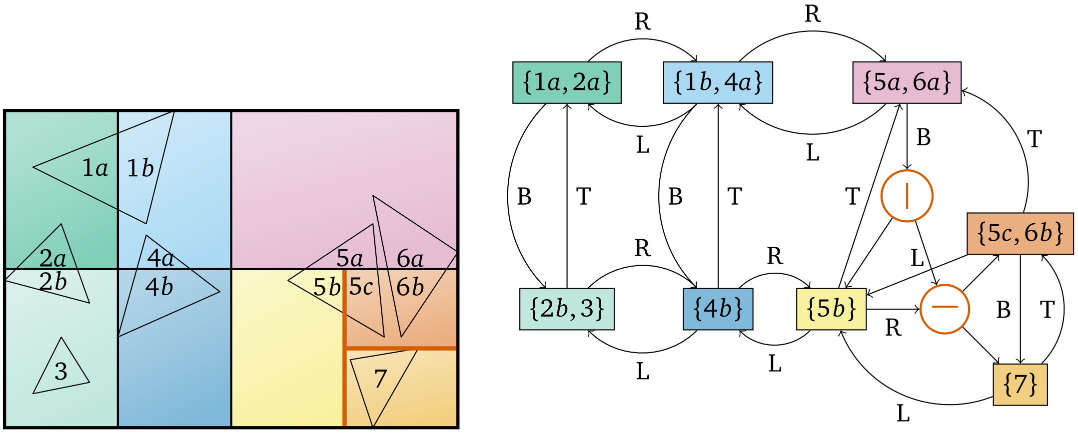


Figure 2.18: 2D example of the graph based acceleration structure from Gribble and Naveros [2013]. Basis is the example kd-tree and scene in Figure 2.15 from the kd-tree section. The original kd-tree leaves are turned into sectors (colored) with explicit bounds. The left (L), right (R), top (T), and bottom (B) face of each sector references its adjacent sector. The bottom face of the purple sector and the right face of the yellow sector have no unambiguous neighbor. Thus, each face references the deepest inner node in the original kd-tree (red nodes) which contains all adjacent sectors in its subtree. The referenced subtree allows to disambiguate ambiguous sectors at traversal time.

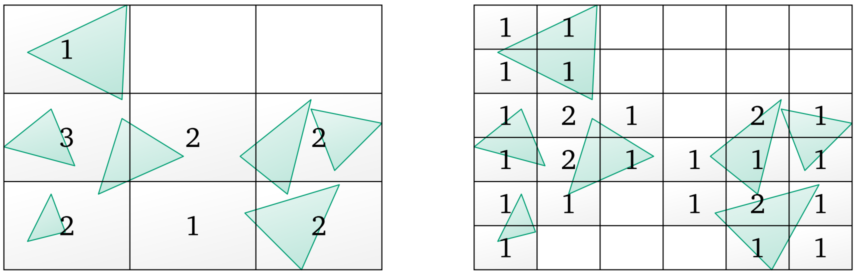


Figure 2.19: Example of two uniform grids with different resolution for a scene with 7 primitives. The left grid has resolution $r = \lceil 1.5 \cdot \sqrt[3]{7} \rceil = 3$ and the left grid has resolution $r = \lceil 3 \cdot \sqrt[3]{7} \rceil = 6$. The numbers in voxels indicate the number of referenced primitives.

Graph-based Acceleration Structure Gribble and Naveros [2013] very briefly hint at a novel graph-based spatial acceleration structure which has been developed with GPU ray tracing in mind. The goal is to get rid of the traversal stack (similar to uniform grids) while at the same time have a structure, which adapts to the geometry distribution (Alexis Naveros, personal communication, March 18, 2013). Algorithmic details can be retrieved from the publicly available implementation (Naveros [2016]). The basis of the structure is a kd-tree, which can be constructed with any construction strategy. For every face of the implicit AABBs of leaves (which are called *sectors* in this structure) a reference to the neighboring sector is stored. If several sectors touch an AABB face the closest common ancestor node of those sectors in the kd-tree is referenced. The resulting graph structure allows to traverse from sector to sector. Figure 2.18 depicts this graph-based acceleration structure for the example kd-tree in Figure 2.15 from the kd-tree section. We can see that all leaves of the original kd-tree have turned into sectors. Two inner nodes of the original kd-tree (orange circles) are still present in the graph to resolve neighborhood ambiguity for the pink and the leftmost yellow sectors. For traversal the ray must start in the sector, which contains the ray origin. If the ray origin is outside of the scene bounds the entry sector has to be identified first. For this, Gribble and Naveros [2013] sketch several methods to identify an initial starting sector in constant time. If those methods fail the kd-tree is traversed down to find the sector. For secondary rays in ray tracing based global illumination algorithms the starting sector is known, as they usually start on surfaces. After, the starting sector has been identified the ray checks all sector primitives for intersection. On intersection traversal can terminate immediately. Otherwise, the ray proceeds to the sector which is referenced by the sector face the ray intersects. If the face references an inner kd-tree node the subtree is traversed to find the next sector. All these operations do not require a traversal stack. Gribble and Naveros [2013] report competitive trace performance with kd-trees.

Uniform Grids Fujimoto et al. [1986] proposed uniform grids as a ray tracing acceleration structure. The scene bounds are subdivided into a grid of equally shaped and sized boxes or *voxels* (volumetric elements). Each voxel stores the list of all primitives which overlap with its volume. As a result a primitive can be referenced in more than one voxel. According to Pharr et al. [2016], the number of voxels optimally is roughly proportional to the number n of primitives. For uniformly distributed geometry this results in a grid resolution $r \in \mathbb{N}$ of $r \approx c\sqrt[3]{n}$ with some constant c for an $r \times r \times r$ grid. Traversal of the grid processes voxels which intersect the ray in front to back order. This does not require a traversal stack. For each voxel the ray visits primitive intersection with all referenced primitives is performed. Traversal can terminate as soon as an intersection has been found inside the current voxel. Figure 2.19 depicts a grid with two different resolutions. Fujimoto et al. [1986] proposed the 3DDDA (3d digital differential analyzer) algorithm as an efficient implementation of this traversal process which Amanatides and Woo [1987] improved on. In the best case a grid can find an intersection after just a couple of steps if the intersection occurs in the starting cell or its neighborhood. This gives essentially an $O(1)$ traversal complexity in the number of grid cells in this case.

Grids have some disadvantages which partially can be seen in Figure 2.19. One problem is that they do not adapt well to the geometry distribution. Only the grid resolution r can be adapted. If no intersection is found tracing parallel to a coordinate axis requires roughly r traversal steps. Tracing along the diagonal of the grid requires roughly $\sqrt{3}r$ traversal steps. Thus the worst case complexity of grid traversal is $O(r)$ if no intersection is found. Using the optimal resolution of $r \approx c\sqrt[3]{n}$ this results in a complexity of $O(\sqrt[3]{n})$, which is much higher than the $O(\log n)$ complexity of tree-based ray tracing. If the resolution r is too high too many traversal steps have to be performed to find an intersection and much time is spent in traversing empty areas. A higher resolution also causes a higher memory consumption due to the increased number of voxels and duplicate primitive references. Sramek and Kaufman [2000] and Es and İşler [2007] proposed to enrich voxels with additional empty space information computed in a preprocess which allows to perform larger leaps over empty regions. This allows to drastically reduce traversal for scenes with large empty regions, but further increases the memory overhead. If r is too low too many primitives are stored per voxel which results in a larger number of unnecessary primitive intersection tests.

While worst case traversal complexity of grids is higher than for trees they have a lower construction complexity. Hapala et al. [2011] analyzed when it is more beneficial to use grids based on the number of rays to trace and determined critical points in the number of rays at which the higher construction cost of trees is amortized. The number of primitives of their scenes ranged from hundreds to millions. For random rays starting outside of the scene the critical point was approximately proportional to the number of primitives. For rays generated from a path tracer critical points where heavily scene dependent and seemingly independent of the number of scene primitives. For more than half of the 24 test scenes grids never paid off or had their critical point at a couple thousands of rays. For other scenes the critical point ranged from 1M to 50M rays. It would be interesting to see how the empty space leaping techniques from Sramek and Kaufman [2000] and Es and İşler [2007] affect the critical point.

2.5 The Surface Area Metric and Surface Area Heuristic

The *surface area metric (SAM)* provides a measure for the expected traversal cost of a given BVH or kd-tree. The *surface area heuristic (SAH)* is derived from SAM and is a means to measure the quality of an object or spatial split during construction. SAM and SAH have been developed in two iterations. We start with presenting the first prototype by Goldsmith and Salmon [1987] and the final iteration as it is known today from MacDonald and Booth [1989,1990]. We proceed with the standard and alternative SAH-based kd-tree and BVH construction algorithms. Further, we will introduce other ray tracing performance metrics of which the *end-point-overlap (EPO)* metric is the most important one for this dissertation.

2.5.1 Goldsmith and Salmon's Approach

The key concept of Goldsmith and Salmon [1987] is that the size of the surface area of bounding volumes is important for ray tracing performance. Till then it has been thought, that volume is more important (e.g. Weghorst et al. [1984]). An example for this observation are planar objects such as triangles, which have zero volume but still can be intersected. Goldsmith and Salmon based their argument on an example from optimal search theory (in the sense of finding a lost object) by Stone [1975]. Stone himself lend this example from Koopman [1956] which used it to introduce the *inverse-cube law of sighting*. Koopman actually discusses the probability of a plane spotting an object on the ocean. He determines that this probability depends on the solid angle of the object, which is proportional to surface area at large distance. From this example, Goldsmith and Salmon state that the probability that some random ray outside of the scene intersects a convex object O is proportional to the surface area of O:

$$p(\text{intersect } O) \sim \texttt{Area}(O). \tag{2.29}$$

Based on this principle they proposed an incremental top-down BVH construction algorithm with runtime $O(n \log n)$ in the number of primitives. The algorithm starts with a BVH, which only consists of a single leaf, that contains one of the input primitives. Then, successively for each primitive heuristically the leaf in the current BVH is identified where adding the current primitive results in the smallest cumulative increase in bounding volume surface area of all ancestors up the hierarchy. A new leaf is created for the current primitive. Then, a common new parent node of the new and the selected leaf is created at the original position of the selected leaf.

To predict traversal performance of the resulting hierarchies Goldsmith and Salmon developed a cost metric. For this, based on Equation 2.29 they defined the conditional probability of intersecting a convex object O_1 given that another convex object O_2, which fully contains O_1, has been intersected as the ratio of their surface areas. Realizing that the probability of intersecting O_2 given that we intersected O_1 is one, we can derive this conditional probability using Bayes' theorem and Equation 2.29:

$$\begin{aligned} p(\text{intersect } O_1 \mid \text{intersect } O_2) &= \frac{p(\text{intersect } O_2 \mid \text{intersect } O_1)p(\text{intersect } O_1)}{p(\text{intersect } O_2)} \\ &= \frac{p(\text{intersect } O_1)}{p(\text{intersect } O_2)} \\ &= \frac{\texttt{Area}(O_1)}{\texttt{Area}(O_2)} \end{aligned} \tag{2.30}$$

We abbreviate $p(\text{intersect } O_1 \mid \text{intersect } O_2)$ with $p_{O_1}^{O_2}$. This conditional probability allows to compute the probability of a ray intersecting a BVH node under the assumption that it intersected the BVH root r. Goldsmith and Salmon assumed that every time an inner node n of a BVH with branching factor k has been intersected with probability p_n^r, its k children have to be intersected, too. This allowed them to define their BVH traversal cost metric $c(\mathcal{I})$ on the set of inner BVH nodes $\mathcal{I}$ as the expected number of node bounds intersection tests:

$$c(\mathcal{I}) = k \sum_{n \in \mathcal{I}} p_n^r = k \sum_{n \in \mathcal{I}} \frac{\texttt{Area}(B_n)}{\texttt{Area}(B_r)}. \tag{2.31}$$

B_n are the bounds of a node $n \in \mathcal{I}$. Empirical tests made by Goldsmith and Salmon [1987] proved this metric relatively accurate. Thus, it gives a measure for the expected traversal cost for a tree without tracing a single ray. They also revealed that the value of $c(\mathcal{I})$ of the BVHs constructed with their algorithm is very sensitive to the order in which primitives are inserted.

2.5.2 MacDonald and Booth's Approach

Based on [Goldsmith and Salmon 1987] MacDonald and Booth [1989,1990] developed the surface area metric (SAM) and the surface area construction heuristic (SAH) that are used today. SAM and SAH are based on the following three assumptions:

1. Rays originate infinitely far away from the scene.
2. Ray directions have a uniform distribution.
3. Rays do not terminate on intersection.

The first and second assumption allow to define the analytical conditional geometric probability of a ray r intersecting a convex body B_1 given that it intersected another convex body B_2, which contains B_1, as follows:

$$p\,(r \text{ intersects } B_1 \mid r \text{ intersects } B_2) = \frac{\texttt{Area}\,(B_1)}{\texttt{Area}\,(B_2)} \tag{2.32}$$

Thus, p is simply the ratio of the surface areas of both convex bodies. This is the same result as Equation 2.30 but with a more proper theoretical foundation. The exact derivation of Equation 2.32 is discussed in Chapter 4.

SAM further has two implementation dependent cost constants c_i and c_t. c_i is the cost for intersecting a scene primitive while c_t is the cost of performing a traversal step. The later includes costs for intersecting children bounding volumes in case of a BVH, or split plane intersection in case of a kd-tree. It can further include costs of traversal stack operations and logic for determining which child to traverse first. The cost itself can be the amount of time or computation cycles needed for performing operations. It is common practice to set c_i to one and express c_t relative to c_i.

The third SAH assumption means that a ray intersects all nodes and primitives in its path regardless of whether they are occluded by the closest hit point. Combined with the intersection probability from Equation 2.32 and the implementation dependent constants the expected traversal cost $c(\mathcal{N})$ for the set of tree nodes $\mathcal{N} = \mathcal{I} \cup \mathcal{L}$ is partitioned into the cost $c_{\mathcal{I}}$ for processing inner nodes $i \in \mathcal{I} \subset \mathcal{N}$ and costs $c_{\mathcal{L}}$ for processing the leaves $l \in \mathcal{L} \subseteq \mathcal{N}$. The cost metric is as follows

$$c_{BVH} = c_{\mathcal{I}} + c_{\mathcal{L}} = c_t \sum_{i \in \mathcal{I}} p_i^{root} + c_i \sum_{l \in \mathcal{L}} p_l^{root} |l|. \tag{2.33}$$

$|l|$ is the number of primitives in a leaf. This cost metric is equally applicable to BVHs and kd-trees. For BVH nodes the bounds needed to compute the conditional probabilities are explicitly defined, while for a kd-tree node bounds are defined implicitly by intersecting the scene bounds with all half spaces of the predecessors of the node.

It is possible to define Equation 2.33 recursively, which also resembles the tree traversal process. This requires the conditional probability p_n that a ray intersects the bounds B_n of a node n given that it intersected the bounds of the parent of n. With Equation 2.32 it can be defined as:

$$p_n = \frac{\texttt{Area}(B_n)}{\texttt{Area}\left(B_{\uparrow(n)}\right)}, \tag{2.34}$$

where $\uparrow(n)$ denotes the parent node of n. Together with the implementation dependent constants the recursive expected traversal cost $c(n)$ for the subtree of a node $n \in \mathcal{N}$ is recursively defined as:

$$c(n) = \begin{cases} c_t + p_{n_l} c(n_l) + p_{n_r} c(n_r) & n \in \mathcal{I} \\ |n| c_i & n \in \mathcal{L} \end{cases} \tag{2.35}$$

Here, n_l and n_r are the left and right child node of n in case of an inner node, and $|n|$ is the number of primitives belonging to n. Starting the recursive evaluation of $c(n)$ at the tree root gives the cost for the whole tree. To see that the result is identical to Equation 2.33 one has to show that in the end every node n gets multiplied with p_n^{root} as in Equation 2.33. Because of the recursion the cost of a node n is multiplied by a series of conditional intersection probabilities of the node and its ancestors up to the root from previous recursion steps. Let $\uparrow^i(n)$ denote the i-th ancestor of a node $n \in \mathcal{N}$ up the tree with $\uparrow^0(n) = n$ and let $d \in \mathbb{N}$ be the depth of n in the tree such that $\uparrow^d(n)$ gives the root. Further we abbreviate the surface area of the bounds of a node n with A_n. This lets us formulate the product as follows:

$$\begin{aligned}\prod_{i=0}^{d-1} p_{\uparrow^i(n)} &= p_n \cdot p_{\uparrow^1(n)} \cdot p_{\uparrow^2(n)} \cdots p_{\uparrow^{d-2}(n)} \cdot p_{\uparrow^{d-1}(n)} \\ &= \frac{A_n}{A_{\uparrow^1(n)}} \cdot \frac{A_{\uparrow^1(n)}}{A_{\uparrow^2(n)}} \cdot \frac{A_{\uparrow^2(n)}}{A_{\uparrow^3(n)}} \cdots \frac{A_{\uparrow^{d-2}(n)}}{A_{\uparrow^{d-1}(n)}} \cdot \frac{A_{\uparrow^{d-1}(n)}}{A_{root}} \\ &= \frac{A_n}{A_{root}} = p_n^{root}\end{aligned} \tag{2.36}$$

The surface areas of all ancestors between n and the root node cancel out resulting in conditional probabilities w.r.t. the root as in Equation 2.33.

As already hinted at in Section 2.2, choosing OBBs or spheres as node bounds which are always tight w.r.t. the geometry in a node's subtree does not guarantee that the node bounds are fully contained in the parent bounds. In this case Equation 2.32 cannot be applied directly to compute the intersection probability of a node. Instead, the surface area of the intersection of a node's bounds with the bounds of all ancestors is needed. This can be computed conveniently when evaluating SAM with the recursive formulation. Computation of the intersection of multiple OBBs or spheres themselves is non-trivial.

In practice at most only a subset of the three SAM assumptions is fulfilled. Usually rays originate inside the scene bounds. Rays originating from a camera or directed light sources such as the sun or spotlights do not have a uniform ray direction distribution. Also, except for multi-hit traversal (see Gribble et al. [2014]), rays usually terminate traversal as soon as the closest intersection point has been found. Still SAM empirically proves to be a good performance predictor for kd-trees. In case of BVHs SAM proved to be less reliable due to effects caused by node overlap which are not captured by SAM. As we will see in Section 2.5.7 Aila et al. [2013] developed the EPO metric as an addition to SAM to better explain the performance of BVHs.

2.5.3 SAH-based Construction

Construction of high quality kd-trees and BVHs aims at reducing SAM cost for the resulting tree. Besides introducing SAM MacDonald and Booth [1989,1990] also developed a greedy top-down construction algorithm for kd-trees which reduces SAM in a more direct manner than Goldsmith's and Salmon's algorithm. In contrast to Goldsmith and Salmon [1987] the output of their algorithm does not depend on the order of input primitives. While MacDonald and Booth [1989,1990] note that their approach could be directly applied to BVHs, Müller and Fellner [1999] where the first to present and evaluate a similar approach for BVHs. Müller and Fellner [1999] were not aware of the prior work from MacDonald and Booth [1989,1990] and proposed a slightly simpler cost model applied to BVHs. Wald et al. [2007] presented a direct adaptation of MacDonald and Booth's approach, which in contrast to Müller and Fellner's approach made traversal performance of BVHs competitive to kd-trees.

On a high level the construction process is the same for kd-trees and BVHs. The problem solving heuristic for the greedy algorithm is intuitively derived from Equation 2.35. It assumes that the set of input primitives $\mathcal{P}$ embedded in a leaf node n is split into a left leaf l with primitives $\mathcal{P}_l$ and a right leaf r with primitives $\mathcal{P}_r$ which share a common parent. The cost for this split is

$$c_{split} = c_t + p_l|l|c_i + p_r|r|c_i, \tag{2.37}$$

where p_x is the conditional probability from Equation 2.34 w.r.t. the original node. Construction aims at finding a partition of $\mathcal{P}$, which gives the lowest c_{split}. The cost for not splitting n is the cost for processing the node as a leaf:

$$c_{leaf} = |n|c_i. \tag{2.38}$$

If the lowest c_{split} is lower than c_{leaf} the partition is executed and construction recursively continues on $\mathcal{P}_l$ and $\mathcal{P}_r$. The recursion terminates as soon as the best c_{split} is larger than or equal to c_{leaf}. Considering that p_l and p_r are both w.r.t. the same parent node area A_n for all split candidates, a common variant of Equation 2.37 avoids the involved division by multiplying with A_n:

$$c_{split} = A_n c_t + A_l|l|c_i + A_r|r|c_i. \tag{2.39}$$

The adapted variant of Equation 2.38 for recursion termination is

$$c_{leaf} = A_n|n|c_i. \tag{2.40}$$

The kd-tree and BVH construction process differs in creation of the partition candidates. We first discuss partition creation for kd-trees and then proceed with BVHs.

kd-tree Partition Candidates Finding the best partition for a kd-tree boils down to finding the split plane, which results in the lowest cost. Figure 2.20 shows the behavior of the cost when sweeping a split plane along the coordinate axes for some example scene. We see that the cost $c(s)$ is a piecewise linear function with respect to split position s. Whenever the plane starts or ends to intersect a primitive there is a discontinuity. Primitive starts cause a discontinuous increase of $c(s)$ as primitives are split, while primitive ends cause a discontinuous decrease as they are not split anymore. Thus, for each dimension we only have to look for the minimum of c_{split} at the $2|\mathcal{P}|$ distinct discontinuities caused by the primitives in $\mathcal{P}$.

Wald and Havran [2006] presented an efficient implementation for finding the best partition candidate. For a fixed sweeping dimension a list $\mathcal{E}$ of events is created which contains the discontinuities of $c(s)$. An event is a pair (x_e, e), where $x_e \in \mathbb{R}$ is the position of the discontinuity and $e \in \{START, END\}$ is an event label which distinguishes between a primitive start or end. For each primitive such a start event $(x_{START}, START)$ and an end event (x_{END}, END) is created and added to $\mathcal{E}$. Then the events in $\mathcal{E}$ are sorted w.r.t. x in ascending order, where in the case that two events have the same position, a *start* event goes before an *end* event. The next step is to extract split candidates from the sorted list. For this we track the number of primitives L and R on the left and right side of a partition by updating them depending on the order of events encountered in the event list. Initially, L is zero and R is $|\mathcal{P}|$. Now, we iteratively inspect the sorted events. On a start event a primitive enters the left side while on an end event a primitive leaves the right side. This gives the following simple update rules for L and R based on the event label e:

$$\begin{aligned} e = START: &\quad L \leftarrow L + 1 \\ e = END: &\quad R \leftarrow R - 1. \end{aligned} \tag{2.41}$$

For every encountered event we compute c_{split} with the current counts L and R, and the bounds associated with this split. To compute the bounds we simply have to set the maximum of the left bounds and the minimum of the right bounds in the sweeping dimension

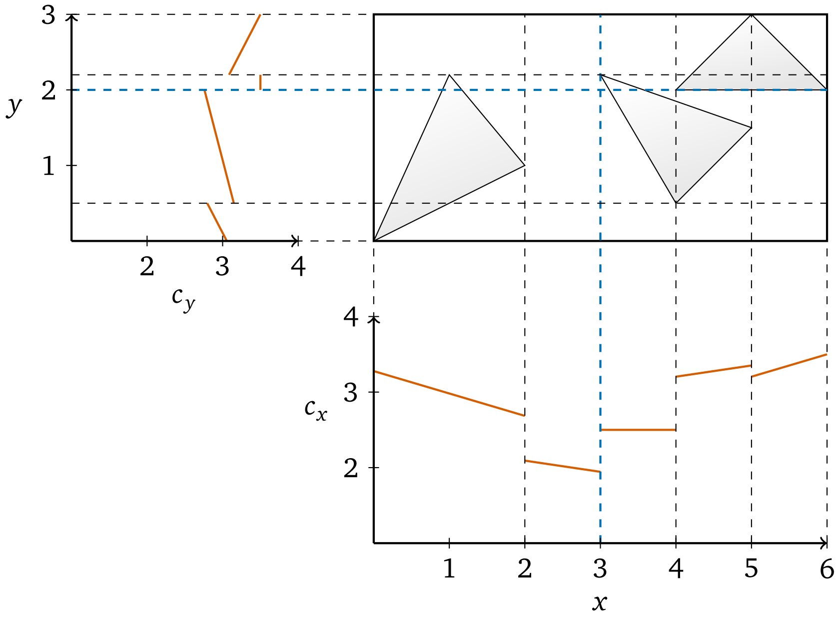


Figure 2.20: Example for the candidate cost function when sweeping the candidate plane along the x- and y-axis. The z-dimension is not depicted. The scene consists of three triangles where the scene bounding box has width, height, and depth (6,3,1). The pair of implementation dependent constants (c_t, c_i) is set to $(\frac{1}{2}, 1)$. From the given box geometry the cost function for split positions on the x-axis is $c_x(x) = \frac{1}{2} + |l|\frac{8x+6}{54} + |r|\left(1 - \frac{8x+6}{54}\right)$. For split positions on the y-axis the cost function is $c_y(y) = \frac{1}{2} + |l|\frac{14y+12}{54} + |r|\left(1 - \frac{14y+12}{54}\right)$. The dashed lines mark discontinuities at the start and end of primitives. Primitive starts cause a discontinuous increase of c_x and c_y as the primitive is split in two parts. At Primitive ends c_x and x_y show a discontinuous decrease as the primitive is not split anymore. $c_x(x)$ has its minimum at $x = 3$ with $c_x(3) \approx 1.94$ and the minimum for $c_y(y)$ is at $y = 2$ with $c_y(2) \approx 2.76$. Thus, the best split candidate is the plane $x = 3$. With $c_i = 1$ the leaf cost $c_{leaf} = 3$ is simply the number of triangles. As the best split has lower cost than the leaf cost it would have been chosen for splitting the node.

to the plane position of the current event. The remaining bound limits are identical to the parent bounds. This process is repeated for the other two sweeping dimensions to find the overall best candidate. The described procedure allows to find the best candidate in the sorted list in $O(n)$. As the event list had to be sorted first the overall complexity of this step is $O(n \log n)$. Wald and Havran [2006] also had a third event for the case when a primitive completely lies in the splitting plane. We left this event out for simplicity. Recursively repeating this procedure on both sides of a partition to construct the whole kd-tree has complexity $O(n \log^2 n)$.

Wald and Havran [2006] proposed a modification to improve on this complexity which requires additional memory. For this approach event lists for each sweeping dimension are created and sorted for all input primitives once before construction. Then, at construction time best candidates can be found in $O(n)$. When partitioning the primitives the sorted event lists are partitioned, too, and merely have to be maintained to preserve the event order for each partition. This maintenance step only has $O(n)$ complexity in contrast to the

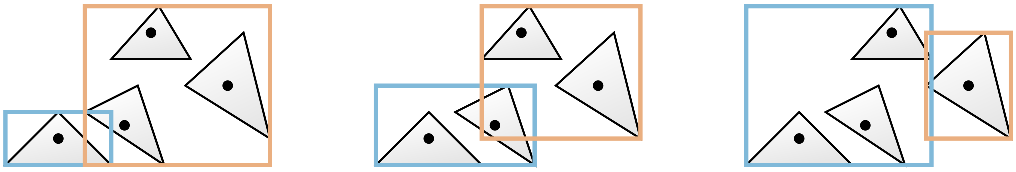


Figure 2.21: Example of generating candidate partitions for greedy SAH-based top-down BVH construction. First, the primitives are sorted w.r.t. a fixed dimension coordinate of the centroid of their AABBs. The initial partition contains the left most sorted primitive on its left side (blue) while all other primitives are put to the right side (red). Additional partitions are generated by iteratively removing a primitive from the right side and adding it to the left side in the order they appear in the sorted primitive list. This process is repeated for all coordinate dimensions as well. The candidate partition which gives the lowest SAH cost is chosen.

$O(n \log n)$ cost for sorting after each partitioning step. Constructing the whole kd-tree has $O(n \log n)$ complexity with this approach. As the sorting pre-process also has $O(n \log n)$ complexity the overall complexity is $O(n \log n)$.

BVH Partition Candidates SAH-based BVH construction has to find a partition $(\mathcal{L},\mathcal{R})$ of $\mathcal{P}$ which has the lowest cost c_{split}. With respect to the power set $\mathbb{P}(\mathcal{P})$ of $\mathcal{P}$ the set of partition candidates of $\mathcal{P}$ is

$$\mathcal{C}_{\mathcal{P}} = \{(\mathcal{L},\mathcal{R}) \in \mathbb{P}(\mathcal{P}) \times \mathbb{P}(\mathcal{P}) \setminus \{(\emptyset,\mathcal{P}),(\mathcal{P},\emptyset)\} \mid \mathcal{L} \cup \mathcal{R} = \mathcal{P} \wedge \mathcal{L} \cap \mathcal{R} = \emptyset\} \tag{2.42}$$

As for every $\mathcal{L} \in \mathbb{P}(\mathcal{P}) \setminus \{\emptyset,\mathcal{P}\}$ there is exactly one $\mathcal{R} \in \mathbb{P}(\mathcal{P}), (\mathcal{L},\mathcal{R}) \in \mathcal{C}_{\mathcal{P}}$ the number of candidate pairs is $|\mathcal{C}_{\mathcal{P}}| = |\mathbb{P}(\mathcal{P})|/2-1 = 2^{|\mathcal{P}|-1}-1$. Thus, in contrast to kd-tree construction the number of candidates is exponential in the number of primitives which is unpractical. Most of these possible partitions are unreasonable as they do not provide a good spatial separation of the primitives.

Instead of testing all partitions, which is infeasible, Wald et al. [2007] propose to test only a reasonable subset which should result in acceptable partitions. The idea is to sweep for candidates along each coordinate axis similar to kd-trees in the previous section. First all primitives are sorted w.r.t. the x-coordinate of the centroid of their AABBs. To reduce memory traffic the algorithm works on primitive references. A primitive reference is a pair (i, B_i) where i and B_i are the index and the bounds of the referenced primitive. For the first partition the left most sorted primitive is put to the left side while all other primitives are put to the right side. Remaining partitions are generated by iteratively removing a primitive from the right side and adding it to the left side in the order they appear in the sorted primitive list. This results in a total of $|\mathcal{P}|-1$ partitions. Every iteration also has to keep track of the partition bounds to compute the SAH cost. Let us assume $B_1, \ldots, B_{|\mathcal{P}|}$ is the array of sorted reference bounds and let $B_a \oplus B_b$ denote the *grow operation* which computes tight bounds for the given bounds B_a and B_b. Then the left bounds B_l^k of the k-th partition are $B_l^k = \bigoplus_{j=1}^{k} B_j$. The respective right bounds are $B_r^k = \bigoplus_{j=k+1}^{|\mathcal{P}|} B_j$. For the left side the partition bounds B_l^k can be incrementally updated when adding primitives to it by simply computing $B_l^k = B_l^{k-1} \oplus B_k$, where $B_l^1 = B_1$. For the bounds B_r^k of the right side it is not possible to simply remove the bounds of the removed primitive from B_r. An efficient way to compute all right partition bounds is to perform a scan from the

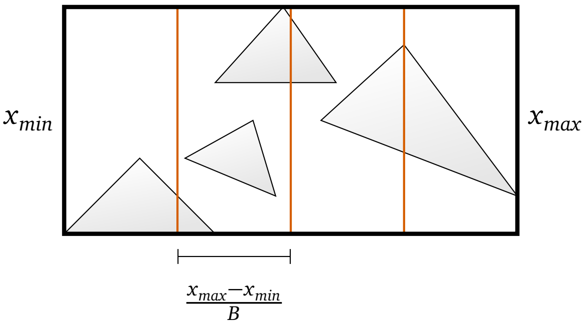


Figure 2.22: Example for a scene which has been subdivided into $B = 4$ bins with three equidistant split planes (red) along the x-axis.

right on the sorted bounds $B_1, \dots, B_{|\mathcal{P}|}$ using the grow operator $\oplus$. That is we compute $B_r^k = B_r^{k+1} \oplus B_{k+1}$, where $B_r^{|\mathcal{P}|-1} = B_{|\mathcal{P}|}$. The resulting array of right partition bounds has to be stored in additional memory. The partition index k is also the number of primitives on the left side of the partition. As BVH construction does not split primitives the number of primitives on the right side of the k-th partition is simply $|\mathcal{P}|-k$. Now we have all required information to compute the SAH cost of all partitions. The candidate partition which gives the lowest SAH cost is chosen. This process is repeated for the y- and z-axis and the overall best candidate is selected. From Figure 2.21 it can already be seen that this procedure can generate partitions with reasonable spatial separation. As with kd-tree construction this sorting-based candidate determination has $O(n \log n)$ complexity resulting in an overall complexity of $O(n \log^2 n)$. It is possible to adapt the concepts of the $O(n \log n)$ approach for kd-tree construction to BVHs to also achieve $O(n \log n)$ SAH-based BVH construction.

2.5.4 Binned Construction

While BVHs and kd-trees constructed with the SAH give significantly higher traversal performance than trees constructed with the spatial- or object-median split strategy SAH-based construction is also the most expensive. Popov et al. [2006] proposed an $O(n \log n)$ SAH-based kd-tree construction algorithm which is faster than the $O(n \log n)$ algorithm from Wald and Havran [2006] as it has a lower constant. This is achieved by sampling the cost function $c(s)$ at a couple of different sample split plane positions s, where the number of sample positions is chosen to be much lower than the number of candidates. This gives an approximation to the full sweep approach. Popov et al. [2006] presented an efficient implementation of this idea which completely replaces all sorting with a $O(n)$ *binning* or *histogram computation* step. They sample $c(s)$ with a number of equidistant split planes with respect to the current bounds of the node to split. Consecutive split planes including the sides of the bounds define equally shaped volumetric bins to which primitives are distributed. Binning with $B \in \mathbb{N}$ bins results in $B-1$ split planes/split candidates. Figure 2.22 depicts this bin construction for an example bin partition with three split planes. Bins have a zero-based index $b \in \{0, \dots, B-1\}$. Exemplary for binning along the x-axis the associated right split plane is at $s_b = x_{min} + (b+1)\frac{x_{max}-x_{min}}{B}$, where $[x_{min}, x_{max}]$ are the current node bounds for this axis. At this point binned kd-tree and BVH construction have minor differences. We will proceed with binned kd-tree construction before we explain binned BVH construction.

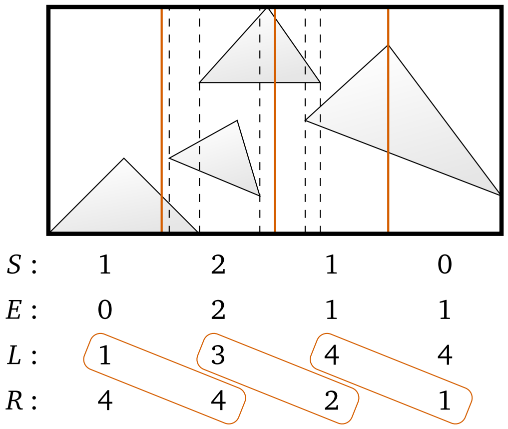


Figure 2.23: Example for binning in kd-tree construction with $B = 4$ bins. Each bin stores the number of start events S and end events E (dashed lines) which fall into it. By performing a prefix sum from the left on S and a prefix sum from the right on E we can efficiently compute the numbers L and R of primitives on the left and right of a candidate partition (red).

Binned Kd-tree Construction For kd-tree construction start and end events of primitives are distributed to bins based on the event position x_e. In the x-axis the target bin b_e can be directly computed from x_e and the node bounds as

$$b_e = \mathtt{inf}\left\{\left\lfloor B\frac{x_e - x_{min}}{x_{max} - x_{min}}\right\rfloor, B-1\right\}. \tag{2.43}$$

Each bin b stores the number of start events S_b and number of end events E_b that fall into it. Thus we have two separate arrays S and E of start and end event counters. Figure 2.23 depicts this for the example bin partitioning in Figure 2.22 including extraction of left and right primitive counts for partition cost computation. To compute the candidate cost for split plane b at position s_b we have to determine the number of primitives to the left and right of this plane. The number of left primitives L_b is the sum $\sum_{i=0}^{b} S_i$ of start event counts of all bins i with index $i \leq b$, while the right count is the sum $\sum_{i=b+1}^{B-1} E_i$ of end event counts of all bins i with index $b < i < B$. These counts can be computed efficiently by performing a prefix sum on S from the left and a prefix sum from the right on E as also depicted in Figure 2.23. As the binning process and the candidate cost computation have both $O(n)$ complexity the whole best split determination has linear time. With a sufficient number of bins hierarchy quality is practically identical to full sweep construction.

Binned BVH Construction Wald et al. [2007] proposed an adaptation of the binned kd-tree construction algorithm to BVHs. Primitive information is distributed to bins w.r.t. the centroid $\mathbf{c}$ of the primitive bounds. Computation of the target bin from the centroid is analogous to event target bin computation of kd-trees. In the x-axis the index of the bin b_c which contains the centroid can be computed from the centroid x-component c_x and the node bounds as

$$b_c = \mathtt{inf}\left\{\left\lfloor B\frac{c_x - x_{min}}{x_{max} - x_{min}}\right\rfloor, B-1\right\}. \tag{2.44}$$

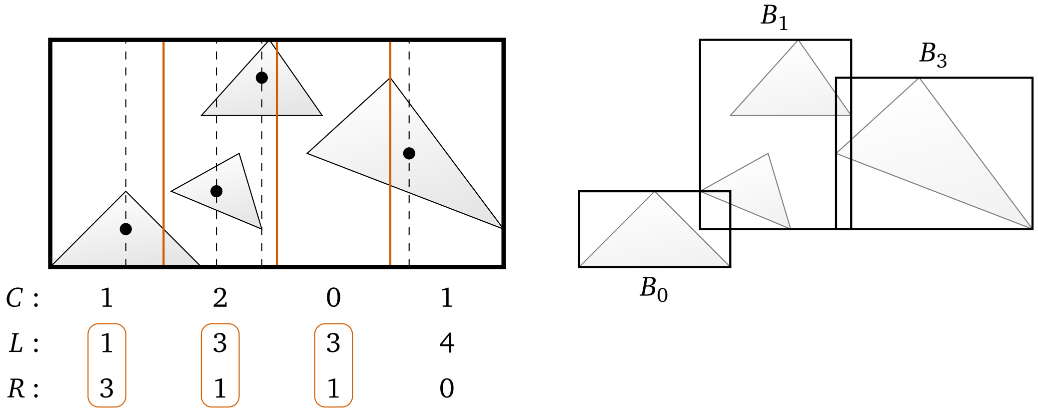


Figure 2.24: Example for binning in BVH construction with $B = 4$ bins. *Left:* Bins store the number C of bounding volume centroids which fall into them. By performing a scan on C we can efficiently compute the number L of primitives on the left of a candidate partition (orange). As primitives are not split the number of primitives on the right of each partition can be directly computed as $R = |\mathcal{P}| - L$, where $|\mathcal{P}|$ is the number of primitives in the current node. *Right:* The bounds B_0 to B_3 stored in each bin are also depicted. The stored bounds B_2 of bin 2 are invalid as no centroid fell into the bin. Bounds for the different partition candidates can be be computed efficiently from the stored bounds.

Each bin b counts the number C_b of centroids it contains and stores tight bounds B_b of all primitives which have their bounds centroid in this bin. This is depicted in Figure 2.24. The number of primitives on the left side of a candidate partition can be computed efficiently by performing a prefix sum on the centroid count array. As with sweeping construction the number of primitives on the right side can be directly computed from the primitives on the left side. Computation of partition bounds is also analogous to sweeping construction. The only difference is that we work on the bounds stored in the bins. Using zero-based numbering of partitions the left bounds of partition k-th can be incrementally computed as $B_l^k = B_l^{k-1} \oplus B_k$, where $B_l^0 = B_0$ and $\oplus$ again is the grow operator. Analogous to the sweeping approach the right partition bounds can be efficiently computed with a scan from the right on the bin bounds using the grow operator. The scan computes $B_r^k = B_r^{k+1} \oplus B_{k+1}$ with $B_r^{B-1} = B_{B-1}$. As was already the case for binned kd-tree construction the whole best split determination process has linear time.

2.5.5 The Minimum-SAM BVH and Treelet-based BVH Optimization

Karras and Aila [2013] proposed a fast and parallel high quality BVH construction algorithm, which first constructs a cheap-to-build low quality BVH, which then is post-processed in a fast and parallel optimization step to yield a high quality BVH. At the core of the optimization step is a minimum-SAM BVH builder, for which Karras and Aila [2013] presented an efficient exhaustive search-based construction algorithm. We briefly introduce the minimum-SAM algorithm and optimization procedure as we refer to them throughout this thesis in different contexts.

For a set of n primitives the algorithm follows a dynamic programming approach which computes optimal splits for smaller subsets to construct optimal BVHs for larger subsets. All intermediate solutions are memoized for reuse. Partitions of the n primitives are en-

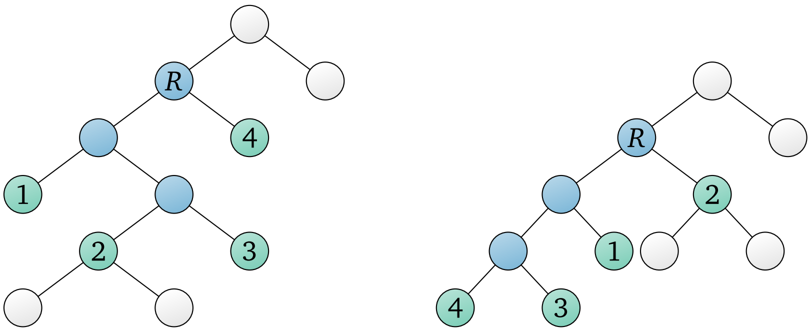


Figure 2.25: Example for treelet formation and optimization in a BVH for a target treelet size of four. *Left:* Starting from some BVH node as the treelet root (R) the treelet adds descendant nodes by some criterion (e.g. largest bounds surface area) until it contains a maximum number of treelet leaves (green). The example treelet has four treelet leaves. *Right:* The inner treelet nodes (blue) are discarded and a new hierarchy is built on the treelet leaves with e.g. the minimum-SAM BVH builder from Karras and Aila [2013].

coded with n-bit bitmasks, where each primitive is assigned a designated bit position, which is set to 0 or 1 to indicate whether the primitive is contained in the partition. The algorithm uses two tables which for each partition store the corresponding optimal BVH SAM cost, and an n-bit mask for the optimal left split of the partition. The bit mask for the optimal right split can be obtained by an XOR of the partition mask and the optimal left mask. The tables are directly indexed by the bitmasks of partitions. As there is a table entry for each bitmask each table has 2^n entries. Thus, the table which stores SAM costs has a space complexity of $O(2^n)$ and the table which stores n-bit bitmasks has a complexity of $O(n2^n)$. This results in an overall space complexity of $O(n2^n)$. The algorithm successively computes optimal splits starting from all 1-sized partitions going up to the full n-sized partitions. In each step a partition looks up all memoized solutions of its previously computed sub-partitions to find an optimal split. The relevant sub-partitions can be extracted from the bitmask of the partition. Starting from the table entry for the complete set of primitives the hierarchy can be extracted from the table, which stores the optimal left splits, by simply following the explicit left and implicit right masks.

As all 2^n table entries have to be computed and each entry requires a non-constant amount of computation the computational complexity of the algorithm is at least $\Omega(2^n)$. Because of the time and space complexity the algorithm is unsuitable even for tiny scenes. For a scene with 32 primitives both auxiliary tables require 16GB each for storing the 32-bit split masks and 32-bit floating point partition subtree costs.

Karras and Aila [2013] locally apply the minimum-SAM algorithm on *treelets*, small sub-hierarchies of a BVH. Starting from some BVH node a treelet is formed by recursively adding node children to the treelet depending on a custom criterion (e.g. largest node bounds surface area). Karras and Aila [2013] grow a treelet until it has 6 to 8 treelet leaves. Then, they optimize the treelet by removing the inner treelet nodes and optimally reorganizing the treelet leaves with the minimum-SAM algorithm. For the targeted treelet sizes the minimum-SAM algorithm is still practical. Treelet formation and optimization is depicted in Figure 2.25. This procedure is applied bottom-up on the whole BVH with each BVH node being a treelet root once. The resulting BVHs achieve quality close to greedy top-down SAH-based construction in much less time.

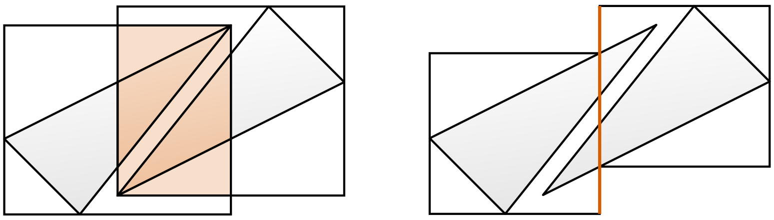


Figure 2.26: Example for partitioning a set of two primitives with an object split(*left*) and a spatial split (*right*). The object split resulted in large node overlap (orange). The spatial split produced smaller node bounds without any overlap. This comes at the cost of duplicated primitives on both sides.

2.5.6 The Spatial Split BVH

The object splits applied by BVH construction algorithms can have problems with separating non-uniformly tessellated geometry or non-axis-aligned shared triangle edges which causes high overlap between nodes. Such geometry is no problem for spatial splits applied by kd-trees as they cannot produce overlap. Stich et al. [2009] proposed to overcome this weakness of BVHs by simply also allowing spatial splits during BVH construction. Figure 2.26 demonstrates both split types with a very simple example similar to a depiction from Stich et al. [2009]. Their algorithm which they call *spatial split BVH* (SBVH) is also an SAH-based top-down construction approach. When searching for a good partition for a set of primitives, they first determine the best object split with a sweeping or binning approach. Next they determine the best spatial split for the same set of primitives. This is done with a binning approach which is a hybrid of kd-tree and BVH binning. It is also computationally much more expensive. As with kd-tree binning each bin counts start and end events. Also each bin stores bounds as with binned BVH construction. But in contrast to binned BVH construction bounds in a bin are not simply grown by the bounds of all primitive references which have their bounds centroid in the bin bounds. Bins store the exact bounds of all geometry clipped to the bin bounds. For this they have to iterate over all primitives and for each primitive have to separately compute its clipped bounds w.r.t. all bins it overlaps. This is a big computational difference to ordinary binning as this incurs a non-constant cost per primitive. Stich et al. [2009] proposed an efficient implementation for this per primitive computation which they called *chopped binning*. An example for the result of chopped binning for a single primitive is shown in Figure 2.27. For more details on chopped binning we refer to the original publication. As a hybrid of kd-tree and BVH binning after all primitives have been binned, scan operations are performed on the start and end event counters and on the stored per bin bounds as described in Section 2.5.4 to compute the SAH costs of the different partitions. An important aspect is that SBVH only applies the best spatial split when it has lower SAH cost than the best object split. BVHs constructed with the SBVH algorithm so far have the highest ray traversal performance. This comes at the cost of a much higher construction time and a higher and unpredictable memory footprint due to duplicate primitives caused by the spatial splits.

2.5.7 The End-Point-Overlap Metric

Aila et al. [2013] analyzed the correlation between measured performance and the SAM cost of BVHs constructed with several BVH construction algorithms. Their motivation

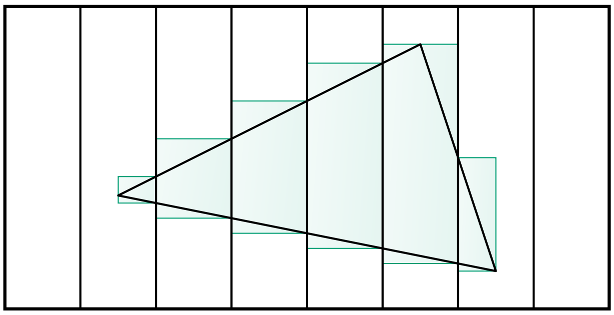


Figure 2.27: Result of chopped binning of a triangle primitive with a node, which is partitioned into eight bins. The procedure computed the tight bounds (green) of the primitive clipped against the bounds of each bin the primitive overlaps. Each bin computes the tight total bounds of all clipped bounds that fall into it.

was the often made observation that more sophisticated construction algorithms that constructed BVHs with lower SAM cost improved measurements less than expected or even decreased performance (e.g., Popov et al. [2009], Walter et al. [2008], and Bittner et al. [2015]). At the same time BVHs with similar SAM cost but constructed with different algorithms can give significantly different trace performance. Aila et al. identified end-point-overlap (EPO) as the missing piece of information and proposed the EPO metric to better predict the performance of BVHs in combination with the SAM. The key observation behind EPO is if the start point and/or end point of a ray lies on a primitive we have to at least process all nodes that overlap with these points. If these points overlap with nodes which are not ancestors of the leaves containing the primitives these nodes cause an extra traversal cost simply because of overlap. The aim of EPO is to directly measure this extra traversal cost caused by overlapping nodes. Regarding the derivation of EPO Aila et al. [2013] "tried many variations of the general idea (e.g. [Stich et al. 2009], [Popov et al. 2009], or using node areas to approximate triangle areas), but EPO was significantly more descriptive than the alternatives". EPO assumes secondary diffuse rays. That is, rays start at surface points. Ray origins and ray intersection points are assumed to be uniformly distributed over all primitive surfaces. The probability density of a surface point $\mathbf{x} \in \mathcal{S}_{\mathcal{P}}$ is $\frac{1}{A_{\mathcal{P}}}$, where $\mathcal{P}$ is the set of scene primitives and $A_{\mathcal{P}}$ is the surface area of the surface $\mathcal{S}_{\mathcal{P}}$ of the union of all primitives in $\mathcal{P}$. The probability of such a random point to be in the bounding volume B_n of a node n is the surface area of the primitive surfaces which overlap B_n times the probability density:

$$p(\mathbf{x} \in \mathcal{S}_{\mathcal{P}} \cap B_n) = \frac{\mathtt{Area}(\mathcal{S}_{\mathcal{P}} \cap B_n)}{A_{\mathcal{P}}}. \tag{2.45}$$

The EPO metric is interested in surface points which are contained in B_n but do not belong to any primitive referenced in the subtree of the node n. These points cause extra traversal costs since n needs to be processed simply because they are contained in the overlap with n. This is depicted in Figure 2.28. Similar to Aila et al. we denote the set of primitives referenced in the subtree of a node n with $\mathcal{Q}(n)$. This allows to define the set of these points as $\mathcal{S}_{\mathcal{P}\setminus\mathcal{Q}(n)} \cap B_n$. Thus, the probability of such a random point to be in a node n is

$$p(\mathbf{x} \in \mathcal{S}_{\mathcal{P}\setminus\mathcal{Q}(n)} \cap B_n) = \frac{\mathtt{Area}(\mathcal{S}_{\mathcal{P}\setminus\mathcal{Q}(n)} \cap B_n)}{A_{\mathcal{P}}}. \tag{2.46}$$

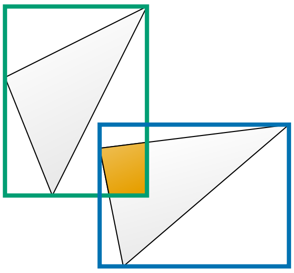


Figure 2.28: Simple primitive overlap example with a scene consisting of two primitives in two leaves. The primitive in the blue leaf overlaps (orange) with the bounds of the green leaf. Rays starting at or intersecting the overlapping surface inevitably also have to process the green leaf. EPO assumes that the probability of this to occur during traversal is the surface area of the overlapping surface divided by the combined surface area of all scene primitives.

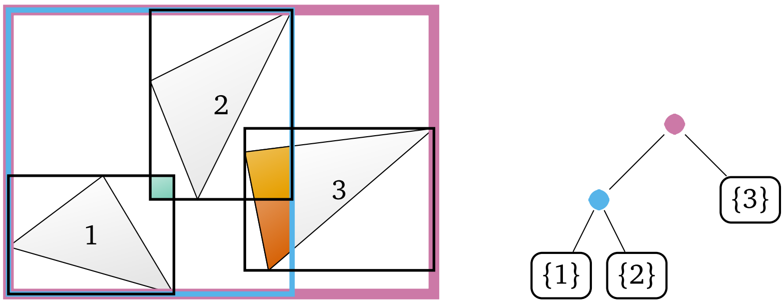


Figure 2.29: Depiction of EPO for a BVH with three primitives in three leaves. In this example EPO is non-zero because of primitive 3. It overlaps with the bounds of leaf 2 (orange) and the blue inner node (orange and red). As the node bounds overlap region of leaf 1 and 2 (green) does not contain any primitive it has no contribution to EPO.

Now, the EPO cost metric for a BVH is defined as the expected extra node processing cost caused by random ray origins or hit points on scene surfaces overlapping with non-ancestral nodes. For a BVH with leaf nodes $\mathcal{L}$ and inner nodes $\mathcal{I}$ this cost is

$$EPO := \sum_{n \in \mathcal{I} \cup \mathcal{L}} C_n \frac{\texttt{Area}(\mathcal{S}_{\mathcal{P} \setminus \mathcal{Q}(n)} \cap B_n)}{A_{\mathcal{P}}}, \tag{2.47}$$

where C_n is defined as

$$C_n = \begin{cases} c_t & n \in \mathcal{I} \\ c_i |n| & n \in \mathcal{L} \end{cases}. \tag{2.48}$$

c_t and c_i are the implementation dependent constants from the SAM. $|n|$ is the number of primitives referenced in a leaf node n. Figure 2.29 depicts EPO computation in a BVH. Aila et al. [2013] proposed a traversal cost predictor p which uses the SAM and EPO cost of a BVH to give a better approximation of traversal cost than SAM alone. The predictor is simply a convex combination of SAM and EPO:

$$p = SAM \cdot (1 - \alpha) + EPO \cdot \alpha. \tag{2.49}$$

Here, $\alpha \in [0,1]$ is a scene dependent constant. Given the actual average measured traversal cost m for a BVH it is assumed that

$$m \sim p \tag{2.50}$$

holds (MacDonald and Booth [1989,1990]). Via the derivation of EPO this predictor is only designed for secondary diffuse rays and scalar traversal. α values as high as 0.98 have been computed by Aila et al. [2013] and also by ourselves (see Table 5.1) meaning that performance can be almost completely governed by EPO in practice. With such high possible values it becomes clear that minimizing SAM is not enough as EPO must be optimized as well. Aila et al. [2013] discovered that top-down greedy SAH-based construction algorithms implicitly reduce node overlap in a way that also minimizes EPO, which gives them an innate superiority over bottom-up or hybrid algorithms. This result is especially important for Chapter 5, where we describe an algorithm which exploits this implicit property. Chapter 5 also explains computation of α, as one of the chapters contributions is the computation of α values, which result in more accurate performance predictions.

2.5.8 Other Metrics

Unlike the EPO metric, which is meant to be used in addition to the SAM, other alternative ray tracing metrics, which aim at introducing more realistic assumptions in the derivation of SAM or SAH, have been developed. Fabianowski et al. [2009] proposed the *scene-interior-ray-origin* (SIRO) heuristic as a possible improvement of the standard SAH. As the name implies this heuristic replaces the assumption of infinitely far away ray origins with the more realistic assumption that ray origins are uniformly distributed inside the scene bounds. This only changes the computation of node intersection probabilities involved in the SAH. Instead of the conventional surface area ratio an elliptic integral has to be solved, which has no closed-form solution. Thus, the authors provide two approximations with a focus on speed or quality. While only evaluated for kd-trees, SIRO should be directly applicable to BVHs as well. Unfortunately, reported kd-tree traversal performance improvement on average is only 2.6% for larger scenes.

Ize and Hansen [2011] proposed an algorithm which improves occlusion ray traversal performance (see Figure 2.4, Section 2.1.1) for a given tree. For every inner node they evaluate an extended SAM. Their metric, which they call ray termination SAH (RTSAH), accounts for the fact that when a ray intersects two sibling nodes traversal can stop when there is a primitive intersection in the first visited sibling's subtree. Each inner node stores a flag to indicate which child more likely produces an intersection with lower cost. The flagged sibling's subtree is traversed first in case both siblings have been intersected during traversal. The technique can be applied to kd-trees as well as BVHs. Speedups of up to a factor of two have been observed. As the left and right child of inner BVH nodes can be swapped without corrupting the hierarchy, we noted that the lower cost child can be indicated by letting it be the left child. This way occlusion ray traversal does not have to be modified and still can always enter the left child first when both children are intersected.

Chapter 3

GPU Hardware Platform

Contents

The GPU algorithms we present in Chapter 7 and Chapter 8 were implemented using the NVIDIA CUDA API[1]. CUDA allows to implement data parallel problems in so called *kernels*, which are executed by millions of GPU threads. According to Flynn's taxonomy [Flynn 1972] on a high level a GPU can be categorized as a multiple-instructions-multiple-data (MIMD) device, as groups of different GPU threads can process different kernel instructions on different data per clock. The groups themselves operate in a single-instruction-multiple-data (SIMD) fashion. Kernels are implemented in CUDA C, which is the language C with some extensions for GPU programming. We give a brief introduction on the basic aspects of GPU programming with CUDA [NVIDIA 2017a]. A non-proprietary platform independent alternative for GPU programming is OpenCL[2]. The Intel SPMD Program Compiler[3] takes a very similar approach to CUDA but is intended for exploiting the vector units of CPUs.

3.1 Kernels, Grids, and Blocks

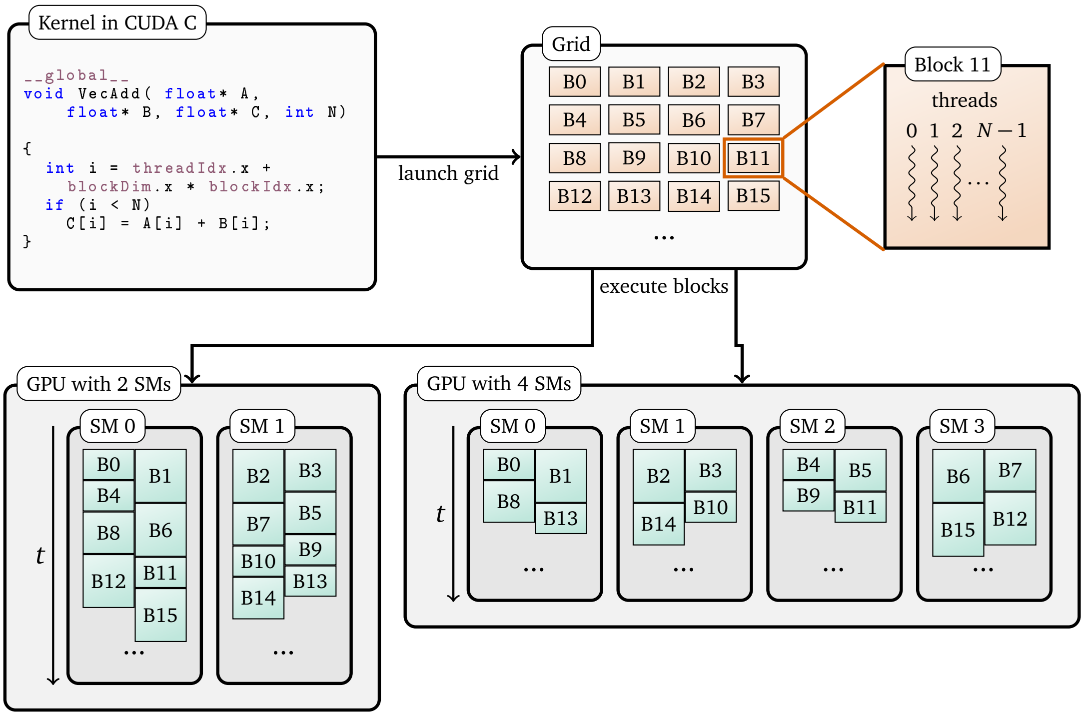
We start our introduction with kernels, grids, blocks, multiprocessors, and their relationships, which are also depicted in Figure 3.1. A CUDA kernel is executed by a *grid* of up to millions of threads and operates on data stored in device memory (GPU memory). The user side which invokes the kernel is called the *host*. A grid is decomposed into user specified equally sized blocks of at most 1024 threads. CUDA C code can access the global

---

[1] `https://developer.nvidia.com/cuda-downloads`

[2] `https://www.khronos.org/opencl/`

[3] `https://ispc.github.io/`



```
__global__
void VecAdd( float* A,
    float* B, float* C, int N)
{
  int i = threadIdx.x +
    blockDim.x * blockIdx.x;
  if (i < N)
    C[i] = A[i] + B[i];
}
```

Figure 3.1: Depiction of the relationship between kernels, grids, blocks, and multiprocessors. To execute a kernel a grid of threads is launched. A grid is decomposed into smaller units called blocks (B) which are waiting (red) to be executed. Each block has the same programmer specified number of threads N. Blocks are executed (green) in parallel by the streaming multiprocessors (SM) of GPUs. The number of blocks a multiprocessor can process at a time depends on different factors. In our example this resulted in two blocks at a time. GPUs can have different numbers of multiprocessors. The example shows two GPUs with 2 and 4 multiprocessors, respectively. A multiprocessor replaces terminated blocks with new blocks waiting for execution. As runtime of blocks can vary new blocks are scheduled dynamically. (Partially based on [NVIDIA 2017a], Figure 5)

index of a block and the block-relative index of threads in a block via built-in variables. Both indices together allow to compute unique global indices to identify the portion of the input data each thread has to process. A GPU possesses $N_{SM} \in \mathbb{N}$ so called multiprocessors which process blocks. A block is mapped to exactly one multiprocessor, which can only process a GPU dependent maximum number of blocks at a time. Thus, only a subset of the millions of threads of a grid is actually active at a time. There is neither a guaranteed processing order for the whole set of blocks nor a predetermined mapping of blocks to multiprocessors. Threads in blocks of different multiprocessors can process different instructions on different data per clock. The GPU dependent maximum number of active threads per multiprocessor is also limited. This maximum number cannot be reached if the block size is not a proper divisor of this limit or is so small that the active block count limit is reached first. In NVIDIA terminology the ratio of active threads to the maximum possible amount of active threads is called *occupancy*. In practice, occupancy is mainly further limited by the *register* and *shared memory* usage of a kernel. Depending on the complexity of a kernel the compiler determines the maximum amount of registers needed per thread for execution of the kernel. Shared memory is a resource that can be allocated per block that we will briefly introduce in Section 3.4. Only a small amount of both resources is available per multiprocessor. At times it can be beneficial to partition kernel code into several smaller kernels which might have higher occupancy and thus possibly higher performance. Using *streams* several kernels can be launched in parallel, if resource usage allows. Streams also allow to asynchronously transfer data to and from the device memory while kernels are running.

3.2 Warps

CUDA implicitly partitions blocks into groups of 32 threads which are called *warps*. Warps are processed by different single-instruction-multiple-data (SIMD) units of a multiprocessor. The actual SIMD width of a SIMD unit varies from GPU to GPU and also depends on the type of instruction. However, to the programmer warps appear to be of SIMD nature with a SIMD width of 32 as threads in a warp are implicitly synchronized after each instruction. Starting with the NVIDIA Fermi GPU architecture each multiprocessor has at least two warp schedulers, which can each issue instructions to different warps at a time. This means that a multiprocessor is a MIMD device itself. As a block consists of more warps than there are warp schedulers only a subset of the active threads per multiprocessor is issuing instructions at a time. But the larger amount of active threads is still necessary. GPUs are optimized for high instruction throughput at the cost of different kinds of high latencies. SIMD units rely on switching between warps in the pool of active threads to hide those latencies by issuing instructions of other ready warps. If enough instruction level parallelism is available warps have to be switched less often and full performance can be achieved with lower occupancy.

3.3 SIMD and SIMT

While multiprocessors essentially posses SIMD units NVIDIA coined the term *SIMT* for *single-instruction-multiple-threads* as there are some differences to traditional SIMD in terms of programming and hardware. On the programming side the *CUDA C* programming language, in which kernels are programmed, mainly expresses kernels from the point of view of a single scalar thread, which has a global ID to identify the data it has to process. Thus, code is independent of the SIMD-width. In contrast, SIMD programming directly exposes the underlying SIMD-width in its separate sets of instructions (or intrinsic functions). The hardware implementations of Intel's SSE, AVX, and AVX-512 technologies have 4-, 8-, and 16-wide SIMD units, respectively [Intel 2017]. Code has to be specifically adapted to the SIMD width of the used technology. On the hardware side each thread or lane in an active warp has its own set of registers and, more importantly, own program counter. This already enables a lane to define a single thread of execution, justifying the term thread. Reading from (gather) or writing to (scatter) individual memory addresses per lane is also directly supported in hardware and in contrast to SIMD programming does not require explicit handling in software.

What SIMT has in common with SIMD is potentially lower SIMD efficiency with conditional code. As the name implies SIMT can only execute a single instruction on multiple threads at a time. Execution diverges when different threads in a warp take different con-

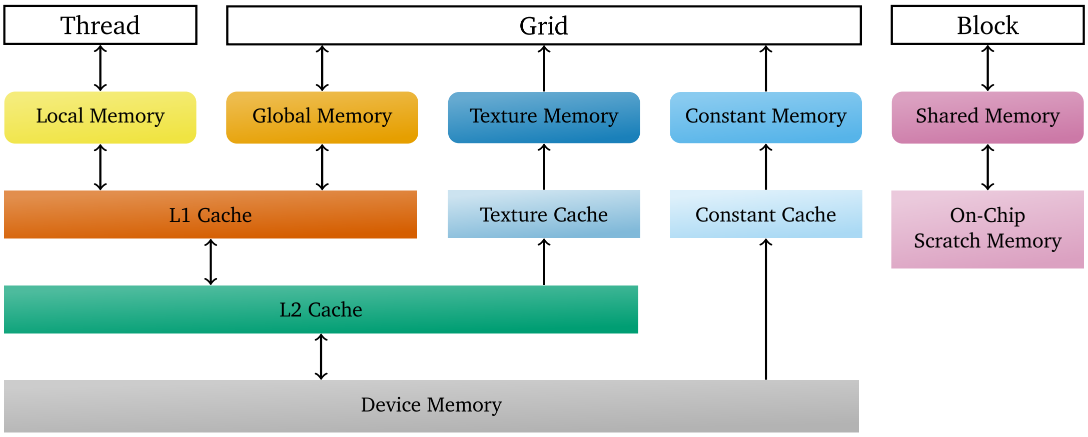


Figure 3.2: Diagram of the different memory spaces (rounded boxes) and their scopes (white boxes). The underlying hierarchy of caches and device memory is depicted as well. Arrows indicate read/write or read-only access. The presence of the L1 cache varies between GPU series and also between models in a series. (Partially based on [NVIDIA 2017a], Figure 7)

trol flow paths. The separate branches have to be executed one after the other. Deeply nested conditional statements can lead to complete serialization of the execution of a warp. While with traditional SIMD units the programmer has to manually create lane masks to deactivate SIMD lanes, SIMT automatically handles masking in hardware. If all threads decide on the same branch of a conditional statement no divergence occurs. But there is a key difference to SIMD. As every SIMT lane has its own program counter execution automatically continues after the conditional statement after the branch has been processed. The SIMD programmer has to explicitly branch past the untaken branch to prevent unnecessary issuing of instructions where all lanes are deactivated.

3.4 Memory Spaces

CUDA provides access to different memory spaces, which differ in purpose, scope, and access characteristics. We give a brief introduction to each memory space. Figure 3.2 gives an overview on the scope of the memory spaces and their hierarchical relationship with caches and device memory.

Device and Global Memory Device memory is the off-chip main memory of a GPU. Essentially every CUDA kernel at least works on device memory, as it is the only memory area which allows to exchange data with the host. While it is possible to directly read from and write to system memory from the GPU, this is unpreferable as bandwidth through the PCI express bus is about two orders of magnitudes lower than for device memory access. From the scope of a grid, device memory is also called *global* memory as it is globally readable and writable for all threads in a grid.

The first generation of CUDA-enabled GPUs had very strict so-called *coalescing rules* for efficient global memory access which we will not discuss here. All following generations

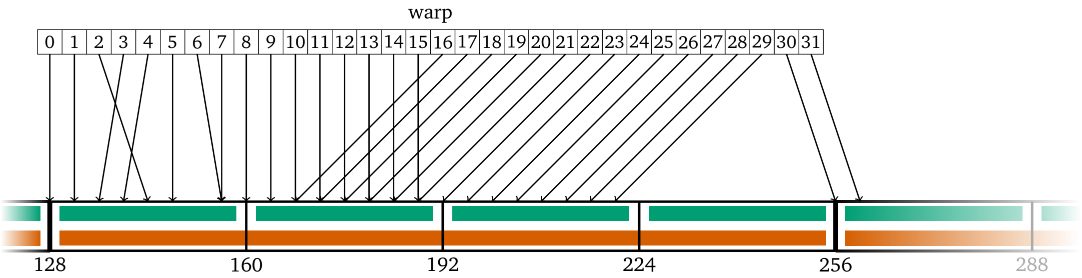


Figure 3.3: Depiction of a warp accessing global memory via the L1 cache (red) or the L2 cache (green). The L1 cache access results in two 128 byte transactions as the warp accesses two 128 byte L1 cache lines. On a miss, 256 bytes would have to be loaded from the L2 cache. The L2 cache access results in four 32 byte transactions as the warp accesses four 32 byte L2 cache lines. (Based on [NVIDIA 2017a], Figure 16)

have an L2 cache and some have an L1 cache which have much simpler coalescing rules. The presence of the L1 cache varies between GPU series and also between models in a series. Global memory is divided into segments of 32 bytes for the L2 cache and 128 bytes for the L1 cache. When threads in a warp access global memory, the access is simply split into as many memory transactions as different segments are accessed. Figure 3.3 depicts this for L1 and L2 accesses. In the worst case each thread in a warp accesses a different segment, which results in 32 transactions. Thus, to keep the number of transactions low threads in a warp should access nearby addresses, or related data should be kept closer together.

Local and Constant Memory *Local memory* and *constant memory* are two additional memory types, which reside in device memory. Local memory derives its name from the fact that it has thread local scope. It is mainly used for register spilling if a kernel uses too many registers, or thread scope arrays, which have no static access pattern. The thread local traversal stack used in the GPU ray tracing kernels from Aila and Laine [2009], for example, ends up in local memory, as it is accessed in an unpredictable manner. Constant memory has a designated cache, which is optimized for multiple simultaneous 4-byte accesses to the same address. Thus, it is meant for constant data that is needed by several threads at the same time. Simultaneous accesses to multiple addresses are serialized into the number of different addresses.

Texture Memory *Texture memory* is the last type of memory, that resides in device memory. It allows 1-, 2-, or 3- dimensional indices for addressing with optimized performance of lookups in a 2D or 3D neighborhood. For the last two variants the input data first has to be converted into a *CUDA Array*, which stores the data in an optimized opaque proprietary memory layout. All NVIDIA GPUs access texture memory via an additional dedicated read-only L1 cache. The CUDA programming guide is unspecific regarding optimal texture memory access patterns. The only hint is that "[if] the memory reads do not follow the access patterns that global or constant memory reads must follow to get good performance, higher bandwidth can be achieved providing that there is locality in the texture fetches" [NVIDIA 2017a]. As we will see in Chapter 7, Section 7.2.1 our experiments with certain access patterns, which are bad for either global or both global and shared memory,

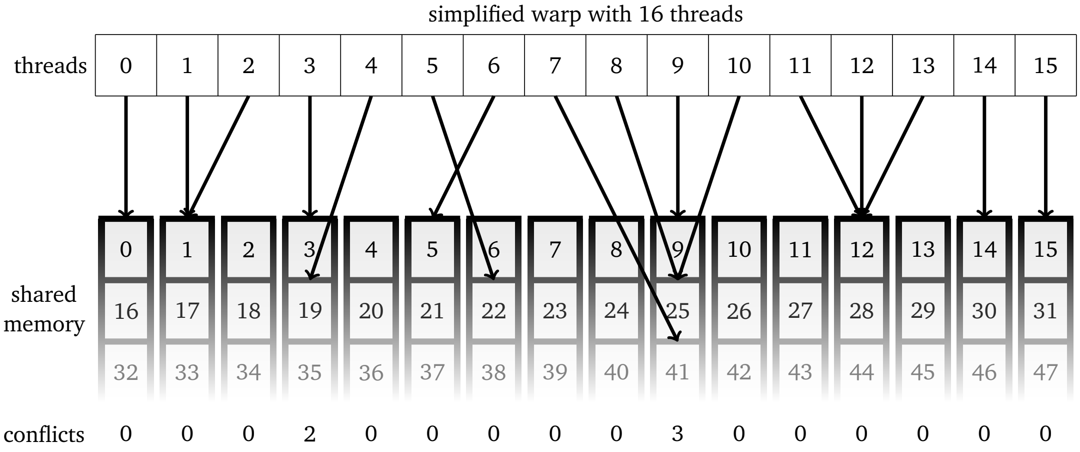


Figure 3.4: Depiction of the organization of shared memory into banks and access patterns in a warp which cause bank conflicts. For clarity the warp size and number of banks is reduced to 16. Consecutive 4-byte words are periodically assigned to the memory banks. Threads 3 and 4 cause a two-way bank conflict as they access two different words in the same bank. Threads 7, 8, 9, and 10 cause a three-way bank conflict as they access three different words in the same bank. (Based on [NVIDIA 2017a], Figure 18)

reveal an almost equal performance for texture memory with either access pattern. From its computer graphics origins texture memory provides some additional hardware features such as different addressing modes, value interpolation, and unpacking of specially stored data. These features are not important in the context of this dissertation.

Shared Memory *Shared memory* is located on-chip and as such has “much higher bandwidth and much lower latency than local or global memory” [NVIDIA 2017a]. It has block scope and is intended to be shared between threads in a block in a cooperative way. The amount of available shared memory is only a couple of kilobytes per multiprocessor. It is organized in 32 banks, which can be accessed simultaneously. Consecutive 4-byte words are periodically assigned to the 32 banks. That is, shared memory addresses which are 128 byte apart are assigned to the same bank. The bandwidth of each bank is one 4-byte word per clock. Multiple threads are allowed to access different banks or the same bank. When several threads access different words which are assigned to the same bank a *bank conflict* occurs. In this case the accesses to the different words have to be serialized, which effectively reduces the instruction throughput. Thus, efficient shared memory usage aims at reducing the number of bank conflicts. Figure 3.4 depicts the shared memory organization and conflicting access patterns. Common strided access patterns of the form `threadIdx*stride` cause bank conflicts if `stride` has common divisors with the number of banks and should be avoided if possible. The worst case is if the stride is the number of banks itself, in which case we have the number of banks as many conflicts.

3.5 Block Cooperation and Synchronization

Threads in a block can work cooperatively by exchanging data via slow global memory or the fast on-chip shared memory. To avoid data hazards between threads, special synchronization instructions have to be explicitly inserted into the kernel code. While choosing larger block sizes allows more threads to cooperate, block synchronization time increases. Also the occupancy of a multiprocessor temporally decreases the more warps arrive at the synchronization barrier. This can reduce the latency hiding efficiency of a multiprocessor.

Section 3.2 mentioned that threads in a warp are implicitly synchronized after each instruction. This implicit synchronization can be used to avoid explicit synchronization if a problem can be partitioned into warps in a meaningful way. The downside of this is that kernel code is not oblivious to SIMT-width anymore and might break if future GPUs have a different SIMT-width. The implementations of our GPU algorithms in Chapter 8 exploit implicit warp-level synchronicity to achieve higher performance. CUDA provides special hardware supported *warp voting* and *warp shuffle* functions which allow to efficiently exchange data between threads in a warp without additional memory and explicit synchronization, but are out of scope of this introduction. According to [NVIDIA 2017a], Section H.6.2 future NVIDIA GPU generations remove the implicit warp synchronization. Instead, special warp synchronization functions, which also allow sub-warp synchronization, have to be used.

Chapter 4

On the Geometric Probability Function of the Surface Area Metric

Contents

In this chapter we will shed some light on the origins and background of the conventional surface area ratio based intersection probability function for intersecting node bounds (Equation 2.32), which is at the core of the surface area metric and heuristic. We will see that there are two equivalent approaches for its derivation, which result in the ratio of node bound surface areas. Considering a fixed ray direction the probability that a ray with random origin intersects a node contained in another node depends on the ratio of the surface area of the *projections* of the bounds w.r.t. the plane, which has the fixed ray direction as its normal. In general, this directional intersection probability varies with the ray direction. The conventional definitions do not take this directional variation into account. Based on the same assumptions underlying the conventional probability we define and analyze an alternative conditional probability which includes the directional dependence in its derivation. Finally we discuss the relationship of the conventional and alternative probability.

4.1 The Conventional Conditional Intersection Probability

In Section 2.5.2 we saw that the conventional conditional intersection probability is based on two assumptions regarding the distribution of ray origins and ray directions:

1. Rays originate infinitely far away from the scene.

2. Ray directions have a uniform distribution.

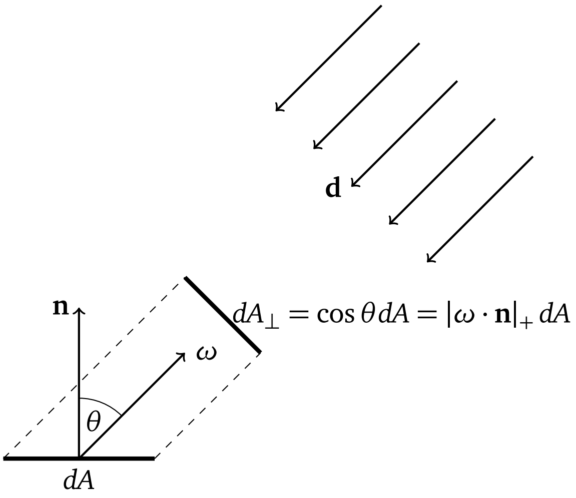


Figure 4.1: Computation of the projected visible surface area $dA_\perp$ of a surface element dA with normal $\mathbf{n}$ for rays originating from direction ω with ray direction $\mathbf{d} = -\omega$.

Based on these two assumptions, the conditional geometric probability that a ray r intersects a convex body B_1 given that it intersects another convex body B_2 with $B_1 \subseteq B_2$ is defined as the ratio of the surface area of B_1 to the surface area of B_2:

$$p\,(r \Cap B_1 \mid r \Cap B_2) := \frac{\texttt{Area}(B_1)}{\texttt{Area}(B_2)}.$$

For brevity we used the notation $r \Cap B$ for the predicate $r \cap B \neq \emptyset$ or "r intersects B".

While this probability is an important part of the SAM and SAH there is no consensus on the origins of this result. Goldsmith and Salmon [1987] and MacDonald and Booth [1989,1990] simply state this result as a fact. Other graphics researchers (Arvo and Kirk [1989], Havran [2000], Wald and Havran [2006], and Fabianowski et al. [2009]) refer to results from different publications in *integral geometry* (Kendall and Moran [1963], Santaló [1976], Solomon [1978], Cazals and Sbert [1997], and Hulst [1981]). These can be boiled down to two essentially equivalent approaches.

4.1.1 Expected Projected Visible Area Approach

The first approach is based on the expected projected visible area of a convex body. As a consequence from Assumption 1 we only have to deal with ray directions to compute the projected area of a convex body B. For convenience we work with directions ω from which infinitely far away rays with direction $\mathbf{d} = -\omega$ originate. Given a surface element dA with surface normal $\mathbf{n}$ the projected visible surface area with respect to rays *from* direction ω is

$$dA_\perp = |\omega \cdot \mathbf{n}|_+\, dA. \tag{4.1}$$

Here, $|\omega \cdot \mathbf{n}|_+$ is the *clamped dot product*, which clamps negative results to zero. That is, we only consider one-sided surfaces. Figure 4.1 depicts the computation of $dA_\perp$. To obtain the projected area for the whole convex body B and a direction ω we have to integrate over its surface $\mathcal{S}_B$:

$$A_\perp(\omega,B)=\int_{\mathcal{S}_B} dA_\perp=\int_{\mathcal{S}_B} |\omega\cdot\mathbf{n}(\mathbf{x})|_+\,dA \tag{4.2}$$

To compute the expected visible projected area of B we have to integrate over all directions $\omega\in\mathcal{S}^2$. As according to Assumption 2 ray directions $\mathbf{d}=-\omega$ are uniformly distributed the probability density function for ω is the constant function $p(\omega)=\frac{1}{4\pi}$. Thus, the expected value of Equation 4.2 is

$$\begin{aligned}\mathtt{E}\left[A_\perp(\omega,B)\right]&=\int_{\mathcal{S}^2} A_\perp(\omega,B)p(\omega)\,d\omega\\&=\frac{1}{4\pi}\int_{\mathcal{S}^2}\int_{\mathcal{S}_B}|\omega\cdot\mathbf{n}(\mathbf{x})|_+\,dA\,d\omega\\&=\frac{1}{4\pi}\int_{\mathcal{S}_B}\int_{\mathcal{S}^2}|\omega\cdot\mathbf{n}(\mathbf{x})|_+\,d\omega\,dA\end{aligned}\tag{4.3}$$

As B is convex there can be no self-occlusion and the inner integral is the same for all $\mathbf{x}\in\mathcal{S}_B$. Integration has only to be done on the positive hemisphere $\Omega(\mathbf{n})$ with respect to the surface normal as the surface is one sided. Arbitrarily choosing a local coordinate system at every point $\mathbf{x}$ where the normal $\mathbf{n}(\mathbf{x})$ corresponds to the y-axis, every direction ω in the hemisphere $\Omega(\mathbf{n})$ can be transformed to a direction ω' in the unifying hemisphere $\Omega_y=\{\omega\in\mathcal{S}^2\mid\omega_y\geq 0\}$. Putting everything together and converting to spherical coordinates we can solve the inner integral:

$$\begin{aligned}\int_{\mathcal{S}^2}|\omega\cdot\mathbf{n}|_+\,d\omega&=\int_{\Omega(\mathbf{n})}\omega\cdot\mathbf{n}\,d\omega\\&=\int_{\Omega_y}\omega'\cdot\begin{pmatrix}0&1&0\end{pmatrix}^T d\omega'=\int_{\Omega_y}\omega'_y\,d\omega'\\&=\int_0^{2\pi}\int_0^{\frac{\pi}{2}}\cos\theta\sin\theta\,d\theta\,d\phi\\&=\int_0^{2\pi}\int_0^1 u\,du\,d\phi\\&=\pi.\end{aligned}\tag{4.4}$$

Reinserting into Equation 4.3 we get:

$$\begin{aligned}\mathtt{E}\left[A_\perp(\omega,B)\right]&=\frac{1}{4\pi}\int_{\mathcal{S}_B}\int_{\mathcal{S}^2}|\omega\cdot\mathbf{n}(\mathbf{x})|_+\,d\omega\,dA\\&=\frac{\pi}{4\pi}\int_{\mathcal{S}_B}1\,dA\\&=\frac{1}{4}\mathtt{Area}(B).\end{aligned}\tag{4.5}$$

That is, the expected projected area of a convex body B and a random ray r is one forth of the surface area of B. From this result the conditional geometric probability that a ray r

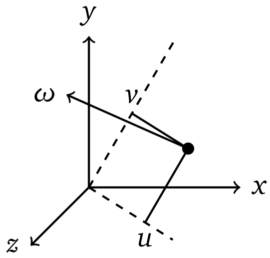


Figure 4.2: Depiction of the line parametrization chosen by Cazals and Sbert [1997] for the differential line measure $dL = d\omega\, du\, dv$. ω is the line direction. u and v are the coordinates of the intersection of the line with a plane through the origin which has the line direction ω as its plane normal. (Based on Cazals and Sbert [1997], Figure 3 (c))

intersects a convex body B_1 given that it intersects another convex body B_2 with $B_1 \subseteq B_2$ is the ratio of the expected projected surface area of B_1 to the expected projected surface area of B_2:

$$\begin{aligned} p(r \Cap B_1 \mid r \Cap B_2) &= \frac{\mathtt{E}[A_\perp(r, B_1)]}{\mathtt{E}[A_\perp(r, B_2)]} \\ &= \frac{\mathtt{Area}(B_1)}{\mathtt{Area}(B_2)}. \end{aligned} \tag{4.6}$$

4.1.2 Measure Theory Approach

The second approach is based on measure theory. It states that the intersection probability is defined as the measure of lines intersecting B_1 divided by the measure of lines intersecting B_2:

$$p\,(r \Cap B_1 \mid r \Cap B_2) := \frac{\mu\,(B_1)}{\mu\,(B_2)}. \tag{4.7}$$

That is, with this interpretation the conditional probability is not an actual probability, but rather the relative "*amount*" or fraction of rays intersecting B_2 which also intersect B_1. Following Cazals and Sbert [1997] the differential measure of a line on the set of all lines $\mathcal{L}$ is $dL = d\omega\, du\, dv$. The corresponding line parametrization is depicted in Figure 4.2. To compute $\mu(B)$ for a convex body B we have to integrate all lines which intersect B. For a given direction ω this simply results in integrating the projected area of B in this direction. This gives

$$\mu(B) = \int_{\mathcal{L} \cap B} dL = \int_{\mathbb{S}^2} \int_{\mathcal{A}_\perp^\omega(B)} du\, dv\, d\omega = \int_{\mathbb{S}^2} A_\perp(\omega, B) d\omega, \tag{4.8}$$

where $\mathcal{A}_\perp^\omega(B)$ is the set of points of the projection of B on the plane with normal ω and $A_\perp(\omega, B)$ is the projected area function from Equation 4.2. We can see that this is the same integral as in Equation 4.3 without the factor of $\frac{1}{4\pi}$. Thus, the solution to the integral is the result of Equation 4.5 multiplied by 4π. This gives

$$\mu(B) = 4\pi \frac{1}{4} \mathtt{Area}(B) = \pi \mathtt{Area}(B). \tag{4.9}$$

Reinserting into the measure-based probability definition gives

$$p\,(r \Cap B_1 \mid r \Cap B_2) := \frac{\mu\,(B_1)}{\mu\,(B_2)} = \frac{\pi \mathtt{Area}\,(B_1)}{\pi \mathtt{Area}\,(B_2)} = \frac{\mathtt{Area}\,(B_1)}{\mathtt{Area}\,(B_2)}. \tag{4.10}$$

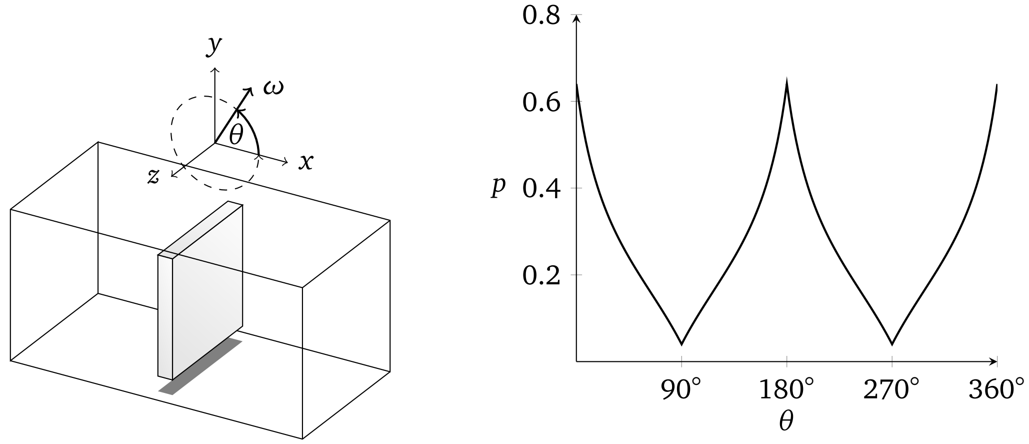


Figure 4.3: Example for the directional conditional intersection probability $p(\omega, B_1, B_2)$ from Equation 4.11 for a pair of axis aligned bounding boxes $(B_1, B_2), B_1 \subseteq B_2$. B_2 has x-, y-, and z-dimension extents $\mathbf{e}_{parent} = (2, 1, 1)$. B_1 has extents $\mathbf{e}_{child} = (0.1, 0.8, 0.8)$. p is evaluated for directions $\omega = (\cos\theta, \sin\theta, 0), \theta \in [0°, 360°]$ in the xy-plane. The left image depicts this setup, while the right plot shows the resulting directional probabilities.

4.1.3 Comparison

We can see that both probability definitions are equivalent though they are derived with different approaches. Both approaches do not consider that the probability of intersecting an object can be different for different directions. The measure approach sums up the amount of all different rays intersecting B_1 and B_2 separately and computes the relative amount intersecting B_1. The projected area approach essentially replaces B_1 and B_2 with spheres which have the same projected area as the respective expected projected area of B_1 and B_2. This eliminates any directional variation, as spheres have the same projected area from all directions.

4.2 Expected Direction Dependent Conditional Probability

Motivated by the observations from the previous section we investigated the possibility to define a conditional probability which accounts for directional variation in the likelihood of intersecting two objects. Our approach is also based on projected areas and can be derived from the first two SAM assumptions as well. For a fixed direction ω which rays originate from and two convex bodies $B_1, B_2, B_1 \subseteq B_2$ we can define a *direction dependent* conditional probability based on projected area:

$$p(\omega, B_1, B_2) := \frac{A_\perp(\omega, B_1)}{A_\perp(\omega, B_2)} \tag{4.11}$$

An example for the possibly strong directional variation of this directional probability is given in Figure 4.3. From this direction dependent conditional probability we can define a direction independent probability as the expected direction dependent conditional probability for uniformly distributed directions ω:

$$p(r \Cap B | r \Cap A) := \mathrm{E}\left[p(\omega, B_1, B_2)\right] = \mathrm{E}\left[\frac{A_\perp(\omega, B_1)}{A_\perp(\omega, B_2)}\right] \tag{4.12}$$

Contrary to Equation 4.5 this is not a ratio of expected values, but the expected value of a ratio, which is not equivalent to the former. We denote this alternative probability p_a to distinguish it from the conventional probability, which we denote p_c. Unfortunately, p_a cannot be evaluated analytically in general and must be evaluated numerically. We identified three special cases, which have an analytical solution. The solution of these cases turned out to be identical to p_c.

Case 1: Uniform Projected Area The first case is when B_2 has uniform projected area, that is, $A_\perp$ constantly has value $c \in \mathbb{R}_+$ for all ω. As a result $p(\omega, B_1, B_2)$ is not a non-polynomial rational function anymore and can be integrated:

$$\mathrm{E}\left[\frac{A_\perp(\omega, B_1)}{A_\perp(\omega, B_2)}\right] = \mathrm{E}\left[\frac{A_\perp(\omega, B_1)}{c}\right] = \frac{\frac{1}{4}\mathtt{Area}(B_1)}{c}. \tag{4.13}$$

The only body which has this property is a sphere, where the projection from all directions is a circle. Thus, for a sphere with radius r_s the projected area function is $A_\perp^{Sphere}(\omega) = c = \pi r_s^2$. Inserting into Equation 4.13 gives

$$\frac{\frac{1}{4}\mathtt{Area}(B_1)}{\pi r_s^2} = \frac{\frac{1}{4}\mathtt{Area}(B_1)}{\frac{4}{4}\pi r_s^2} = \frac{\frac{1}{4}\mathtt{Area}(B_1)}{\frac{1}{4}\mathtt{Area}(Sphere)}, \tag{4.14}$$

which is the conventional probability for a convex body in a sphere.

Case 2: Isotropically Scaled Bounds The second case is when B_1 is an isotropically scaled version of B_2. In this case $A_\perp(\omega, B_1) = sA_\perp(\omega, B_2), s \in \mathbb{R}$ holds and $p(\omega, B_1, B_2)$ constantly has a fixed value $s \in [0, 1]$. This case is depicted for two AABBs in Figure 4.4. As can be seen in the example both probabilities also do not deviate much from each other. While p_a is about 25% larger when the child box degenerates to a quad the difference of p_a and p_c is very low.

Case 3: Axis Aligned Bounding Box in Axis Aligned Bounding Cube The third case with an analytical solution of p_a is when B_1 is an AABB and B_2 is an *axis aligned bounding cube* (AABC). The projected area function of an AABB can be defined as

$$A_\perp^{AABB}(\omega) = A_x|\omega_x| + A_y|\omega_y| + A_z|\omega_z|, \tag{4.15}$$

where $A_d, d \in \{x, y, z\}$ is the area of an AABB face with normal direction parallel to the d-coordinate axis. In case of a cube all faces have the same area A_c. Thus, the directional probability is

$$p(\omega, B_1, B_2) := \frac{A_\perp^{AABB}(\omega)}{A_\perp^{AABC}(\omega)} = \frac{A_x|\omega_x| + A_y|\omega_y| + A_z|\omega_z|}{A_c(|\omega_x| + |\omega_y| + |\omega_z|)} \tag{4.16}$$

For uniformly distributed random ray directions the expected value of this function is

$$\mathrm{E}\left[\frac{A_\perp^{AABB}(\omega)}{A_\perp^{AABC}(\omega)}\right] = \frac{1}{4\pi}\int_{S^2} \frac{A_x|\omega_x| + A_y|\omega_y| + A_z|\omega_z|}{A_c(|\omega_x| + |\omega_y| + |\omega_z|)}\, d\omega \tag{4.17}$$

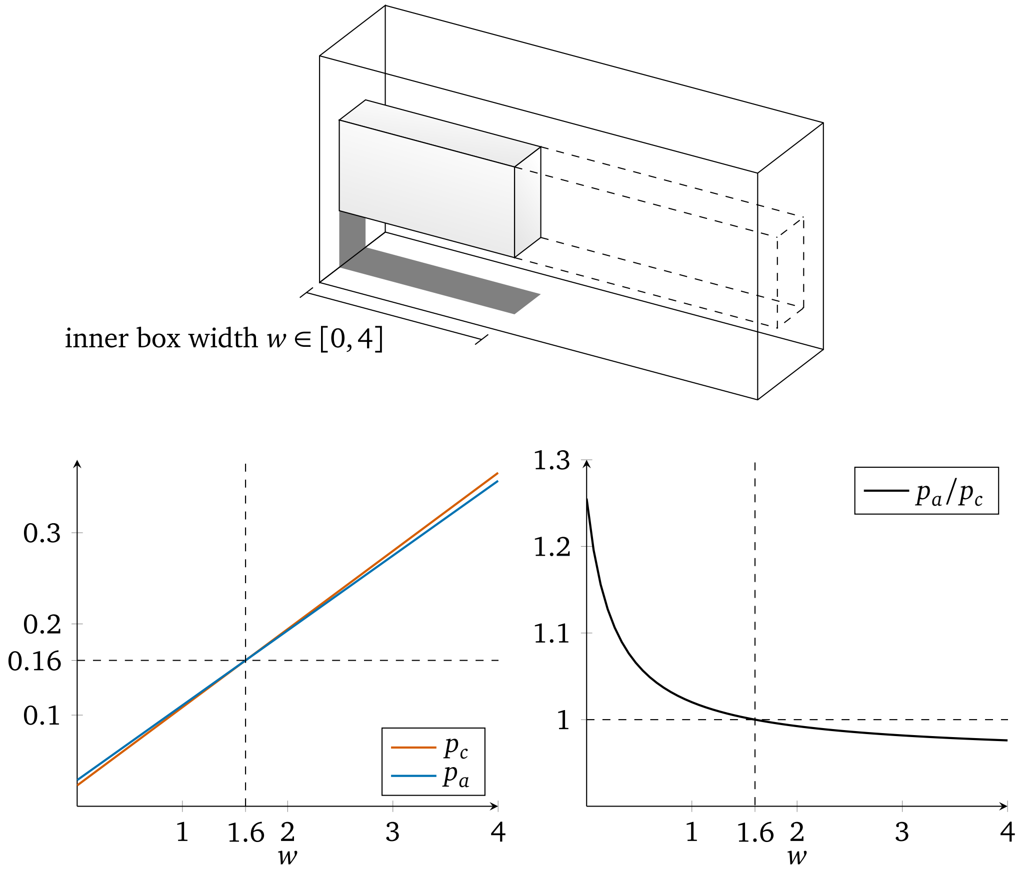


Figure 4.4: Exemplary comparison of the conventional intersection probability p_c and our numerically evaluated alternative intersection probability p_a for a pair of axis aligned bounding boxes. The parent box has fixed extents $\mathbf{e}_{parent} = (4,2,1)$. The child box has extents $\mathbf{e}_{child} = (w, 0.8, 0.4)$ with varying width $w \in [0,4]$. At $w = 1.6$ the child box is an isotropically scaled version of the parent box with $\mathbf{e}_{child} = 0.4\,\mathbf{e}_{parent}$. Thus, both intersection probabilities have an identical value of $0.4^2 = 0.16$ at this width.

Due to the symmetry of the absolute values, the integral can be reduced to integration over the set $\Omega^+ = \{\omega \in \mathcal{S}^2 \mid \omega_x \geq 0 \wedge \omega_y \geq 0 \wedge \omega_z \geq 0\}$ of directions with positive components. This allows us to write the integral as follows:

$$\begin{aligned} \mathrm{E}\left[\frac{A_\perp^{AABB}(\omega)}{A_\perp^{AABC}(\omega)}\right] &= \frac{1}{4\pi} 8 \int_{\Omega^+} \frac{A_x\omega_x + A_y\omega_y + A_z\omega_z}{A_c(\omega_x + \omega_y + \omega_z)}\, d\omega \\ &= \frac{2}{A_c\pi}\left(A_x\Phi_x + A_y\Phi_y + A_z\Phi_z\right), \\ \Phi_d &= \int_{\Omega^+} \frac{\omega_d}{\omega_x + \omega_y + \omega_z}\, d\omega, \quad d \in \{x,y,z\}. \end{aligned} \tag{4.18}$$

To compute the expected value we have to solve the integrals Φ_d. Though, to the best of our knowledge, these integrals cannot be integrated analytically we still managed to find a solution. The first step is that we can compute the sum of all Φ_d:

$$\Phi_x + \Phi_y + \Phi_z = \int_{\Omega^+} \frac{\omega_x + \omega_y + \omega_z}{\omega_x + \omega_y + \omega_z} \, d\omega = \int_{\Omega^+} 1 \, d\omega = \frac{\pi}{2}. \tag{4.19}$$

The next step is based on the observation that sphere parametrization with spherical coordinates is not unique. That is, for every Φ_d we can choose a different spherical coordinates-based parametrization for ω. By setting ω_d to $\cos\theta$ and the other components to $\sin\theta\cos\phi$ and $\sin\theta\sin\phi$ we can write every $\Phi_d, d \in \{x, y, z\}$ in the same form:

$$\Phi_d = \int_{\Omega^+} \frac{\omega_d}{\omega_x + \omega_y + \omega_z} \, d\omega = \int_0^{2\pi} \int_0^{\frac{\pi}{2}} \frac{\cos\theta \sin\theta}{\sin\theta\cos\phi + \cos\theta + \sin\theta\sin\phi} \, d\theta \, d\phi. \tag{4.20}$$

Consequently, the value of all Φ_d are identical. This allows us to solve Equation 4.19 for Φ_d:

$$\begin{aligned} \Phi_x + \Phi_y + \Phi_z = 3\Phi_d = \frac{\pi}{2} \\ \Rightarrow \Phi_d = \frac{\pi}{6}. \end{aligned} \tag{4.21}$$

Inserting this result in Equation 4.18 we finally arrive at

$$\begin{aligned} \mathrm{E}\left[\frac{A_\perp^{AABB}(\omega)}{A_\perp^{AABC}(\omega)}\right] &= \frac{2}{A_c\pi}\left(A_x\Phi_x + A_y\Phi_y + A_z\Phi_z\right) \\ &= \frac{2}{A_c\pi}\frac{\pi}{6}\left(A_x + A_y + A_z\right) \\ &= \frac{2\left(A_x + A_y + A_z\right)}{6A_c} \\ &= \frac{\mathtt{Area}(AABB)}{\mathtt{Area}(AABC)}. \end{aligned} \tag{4.22}$$

That is, in the particular combination of an AABB contained in an AABC p_a is identical to the conventional probability.

Evaluation and Discussion

We evaluated the applicability of p_a for SAH-based BVH construction with the plane-sweep algorithm from MacDonald and Booth [1989,1990] and AABBs as bounding volumes. As we had to numerically evaluate p_a for every split candidate construction time increased drastically. At the same time the traversal performance was essentially the same as for hierarchies constructed with p_c. This is not surprising considering the results from the example in Figure 4.4, where deviations from the isotropically scaled case did not result in significant absolute differences between p_a and p_c. Case 2 also should result in almost identical probabilities with nearly cubic parent bounds, which additionally causes the small differences in the hierarchies. Thus, unfortunately, at least in our experiments p_a does not provide a clear advantage over p_c.

4.3 Including Parent Intersection Likelihood

The directional conditional probability $p(\omega, B_1, B_2)$ from Equation 4.11 can display a strong directional variation as can be seen in the example in Figure 4.3. It is conditionally more likely to intersect the child box from the x-direction than from the y-direction. But, at the same time only half as much random rays actually intersect the parent box from the x-direction than from the y-direction as A_x of the parent is only half the size of A_y. That is, for different directions ω the directional probability $p(\omega, B_1, B_2)$ should be weighed differently.

Our attempt to account for this is to redefine the direction probability density function $p(\omega)$. We redefine it as the probability that from all randomly generated rays with uniform direction distribution *which intersect* B_2, a ray has direction ω. As for a particular direction ω the amount of randomly generated rays which intersect B_2 is proportional to the size of the projected area of B_2 w.r.t. ω the redefined probability $p(\omega)$ is of the form $p(\omega) = A_\perp(\omega, B_2)/c$, where c is the normalization constant. To compute c we have to integrate $A_\perp(\omega, B_2)$ over $\mathbb{S}^2$. This is the same integral as for the computation of the measure of lines intersecting a convex body. Thus, identical to the solution of Equation 4.8 we get:

$$\begin{aligned} c &= \int_{\mathbb{S}^2} A_\perp(\omega, B_2)\, d\omega \\ &= \int_{\mathbb{S}_{B_2}} \int_{\mathbb{S}^2} |\omega \cdot \mathbf{n}(\mathbf{x})|_+ \, d\omega\, dA \\ &= \pi \int_{\mathbb{S}_{B_2}} 1\, dA \\ &= \pi\, \mathtt{Area}(B_2). \end{aligned} \tag{4.23}$$

Now we use this adapted probability density to compute the expected value of $p(\omega, B_1, B_2)$:

$$\begin{aligned} \mathtt{E}\left[p(\omega, B_1, B_2)\right] &= \int_{\mathbb{S}^2} p(\omega, B_1, B_2)\, p(\omega)\, d\omega = \int_{\mathbb{S}^2} \frac{A_\perp(\omega, B_1)}{A_\perp(\omega, B_2)}\, p(\omega)\, d\omega \\ &= \int_{\mathbb{S}^2} \frac{A_\perp(\omega, B_1)}{A_\perp(\omega, B_2)}\, \frac{A_\perp(\omega, B_2)}{\pi\, \mathtt{Area}(B_2)}\, d\omega \\ &= \frac{1}{\pi\, \mathtt{Area}(B_2)} \int_{\mathbb{S}^2} A_\perp(\omega, B_1)\, d\omega \\ &= \frac{\pi\, \mathtt{Area}(B_1)}{\pi\, \mathtt{Area}(B_2)} = \frac{\mathtt{Area}(B_1)}{\mathtt{Area}(B_2)} \end{aligned} \tag{4.24}$$

This result corresponds to the conventional intersection probability. As a consequence, in reverse the conventional intersection probability has the unrecognized property that it can be interpreted as the expected value of the direction dependent conditional intersection probability from Figure 4.3 with our adapted ray direction probability density. This property might partially explain the success of the surface area metric in predicting traversal performance despite being based on unrealistic assumptions.

Chapter 5

Temporary Subtree SAH-based Bounding Volume Hierarchy Construction

Contents

The previous chapter aimed at defining a more accurate conditional node intersection probability to improve SAM and SAH. As our probability function gave values similar to the conventional probability it resulted in essentially identical BVHs. The focus of this chapter is on construction of higher quality BVHs w.r.t. SAM and SAH with the conventional probability. State-of-the-art SAH-based BVH builders are the greedy top-down plane-sweeping algorithm from MacDonald and Booth [1990] and the extension of this algorithm with spatial splits proposed by Stich et al. [2009] (see Section 2.5.3 and Section 2.5.6). More sophisticated algorithms have been developed (see the summary in Aila et al. [2013]) that produce higher quality BVHs with respect to SAH. But the improvements do not translate well to actual measured performance and can in fact even decrease performance. As described in Section 2.5.7, Aila et al. [2013] identified geometry that overlaps bounds of subtrees to which it does not belong as a second major factor and proposed the end-point-overlap metric (EPO) to measure this effect. They also revealed the unique characteristic of greedy top-down SAH builders that they not only optimize SAH but also implicitly minimize EPO, which explains why they perform so well in practice.

To the best of our knowledge no approach has been proposed to date, which directly takes advantage of this implicit correlation of SAH and EPO for greedy top-down builders to construct better BVHs. We examine the possibility to improve EPO further by using recursive SAH evaluation on temporarily built BVHs as an accurate prediction for the surface area metric (SAM) cost of subtrees during construction. Further, we reason why the temporary BVHs themselves have to be constructed with SAH to gain any benefit and propose an algorithm that can construct BVHs with recursive SAH in $O(N \log^2 N)$. Due to the computational complexity the algorithm is mainly suitable for static scenes and global illumination algorithms.

Our main contributions are as follows:

- a BVH construction algorithm that produces BVHs with better average performance than state-of-the-art methods,
- a complexity analysis of our algorithm that reveals subquadratic runtime in the number of primitives,
- a spatial split-based algorithm, which applies temporary spatial splits to push quality of BVHs even further,
- an approach for reducing the forecasting error of the ray tracing performance predictor from Aila et al. [2013] which also enables more accurate predictions for primary rays, and
- a comparison with a related algorithm proposed by Popov et al. [2009] and a hybrid of their algorithm with ours.

This chapter is based on the paper by Wodniok and Goesele [2017]. The paper from Wodniok and Goesele [2017] itself is an extended version of an earlier conference paper by Wodniok and Goesele [2016] and added the last three contributions.

5.1 Background and Related Work

From the in-depth introduction in Section 2.5 we know that SAM provides an approximation for the expected cost of traversing a given kd-tree or BVH. The conditional node intersection probability p_n for a node n, combined with implementation dependent constants c_t for traversal step costs and c_i for primitive intersection test cost, the recursive definition of the expected traversal cost of the subtree of n is

$$c(n) = \begin{cases} c_t + p_l c(l) + p_r c(r) & \text{inner node} \\ |n| c_i & \text{leaf node} \end{cases}. \tag{5.1}$$

Here, l and r are the left and right child of n in case of an inner node, and $|n|$ is the number of primitives belonging to n. Evaluating $c(n)$ for the tree root yields the expected cost of the whole tree.

The state-of-the-art greedy top-down plane-sweeping construction from MacDonald and Booth [1990] locally applies an approximation of Equation 5.1 when splitting a node. Several candidate partitions are generated and their expected traversal cost is evaluated with $c(n)$ under the assumption that the newly generated children are leaf nodes. That is we compute:

$$c_{split} = c_t + p_l |l| c_i + p_r |r| c_i \tag{5.2}$$

The partition with smallest c_{split} is chosen and construction recursively proceeds with the children. The recursion terminates as soon as the smallest c_{split} is higher than the cost for creating a leaf node. Partitions are typically generated by sweeping axis aligned planes through every dimension and checking on which side the bounding volume centroids of primitives fall. With this approach, only planes which contain bounding volume centroids are relevant.

Though the assumptions underlying SAH generally do not apply in practice, SAH guided construction empirically produces the best performing BVHs to date. Unfortunately, SAH-based construction is also the most expensive. In Section 2.5.4 we described $O(n\log(n))$ SAH-based kd-tree and BVH construction with binning from Popov et al. [2006] and Wald et al. [2007]. Both algorithms replace the sorting step with an $O(n)$ primitive binning step. With a sufficient number of bins hierarchy quality is practically identical to full sweep construction. Fabianowski et al. [2009] changed the SAH assumption of infinitely far away ray origins to origins uniformly distributed in the scene bounds. On average ray tracing performance increases of 2.6% have been reported for kd-trees.

5.1.1 Fast High Quality Construction

Lauterbach et al. [2009] proposed three GPU-based BVH construction algorithms with different trade-offs between tree quality and construction time: The median split-based linear BVH (LBVH) algorithm is fast but has poor tree quality. The second algorithm is a parallel approach for full binned-SAH BVH construction (see Wald [2007]) with high tree quality but slower construction. The third algorithm, a hybrid of the former two, strikes a balance: Upper levels are constructed according to the highly parallel first algorithm while the remaining levels expose enough parallelism to be efficiently constructed according to the second one. Pantaleoni and Luebke [2010] and Garanzha et al. [2011] proposed much faster implementations for the median split and the hybrid algorithm called hierarchical LBVH (HLBVH) which allow real-time rebuilds for scenes with up to 2 million triangles. An important change to the hybrid algorithm is, that LBVH is used to build the lower levels of the tree first. The roots of the subtrees themselves are then used for binned top-down SAH BVH construction. Thus the expensive part of the algorithm is performed on much less input elements and tree quality is improved in the important upper levels.

Ganestam et al. [2015] proposed a hybrid algorithm called *BONSAI*. Similar to the original hybrid algorithm from Lauterbach et al. [2009] it first performs a spatial-median split partition on the input to produce sufficiently small chunks of spatially coherent primitives. Then for each chunk top-down plane-sweeping SAH-based BVHs are constructed in parallel. Finally, in the spirit of HLBVH an SAH-based top-level BVH is constructed on the chunks. Quality of the final hierarchy on average is identical to full sweep-based construction but construction time is much lower.

Bittner et al. [2015] presented the first incremental BVH construction algorithm which can produce high quality BVHs. While average hierarchy quality w.r.t. SAM is higher than for a full sweep construction, the actual average measured performance is slightly lower. Nonetheless, construction is faster than top-down sweep construction.

As dynamic scenes are not in focus of our work we only give a very brief overview on related algorithms. One approach is to simply construct a new BVH each frame as fast as possible. This was the purpose of LBVH from Lauterbach et al. [2009] and derived work (Pantaleoni and Luebke [2010], and Garanzha et al. [2011]). For state-of-the-art in refitting-based approaches we refer to the algorithm from Yin and Li [2014] and the references therein.

5.1.2 Higher Quality BVHs

The offline spatial split BVH (SBVH) algorithm (see Section 2.5.6) from Stich et al. [2009] drastically improves tree quality for scenes with a widely varying degree of tessellation. Their key idea is to either use spatial splits or object partitions depending on which of them yields a better SAH value. When searching for a node split, the best spatial split is determined in addition to the best object split. Spatial splits are only applied when considered beneficial. To date no efficient GPU implementation of this algorithm has been presented. Karras and Aila [2013] proposed an approximate but real-time construction algorithm for GPUs that takes any BVH (i.e., LBVH) as input and performs local optimizations on small node subsets (treelets) w.r.t. SAM. They also present a triangle pre-splitting heuristic with a strong focus on producing splits which are likely to be beneficial for tree quality. The resulting trees achieve about 90% of SBVH tree quality. Ganestam and Doggett [2016] present an alternative triangle pre-splitting algorithm, which can optionally directly be integrated into the clustering phase of the BONSAI algorithm. They report traversal performance improvements to be similar to Karras and Aila [2013] while producing less duplicate triangles.

Plane sweeping only generates left-right partitions with respect to a splitting plane. Popov et al. [2009] proposed to allow more general partitions in order to achieve smaller SAH cost partitions if possible. This is done by pre-generating a set of more general child bound pairs and distributing the primitives to these sets. They call this process geometric partitioning. Though achieving smaller total SAM values, trace performance did not improve equally or even decreased. Further, they also tried to improve their general partition BVH constructor by rating partitions with recursive SAM computed from temporarily built spatial-median split BVHs. This improved measurements but results were still inferior to the standard plane sweeping algorithm.

An alternative construction approach is the agglomerative clustering algorithm from Walter et al. [2008]. Initially each primitive is in its own leaf node. Then, in a bottom-up fashion the algorithm iteratively generates a new parent for the pair of nodes which produces the smallest parent bounds surface area. The nodes of the selected pair are removed from the list of candidates while the new parent node is added to this list. The authors empirically show that runtime of their implementation is somewhere between linear and quadratic. While this algorithm can produce hierarchies with higher quality than top-down SAH-based construction, Aila et al. [2013] observed that it can also produce hierarchies with drastically lower quality, or expose low traversal performance even when the SAM cost is low. Gu et al. [2013] presented a more efficient but approximative implementation of this algorithm, which also inherits its downsides.

In Section 2.5.7 we introduced the EPO metric proposed by Aila et al. [2013]. The metric was developed to better explain and predict performance of different BVHs constructed with different construction algorithms. EPO is a measure for the extra traversal cost caused by intersection with primitives which intersect bounds of subtrees they do not belong to. We refer to Aila et al. [2013] and Section 2.5.7 for computation of EPO for a BVH. The traversal cost predictor p is a convex combination of SAM and EPO:

$$p = SAM \cdot (1 - \alpha) + EPO \cdot \alpha, \tag{5.3}$$

where α is a scene dependent constant. It is assumed that

$$m \sim p \tag{5.4}$$

holds for the given actual average measured traversal cost m for a BVH (MacDonald and Booth [1989,1990]). The predictor is only designed for secondary diffuse rays and scalar traversal. Aila et al. [2013] computed α values as high as 0.98. With such a high possible weight for EPO it becomes clear that optimizing SAM alone is not enough and that even the BVH with total minimum SAM is not necessarily the best performing one. Especially important for this chapter, and the next section where we describe our algorithm, is the discovery from Aila et al. [2013] that top-down greedy SAH-based construction algorithms implicitly reduce node overlap in a way that also minimizes EPO. This gives them an innate superiority over bottom-up or hybrid algorithms.

Finally, for SAH kd-tree construction Havran [2000] examined a possibly better prediction model than Equation 5.2 for subtree SAM costs. He assumed that geometry is distributed uniformly in space, that a spatial-median split strategy is used, and that the subtree root has cube shaped bounds. He proved that the predicted cost is in $O(N)$, which renders the classic linear cost model (Equation 5.2) sufficient under these assumptions. Havran further elaborates on minimum total SAM cost kd-tree construction. This requires to recursively evaluate SAM for each split candidate and the recursive evaluation itself has to recursively apply recursive SAM evaluation. This results in a combinatorial explosion which Havran states to be *NP*-hard and also translates to BVH construction. As a side note, a consequence of the work of Aila et al. [2013] would be that the minimum total SAM cost kd-tree would probably also be the best performing one as EPO is always zero for kd-trees.

5.2 Algorithm

The goal of the approach presented in this section is to give a better predictor for split candidates than the classic linear model given in Equation 5.2. To achieve this, the model has to improve both SAM as well as EPO. So far greedy top-down builders are the only known builders which, at least implicitly, minimize EPO. We want to take advantage of this characteristic during construction. As EPO minimization only occurs implicitly during construction we cannot estimate it, e.g., just from the number of primitives and bounds of the node to split. Thus we propose to actually construct temporary, greedily built BVHs for the left and right side of each split candidate and use their recursive SAM values. The rating function for a split then becomes

$$c_{split} = c_t + p_l c(t_l) + p_r c(t_r), \tag{5.5}$$

where t_l and t_r are the roots of the temporarily constructed BVHs and c is the recursive SAM function introduced in Equation 5.1. Note that EPO does not directly appear in this rating function. We rely on the assumed correlation of SAM and EPO of the greedily top-down built temporary BVHs to find the split with minimal EPO by finding the split with minimal c_{split}. This also should implicitly guide global construction into directions of lower EPO. Pseudo-code for determining the best split is given in Algorithm 3. A depiction of candidate cost computation with temporary BVH construction is given in Figure 5.1. Note that `BuildTemporaryBVH` and `ComputeSAM` can be merged into a single function as SAM cost can be computed incrementally during construction. There is no need to store the temporary hierarchy at any point. It is also not beneficial to store the hierarchy of the best candidate for reuse as it only has the quality of the standard algorithm and different hierarchies on smaller subsets have to be constructed for the newly created child nodes.

Algorithm 3: Pseudo-code for determining the best node split with recursive SAH.

```
input  : node                                         // node to split
input  : $c_t$                                        // cost of a traversal step
input  : $c_i$                                        // cost of intersecting a primitive
output: bestP                                         // Best primitive partition
output: bestC                                         // Best partition costs

(bestP, bestC) ← ($\emptyset$, $\infty$)
partitions ← GeneratePartitions(node)
foreach partition ∈ partitions do
    $t_l$ ← BuildTemporaryBVH(partition.left, $c_t$, $c_i$)
    $t_r$ ← BuildTemporaryBVH(partition.right, $c_t$, $c_i$)
    $c_l$ ← ComputeSAM($t_l$, $c_t$, $c_i$)
    $c_r$ ← ComputeSAM($t_r$, $c_t$, $c_i$)
    ($p_l$, $p_r$) ← ComputeIntersectionProbabilities(partition)
    $c_{split} \leftarrow c_t + p_l c_l + p_r c_r$
    if $c_{split}$ < bestC then
        (bestP, bestC) ← (partition, $c_{split}$)
    end
end
```

Seen from a different angle, our approach can be interpreted as a middle ground between the *NP*-hard algorithm proposed by Havran [2000] and the recursive SAH evaluation on temporary spatial-median split trees used by Popov et al. [2009] in terms of computational complexity and BVH quality. The difference is, that we give a more representative cost estimate for the split candidates than a spatial-median split as it much closer reflects the way the main BVH itself is constructed. Spatial-median split construction does not incorporate SAM in any way. Thus, SAM values retrieved from temporary BVHs constructed in this way are unreliable for construction that aims at reducing SAM.

We will now proceed with the algorithmic aspects of our approach, which we call recursive SAH-based BVH construction (RBVH). The `GeneratePartitions` function in Algorithm 3 determines if the main BVH is constructed with plane-sweeping or binning, though more general partitions such as in Popov et al. [2009] are also possible. The `BuildTemporaryBVH` function for temporary BVH construction can also use arbitrary construction schemes but we only consider the EPO-reducing greedy top-down plane-sweeping or binning construction. Alternatives would be the fast to construct bottom-up LBVH and HLBVH approaches or the agglomerative algorithm from Walter et al. [2008]. Data collected by Aila et al. [2013] shows that these algorithms can increase EPO drastically, which renders them a poor choice for `BuildTemporaryBVH`. Depending on whether we choose sweeping or binning for `GeneratePartitions` and `BuildTemporaryBVH` we have four different RBVH algorithms with their own asymptotic computational complexities. We will proceed with deriving complexities for all four cases.

5.2.1 Computational Complexities

We first recap the computational complexity of the standard SAH-based construction approach. The common plane-sweeping algorithm implementation which sorts in every dimension needs $O(n \log n)$ steps to find a split and $O(n \log^2 n)$ steps in total. Adapting

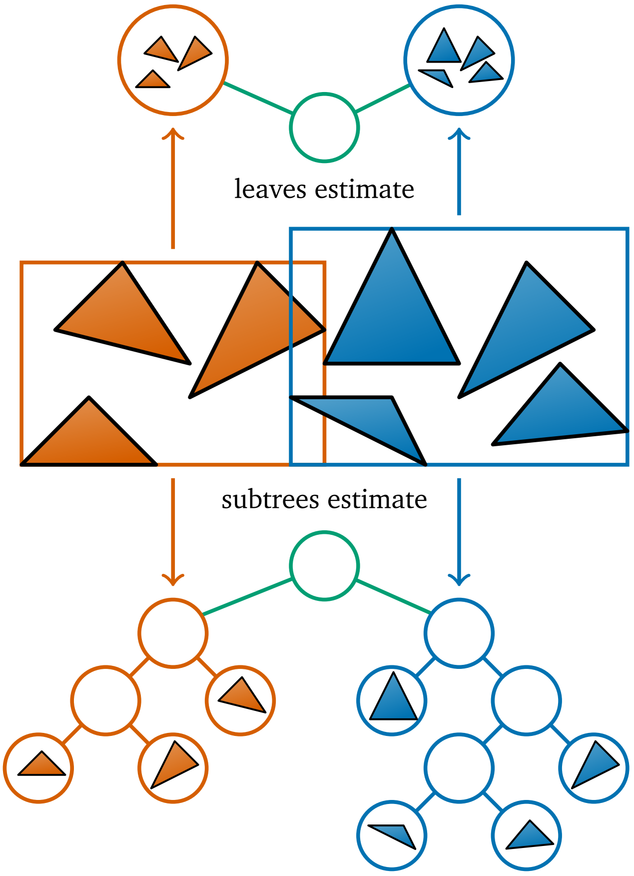


Figure 5.1: Depiction of candidate cost evaluation in Algorithm 3 for a candidate partition (red and blue boxes) generated by `GeneratePartitions` for a set of input primitives with e.g. a plane-sweeping or binning approach. Classic SAH candidate cost assumes that both sides of the partition will stay leaves. With the `BuildTemporaryBVH` function we construct separate temporary BVHs for each side of the partition and compute their SAM cost to obtain a candidate cost estimate. Depending on the construction strategy used in `BuildTemporaryBVH` quality of the estimate can be better or worse than for the classic candidate cost.

concepts of Wald and Havran [2006] for kd-tree construction to BVHs allows to implement an algorithm with lower complexity. For this, three copies of all primitives have to be sorted in each dimension in a pre-computation step. This allows to find the best split in $O(n)$ steps. The arrays which do not correspond to the dimension of the best split can then be updated in $O(n)$. As a result the whole algorithm has a complexity of $O(n \log n)$. Using binning construction also has $O(n \log n)$ complexity, but with a smaller constant. To simplify derivation of the complexities we make the common assumption that a split produces two new nodes with roughly the same number of primitives and that the number of scene primitives N is a power of two.

Sweep-Sweep / Sweep-Binning Construction We start with the derivation of the complexity of sweep-sweep construction (sweeping construction for the main and temporary BVHs). For a node with N primitives a sweep-based construction generates $N - 1$ candidate partitions. This results in $2(N-1)$ temporary BVHs that need to be constructed. With our proposed approach akin to Wald and Havran [2006] each temporary BVH can be constructed in $O(n\log n)$, where n is the number of primitives of each side of a candidate partition. Using the hyper factorial $H(x) = \prod_{i=1}^{x} i^i$ and $O(n\log n) = O(\log n^n)$ we can define the recurrence relation $T(N)$ of the algorithm:

$$\begin{aligned}
T(N) &= 2\left(\sum_{i=1}^{N-1} i\log i\right) + 2T\left(\frac{N}{2}\right)\\
&= 2\log\left(H\left(N\right)\right) + 2T\left(\frac{N}{2}\right)\\
&= 2\log\left(H\left(N\right)\right) + 2\left(\log\left(H\left(\frac{N}{2}\right)\right) + 2T\left(\frac{N}{4}\right)\right)\\
&= 2\sum_{i=0}^{\log N} 2^i \log\left(H\left(\frac{N}{2^i}\right)\right)
\end{aligned}$$

Using the simple-to-derive upper bound $\log(H(x)) < x^2\log x$ we get:

$$\begin{aligned}
T(N) &< 2\sum_{i=0}^{\log N} 2^i \left(\frac{N}{2^i}\right)^2 \log\left(\frac{N}{2^i}\right)\\
&= 2\sum_{i=0}^{\log N} N^2 2^{-i}\left(\log(N) - i\right)\\
&= 2N^2\left(\log(N)\left(\sum_{i=0}^{\log N} 2^{-i}\right) - \sum_{i=0}^{\log N} 2^{-i} i\right)\\
&\to O\left(N^2\log(N)\right)
\end{aligned} \tag{5.6}$$

As we only found an upper bound for $\log(H(x))$ the asymptotic complexity $O(N^2\log N)$ is not tight. Using $\log(H(x)) > x^2/2$ we get the lower bound $\Omega(N^2)$ for the asymptotic complexity. An algorithm based on the common naïve approach which sorts in every step has a higher complexity of $O\left(N^2\log^2 N\right)$. A derivation of this result is provided in Appendix A.1.

Asymptotic complexity of binning-based temporary BVH construction is the same as for sweep-based construction akin to Wald and Havran [2006]. Consequently the sweep-binning approach has the same complexity as the sweep-sweep approach.

Binning-Binning / Binning-Sweep Construction Let $B = 2^b, b \in \mathbb{N}$ denote the number of bins for the main BVH. The number of bins for the temporary BVHs is not relevant, as it does not appear in the complexity of binned construction. For simplicity we assume that geometry is roughly distributed uniformly in space such that a node with N primitives generates B bins with N/B primitives in each bin after binning. This results in $B-1$ candidate partitions and thus $2(B-1)$ temporary BVHs that need to be constructed. Each temporary BVH itself is constructed in $O(n\log n)$, where n is the number of primitives in the union of all bins on each side of a candidate partition. This results in the following recurrence relation:

$$
\begin{aligned}
T(N) &= 2\left(\sum_{i=1}^{B} i\frac{N}{B}\log\left(i\frac{N}{B}\right)\right) + 2T\left(\frac{N}{2}\right) \\
&= 2\sum_{i=0}^{\log N} 2^i \left(\sum_{j=1}^{B} j\frac{N}{2^i B}\log\left(j\frac{N}{2^i B}\right)\right) \\
&\in O(N\log^2 N + BN\log(B)\log(N)) = O(N\log^2 N)
\end{aligned}
\tag{5.7}
$$

We used the upper bound of $\log H(x)$ for the derivation. This has no effect on the asymptotic complexity. The full derivation is described in Appendix A.2. As we can see, the binning-binning construction algorithm has subquadratic complexity and thus more relevance in practice. Though the number of bins asymptotically has no effect on runtime it increased runtime linearly in our experiments for problem sizes we used in our tests. The reason for this is that the $BN\log(B)\log(N)$ of T is dominating up to a certain problem size. We proceed with computing bounds for the number of input primitives N for which the number of bins causes the second most dominating term to dominate the $N\log^2 N$ term. Using the lower bound for $\log H(B)$ the second term becomes $BN\log(N)$. Equating the two dominating terms of T for the upper and lower bound of $\log H(B)$ we get:

$$N\log^2 N = BN\log(N) \tag{5.8}$$

$$N\log^2 N = BN\log(B)\log(N) \tag{5.9}$$

Solving for N we get the bounds $2^B < N < B^B$. As a result even for the small number of $B = 32$ bins the $BN\log(B)\log(N)$ term dominates till $2^{32} < N < 2^{160}$ primitives. Thus, B keeps impacting construction time even for scenes which have a several orders of magnitude higher number of primitives than current scenes in production rendering.

Again, due to the same asymptotic complexity of binning construction and our approach for sweep construction of temporary BVHs, binning-sweep construction has the same complexity as binning-binning construction.

5.2.2 Spatial Splits

To also take advantage of spatial splits akin to SBVH from Stich et al. [2009] we cannot simply treat them as an additional technique to RBVH. SBVH uses the linear cost model from Equation 5.2 which is an upper bound on the cost model of RBVH (Equation 5.5). This does not allow us to compare split candidates from those techniques in a meaningful way. Instead, we simply have to adapt the `GeneratePartitions` function to also generate spatial partitions in order to remove this problem. This requires to temporarily split primitives for each candidate partition, but potentially allows to find even better split candidates. This variant is included into the evaluation, where we call it recursive spatial split bounding volume hierarchy (RSBVH).

Going one step further we can include spatial splits into `BuildTemporaryBVH` to construct temporary SBVHs. This again allows to find better split candidates. To unfold the full potential of this approach temporary SBVHs have to be constructed for spatial partitions as well as object partitions generated by `GeneratePartitions`. Constructing temporary SBVHs but not generating spatial partitions in `GeneratePartitions` is detrimental for BVH quality as the main BVH might be misguided into directions which are only beneficial if spatial splits are enabled. RSBVH construction with temporary SBVHs is also included into the evaluation.

5.3 Improving Accuracy of the SAM-EPO Predictor

In our previous work [Wodniok and Goesele 2016] we observed that while the achieved correlations of predicted cost p from Equation 5.3 and average measured traversal cost m are higher than 0.99 in most scenes, the average predicted performance improvements were up to 50% higher than the actual average measured performance improvement, but could not provide an explanation for this. Before presenting our solution to this problem we first recap how the scene dependent α value of Equation 5.3 is computed.

Given a set of BVHs $\mathcal{B}$ constructed for a scene and triples (SAM_i, EPO_i, m_i) of SAM, EPO, and measured traversal cost for each BVH $i \in \mathcal{B}$ Aila et al. [2013] we have to find the α value which maximizes the correlation of predicted and measured costs. This can be formulated as

$$\alpha_{best} = \underset{\alpha \in [0,1]}{\arg\max} \;\; r\left(\{(p_i(\alpha), m_i) \,|\, i \in \mathcal{B}\}\right), \tag{5.10}$$

where r is the centered sample Pearson correlation coefficient and $p_i(\alpha) = SAM_i \cdot (1 - \alpha) + EPO_i \cdot \alpha$ corresponds to Equation 5.3 for a pair (SAM_i, EPO_i) of a BVH i for some tentative α. We computed α by simply sampling the $[0,1]$ range and selecting the α which gives the highest correlation. For a chosen α we can then compute the *mean absolute percentage error* (MAPE) of the expected and measured pairwise *speedups* $p_{ij} = p_i/p_j$ and $m_{ij} = m_i/m_j$ for pairs of BVHs i, j:

$$MAPE = \frac{1}{|\mathcal{B}|^2 - |\mathcal{B}|} \sum_{\substack{i,j \in \mathcal{B} \\ i \neq j}} \left| \frac{p_{ij} - m_{ij}}{m_{ij}} \right|. \tag{5.11}$$

Computing MAPE of the predicted and measured speedups allows us to quantify the prediction error of a given α without having to know the constant of proportionality of the assumed proportionality of p and m. Computing α with the centered Pearson correlation resulted in average MAPE errors as high as 65%.

The reason for this high error is that solving Equation 5.10 is equivalent to finding the α, which gives the smallest least squares error of a simple linear regression with an intercept term. The intercept term breaks, however, the proportionality assumption between measured and estimated cost (Equation 5.4) as it allows to not contain the origin. To force regression through the origin we propose to use the *uncentered sample Pearson correlation* (equivalent to the *cosine similarity* function):

$$r_{uncentered} = \frac{\sum_i p_i m_i}{\sqrt{\sum_i (p_i)^2} \sqrt{\sum_i (m_i)^2}} \tag{5.12}$$

Solving Equation 5.10 with the uncentered correlation is equivalent to finding the α, which gives the smallest least squares error of a simple linear regression without an intercept term. Table 5.1 shows the computed α values along with their corresponding correlation and MAPE for the centered and uncentered sample Pearson correlation coefficients for all test scenes. Though only intended for secondary diffuse rays we also computed a separate α for primary rays.

	Primary rays						Diffuse rays					
	Centered			Uncentered			Centered			Uncentered		
Scene	α	corr.	MAPE	α	corr.	MAPE	α	corr.	MAPE	α	corr.	MAPE
Babylon	0.98	0.993	116.5%	0.53	0.997	**9.4%**	0.89	0.997	60.2%	0.38	0.998	**6.2%**
Bubs	0.00	0.935	**5.7%**	0.00	0.998	**5.7%**	0.25	0.989	8.0%	0.00	0.999	**4.6%**
Conference	0.49	0.971	9.2%	0.45	0.998	**8.9%**	0.64	0.999	9.9%	0.27	0.999	**2.5%**
Epic	1.00	0.936	62.2%	0.66	0.996	**10.9%**	1.00	0.955	65.9%	0.60	0.998	**8.4%**
Fairy	0.48	0.855	**7.4%**	0.80	0.997	8.0%	0.75	0.928	3.6%	0.68	0.999	**3.4%**
Hairball	1.00	0.886	74.9%	0.87	0.993	**13.2%**	0.96	0.993	32.2%	0.72	0.999	**2.8%**
Powerplant	0.75	0.997	23.0%	0.47	0.999	**4.1%**	0.61	0.999	17.3%	0.28	0.999	**3.4%**
Rungholt	0.00	0.602	**2.6%**	0.00	0.999	**2.6%**	0.00	0.664	**2.6%**	0.00	0.999	**2.6%**
San Miguel	0.61	0.990	16.7%	0.43	0.999	**9.0%**	0.66	0.993	28.2%	0.28	0.999	**7.8%**
Sibenik	0.47	0.995	5.9%	0.62	0.999	**3.5%**	0.74	0.997	12.7%	0.38	0.999	**3.1%**
Soda	0.61	0.975	14.0%	0.45	0.998	**9.7%**	0.61	0.998	11.8%	0.35	0.999	**4.6%**
Sponza	0.74	0.920	**12.3%**	0.75	0.995	12.4%	0.79	0.979	10.0%	0.62	0.999	**5.9%**
Average	0.59	0.921	29.2%	0.50	0.998	**8.1%**	0.66	0.958	21.9%	0.38	0.999	**4.6%**

Table 5.1: Listing of determined α values for primary and diffuse rays for all scenes used for benchmarking our algorithms obtained with the centered and uncentered sample Pearson correlation. Each α is accompanied by its corresponding correlation coefficient and mean absolute percentage error (MAPE). Lowest MAPE is highlighted for each combination of ray type and scene.

Table 5.1 shows that our approach reduces MAPE significantly. The obtained α values can differ drastically from the original approach with an extreme case of 0.89 and 0.38 for *Babylon* with diffuse rays, where MAPE is reduced from 60.2% to 6.2%. Overall we can observe that our approach gives less of an emphasis on EPO for diffuse rays in all scenes. On average we reduce MAPE from 22% to 4.6% for diffuse rays.

Our measurements show that the SAM-EPO predictor indeed gives higher errors for primary rays for which it is not intended. MAPE is highest for *Babylon* with 116.5%. But our approach was able to reduce the error to 9.4%. On average we reduced MAPE from 29.2% to 8.1%. Unlike for diffuse rays, we cannot observe the overall trend that α values with our approach are lower than with the centered correlation. Note that average MAPE for primary rays with the uncentered correlation is also lower than average MAPE with the centered correlation and diffuse rays.

5.4 Evaluation Setup

Our test platform is equipped with two Intel Xeon E5-2650 v2 octa-core CPUs. Construction timings are included in our results. Our implementation of the algorithms only parallelized the for-each loop in Algorithm 3. This is not optimal as it introduces global synchronization between every node split. With additional effort it is possible to parallelize the whole construction process, though.

5.4.1 Scenes and Algorithms

To evaluate our proposed construction algorithm we measured the impact on SAM, EPO and traversal performance. We used a number of freely available test scenes (see Fig-

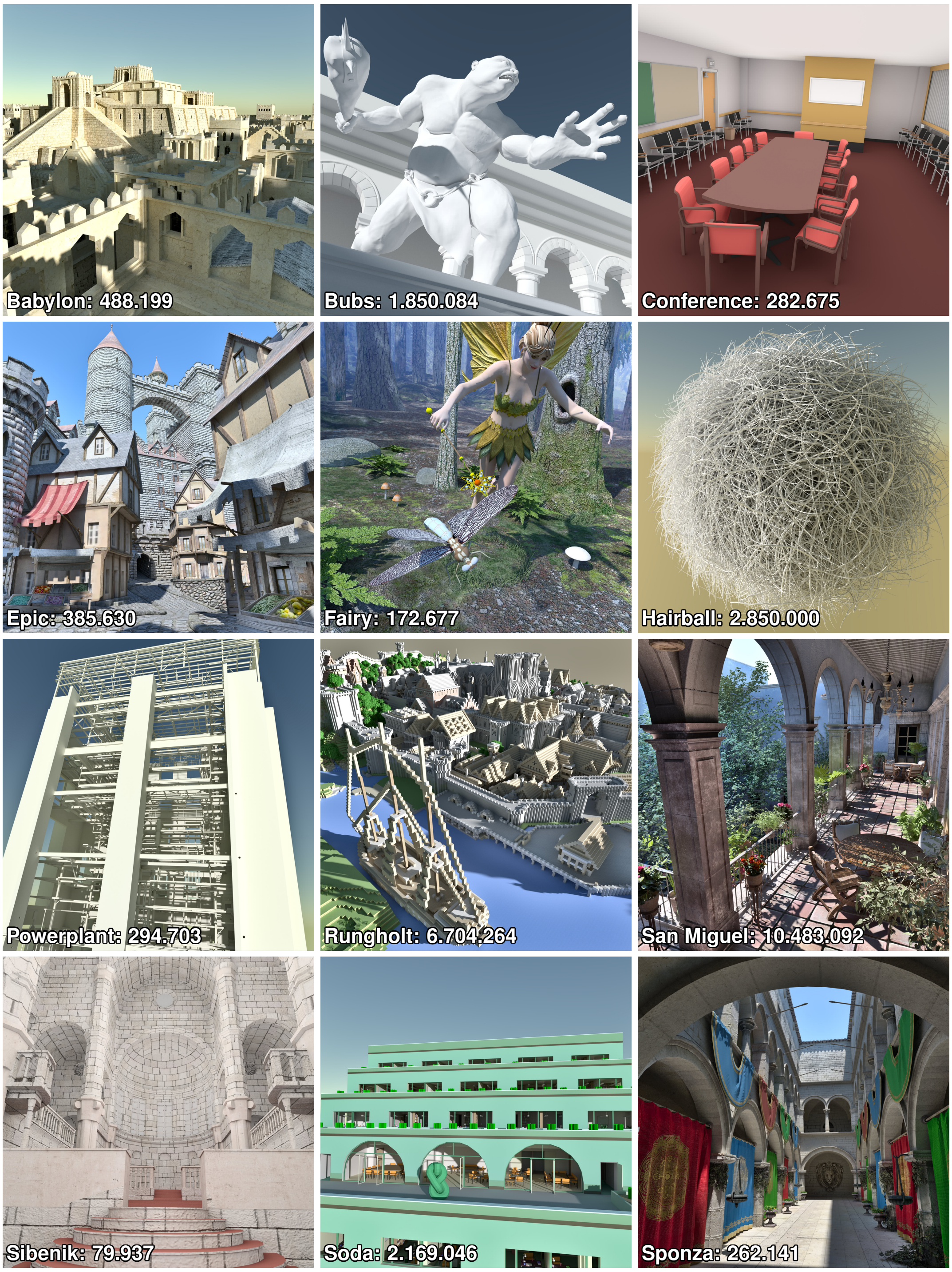


Figure 5.2: Listing of all twelve scenes used for benchmarking our algorithms along with their number of primitives.

Abbr.	Algorithm	o	s	t
BBVH	Baseline Plane-Sweep [MacDonald and Booth 1990]	-	-	-
SBVH	Baseline SBVH [Stich et al. 2009]	256	128	-
RBVH	RSAH	256	-	32
RMBVH	BBVH + temp. median splits	256	-	-
RSBVH	RSAH + SBVH	256	128	32
RSSBVH	RSBVH + temp. SBVH	256	128	32
GBVH	Geometric splits [Popov et al. 2009]	256	-	-
RMGBVH	GBVH + temp. median splits	256	-	-
RGBVH	GBVH + RSAH	256	-	32

Table 5.2: List of algorithms and their configurations we used for evaluation. o and s denote the numbers of object and space partitioning bins used for construction of the main BVH. t is the number of object partitioning bins used for the construction of temporary BVHs. In case of RSSBVH the number of temporary spatial bins is equal to t.

ure 5.2) partly from McGuire [2011] and the Mitsuba renderer [Jakob 2010]. We only evaluated the $O(N \log^2 N)$ binning-binning algorithm as the superquadratic complexity of the sweep-sweep and and sweep-binning algorithm proved to be impractical. The RSBVH algorithm and the extension with temporary SBVHs (RSSBVH) from Section 5.2.2 is also included into the evaluation. As the baseline construction algorithm we chose the standard plane-sweeping approach. We also evaluated our RBVH algorithm with recursive SAM evaluation on temporarily built spatial-median split BVHs (RMBVH).

Further, we included the geometric partitions with (RMGBVH) and without (GBVH) temporarily built spatial-median split BVHs from Popov et al. [2009]. In this connection we also evaluated the inclusion of geometric partitions into the `GeneratePartitions` function of our RBVH algorithm (RGBVH).

As the baseline for construction with spatial splits we chose the SBVH algorithm from Stich et al. [2009]. SBVH allows to specify a parameter which guides spatial split attempts. We follow the authors recommendation and use a value of 10^{-5} for all scenes. Exceptions were *Hairball* where we used 10^{-4} to avoid excessive primitive duplication and *San Miguel* where we had to use 10^{-6} for any spatial splits to occur. The same parameter values are also applied to the temporary SBVH construction of the RSSBVH algorithm.

In total we have up to nine different BVHs per scene. We omit results for RGBVH and RMGBVH for the five largest scenes as the expected total computation time is several months (a year for RGBVH on San Miguel) and we expect the results to be similar to the results for the smaller scenes.

For the main BVH we have 256 object split bins and 128 spatial bins. The number of object and spatial bins for temporary BVH construction is 32. SAH build constants were set to $(c_t, c_i) = (1.2, 1.0)$. They correspond to the constants of the GPU ray tracing kernels from Aila and Laine [2009], which we used for collecting traversal statistics. All BVH algorithms and configurations along with abbreviations we used for them are listed in Table 5.2.

		Avg. reduction (%)				
Algo.	Bins	Time	SAM	EPO	m_p	m_d
RBVH	64	-64.5	+0.7	+0.3	+0.2	+0.5
	32	-78.1	+1.2	+0.3	-0.2	+0.5
RSBVH	64	-57.3	-0.2	0.0	+1.7	+0.4
	32	-75.5	+0.6	+4.3	+2.2	+1.0
RSSBVH	64	-61.3	0.0	0.0	-0.5	-0.2
	32	-79.0	+0.1	+5.8	+0.7	-0.2

Table 5.3: Average relative difference of construction time, SAM, EPO, as well as measured traversal cost of primary (m_p) and diffuse (m_d) rays for the RSAH-based algorithms in percent when the number of object- and space partitioning bins for the main BVH are reduced to 64 or 32.

5.4.2 Performance Measurements

We measure performance of front-to-back traversal for primary rays and secondary diffuse rays to compare quality of the different BVHs. To get implementation and platform independent measurements we measured the average number of traversal steps $\overline{n}_s$ and the average number of intersected triangles $\overline{n}_t$ over a varying number of views for each scene and BVH. Combined with the SAH constants we define the average measured traversal cost

$$m = \overline{n}_s c_t + \overline{n}_t c_i. \tag{5.13}$$

We also give results for predicted traversal cost with EPO according to Equation 5.3. For this we computed the scene dependent α values with the novel approach described in Section 5.3 using the uncentered Pearson correlation (see Table 5.1 for the specific α values).

5.5 Results

All measurements are collected in Table 5.5 and Table 5.6. To give a more condensed view of the results Table 5.4 shows relative improvements averaged over all scenes with BBVH and SBVH as baseline. Measurements for each scene are sorted from best to worst with respect to measured traversal cost m of diffuse rays.

Our spatial split-based algorithms improved SAM, EPO and trace performance in all scenes compared to BBVH and SBVH. Performance of primary and diffuse rays improves roughly by the same amount on average with a slightly higher improvement for primary rays. The average improvement of SBVH is higher than for our RBVH algorithm without spatial splits. SBVH improves SAM only slightly compared to RBVH, but outperforms RBVH in EPO improvements. The RMBVH algorithm which, unlike RBVH, uses temporary spatial median splits performs worse than BBVH and overall performs second worst of all evaluated algorithms. The SAM costs of temporarily constructed spatial median split BVHs were less representative for subtree costs than the standard cost estimate from Equation 5.2 and misguided the construction process.

		Avg. (Min/Max) reduction (%)					
				Primary rays		Diffuse rays	
	Algorithm	SAM	EPO	p	m	p	m
Baseline BBVH	RSSBVH	**-21.6** (-4.0/-35.8)	**-71.9** (-20.8/-90.6)	**-32.6** (-4.0/-47.0)	**-24.0** (-2.2/-37.1)	**-28.3** (-4.0/-39.7)	**-22.5** (-0.9/-32.2)
	RSBVH	-19.4 (-3.9/-34.7)	-68.1 (-19.6/-85.8)	-30.1 (-3.9/-41.3)	-23.9 (+0.6/-37.0)	-26.0 (-3.9/-35.9)	-22.4 (+0.3/-32.6)
	SBVH	-12.5 (+3.8/-26.8)	-58.2 (0.0/-82.8)	-22.3 (0.0/-38.7)	-17.7 (+1.7/-33.7)	-18.6 (0.0/-33.3)	-18.5 (0.0/-32.0)
	RBVH	-11.7 (-2.7/-33.3)	-22.2 (-1.4/-65.3)	-13.3 (-2.7/-33.3)	-10.4 (+0.9/-28.3)	-12.7 (-2.7/-33.3)	-8.9 (+0.4/-23.7)
	RGBVH	-10.6 (-4.0/-17.8)	-11.4 (+8.3/-42.5)	-11.1 (-2.7/-24.9)	-7.7 (+2.3/-20.2)	-11.0 (-3.2/-22.0)	-7.5 (-1.9/-22.8)
	GBVH	+2.6 (+21.1/-21.8)	+31.7 (+96.0/-63.6)	+9.8 (+36.4/-21.8)	+22.2 (+46.2/-6.2)	+7.1 (+30.4/-21.8)	+11.9 (+31.6/-13.8)
	RMBVH	+22.1 (+69.4/-12.9)	+60.6 (+171.0/-7.3)	+29.6 (+90.7/-12.9)	+33.4 (+84.5/-7.6)	+26.4 (+80.1/-12.9)	+26.6 (+74.8/-4.9)
	RMGBVH	+26.7 (+70.0/-0.5)	+113.7 (+228.6/+29.2)	+45.0 (+103.3/+10.4)	+60.8 (+124.3/-1.0)	+37.2 (+86.8/+7.3)	+44.9 (+93.7/+2.6)
Baseline SBVH	RSSBVH	**-10.3** (-4.0/-19.4)	**-35.1** (-1.7/-70.6)	**-13.5** (-4.0/-27.5)	**-7.4** (-1.7/-17.6)	**-12.1** (-4.0/-20.1)	**-4.7** (+0.2/-15.1)
	RSBVH	-7.8 (-2.0/-15.6)	-18.6 (+52.6/-70.6)	-9.8 (-0.7/-28.4)	-7.1 (+2.1/-18.3)	-8.9 (-2.4/-18.6)	-4.6 (+0.3/-15.4)
	RBVH	+1.7 (+23.8/-12.1)	+143.7 (+394.0/-20.1)	+13.4 (+46.9/-8.9)	+10.7 (+46.0/-9.3)	+8.5 (+35.9/-8.9)	+12.9 (+36.0/-3.7)
	RGBVH	+4.9 (+21.2/-7.5)	+198.5 (+451.3/+25.0)	+22.4 (+50.2/+0.2)	+18.5 (+51.3/-8.3)	+15.0 (+37.5/-3.0)	+18.4 (+39.8/-1.3)
	GBVH	+18.3 (+46.0/-0.2)	+331.6 (+894.5/+22.3)	+44.2 (+98.1/-0.2)	+51.1 (+93.2/+0.4)	+33.4 (+75.3/-0.2)	+39.4 (+72.2/-0.2)
	RMBVH	+42.4 (+124.4/-0.4)	+459.9 (+1081.3/+2.0)	+72.7 (+194.9/+3.2)	+66.9 (+178.5/-0.7)	+59.3 (+157.3/+3.2)	+59.1 (+149.0/+2.7)
	RMGBVH	+51.4 (+125.2/-1.0)	+700.9 (+1710.1/+60.3)	+104.7 (+219.3/+13.7)	+111.3 (+227.8/-2.6)	+80.8 (+171.9/+7.6)	+88.7 (+176.0/+5.0)

Table 5.4: Average, minimum, and maximum relative reduction of SAM, EPO, as well as predicted (p) and measured (m) traversal cost of primary and diffuse rays over all scenes. Algorithms are either compared against BBVH or SBVH as baseline. For each baseline, algorithms are sorted from highest to lowest reduction in traversal cost of diffuse rays. The highest reduction of each attribute is highlighted per baseline.

In the *Rungholt* scene no algorithm was able to significantly outperform the other algorithms. The baseline BBVH already gives the third best results and its output is identical to SBVH, which was not able to find any beneficial spatial splits. The scene mainly consists of triangles of the same shape and size which are also parallel to the main coordinate system planes. BBVH seems to be sufficient for scenes with such characteristics. This is also reconfirmed by identical results we obtained for the *Lost Empire* scene available from McGuire [2011] which has the same characteristics but is not included in our results in the interest of brevity.

5.5.1 Geometric Object Partitions

On average our algorithms perform better than algorithms based on the geometric splits from Popov et al. [2009]. In fact, we can reconfirm results from Popov et al. that GBVH and RMGBVH perform worse than standard plane-sweeping. Given that spatial median splits already decreased performance of RMBVH over BBVH, it is no surprise that RMGBVH performs worse than GBVH. RMGBVH proved to be the worst performing of all evaluated algorithms. Only RGBVH, our RBVH algorithm extended with geometric splits, managed to produce better results than BBVH on average from all algorithms with geometric splits. Still, RGBVH proved to be less effective than RBVH.

5.5.2 Construction Time

RBVH- and RSBVH- based construction time is one to two orders of magnitude higher than for the baseline. RSSBVH additionally takes 2 to 8 times longer than RSBVH. One possibility to reduce construction time is to reduce the number of object and spatial bins for the main and temporary BVHs of our algorithms, as this reduces the number of temporary BVHs to construct. Therefore we evaluated configurations where the number of object and spatial bins is set to 64 or 32. Average results for relative construction time, quality, and measured cost are collected in Table 5.3. On average construction time is reduced by a factor of 2.5 and 4.5 with 64 and 32 bins respectively. Surprisingly, the effects on quality and measured performance are small on average. Performance of RBVH decreases by just half a percent with both reduced bin counts for diffuse rays. RSBVH performance decreases by up to one and two percent for diffuse and primary rays, while RSSBVH performance actually improved slightly for diffuse rays despite having the largest average increase in EPO.

All geometric split-based algorithms are the most expensive ones with RGBVH already taking 12 hours to construct for our smallest scene *Sibenik* and more than two days for *Babylon*.

5.5.3 Construction Complexity

To evaluate the validity of our derived construction complexity of $O(N \log^2 N)$ we compared the measured and predicted construction time of RBVH, RSBVH, and RSSBVH w.r.t. the number primitives. Results are shown in Figure 5.3. The measurements seem to validate our derivation for RBVH though it built on the unrealistic assumption that the number of primitives is halved on every split. The prediction accuracy for RSBVH is as good as for RBVH. While RSSBVH exhibits a less predictable construction time behavior our predictions do not systematically underestimate its cost. Considering that our complexity derivation did not consider spatial splits and the extra cost from chopped binning (see Section 2.5.6) the results for RSBVH and RSSBVH are remarkable.

5.6 Discussion

The proposed RBVH, RSBVH and RSSBVH builders managed to reduce SAM as well as EPO by a significant amount. All three algorithms do not handle EPO reduction directly but rely on the implicit correlation of SAM and EPO minimization of greedy top-down builders. Thus, our results also reconfirm the results from Aila et al. [2013]. Though SBVH already gives high average reduction in SAM and more so in EPO, RSBVH managed to push both reductions even further. Although RBVH manages to reduce SAM and EPO well, the spatial splits of SBVH prove to be a superior splitting strategy. Considering that

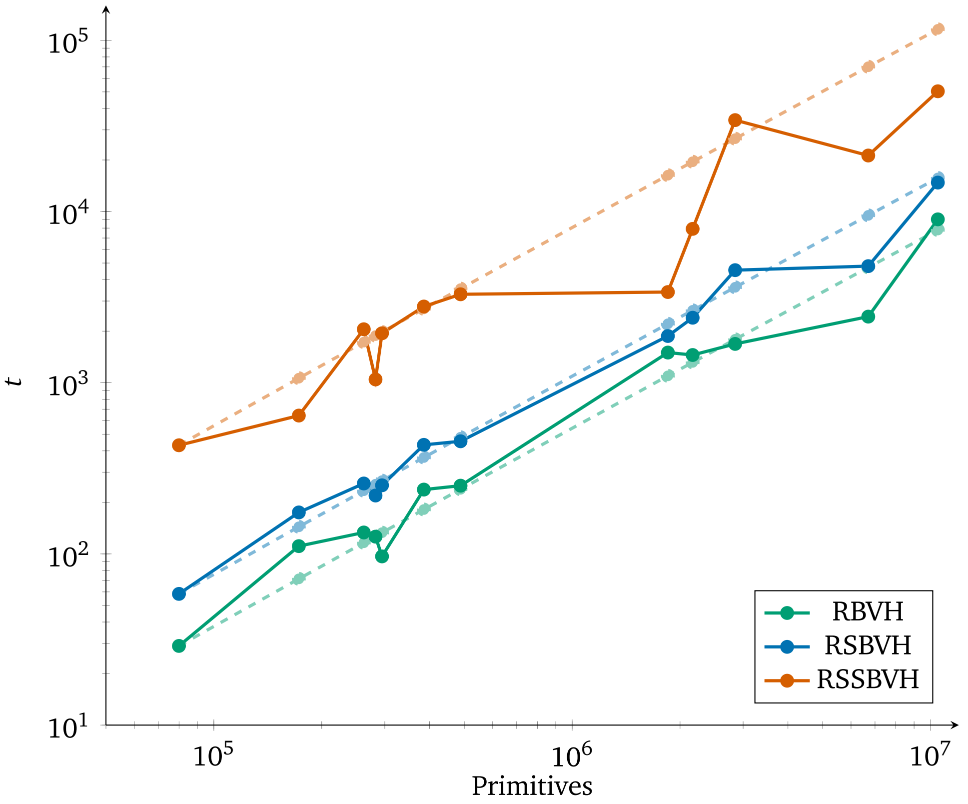


Figure 5.3: Measured construction time in seconds of the RBVH, RSBVH, and RSSBVH algorithms with respect to the number of primitives of the different scenes. The dashed line indicates the predicted construction time using our derived $O(N \log^2 N)$ complexity. For each construction algorithm the computational constant is adjusted such that the prediction agrees for the smallest scene.

RBVH only performs object splits, an EPO reduction of up to 65% and 22% on average is quite impressive. If primitive duplication is not desired RBVH might be an alternative to SBVH.

Compared with SBVH, RSBVH and RSSBVH on average produce twice the number of duplicates. An exception to this is *Hairball*, where we measured five times as many duplicates. Our subtree cost estimate predicts more accurately whether a spatial split will pay off and seems to find more opportunities where a spatial split is more beneficial than an object split. As a result more spatial splits are performed. The slightly more accurate cost estimate of RSSBVH results in a slightly higher number of duplicates compared to RSBVH.

The downside of our proposed algorithms is the large increase in construction time, which renders them unsuitable for realtime applications. Our largest test scene, *San Miguel*, took almost 4 hours to construct with RSBVH and almost 14 hours with RSSBVH. However, global illumination computations are one application that requires many intersection tests and can thus offset the initial costs of acceleration structure construction. We also have to remark that our implementation was not heavily optimized and not entirely parallelized. With enough implementational effort it should be possible to significantly increase construction performance. Results from Section 5.5.2 showed that reducing the number of bins might be an option as construction can be sped up by a factor of 4.5 without severe effects on measured performance. Alternatively, the construction of temporary BVHs could be completely offloaded to a GPU. Only primitive bounds are needed for construction. When primitive bounds are reordered according to the bins they fall into, the GPU can incrementally construct all temporary BVHs without further reloading or reordering. This also should give a significant performance boost.

5.6.1 Insufficiency of the SAM-EPO Metric

Compared with RSBVH, RSSBVH manages to reduce SAM 1.3 times and EPO twice as much over SBVH. But disappointingly the average reduction of measured traversal cost is only about 3% higher. Actually, RSSBVH only performed best in five out of twelve scenes for diffuse rays, while RSBVH performed best in the remaining seven scenes. Considering that RSSBVH managed to achieve the highest reduction in SAM *and* EPO at the same time for eight scenes, this is an unexpected result. Though both SAM and EPO are clearly smaller in *Babylon*, *Bubs*, *Conference*, and *Sponza* than with any other builder, measured traversal cost is higher than for RSBVH in those scenes. The most severe example for this observation is *Sponza*, where RSSBVH also falls behind SBVH. RSSBVH has 14% lower SAM and 60% lower EPO cost than SBVH for this scene. The predicted decrease in traversal cost is 17%. But measured traversal cost is slightly higher than for SBVH. Both, the SAM metric and the combined SAM-EPO metric (which is designed for diffuse rays), fail to explain these observations for RSSBVH. Thus, there must be an unidentified effect which is not captured in the former metrics. We were unable to identify this extra cost and consider it an open question for future work.

5.6.2 Inferiority of Geometric Object Splits

The GBVH, RMGBVH, and RGBVH algorithm can be interpreted as extending BBVH, RMBVH, and RBVH with geometric object splits from Popov et al. [2009]. In all three cases adding geometric object splits resulted in a decrease in the quality of the original algorithms. Considering that geometric splits only extend the set of available partitions of the original algorithms and are only selected if considered beneficial this seems perplexing at first. Analyzing constructed BVHs we found that GBVH indeed finds geometric splits where c_{split} from Equation 5.2 is up to 40% lower than the best split found with plane-sweeping. To explain why the overall quality is nonetheless lower we first write Equation 5.1 for the SAM cost of a BVH in its iterative form. This partitions the cost of a BVH in costs $c_{\mathcal{I}}$ for processing inner nodes and costs $c_{\mathcal{L}}$ for processing leaves:

$$c_{BVH} = c_{\mathcal{I}} + c_{\mathcal{L}} = c_t \sum_{i \in \mathcal{I}} p_i + c_i \sum_{l \in \mathcal{L}} p_l |l|, \tag{5.14}$$

where $\mathcal{I}$ is the set of inner nodes and $\mathcal{L}$ is the set of leaves. p_i and p_l are the probabilities of intersecting an inner or leaf node w.r.t. the root bounds and $|l|$ is the number of primitives in a leaf. We found that while $c_{\mathcal{L}}$ for GBVH is essentially identical to BBVH larger SAM costs for GBVH mainly stem from a larger $c_{\mathcal{I}}$. Further analysis revealed that GBVH has a strong tendency to create children with a larger discrepancy in surface area of the children bounds than BBVH. Let $\underline{A}$ and $\overline{A}$ denote the surface area of the smaller and larger child respectively. On average $\underline{A}$ is smaller and $\overline{A}$ is bigger for GBVH compared with BBVH. This seems to be necessary to allow GBVH to locally find smaller c_{split}. But at the same time

$\underline{A}+\overline{A}$ turns out to be larger than for BBVH on average which directly causes the increase in $C_{\mathcal{J}}$. That is, GBVH is worse in reducing the average area of the children bounds, an effect not captured in Equation 5.2. This increase in $c_{\mathcal{J}}$ is most severe in the upper levels, where surface area of bounds is larger anyway. Our results comply with the observation from Aila et al. [2013]: It is not clear that locally minimizing Equation 5.2 globally reduces SAM, but the minimization maximizes saved worst-case triangle cost. On average overlap of children bounds is also increased. This might be a direct consequence of the larger average child bounds and explains the increase in EPO of algorithms with geometric splits.

We also observed the increase in average children bounds area for RMGBVH and RGBVH. Both algorithms are based on Equation 5.5, which includes the cost of temporarily constructed hierarchies, but $c_{\mathcal{J}}$ is also increased for those algorithms. Though this time the increase in cost for the direct child nodes is included in the candidate cost, the extra cost gets lost in the costs for the temporary hierarchies. Concluding, the supremacy of greedy top-down builders declared by Aila et al. [2013] does not hold for algorithms based on geometric splits.

We conducted experiments with GBVH where we included traversal costs for children into Equation 5.2. This proofed to be insufficient as quality decreased. Overcoming the blindness of Equation 5.2 towards increases in $c_{\mathcal{J}}$ should also benefit the plane-sweeping algorithm. Section 10.9 in the chapter on overall future work of this dissertation proposes an attempt in this direction and gives promising preliminary results on an experimental surface area heuristic, which is also applicable to GBVH.

5.7 Future Work

There are several directions for future work. One direction would be to find a BVH quality metric which explains the observations made for RSSBVH in Section 5.6.1.

Another direction would be to directly include EPO into the construction process. We can readily compute EPO of a candidate partition from the temporarily built BVHs combined with the node to split. This would allow us to directly use Equation 5.3 to guide construction into directions of low p. An unpleasant aspect of this approach is that construction depends on prior knowledge of α. Further, we would have to actually store temporary hierarchy nodes, which so far is not necessary with our algorithms (see Section 5.2). In this regard fast and accurate determination of α for unknown scenes is also an interesting problem.

The treelet-based BVH optimization algorithm proposed by Karras and Aila [2013] is only practical for small treelet sizes as their minimum-SAM BVH construction algorithm has $\Omega(\exp n)$ computational complexity and $O(\exp n)$ space requirements (see Section 2.5.5). While our RBVH algorithm does not construct minimum-SAM BVHs, its computational complexity and space requirements allows for much larger treelets. Combined with its high quality output it would be interesting to see what can be achieved.

Finally, additionally to the EPO metric Aila et al. [2013] proposed the leaf count variability (LCV) metric, which in a convex combination with SAM and EPO can explain SIMD performance of the efficient GPU ray tracing kernels of Aila and Laine [2009]. Aila et al. [2013] showed that top-down greedy SAH-based construction algorithms implicitly reduce LCV besides EPO. We assume that this property should be naturally inherited and boosted by our algorithms. That is, our BVHs might be specially suited for SIMD traversal. Experimental validation is left for future work.

						Primary rays				Diffuse rays			
	Builder	Time	Dupl.	SAM	EPO	$\overline{n}_s$	$\overline{n}_t$	p	m	$\overline{n}_s$	$\overline{n}_t$	p	m
Babylon	RSBVH	454.8s	69.2%	39.1	2.1	27.7	**2.8**	19.5	36.1	**29.5**	**4.2**	25.0	**39.6**
	RSSBVH	3285.2s	71.9%	**37.1**	**1.7**	**27.1**	2.9	**18.3**	**35.3**	29.6	4.3	**23.6**	39.8
	SBVH	16.8s	39.7%	40.5	2.6	27.7	3.4	20.4	36.7	30.4	4.9	26.0	41.4
	RBVH	250.3s	-	49.2	12.9	38.0	5.6	29.9	51.2	39.7	6.6	35.3	54.3
	RGBVH	63.8h	-	49.0	14.4	39.0	6.3	30.6	53.1	41.4	7.3	35.7	57.0
	BBVH	3.8s	-	53.7	15.1	38.7	5.5	33.2	51.8	42.6	7.0	39.0	58.2
	GBVH	268.7s	-	59.1	23.8	50.2	6.5	40.3	66.8	53.2	7.4	45.6	71.3
	RMBVH	43.9s	-	79.2	28.4	61.8	5.0	52.2	79.1	63.1	6.7	59.8	82.5
	RMGBVH	10.1h	-	85.2	47.1	90.7	7.6	65.0	116.4	87.1	7.7	70.7	112.2
Bubs	RSBVH	1874.2s	6.4%	15.8	1.9	**23.1**	2.9	15.8	30.6	**25.5**	**4.1**	15.8	**34.7**
	RSSBVH	3385.5s	6.5%	**15.6**	**1.0**	23.2	**2.7**	**15.6**	**30.5**	25.7	**4.1**	**15.6**	35.0
	RBVH	1500.2s	-	16.2	2.9	24.3	2.9	16.2	32.0	26.8	4.3	16.2	36.5
	SBVH	22.9s	2.5%	17.7	1.8	26.6	3.0	17.7	34.9	28.1	4.2	17.7	37.9
	GBVH	7992.0s	-	19.0	3.1	31.6	3.8	19.0	41.8	30.7	4.5	19.0	41.3
	RMBVH	236.9s	-	21.1	8.0	31.7	3.1	21.1	41.2	34.3	4.4	21.1	45.5
	BBVH	16.5s	-	24.2	8.4	34.3	3.4	24.2	44.6	36.0	4.7	24.2	47.9
Conference	RSBVH	219.3s	82.8%	33.4	3.0	21.8	**3.2**	19.8	**29.4**	24.4	**5.3**	25.2	**34.5**
	RSSBVH	1045.0s	81.2%	**33.0**	**2.8**	22.4	4.1	**19.5**	31.0	**24.3**	5.7	**24.9**	34.8
	SBVH	5.7s	30.1%	38.4	3.5	23.6	6.6	22.8	34.9	26.3	7.7	29.0	39.3
	RBVH	126.1s	-	38.6	7.1	**21.2**	6.9	24.5	32.4	26.3	10.3	30.1	41.8
	RGBVH	18.9h	-	38.2	7.3	22.6	7.3	24.3	34.4	26.2	10.7	29.9	42.1
	BBVH	2.1s	-	46.4	9.8	26.8	7.3	30.0	39.5	32.3	10.8	36.6	49.5
	GBVH	73.9s	-	47.3	10.6	28.7	7.9	30.9	42.4	34.0	11.0	37.5	51.7
	RMGBVH	3.1h	-	54.1	15.9	42.1	8.3	37.0	58.9	42.3	11.7	43.8	62.5
	RMBVH	25.0s	-	65.6	19.0	47.7	7.0	44.7	64.2	50.6	10.4	53.1	71.1
Epic	RSSBVH	2786.2s	66.6%	**17.7**	**1.6**	**41.8**	**4.2**	**7.2**	**54.3**	**41.4**	**6.8**	**8.2**	**56.5**
	RSBVH	433.5s	64.9%	18.2	1.8	44.2	4.3	7.4	57.3	41.8	**6.8**	8.4	57.0
	SBVH	11.3s	32.7%	19.7	3.0	42.5	5.0	8.7	56.1	41.4	7.9	9.7	57.6
	RBVH	237.5s	-	19.5	6.0	52.2	6.6	10.6	69.2	49.8	9.4	11.5	69.2
	RGBVH	69.5h	-	19.3	6.3	52.1	6.9	10.8	69.4	51.0	9.8	11.6	70.9
	BBVH	3.3s	-	21.3	7.1	54.7	7.0	12.0	72.6	52.7	9.9	12.8	73.1
	RMBVH	44.2s	-	21.3	8.1	68.2	7.0	12.6	88.8	64.6	9.8	13.4	87.3
	GBVH	134.9s	-	21.6	9.9	71.3	8.3	13.9	93.9	66.0	10.2	14.7	89.4
	RMGBVH	11.6h	-	21.2	9.8	82.4	8.4	13.7	107.3	73.3	10.2	14.4	98.2
Fairy	RSBVH	174.7s	32.2%	**31.3**	**2.7**	**27.3**	**4.8**	**8.4**	**37.5**	**30.6**	8.5	**12.0**	**45.2**
	RSSBVH	642.8s	33.8%	31.5	**2.7**	27.8	**4.8**	**8.4**	38.1	31.0	**8.4**	12.1	45.6
	RBVH	111.0s	-	31.5	3.0	28.9	5.4	8.7	40.0	31.8	9.0	12.3	47.2
	RGBVH	12.3h	-	32.0	3.4	29.0	5.6	9.1	40.5	32.0	9.1	12.7	47.5
	SBVH	3.2s	9.3%	34.7	**2.7**	32.6	5.0	9.1	44.2	32.9	8.6	13.1	48.1
	BBVH	1.4s	-	33.4	3.4	31.7	5.4	9.4	43.4	33.5	9.1	13.1	49.3
	RMGBVH	7258.4s	-	34.3	4.4	31.3	5.4	10.3	43.0	34.5	9.1	14.1	50.5
	RMBVH	18.5s	-	34.5	4.3	32.1	5.4	10.3	43.8	34.8	9.1	14.1	50.9
	GBVH	34.5s	-	35.7	4.8	42.5	6.2	11.0	57.2	40.7	9.1	14.8	58.0
Hairball	RSBVH	4544.3s	191.4%	**386.5**	**8.3**	**74.6**	**25.8**	**57.3**	**115.4**	**72.1**	29.8	**113.5**	**116.3**
	RSSBVH	9.5h	205.4%	392.1	**8.3**	75.4	**25.8**	58.0	116.3	73.3	**28.8**	115.1	116.8
	SBVH	136.4s	40.2%	428.0	28.3	80.6	44.5	80.0	141.2	75.8	46.6	139.5	137.5
	RBVH	1685.2s	-	454.0	36.7	82.0	55.0	90.8	153.3	78.3	56.2	152.8	150.1
	BBVH	23.8s	-	466.4	37.8	85.6	56.1	93.3	158.8	80.2	56.8	157.0	152.0
	RMBVH	377.1s	-	494.6	44.2	95.8	56.0	102.5	170.9	90.5	57.1	169.5	165.6
	GBVH	2.4h	-	470.7	49.8	130.6	75.4	104.3	232.1	95.5	58.8	166.9	173.4

Table 5.5: Results for the first two rows of scenes in Figure 5.2. $\overline{n}_s$, and $\overline{n}_t$ average the average number of measured traversal steps and triangle intersection tests. p is the EPO measure for BVH performance (Equation 5.3) and m the average measured traversal cost (Equation 5.13). For each scene builders are sorted from smallest to largest m. The highest reduction of each attribute is highlighted per scene.

						Primary rays				Diffuse rays			
	Builder	Time	Dupl.	SAM	EPO	$\overline{n}_s$	$\overline{n}_t$	p	m	$\overline{n}_s$	$\overline{n}_t$	p	m
Powerplant	RSSBVH	1944.7s	149.4%	**30.7**	**2.2**	**29.8**	**3.8**	**17.3**	**39.6**	**31.2**	**5.3**	**22.6**	**42.8**
	RSBVH	251.6s	149.0%	32.4	2.3	31.5	4.0	18.2	41.7	32.4	5.4	23.8	44.3
	SBVH	11.9s	84.5%	33.2	3.2	31.0	4.5	19.1	41.7	31.8	6.2	24.6	44.3
	RBVH	96.7s	-	41.1	13.0	39.7	13.1	27.9	60.8	39.8	12.6	33.1	60.3
	RGBVH	21.1h	-	40.2	14.2	40.9	13.9	28.0	63.0	40.9	12.9	32.8	62.0
	BBVH	2.0s	-	43.9	13.2	41.6	13.0	29.5	62.9	42.0	12.8	35.2	63.2
	GBVH	88.8s	-	47.0	22.0	52.3	14.5	35.3	77.3	51.4	13.2	39.9	74.0
	RMBVH	21.9s	-	74.4	35.7	85.0	14.1	56.2	116.1	80.7	13.5	63.4	110.4
	RMGBVH	2.6h	-	74.7	43.2	99.4	17.3	59.9	136.6	90.2	14.1	65.7	122.4
Rungholt	RSSBVH	5.9h	3.1%	**105.5**	**2.6**	**35.4**	**2.1**	**105.5**	**44.6**	**37.3**	**3.3**	**105.5**	**48.1**
	GBVH	4.1h	-	109.7	4.2	36.3	2.2	109.7	45.7	37.5	3.4	109.7	48.4
	BBVH	54.3s	-	109.9	3.4	36.1	2.2	109.9	45.6	37.7	**3.3**	109.9	48.5
	SBVH	160.0s	0.0%	109.9	3.4	36.1	2.2	109.9	45.6	37.7	**3.3**	109.9	48.5
	RSBVH	4797.4s	3.1%	105.6	**2.6**	36.4	**2.1**	105.6	45.8	37.8	**3.3**	105.6	48.6
	RBVH	2432.7s	-	**105.5**	2.7	36.6	**2.1**	**105.5**	46.0	37.8	**3.3**	**105.5**	48.7
	RMBVH	493.0s	-	113.4	3.5	37.3	**2.1**	113.4	46.9	38.8	**3.3**	113.4	49.8
San Miguel	RSBVH	4.1h	21.6%	16.6	**1.6**	**60.8**	6.6	10.2	79.6	**57.5**	**9.2**	12.3	**78.2**
	RSSBVH	14.0h	22.7%	**15.8**	2.1	**60.8**	**6.5**	**10.0**	**79.5**	59.2	**9.2**	**11.9**	80.3
	SBVH	211.5s	13.7%	19.6	3.1	61.8	6.8	12.6	80.9	60.2	9.9	14.9	82.2
	RBVH	2.5h	-	17.3	7.5	71.8	9.6	13.1	95.8	68.4	12.9	14.5	95.0
	BBVH	130.5s	-	20.3	10.2	83.1	9.8	16.0	109.5	76.5	13.2	17.4	104.9
	GBVH	28.5h	-	24.6	18.1	119.9	12.4	21.8	156.3	103.8	13.5	22.7	138.1
	RMBVH	2042.7s	-	27.6	20.5	142.4	10.8	24.6	181.7	120.6	13.8	25.6	158.5
Sibenik	RSSBVH	431.1s	80.9%	**44.6**	**1.2**	**32.9**	**3.5**	**17.6**	**43.0**	**34.0**	**5.8**	**28.0**	**46.7**
	RSBVH	58.4s	78.9%	46.6	1.3	33.2	**3.5**	18.4	43.4	34.8	**5.8**	29.2	47.6
	SBVH	2.2s	33.8%	47.5	1.6	35.9	4.3	19.0	47.5	34.6	6.6	29.9	48.1
	RBVH	29.0s	-	48.8	4.2	37.4	6.3	21.1	51.2	37.0	7.7	31.7	52.1
	RGBVH	2.1h	-	49.1	4.7	36.5	6.5	21.5	50.4	37.1	7.8	32.1	52.3
	BBVH	0.5s	-	53.6	5.0	42.4	6.4	23.4	57.3	39.4	7.6	35.0	54.8
	GBVH	10.3s	-	54.3	6.3	46.7	7.0	24.5	63.0	43.1	8.1	35.9	59.8
	RMBVH	6.6s	-	68.5	9.5	61.3	6.1	31.9	79.7	52.9	7.5	46.0	71.0
	RMGBVH	1027.9s	-	69.9	14.7	65.5	7.3	35.6	85.9	60.2	8.3	48.8	80.5
Soda	RSSBVH	2.2h	25.7%	**58.7**	**1.9**	31.6	**3.3**	**33.1**	41.3	**30.4**	**5.1**	**38.6**	**41.6**
	RSBVH	2396.6s	25.7%	61.1	2.4	31.3	3.5	34.6	**41.1**	**30.4**	5.2	40.3	41.8
	SBVH	45.6s	14.0%	66.6	2.7	32.8	3.8	37.8	43.1	30.7	5.6	43.9	42.4
	RBVH	1451.7s	-	66.2	10.2	**31.2**	5.2	40.9	42.6	34.3	7.8	46.3	49.0
	BBVH	21.4s	-	77.9	13.7	36.0	5.7	49.0	48.8	38.8	8.5	55.2	55.1
	GBVH	2.1h	-	85.4	26.9	53.4	6.9	59.0	71.0	49.0	8.8	64.7	67.6
	RMBVH	252.3s	-	110.3	31.9	64.8	6.6	74.9	84.4	62.2	8.5	82.5	83.1
Sponza	RSBVH	257.8s	57.6%	65.0	4.6	**39.8**	3.9	19.8	**51.7**	41.7	**6.0**	27.6	**56.1**
	SBVH	7.4s	28.8%	70.2	3.0	45.1	**3.8**	19.9	57.9	41.9	6.3	28.6	56.6
	RSSBVH	2052.2s	61.4%	**60.9**	**1.2**	42.9	4.3	**16.2**	55.8	**41.6**	6.9	**23.9**	56.7
	RGBVH	36.0h	-	69.2	7.5	48.6	7.2	23.0	65.5	46.6	8.3	30.9	64.2
	RBVH	133.4s	-	70.9	7.9	47.0	7.1	23.7	63.5	48.4	9.1	31.8	67.2
	GBVH	96.9s	-	78.1	9.9	66.4	8.5	27.1	88.2	57.7	8.7	35.9	77.9
	RMBVH	25.9s	-	85.6	12.0	61.7	5.2	30.5	79.1	62.0	7.1	40.0	81.5
	BBVH	2.1s	-	83.1	13.0	63.6	5.8	30.6	82.1	62.3	8.4	39.7	83.2
	RMGBVH	4.5h	-	90.8	16.9	86.6	9.4	35.5	113.3	73.5	9.4	45.0	97.5

Table 5.6: Results for the last two rows of scenes in Figure 5.2. See Table 5.5 for a description of each measurement.

Chapter 6

An SAM-Driven Approach to Agglomerative Clustering

Contents

The construction algorithms we presented in the previous chapter where all top-down approaches. Unlike kd-trees, BVHs can also be build bottom-up. The highest quality bottom-up approach is the *agglomerative clustering* algorithm from Walter et al. [2008]. Initially, for each input primitive the algorithm creates a leaf node, which references the respective primitive. Then, iteratively nodes are clustered by creating a new parent node for a pair of nodes. The authors proposed to always cluster the two nodes where the tight parent bounds have the smallest surface area. This gives the following clustering criterion for the current set $\mathcal{N}$ of clusterable nodes:

$$x_{best} = \operatorname*{arg\,min}_{(n,m)\in\mathcal{N}\times\mathcal{N},\, n\neq m} \texttt{Area}(B_{n\circ m}). \tag{6.1}$$

Here, $B_{n\circ m}$ are the tight bounds for the bounds B_n and B_m of nodes n and m. The intuition behind this clustering given by Walter et al. [2008] is that nodes with small surface areas should be preferred in the spirit of the surface area metric. As with this construction algorithm each leaf contains exactly one primitive the authors further proposed to perform collapsing of clustered nodes into leaves during construction or as a post processing pass on the constructed BVH if this is beneficial w.r.t. the SAM cost of the subtree.

While the authors report positive results throughout, reevaluation by Aila et al. [2013] showed very mixed results. Especially, the new EPO cost proposed by Aila et al. [2013] seems to be comparatively high with agglomerative clustering. Related work only focused on faster agglomerative clustering-based (e.g. Gu et al. [2013] and Meister and Bittner [2016]) construction. While Walter et al. [2008] only used a SAM-inspired approach we investigate an agglomerative clustering approach, which includes the SAM throughout. We start by introducing the SAM cost for a forest of BVHs. Then we propose two clustering heuristics, which are derived from the forest cost. Finally, we evaluate both heuristics and conclude with a discussion of the results.

6.1 SAM Cost of a BVH Forest

Like the approach from Walter et al. [2008] we start by constructing separate leaf nodes for each input primitive. We can interpret this set of leaves as a forest of roots $\mathcal{R}$ of BVHs, where initially each BVH is just a leaf. To find the intersection of a ray with this forest, each forest root has to be tested for intersection. For each intersected root the subtree has to be processed. To define the cost of processing a forest we have to introduce the cost constant c_r for the cost of testing a forest root for intersection. As there is no hierarchy yet, the initial cost for the forest of leaf BVHs is:

$$c_{leafforest} = |\mathcal{R}|c_r + \sum_{n \in \mathcal{R}} p_n c_i. \tag{6.2}$$

That is we have the number of forest roots $|\mathcal{R}|$ times the cost of intersecting a forest root plus the sum of the probability of intersecting each root times the cost of intersecting a single primitive. To construct a single BVH from the forest, pairs of forest roots have to be iteratively clustered. There are two possible clustering operations for how two nodes can be clustered. We can either create a leaf node which contains all primitives of the two original nodes, or we create a new common parent node for the pair. Before we discuss possible clustering criteria for which nodes should be clustered let us assume that several clustering iterations have been performed with an arbitrary clustering criterion. The resulting forest has a smaller number of roots and the roots can have subtrees with more than one node. Thus, the cost of a BVH forest is

$$c_{forest} = |\mathcal{R}|c_r + \sum_{n \in \mathcal{R}} c(n), \tag{6.3}$$

where $c(n)$ is the SAM cost of the subtree of a forest root n.

6.2 Clustering Criteria

To derive our clustering criteria we first analyze the subtree cost of the two clustering operations. When creating a leaf for a pair $(n, m) \in \mathcal{R} \times \mathcal{R}$ of roots the cost of the leaf is

$$c_{leaf}(n, m) = c_r + (|n| + |m|) p_{n \circ m} c_i, \tag{6.4}$$

where $p_{n \circ m}$ is the probability of intersecting the tight bounds of the new leaf node $n \circ m$ and $|n|$ and $|m|$ are the total numbers of primitives in the subtrees of n and m. Instead, creating a new common parent node for a pair (n, m) results in a new subtree with the cost

$$c_{subtree}(n, m) = c_r + p_{n \circ m} c_t + c(n) + c(m). \tag{6.5}$$

The operation which results in lower costs should be chosen. Thus, the cost for clustering two nodes is

$$c_{cluster}(n, m) = \texttt{inf}\{c_{leaf}(n, m), c_{subtree}(n, m)\} \tag{6.6}$$

Based on these results we found two possible criteria for clustering nodes. The first criterion is to find the pair $x \in \mathcal{R} \times \mathcal{R}$ which gives the node with smallest subtree cost. That is, we have to compute

$$x_{best} = \underset{(n,m)\in\mathcal{R}\times\mathcal{R},\, n\neq m}{\arg\min} \; c_{cluster}(n,m). \tag{6.7}$$

The idea behind this heuristic is that it should incrementally produce subtrees with small SAM cost and that clustering of several subtrees with low cost again produces low cost subtrees. The second criterion is to find the pair $x \in \mathcal{R}\times\mathcal{R}$ which gives the largest reduction in the forest cost function from Equation 6.3. When clustering a node pair the change in forest costs is the change in the cost contribution caused by those nodes. The contribution of a pair of root nodes (n,m) to the total forest cost is

$$c_{pair}(n,m) = 2c_r + c(n) + c(m). \tag{6.8}$$

The cost delta when clustering (n,m) is simply the difference of the clustering cost and the pair cost:

$$\Delta c_{forest}(n,m) = c_{cluster}(n,m) - c_{pair}(n,m) \tag{6.9}$$

Now, to find the best clustering candidate pair we have to compute

$$x_{best} = \underset{(n,m)\in\mathcal{R}\times\mathcal{R},\, n\neq m}{\arg\min} \; \Delta c_{forest}(n,m). \tag{6.10}$$

Expanding the functions in the definition of $\Delta c_{forest}(n,m)$ in Equation 6.9 we get:

$$\begin{aligned}
\Delta c_{forest}(n,m) &= \mathtt{inf}\{c_{leaf}(n,m), c_{subtree}(n,m)\} - c_{pair}(n,m)\\
&= \mathtt{inf}\{c_r + (|n|+|m|)p_{n\circ m}c_i, c_r + p_{n\circ m}c_t + c(n) + c(m)\} - c_{pair}(n,m)\\
&= c_r + \mathtt{inf}\{(|n|+|m|)p_{n\circ m}c_i, p_{n\circ m}c_t + c(n) + c(m)\} - 2c_r - c(n) - c(m)\\
&= -c_r + \mathtt{inf}\{(|n|+|m|)p_{n\circ m}c_i - c(n) - c(m), p_{n\circ m}c_t\}.
\end{aligned} \tag{6.11}$$

We can see that independent of the clustering operation the forest cost is always reduced by the forest root intersection cost c_r as there is one less root node after clustering. Thus, this addend is shared by all candidates and can be ignored. Consequently, c_r does not have to be specified at all for construction. This is also true for construction with our first criterion. Another observation is that if clustering decides to create a new parent node the resulting cost delta is $\Delta c_{forest}(n,m) = p_{n\circ m}c_t - c_r$. Ignoring the common c_r addend this delta is proportional to $\mathtt{Area}(B_{n\circ m})$. That is, the algorithm of Walter et al. [2008] is included as a special case. Our SAM based approach also easily integrates leaf creation into construction in a meaningful way which is not possible with the purely area based heuristic from Walter et al. [2008].

A naïve implementation of agglomerative clustering has runtime $O(n^3)$. Walter et al. [2008] presented an implementation which uses an auxiliary min-heap for fast retrieval of best clustering candidate pairs and a special kd-tree with extra information for empirically fast determination of an optimal clustering partner for a given node. Performance of their implementation has empirically shown to be sub-quadratic. The implementation can directly be adapted to use our clustering criteria.

Abbreviation	Algorithm
BBVH	Baseline Plane-Sweep [MacDonald and Booth 1990]
AGGLO	Aggl. clustering with pure surface area criterion [Walter et al. 2008]
SAGGLO	Aggl. clustering with subtree SAM cost criterion (Equation 6.7)
ΔSAGGLO	Aggl. clustering with forest SAM cost delta criterion (Equation 6.10)
RBVH	RSAH-based algorithm without spatial splits (Chapter 5)

Table 6.1: List of algorithms and their abbreviations used for evaluation of our proposed clustering criteria for agglomerative clustering.

	Avg. (Min/Max) reduction (%)					
			Primary rays		Diffuse rays	
Algorithm	SAM	EPO	*p*	*m*	*p*	*m*
RBVH	-11.7 (-2.7/-33.3)	-22.2 (-1.4/-65.3)	-13.3 (-2.7/-33.3)	-10.4 (+0.9/-28.3)	-12.7 (-2.7/-33.3)	-8.9 (+0.4/-23.7)
ΔSAGGLO	-3.0 (+36.2/-34.4)	+82.3 (+655.9/-55.0)	+5.0 (+36.2/-34.4)	+16.5 (+83.7/-18.8)	+1.7 (+36.2/-34.4)	+13.3 (+66.6/-15.9)
AGGLO	-2.6 (+36.0/-34.3)	+82.6 (+643.7/-53.8)	+5.4 (+36.0/-34.3)	+17.0 (+82.9/-16.7)	+2.0 (+36.0/-34.3)	+13.5 (+63.2/-14.8)
SAGGLO	+56.2 (+147.4/-6.0)	+594.6 (+3346.3/+26.8)	+122.6 (+316.9/-6.0)	+154.9 (+386.8/+30.7)	+95.3 (+198.8/-6.0)	+125.3 (+319.2/+23.4)

Table 6.2: Average, minimum, and maximum reduction of SAM, EPO, as well as predicted (p) and measured (m) traversal cost of primary and diffuse rays over all scenes relative to BBVH as baseline. Algorithms are sorted from highest to lowest reduction in traversal cost of diffuse rays.

6.3 Evaluation

For evaluation of our proposed clustering criteria we used the same hardware and measurement setup as in Section 5.4. We give SAH and EPO cost, and predicted and measured traversal cost. For the predicted cost we used the alpha values from Table 5.1 in Chapter 5. Table 6.1 lists all evaluated BVH construction algorithms along with their abbreviations. The baseline construction algorithm is the standard plane-sweep algorithm (BBVH). The first agglomerative clustering based algorithm uses the surface area based criterion from Equation 6.1 as proposed by Walter et al. [2008] (AGGLO). The other two algorithms use either our subtree SAM cost criterion (SAGGLO) from Equation 6.7 or our BVH forest SAM cost delta criterion (ΔSAGGLO) from Equation 6.10. Finally, we also included our RSAH-based RBVH algorithm without spatial splits from Chapter 5 for comparison. All results are compiled in Table 6.3 and Table 6.4. Table 6.2 gives a more condensed view of the results.

Unexpectedly, SAGGLO by far results in the highest SAM, EPO, and measured traversal cost for all scenes. Our RBVH algorithm achieves the best result in all scenes except for *Conference*, where ΔSAGGLO is best, and *Rungholt*, where the baseline BBVH is best. ΔSAGGLO performs slightly better than AGGLO in eight out of twelve scenes. Though compared to BBVH ΔSAGGLO and AGGLO give a slightly better result in *Babylon* and much better results in *Bubs*, *Conference*, and *Sponza*, they are inferior to BBVH in the

					Primary rays		Diffuse rays	
Scene	Builder	Time	SAM	EPO	p	m	p	m
Babylon	RBVH	250.25	49.22	**12.86**	**29.91**	51.15	35.31	**54.25**
	ΔSAGGLO	15.79	**47.98**	14.23	30.05	**51.03**	**35.07**	55.35
	AGGLO	15.85	48.97	15.19	31.02	53.10	36.05	56.98
	BBVH	3.82	53.72	15.12	33.21	51.89	38.95	58.15
	SAGGLO	24.47	78.99	49.58	63.36	106.43	67.74	107.64
Bubs	RBVH	1500.20	16.16	**2.91**	16.16	**31.96**	16.16	**36.53**
	ΔSAGGLO	107.55	**15.90**	3.77	**15.90**	36.19	**15.90**	40.54
	AGGLO	92.97	15.91	3.87	15.91	37.12	15.91	40.77
	BBVH	16.52	24.23	8.38	24.23	44.55	24.23	47.87
	SAGGLO	129.79	22.76	10.63	22.76	68.32	22.76	67.22
Conference	ΔSAGGLO	12.20	**37.83**	7.62	**24.30**	32.70	**29.74**	**41.59**
	RBVH	126.05	38.56	**7.09**	24.47	**32.35**	30.14	41.80
	AGGLO	9.15	38.11	7.66	24.47	33.48	29.96	42.22
	BBVH	2.11	46.44	9.79	30.03	39.50	36.63	49.47
	SAGGLO	12.82	53.35	18.56	37.77	51.63	44.03	61.03
Epic	RBVH	237.50	**19.47**	**6.02**	**10.63**	**69.24**	**11.47**	**69.20**
	BBVH	3.34	21.33	7.06	11.95	72.62	12.84	73.08
	ΔSAGGLO	17.04	20.22	9.16	12.96	77.58	13.64	77.23
	AGGLO	15.41	20.21	9.04	12.88	80.03	13.57	77.38
	SAGGLO	25.34	33.15	31.53	32.09	170.83	32.19	153.32
Fairy	RBVH	111.01	**31.48**	**2.97**	**8.66**	**40.03**	**12.25**	**47.20**
	BBVH	1.40	33.38	3.37	9.36	43.42	13.14	49.26
	AGGLO	5.97	36.67	6.45	12.48	48.18	16.29	55.20
	ΔSAGGLO	9.37	36.79	6.69	12.69	51.63	16.48	57.07
	SAGGLO	9.51	51.19	17.72	24.40	84.12	28.61	84.91
Hairball	RBVH	1685.17	**453.97**	**36.72**	**90.75**	**153.29**	**152.77**	**150.12**
	BBVH	23.80	466.36	37.82	93.31	158.78	157.01	152.96
	ΔSAGGLO	1104.88	474.34	64.55	117.62	224.48	178.53	210.06
	AGGLO	887.52	476.31	65.18	118.41	229.78	179.53	210.58
	SAGGLO	1875.35	858.73	319.07	388.95	720.72	469.17	583.76

Table 6.3: Results for agglomerative clustering and the first two rows of scenes in Figure 5.2 of the previous chapter. p is the EPO-based measure for BVH performance (Equation 5.3) and m the average measured traversal cost (Equation 5.13). For each scene builders are sorted from smallest to largest m of diffuse rays. The highest reduction of each attribute is highlighted per scene.

remaining majority of scenes. Drastic increases in SAM and/or EPO can be observed in *Fairy*, *Hairball*, *Powerplant*, *Rungholt*, *Sibenik*, and *Soda*.

6.4 Discussion

The clustering criterion based on subtree SAM cost from SAGGLO turned out to be detrimental for quality. Analysis of constructed trees revealed that with this criterion the bounds of clusters where on average bigger than with AGGLO or ΔSAGGLO. The total leaf cost contribution $c_{\mathcal{L}} = \sum_{l \in \mathcal{L}} p_l |l| c_i$ of all leaves $\mathcal{L}$ is essentially the same and at times even lower than with the other algorithms. That is, SAGGLO essentially constructed similar leaves as the other algorithms. Thus, the main difference must be the inner node case cost $c_{subtree}(n, m)$ of the clustering criterion in Equation 6.7. Ignoring the forest root

					Primary rays		Diffuse rays	
Scene	Builder	Time	SAM	EPO	p	m	p	m
Powerplant	RBVH	96.65	**41.06**	**12.97**	**27.85**	**60.83**	**33.08**	**60.28**
	BBVH	2.01	43.93	13.16	29.46	62.90	35.19	63.17
	ΔSAGGLO	6.03	46.11	20.36	34.00	74.51	38.80	74.52
	AGGLO	6.92	46.44	20.37	34.18	79.48	39.04	75.27
	SAGGLO	9.35	73.60	58.26	66.39	144.20	69.24	131.15
Rungholt	BBVH	54.26	109.86	3.43	109.86	**45.56**	109.86	**48.51**
	RBVH	2432.67	**105.49**	**2.74**	**105.49**	45.99	**105.49**	48.68
	AGGLO	231.65	149.39	25.53	149.39	83.32	149.39	79.16
	ΔSAGGLO	292.97	149.64	25.95	149.64	83.69	149.64	80.82
	SAGGLO	421.88	271.83	118.31	271.83	221.80	271.83	203.34
San Miguel	RBVH	8985.24	17.25	**7.52**	**13.08**	**95.79**	**14.48**	**95.01**
	BBVH	130.53	20.28	10.21	15.97	109.46	17.42	104.94
	ΔSAGGLO	1076.92	**17.04**	10.26	14.14	109.22	15.12	110.36
	AGGLO	881.36	17.21	10.28	14.24	107.64	15.24	111.13
	SAGGLO	1558.25	29.76	39.86	34.09	259.63	32.63	237.68
Sibenik	RBVH	28.97	**48.75**	**4.16**	**21.06**	**51.21**	**31.69**	**52.07**
	BBVH	0.52	53.64	5.00	23.43	57.25	35.03	54.81
	ΔSAGGLO	2.41	55.42	9.53	26.92	59.12	37.86	62.26
	AGGLO	1.75	55.69	10.09	27.37	59.51	38.25	62.84
	SAGGLO	3.10	87.38	34.75	54.69	113.15	67.24	106.18
Soda	RBVH	1451.70	**66.15**	**10.17**	**40.90**	**42.61**	**46.32**	**48.96**
	BBVH	21.35	77.93	13.70	48.96	48.82	55.18	55.05
	AGGLO	86.82	82.41	28.51	58.10	80.41	63.32	77.38
	ΔSAGGLO	88.37	81.99	27.85	57.57	84.78	62.81	77.53
	SAGGLO	127.50	133.87	92.06	115.01	176.43	119.06	155.09
Sponza	RBVH	133.39	**70.86**	**7.85**	**23.72**	**63.54**	**31.83**	**67.20**
	ΔSAGGLO	13.87	72.04	8.58	24.57	73.87	32.74	76.82
	AGGLO	10.36	72.05	8.39	24.42	75.45	32.62	78.32
	BBVH	2.06	83.14	12.95	30.63	82.07	39.66	83.20
	SAGGLO	16.30	107.31	29.50	49.10	143.47	59.12	134.42

Table 6.4: Results for agglomerative clustering and the last two rows of scenes in Figure 5.2. See Table 6.3 for a description of each measurement.

constant c_r the inner node cost is

$$c_{subtree}(n,m) = p_{n \circ m} c_t + c(n) + c(m).$$

When searching for a best clustering partner for a node n higher values of $p_{n \circ m} c_t$ can be compensated with lower $c(m)$ of other nodes m. Thus this criterion is to some extent blind to the larger cluster bounds. This effect propagates up the tree resulting in bounds which are almost as big as the scene bounds in the first few upper BVH levels.

While on average better than AGGLO, ΔSAGGLO turned out to be not a significant improvement. Though ΔSAGGLO more directly aims at reducing SAM cost clustering decisions are made very local due to the greedy nature. This can result in a seemingly beneficial series of local clustering operations which still at times result in higher global cost compared to AGGLO. The higher SAM and EPO cost of AGGLO and ΔSAGGLO compared to BBVH for most of the scenes can be explained by the ignorance of the clustering process towards the inner node bounds in the upper levels from the completely local clustering decision making. Clustering decisions in lower levels can prevent good separation of nodes in upper levels. BVH analysis showed that the drastic increases in SAM and/or EPO of these algorithms observed in the previous section for most scenes are caused by larger bounds and/or higher overlap in the upper levels compared to the BVHs constructed with BBVH and RBVH. In the light of this the better results of AGGLO and ΔSAGGLO than BBVH for four scenes could be considered accidental as their clustering order happened to produce good upper bounds.

Though by far not as severe as in the previous chapter we can observe two cases where the combined SAM-EPO predictor is not sufficient to predict traversal cost for clustering based construction. In *Epic* AGGLO has slightly lower SAM and EPO than ΔSAGGLO, but slightly higher measured traversal cost for diffuse rays and a more pronounced difference for primary rays. This can also be observed in *Soda* where ΔSAGGLO has lower SAM and EPO than AGGLO but higher measured cost. In the previous chapter we only observed this behavior for the RSSBVH algorithm, which also applies spatial splits. From this, one might have concluded that spatial splits cause some uncaptured extra traversal cost. This makes it all the more interesting that we observed this behavior without spatial splits for AGGLO and ΔSAGGLO.

Future Work An interesting venue for future work might be to find a way to make agglomerative clustering sensitive to the bounds in upper tree levels. One possibility would be a hybrid top-down/bottom-up construction, which performs SAH-based plane-sweep construction in the top levels and agglomerative clustering in the remaining levels. Gu et al. [2013] did a spatial median split pre-partitioning to speed-up agglomerative clustering by essentially performing less clustering.

An alternative clustering approach might be to restrict the set of possible clustering partners to candidates from a plane-sweeping partition. For this, three separate arrays of cluster root references would have to be maintained. Each array sorts the cluster references w.r.t. the x, y, or z coordinate of the cluster bounds centroid, respectively. To find the best clustering partner each cluster only considers neighboring references in each array. After a new cluster has been created its two subclusters are removed from the arrays and a new cluster reference for the new cluster is inserted into the arrays. An array maintenance step is required to keep the clusters sorted. The implicit partition planes encountered in each step of this algorithm would also be considered by a plane-sweeping algorithm and should at least result in a better separation of clustered nodes. Though EPO should still be a problem, it would be interesting to see if this restricted clustering results in better BVHs compared to standard agglomerative clustering. Given the unreliability regarding the quality of BVHs produced by clustering it is unclear if it is worthwhile to use agglomerative clustering at all.

According to Ize and Hansen [2011] their RTSAH metric, which we briefly discussed in Section 2.5.8, might also be used for acceleration structure construction. The authors did not present approaches for this. We think agglomerative clustering is a perfect candidate for inclusion of RTSAH as it can be directly integrated into our BVH forest SAM cost delta criterion from Equation 6.10. Also, due to the bottom-up nature of clustering it is very simple to compute the RTSAH cost of subtrees during construction. On the other hand inclusion of RTSAH would not remove the problem of too large bounds in the upper tree. Still, it would be interesting if there is any benefit.

Chapter 7

Cache-Optimized BVH GPU Memory Layouts for Tracing Incoherent Rays

Contents

The last three chapters aimed at constructing higher quality BVHs to increase ray tracing performance. This traversal performance increase comes from a reduced average number of intersection tests that have to be performed. In this chapter we shift the focus on the ray tracing performance aspect of parallel acceleration structure traversal. Theoretically, ray tracing is embarrassingly parallel as different rays can be traced independently. On multi-core systems it is implemented in a straightforward manner by letting each thread process its own batch of rays. Further parallelization can be achieved by taking advantage of SIMD capabilities of multi-core architectures or the massive parallelism of many-core architectures such as GPUs, which are the focus of this chapter. Efficient parallelization on SIMD architectures is, however, much harder due to *incoherent rays* whose origins and directions vary widely.

Tracing incoherent rays requires traversing different paths through the acceleration structure, resulting in incoherent memory accesses since different nodes are traversed and different primitives are tested. As incoherent rays form an absolute majority they pose a serious challenge. It is thus important to carefully choose where (i.e., in which memory area) and how to layout data and to use special instructions to unlock the hardware's full potential. GPUs typically achieve their massive parallelism by a wide SIMD width (in our case 32 lanes, see Section 3.2) yielding the following challenges for an efficient implementation:

- *SIMD efficiency (ratio of active to total number of SIMD lanes)*: Especially for incoherent rays, the SIMD efficiency can be low since the number of acceleration structure nodes that a ray has to test in order to find the nearest intersection can vary significantly. Some rays terminate earlier than others, leaving a number of SIMD lanes idle.

- *SIMD divergence:* Even if all SIMD lanes have active rays, some may want to test geometry while others are still traversing the acceleration structure. In that case the execution paths of the lanes diverge and SIMD efficiency is temporarily lower until the execution paths re-converge.

- *Memory bandwidth/latency:* As incoherent rays access many different memory addresses, the number of different cache lines accessed increases, too. On current GPUs only a single cache line can be read at a time. In the worst case, each SIMD lane accesses a different cache line, resulting in serialization of the accesses and increased latency [Aila and Karras 2010].

We focus on the memory effects of tracing incoherent rays on NVIDIA GPUs. “Real-world” incoherent rays are generated by a basic path tracer. Presumably, the cache efficiency when tracing incoherent rays is low. We analyze our GPU path tracer and the effects of rearranging the nodes of the acceleration structure (a bounding volume hierarchy) on cache efficiency using previously recorded access statistics. Our goal is to increase cache hit rates and reduce the number of cache lines read per access. Our contributions are the analysis of the cache behavior when tracing incoherent rays in real-world scenarios. In particular, we show that the commonly used depth-first search memory layout performs worst and we present several alternative layouts. None of those performs, however, best in all cases.

7.1 Related Work

Plunkett and Bailey [1985] first implemented ray tracing on a vector processor. With the widespread availability of SIMD architectures, research on efficiently implementing ray tracing on such architectures proliferated. Wald et al. [2001b] presented an SIMD implementation of a ray tracer using Intel’s SSE instructions. Their packet tracing technique exploits ray coherence by tracing rays in packets of SIMD width size (4 for SSE) which achieves good caching behavior and yields a speed-up of roughly half an order of magnitude. Memory bandwidth is reduced by loading a node only once for packets of 4 rays. Later, Wald et al. [2007] proposed a combination of packet and frustum tracing. Using a packet size larger than the native SIMD width and different optimizations, they reported 3.3-10.7× speed-ups over the native SIMD packet size. Purcell et al. [2002] first presented a complete GPU ray tracing pipeline which had to map all computations to the GPU’s rendering pipeline. First ray tracing implementations using NVIDIA’s CUDA [NVIDIA 2016a] were proposed by Gunther et al. [2007] and Popov et al. [2007].

Aila and Laine [2009] presented different trace loop implementation organizations. The key difference to packet tracing is that essentially single ray tracing in an SIMD manner is performed using scatter/gather operations and hardware SIMD divergence handling. Rays only visit nodes which they actually intersect, but memory accesses become more incoherent. The *speculative while-while* loop organization performed best. It processes rays in one of two phases at a time: traversal or triangle intersection. During traversal, an SIMD lane traverses the tree until it finds a leaf. If some SIMD lanes have not yet found a leaf, the SIMD lane stores its found leaf and speculatively continues traversal until every SIMD lane found a leaf. Though this may result in superfluous memory accesses, the memory bandwidth overhead is generally low enough and the higher SIMD efficiency results in a 10% lower runtime.

Ray grouping and reordering Simply grouping rays into packets only works well for coherent rays. Therefore, techniques that extract hidden ray coherence using regrouping or reordering ray packets have been developed. Pharr et al. [1997] and Navratil et al. [2007] proposed to defer ray processing at certain queue points. Queue processing is scheduled to minimize and amortize cache misses, and reduce memory bandwidth demand when computing intersections with scene geometry. Mansson et al. [2007] investigated several regrouping algorithms for secondary rays. Further strategies are regrouping by ray type [Boulos et al. 2007], by hashes generated from a ray's geometry [Garanzha and Loop 2010], by approximations of ray intersection points [Moon et al. 2010], or by ray packet filtering [Boulos et al. 2008].

Cache efficient algorithms We can distinguish two types of cache-efficient algorithms: *Cache-aware* algorithms explicitly use prior knowledge about caches (e.g., cache-line size). *Cache-oblivious* algorithms [Prokop 1999] only assume that a cache is present without knowing any of its properties.

Aila and Karras [2010] presented a massively parallel hardware architecture which is to some extent based on NVIDIA's Fermi GPU architecture. They developed a cache-aware traversal algorithm specifically designed for this architecture, which achieves up to a 90% reduction in total memory bandwidth for tracing incoherent rays. A major assumption of the algorithm is, that the L1 cache can access multiple cache lines per clock (otherwise L1 fetches are a serious bottleneck). However, according to Aila and Karras [2010] this was not the case at the time of their publication. To our knowledge, L1 caches of current hardware still do not have such capabilities. While the latest NVIDIA Nsight Profiler User Guide [NVIDIA 2017b] states in its memory statistics section, that memory accesses which spread over several L1 or texture cache lines incur several transactions per request, it is left open whether multiple transactions can be performed per clock.

Wald et al. [2001b] and Havran [1999] optimized cache efficiency by either storing just one child pointer or completely omitting them through special node arrangements, thus reducing node size. Kim et al. [2010] proposed a random-accessible compressed BVH with context-based arithmetic coding. Combined with random accessible compressed triangle meshes [Yoon and Lindstrom 2007] they achieve an average rendering time improvement of 35-54% due to increased cache efficiency and hit rate as more nodes fit into the cache. Yoon and Manocha [2006] proposed a cache-oblivious BVH layout for collision detection. They also conducted raytracing experiments with coherent rays where they adopted their layout to k-d trees. This resulted in a 44% runtime improvement compared to a kd-tree with depth-first layout. Emde Boas [1975] derived a cache-oblivious tree memory layout built by recursively subdividing the height of the tree in half yielding a number of sub-trees per step. This clusters nodes and is beneficial for caches since traversing a node causes nodes of the subtree below the current node to be loaded into the cache which are likely

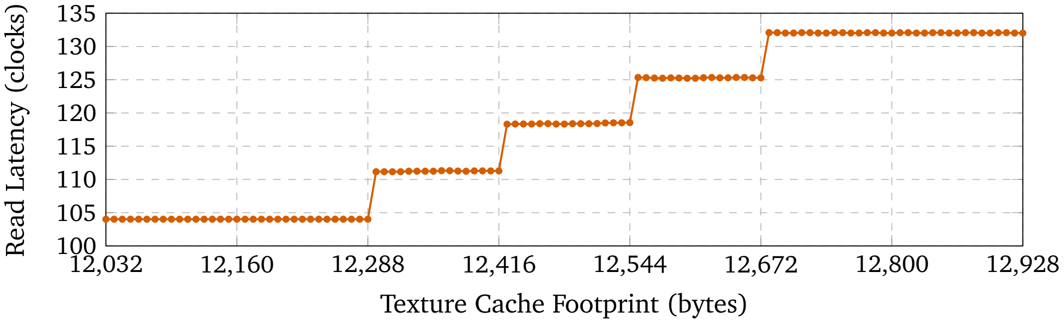


Figure 7.1: Average texture memory L1 cache latency in cycles of a Geforce GTX 680 revealing the cache properties.

to be traversed as well. Gil and Itai [1999] proposed a dynamic programming algorithm which allocates tree nodes to memory pages, minimizing the number of visited memory pages and page faults. Bender et al. [2002] present faster but approximate algorithms for solving the same problem in a cache-oblivious manner. Multi-branching BVHs [Ernst and Greiner 2008, Dammertz et al. 2008, Wald et al. 2008] improve cache efficiency by simply requiring less memory thus reducing bandwidth demand and keeping more nodes in the cache. Contrary to packet tracing a single ray is tested against SIMD width size number of bounding boxes and triangles. This is beneficial for incoherent rays but slower for coherent rays compared to packet tracing.

7.2 GPU Hardware Details / Test Setup

We made minor modifications to the micro-benchmarking code from Wong et al. [2010] and ran it on an NVIDIA Geforce GTX 680 to determine cache properties and access latencies. We evaluated the GPUs and performed all benchmarks using CUDA Version 5.0 [NVIDIA 2016a], the NVIDIA Nsight Visual Studio Edition 3.0 Beta, and the CUDA Profiler Tools Interface (CUPTI). The test system is equipped with an Intel Core i7-960, 32 GB RAM, and an NVIDIA Geforce GTX 480 (primary device) as well as a GTX 680 with 2 GB RAM (headless device). All tests were performed on the GTX 680 using driver version 306.94.

7.2.1 Cache Properties

The Geforce GTX 680 consists of eight Streaming Multiprocessors (SMX) with 192 CUDA cores each. It provides 2048 MB of global/texture memory, 16, 32 or 48 KB of shared memory or L1 cache for local memory (depending on the runtime configuration) and 65536 registers per SMX. The fetch latency for a global memory load of 4 bytes hitting the L2 cache takes ≈160 cycles while a miss results in a latency of ≈290 cycles.

Given the average texture memory cache access latency in Figure 7.1 retrieved from the micro-benchmark, we can deduce that the texture cache size is 12 KB, consisting of 4 cache sets with a cache line size of 128 bytes and is 24-way set associative. L1 hit latency for reading 4 bytes is ≈105 cycles, L2 hit latency is ≈266 cycles and missing both L1 and

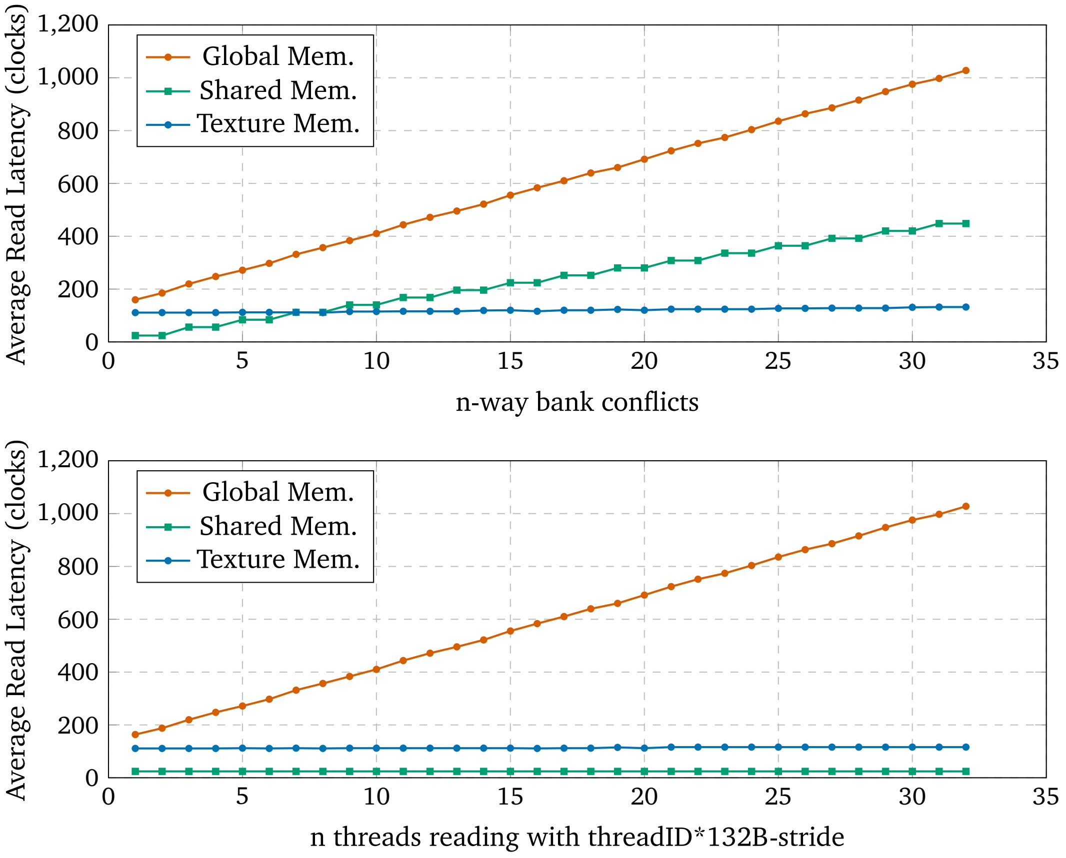


Figure 7.2: Average read latency in cycles of a Geforce GTX 680 using a memory access pattern which would cause n-way bank conflicts (top) in shared memory and a shared memory optimal access pattern (bottom).

L2 incurs a latency of ≈350 cycles. Figure 7.2 shows the latency of two different access patterns evaluated in three different memory areas. One causes n-way bank conflicts in shared memory and the other is optimal for shared memory. We can see that while the patterns are bad for either global or both global and shared memory, texture memory performs almost equally well with either access pattern.

7.3 GPU Path Tracer Implementation

The GPU path tracer implementation essentially follows van Antwerpen's streaming design [Antwerpen 2011]. A large batch of 2^{20} samples is processed in parallel using one thread per sample. Sample paths are iteratively extended. Samples that have finished computation due to termination via Russian Roulette or because their paths have escaped the scene are removed from the stream in each iteration and the remaining samples are compacted. To keep the amount of work constant, each finished sample is regenerated by appending a new sample to the compacted stream. As previously observed [Antwerpen 2011], appending regenerated samples exploits primary ray or even higher order ray coherence due to specular reflection or refraction and improves SIMD efficiency. To include such effects on performance in our analysis we use the Ashikhmin-Shirley BRDF [Ashikhmin and Shirley 2000] to model scenes with materials ranging from diffuse to highly specular as well as refractive rough surfaces [Walter et al. 2007]. For ray traversal, we use the fast speculative-while-while traversal kernel design [Aila and Laine 2009]. For the underlying ray traversal acceleration structure we use the established high quality split-BVH [Stich et al. 2009]. The whole path tracer is implemented in four kernels (sample initialization/regeneration, ray tracing, path extension, connection validation). As tracing is done in a designated kernel, all statistics are only affected by ray traversal and not by other computations such as shading.

7.4 BVH Data Structures and Layouts

For our analysis we focus on binary bounding volume hierarchies with axis aligned bounding boxes and include several memory layouts for the node data and the tree itself.

7.4.1 Node Layouts

The classic BVH node data structure stores a bounding volume along with pointers to its children. We follow Aila and Laine [2009], i.e., a node does not store its bounding box, but the bounding boxes of its children. Both children are fetched and tested together, which is more efficient for GPUs due to increased instruction level parallelism. This also allows to implement rough front to back traversal as the nodes can be sorted by distance while testing for intersection. Depending on the data layout, the size of such a node is at least 56 bytes (2 float values for minimum/maximum per dimension and child plus pointers). We implemented one array-of-structures (AoS) layout and two structure-of-arrays (SoA) layouts:

- **AoS:** 64 bytes, including 8 bytes padding (fitting 2 nodes in one 128B cache line)
- **SoA32_24:** $32 + 24$ bytes, min/max x/y both children, min/max z both children and pointers, plus 8 bytes padding (fitting 4 nodes across 2 128B cache lines)
- **SoA16_8:** $3 \times 16 + 8$ bytes, min/max x/y child1, min/max x/y child2, min/max z both children, pointers (fitting 8 nodes across 4 128B cache lines)

We also analyzed an SoA8 layout which fitted 16 nodes in 7 cache lines. As it consistently performed much worse than the other layouts, we excluded it from our experiments.

7.4.2 Tree Layouts

A tree layout describes how nodes are grouped in memory. We analyzed six different tree layouts. The first four layouts are two common layouts and two cache-efficient layouts. We further propose two more layouts. The idea behind them is to compute a path traced image at a relatively low sample rate as a pre-process, recording the number of accesses for each BVH node. We then use the access statistics to guide the two layouting methods. In the following we describe the layouts in more detail. They are also illustrated in Figure 7.3. Layouts not using statistics are:

- **Depth-first-search (DFS):** Nodes are ordered as visited by a pre-order traversal. This layout performs best with coherent rays since a cache line is potentially filled with nodes on the path to the leaf.

- **Breadth-first-search (BFS):** Nodes are ordered as visited by a breadth-first traversal visiting the left child node first. This fits best for rays traversing neighboring branches.

- **van Emde Boas (vEB):** A cache-oblivious tree layout [Emde Boas 1975] described in Section 7.1.

- **COLBVH (COL):** A cache-oblivious tree layout mainly used for collision detection [Yoon and Manocha 2006] but also applicable to ray tracing. Beginning with all n nodes in a root cluster, the tree is recursively decomposed into clusters of $\lceil \sqrt{n+1} - 1 \rceil$ nodes. Nodes are merged into root clusters depending on their access probability computed from the ratio of the surface areas of its grand-parent and parent.

Next we describe our two proposed layouts depending on node access statistics collected in a pre-process. Both use a preset empirical threshold p:

- **Swapped subtrees (SWST):** Swap the sub-trees of a node in a depth-first layout if the fraction of left child accesses compared to all child accesses is below a fixed threshold $p \in [0, 0.5]$. Left children of the nodes form a path whose nodes are accessed the most and are spread over fewer cache lines.

- **Treelet based DFS/BFS (TDFS/TBFS):** A treelet is a connected sub-tree of a BVH. For this layout treelets of nodes that were accessed above the threshold p are built. This decomposes the BVH into treelets whose nodes are accessed the most. The treelet decomposition algorithm works with two queues: a merge queue and a deferred queue. The merge queue contains nodes which will be added to the current treelet and the other queue contains nodes which are deferred for creating additional treelets. Initially the current treelet and deferred queue are empty, and the merge queue contains the BVH root. Nodes are removed from the merge queue and added to the current treelet as long as the merge queue is not empty. When a node is removed its children are added to one of the queues. If the percentage of rays that continued to descend to a child node is larger than a fixed threshold $p \in [0\%, 100\%]$ the child is added to the merge queue, otherwise to the deferred queue. If the merge queue is empty, a new treelet is created by moving a node from the deferred queue to the merge queue and repeating the process. Once no more nodes are present in either queue the algorithm is done. The internal memory layout of a treelet can be chosen freely. By always adding nodes just to the front or the back of the merge queue we automatically obtain a treelet in DFS or BFS order. Finally the node order of the whole tree is obtained by lining up the nodes of all treelets. Thus treelets are only used as a means for grouping nodes and are not stored explicitly.

Note that there are other possible treelet construction algorithms such as the construction algorithm described by Aila and Karras [2010]. As mentioned previously, this approach is to our knowledge not supported by current hardware and therefore not included in our analysis.

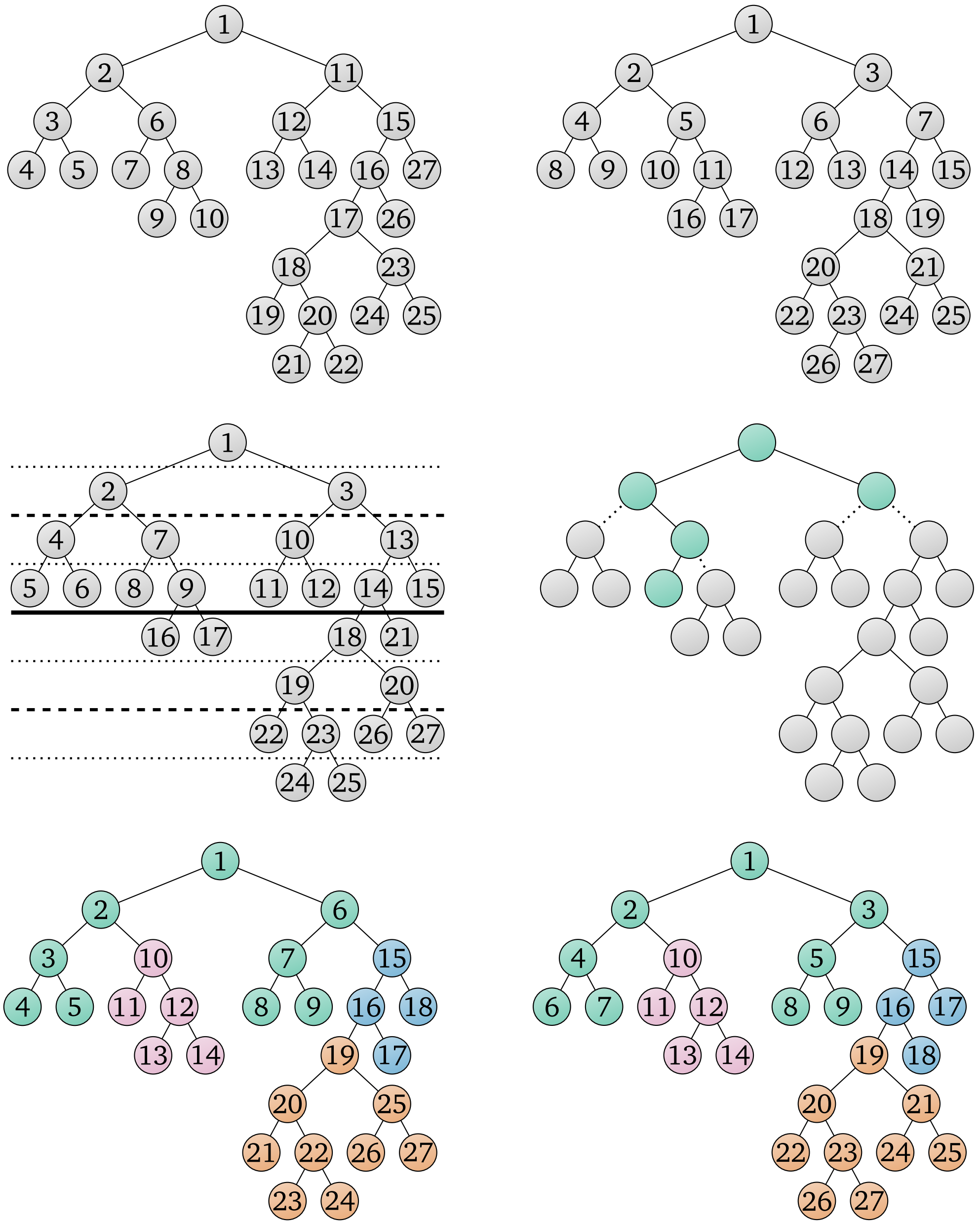


Figure 7.3: The different tree layouts described in Section 7.4.2 for a tree with $n = 27$ nodes. SWST is omitted as it is essentially the same as DFS but the left child is always the more likely one to be visited. *Top left:* DFS. *Top right:* BFS. *Center left:* van Emde Boas layout. The tree height is recursively split in half resulting in a number of subtrees in each step. *Center right:* COLBVH layout from Yoon and Manocha [2006]. The first cluster decomposition step is shown. Based on the number of tree nodes it creates a root cluster of size $s = \lceil \sqrt{n+1} - 1 \rceil = 5$ starting from the root node. The root cluster greedily collects nodes with highest access probability until it is full. The subtrees of root cluster leaf children form child clusters. This decomposes the tree into roughly $\sqrt{n} \approx 5$ clusters. The decomposition recursively proceeds with all clusters starting with the root cluster. The postorder root cluster first traversal gives the order of the nodes. *Bottom:* Our TDFS (*left*) and TBFS (*right*) treelet layouts, where nodes in treelets are either stored in DFS or BFS order.

			
Crytek Sponza	Kitchen	Hairball	San Miguel
262269 triangles 99127 nodes (6.1MB)	425504 triangles 150219 nodes (9.2MB)	2880012 triangles 1021548 nodes (62.4MB)	7880512 triangles 2723017 nodes (166.2MB)

Table 7.1: Scenes used for benchmarking. The number of scene triangles and BVH nodes including the memory footprint of the BVH is shown as well.

7.5 Evaluation

We evaluated the performance of the BVH and node layouts on four different scenes of varying complexity and materials (see Table 7.1). The *Kitchen* scene consists of a mixture of diffuse, glossy, and translucent materials. *Hairball* uses a nearly specular refractive material. *Sponza* and *San Miguel* use diffuse materials throughout the scenes. Several different metrics are computed using the event counters from CUPTI. Some of them can be found in the CUPTI User's Guide, others were deduced from values in Nsight Visual Studio Edition and reconstructed with events from CUPTI. A short explanation of each metric follows (see Schulz et al. [2013] for more details):

- **Runtime**, trace kernel runtime in milliseconds, measured using CUPTI's activity API.
- **GM/L1 ← L2 cache load hit rate**, percentage of global memory (GM) loads that hit in L2 cache. It is slightly diluted as the event counters include both global and local memory loads. (Local memory makes up ∼10% of the sum of global and local memory traffic.)
- **Tex cache hit rate**, percentage of texture memory loads that hit the texture memory cache.
- **Instruction replay overhead**, percentage of instructions that were issued due to replaying memory accesses, such as cache misses (lower is better).
- **SIMD efficiency (warp execution efficiency)**, percentage of average active to total number of threads per warp.
- **Branch efficiency**, ratio of non-divergent branches to all branches (SIMD divergence).
- **Load efficiency**, the ratio of requested memory load throughput to actual memory load throughput. That is, the ratio of the ideal number of memory transactions for a warp-wide memory load to the average measured number of transactions per load (see Section 3.4).

7.5.1 Baseline Performance Analysis

The baseline BVH is laid out in DFS order and stores nodes in AoS format. The AoS node format was chosen because Aila and Laine [2009] are using it in their GPU ray traversal routines which are amongst the fastest. Tree nodes are accessed via global memory and geometry via texture memory. Furthermore, Aila et al. [2012] proposed to store the BVH in texture memory but did not state expected speed-ups. We evaluated the gain of storing BVHs in texture memory and included the results in the layout evaluation.

Figure 7.4 and Figure 7.5 give an overview of the baseline performance for each scene and render loop iteration. We can see the effect of incoherent rays immediately after the first iteration. Runtime increases fivefold from 2.85 ms to 14.36 ms for the *Kitchen* scene. The number of primary rays per batch drops to 20%. Despite the huge number of incoherent rays the branch efficiency only decreases slightly to 85%. This indicates that threads in a warp mostly agree on their execution path. The cache hit rate does not suffer noticeably. The amount of data transferred between the different caches in the memory hierarchy shows that most data requests are serviced by caches. We notice a significantly lower load and SIMD efficiency after the first iteration, which later on increases with the number of primary rays per batch. The achieved occupancy stays relatively close to the theoretical maximum of 75% which means that work is well spread over the GPU's multiprocessors. For the *Hairball* scene we made a similar observation. Runtime increases from 2.49 ms up to 75.29 ms after a significant number of rays did not hit the environment map but the geometry. Especially the load efficiency as well as the SIMD efficiency collapses due to a combination of the incoherent memory access pattern and the depth complexity of the BVH tree. Observations for *Sponza* and *San Miguel* are analogous.

7.5.2 BVH and Node Layouts

Table 7.2 shows a ranking of all BVH and node layout combinations which were accessed via global memory or texture memory. The ranking is performed w.r.t. the average achieved speedup compared to the DFS layout in the respective memory area. The SWST, TDFS and TBFS layouts require a threshold probability p. We have tested a number of different values to find the best performing one. The best threshold is required to perform well for all scenes in our data set so that its performance extends to unknown data sets. We use the sum of the scene runtimes to measure the performance of a threshold and choose the best performing ones. The determined thresholds are stated in the table caption. Table 7.3 additionally shows per scene rankings. Following, we will compare the best performing combinations of threshold, BVH and node layout in each memory area to the other introduced BVH layouts.

Global Memory Overall, the node layout has the biggest impact on runtime. The AoS layout performs best followed by SoA32_24 and SoA16_8, except for the *Kitchen* scene where it is roughly the other way around. Runtime of AoS-based layouts is 10% – 35% lower than SoA16_8 based layouts. For AoS performance differences between tree layouts are only up to 2%. Only in the *San Miguel* scene the treelet DFS layout manages to achieve 6%. Our baseline already performs quite well. For the other node layouts tree layouts improve performance up to 9%, i.e., without access statistics the simple BFS layout performs best for all node layouts, followed by the more complex vEB layout, whereas DFS on average performs worst for all node layouts. Interestingly the AoS SWST layout, which is

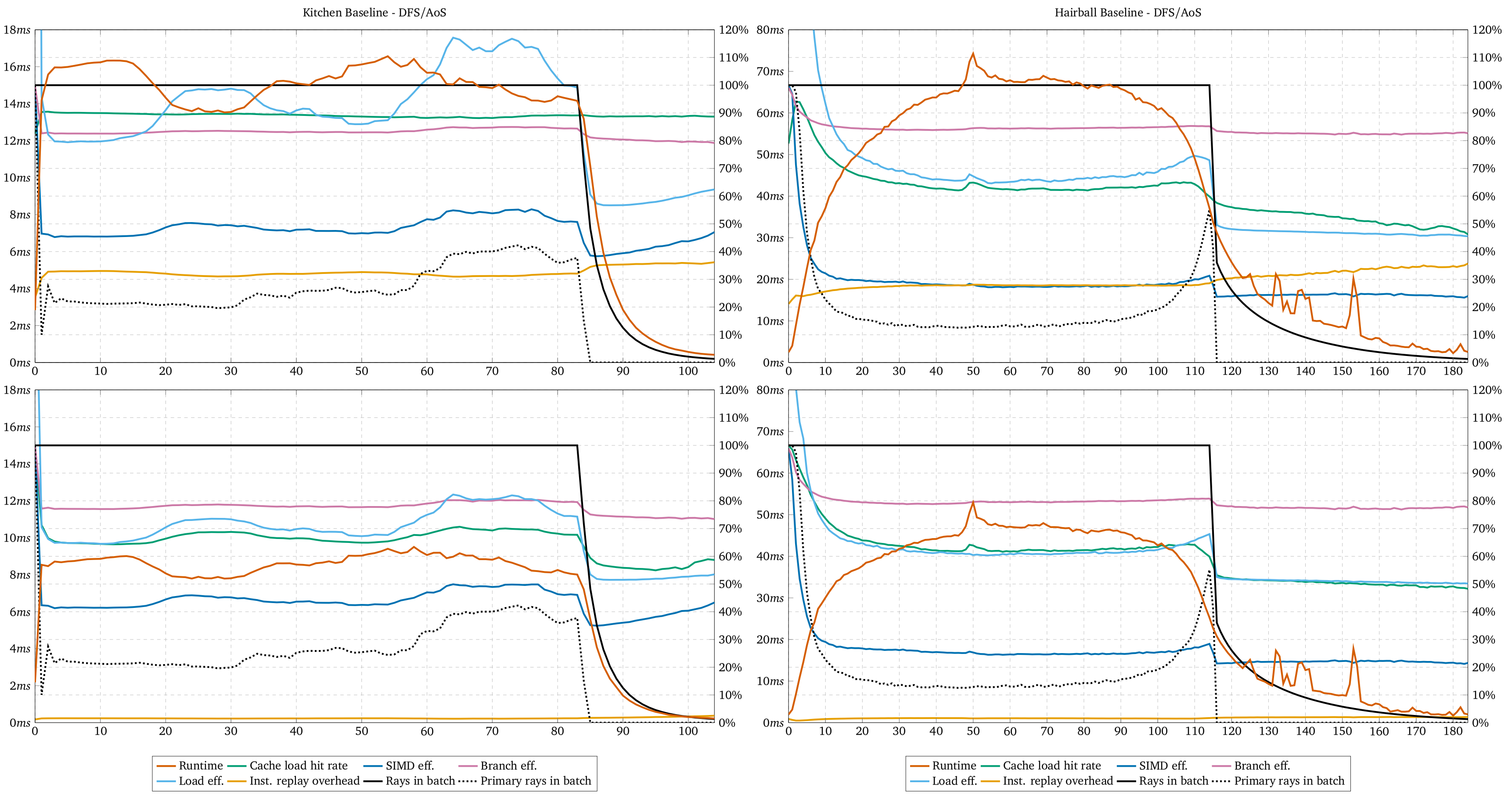


Figure 7.4: Trace kernel profiling graphs of the baseline for the *Kitchen* and *Hairball* scenes using a Geforce GTX 680. Resolution is 1024x768 with 32spp. Nodes have AoS format and are stored in DFS order. The figure shows the runtime behavior (*left y axis*) and GPU metrics (*right y axis*) over all rendering loop iterations (*x axis*). BVH nodes are either stored in global memory (*top*) or texture memory (*bottom*).

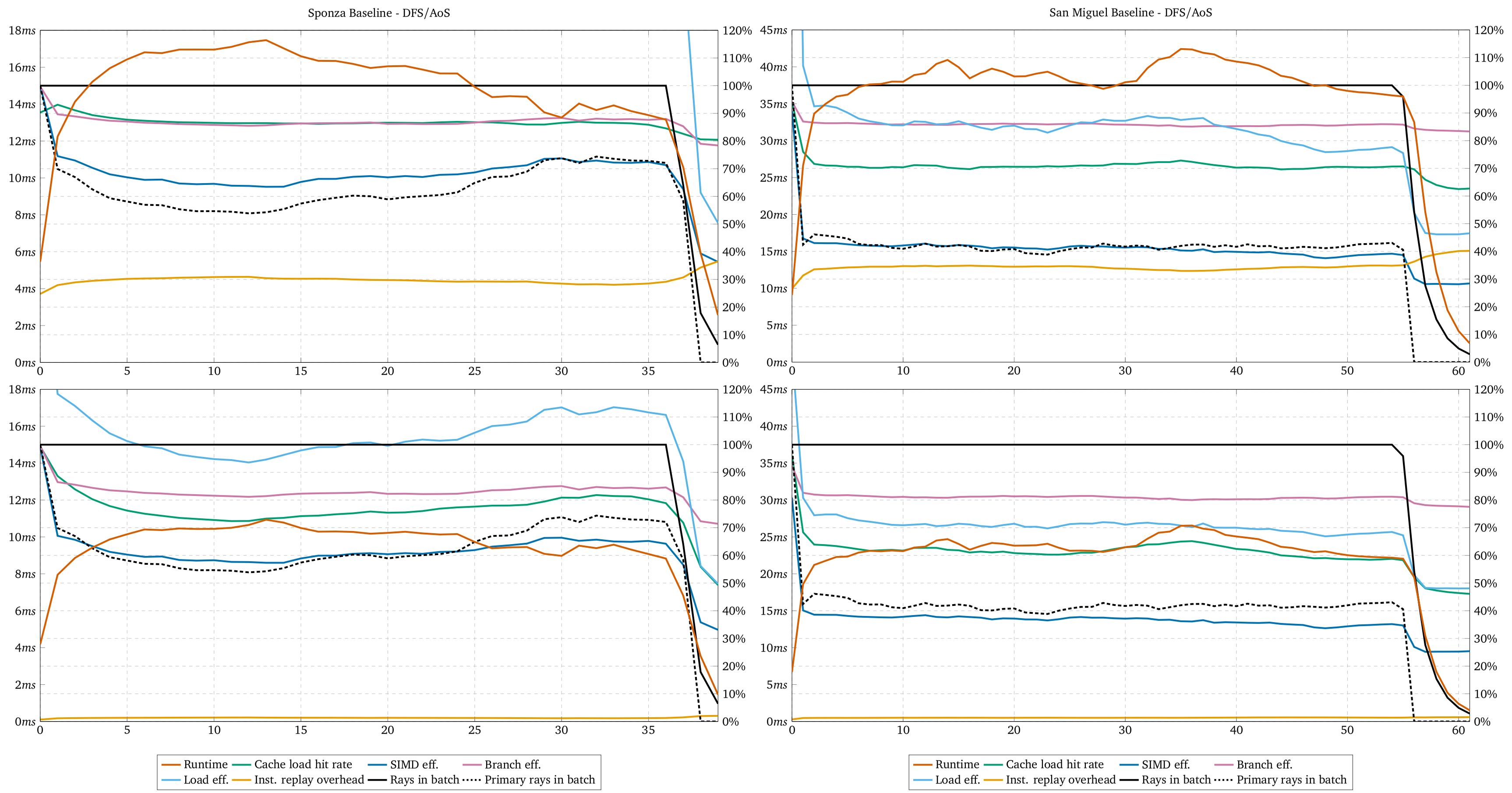


Figure 7.5: Trace kernel profiling graphs of the baseline for the *Crytek Sponza* and *San Miguel* scenes using a Geforce GTX 680. Resolution is 1024x768 with 32spp. Nodes have AoS format and are stored in DFS order. The figure shows the runtime behavior (*left y axis*) and GPU metrics (*right y axis*) over all rendering loop iterations (*x axis*). BVH nodes are either stored in global memory (*top*) or texture memory (*bottom*).

layout		Crytek Sponza			Kitchen			Hairball			San Miguel		
BVH	node	R	H	SZ	R	H	SZ	R	H	SZ	R	H	SZ
GMem													
TDFS	AoS	581.7	86.7	94.3	1386.2	89.1	218.6	7504.9	60.5	948.8	2071.4	71.4	287.4
BFS	AoS	583.3	86.5	94.3	1364.6	89.0	218.6	7462.2	60.4	948.8	2131.1	70.7	287.6
TBFS	AoS	582.6	86.6	94.3	1371.8	89.1	218.6	7469.5	60.4	948.8	2142.8	70.8	287.7
vEB	AoS	582.5	86.6	94.3	1374.6	89.0	218.6	7469.4	60.6	948.6	2165.4	70.7	287.7
COL	AoS	582.5	86.7	94.3	1385.5	89.1	218.6	7539.5	60.6	949.1	2166.9	70.9	287.7
DFS	AoS	583.5	86.6	94.3	1394.9	89.0	218.6	7576.0	60.7	949.1	2205.3	70.6	287.9
SWST	AoS	582.2	86.8	94.3	1391.9	89.1	218.6	7638.8	60.8	949.3	2267.9	71.0	288.0
BFS	SoA32_24	581.0	85.1	94.1	1310.1	88.4	217.8	9099.2	55.3	950.3	2683.3	64.4	287.4
vEB	SoA32_24	583.9	85.3	94.2	1355.1	88.7	217.9	9059.4	55.7	950.1	2824.0	64.6	287.6
COL	SoA32_24	585.1	85.4	94.2	1335.2	88.8	217.9	9269.1	55.4	950.7	2859.4	64.9	287.7
DFS	SoA32_24	595.3	84.6	94.2	1357.9	88.3	217.9	9320.0	54.8	950.7	2932.9	63.3	288.2
BFS	SoA16_8	637.4	77.2	93.4	1346.1	81.7	213.4	10747.6	38.1	942.2	3270.6	48.3	285.3
vEB	SoA16_8	641.3	77.9	93.3	1364.4	83.4	215.3	10555.1	40.0	942.8	3376.6	48.6	286.1
COL	SoA16_8	651.3	77.8	93.2	1371.4	84.0	215.3	10780.1	39.6	943.4	3437.5	49.1	286.3
DFS	SoA16_8	680.1	75.8	93.6	1436.5	82.0	217.1	10969.3	38.3	943.0	3667.2	45.4	287.1
TMem													
TDFS	AoS	372.4	76.5	136.3	812.6	65.6	268.1	5369.4	59.8	1053.8	1300.4	61.0	337.3
BFS	AoS	373.1	76.4	136.3	807.7	65.6	268.1	5356.7	59.9	1053.8	1315.0	61.2	337.3
TBFS	AoS	373.1	76.5	136.3	810.2	65.7	268.1	5359.3	59.9	1053.8	1315.0	61.2	337.3
vEB	AoS	373.0	76.5	136.3	806.7	65.5	268.1	5357.3	59.8	1053.8	1326.4	61.1	337.3
COL	AoS	373.4	76.5	136.3	804.2	65.6	268.1	5386.1	59.8	1053.8	1334.8	61.0	337.3
SWST	AoS	373.8	76.5	136.3	805.1	65.5	268.1	5394.3	59.9	1053.8	1353.1	60.9	337.2
DFS	AoS	374.2	76.4	136.3	806.2	65.4	268.1	5394.9	59.8	1053.8	1356.4	60.9	337.3
BFS	SoA32_24	412.9	73.0	136.4	845.6	61.5	268.3	6868.8	56.6	1056.5	1877.0	56.4	337.3
vEB	SoA32_24	417.0	73.0	136.4	837.4	61.1	268.3	6839.6	56.4	1056.6	1955.8	56.2	337.4
COL	SoA32_24	417.1	73.0	136.4	852.9	61.2	268.3	6956.5	56.3	1056.5	1978.8	56.2	337.4
DFS	SoA32_24	423.4	72.7	136.4	852.7	60.7	268.3	6971.4	56.3	1056.5	2023.4	55.9	337.3
BFS	SoA16_8	497.0	60.2	135.7	988.0	43.9	264.8	9570.8	36.7	1048.7	2837.8	34.7	335.0
vEB	SoA16_8	495.3	61.1	135.6	981.6	43.9	266.5	9261.5	37.8	1048.9	2932.9	35.2	335.3
COL	SoA16_8	506.1	61.2	135.6	973.1	44.2	266.6	9515.5	37.7	1048.6	2999.3	35.3	335.4
DFS	SoA16_8	535.7	60.5	135.7	1042.8	42.9	267.6	9663.3	38.4	1049.5	3229.9	35.1	335.9

Table 7.2: Ranking of BVH and node layout combinations w.r.t. average speedup, where nodes are either stored in global memory (GMem) or texture memory (TMem). Runtime (R) in milliseconds, cache hit rate (H) in percent and transferred data size (SZ) in gigabytes are shown. The thresholds of TDFS, TBFS, and SWST in global memory are 0.6, 0.3, and 0.5, respectively. In texture memory the thresholds are 0.6, 0.2, and 0.4, respectively.

basically a DFS layout, on average performed slightly worse than DFS. The better performing tree layouts have a number of effects. On average, slightly less data is transferred by global load instructions, the average number of transactions per global load request is decreased and instruction replay overhead dropped minimally (see Schulz et al. [2013] for details). This is reflected in a higher IPC count because fewer instructions have to be issued due to memory replays. We can see the impact that the SoA32_24 node layout has on the caches in a lower L2 global load hit rate. A L2 cache miss is more expensive and displaces twice as many nodes than when using the AoS node layout. The situation is similar but exacerbated for the SoA16_8 layouts which all exhibit worse performance due to their even higher miss penalty which results in very low cache hit rates.

Texture Memory Again we can see that the node layout has the biggest performance impact with the AoS layout performing best followed by SoA32_24 and SoA16_8. Performance differences of the AoS and SoA16_8 based layouts range from 17% – 50%. The best performing combination is the TDFS BVH layout with a threshold of 0.6 using the

Crytek Sponza			Kitchen			Hairball			San Miguel		
BVH	node	R	BVH	node	R	BVH	node	R	BVH	node	R
GMem											
BFS	SoA32_24	581.0	BFS	SoA32_24	1310.1	BFS	AoS	7462.2	TDFS	AoS	2071.4
TDFS	AoS	581.7	COL	SoA32_24	1335.2	vEB	AoS	7469.4	BFS	AoS	2131.1
SWST	AoS	582.2	BFS	SoA16_8	1346.1	TBFS	AoS	7469.5	TBFS	AoS	2142.8
vEB	AoS	582.5	vEB	SoA32_24	1355.1	TDFS	AoS	7504.9	vEB	AoS	2165.4
COL	AoS	582.5	DFS	SoA32_24	1357.9	COL	AoS	7539.5	COL	AoS	2166.9
TBFS	AoS	582.6	vEB	SoA16_8	1364.4	DFS	AoS	7576.0	DFS	AoS	2205.3
BFS	AoS	583.3	BFS	AoS	1364.6	SWST	AoS	7638.8	SWST	AoS	2267.9
DFS	AoS	583.5	COL	SoA16_8	1371.4	vEB	SoA32_24	9059.4	BFS	SoA32_24	2683.3
vEB	SoA32_24	583.9	TBFS	AoS	1371.8	BFS	SoA32_24	9099.2	vEB	SoA32_24	2824.0
COL	SoA32_24	585.1	vEB	AoS	1374.6	COL	SoA32_24	9269.1	COL	SoA32_24	2859.4
DFS	SoA32_24	595.3	COL	AoS	1385.5	DFS	SoA32_24	9320.0	DFS	SoA32_24	2932.9
BFS	SoA16_8	637.4	TDFS	AoS	1386.2	vEB	SoA16_8	10555.1	BFS	SoA16_8	3270.6
vEB	SoA16_8	641.3	SWST	AoS	1391.9	BFS	SoA16_8	10747.6	vEB	SoA16_8	3376.6
COL	SoA16_8	651.3	DFS	AoS	1394.9	COL	SoA16_8	10780.1	COL	SoA16_8	3437.5
DFS	SoA16_8	680.1	DFS	SoA16_8	1436.5	DFS	SoA16_8	10969.3	DFS	SoA16_8	3667.2
TMem											
TDFS	AoS	372.4	COL	AoS	804.2	BFS	AoS	5356.7	TDFS	AoS	1300.4
vEB	AoS	373.0	SWST	AoS	805.1	vEB	AoS	5357.3	TBFS	AoS	1315.0
TBFS	AoS	373.1	DFS	AoS	806.2	TBFS	AoS	5359.3	BFS	AoS	1315.0
BFS	AoS	373.1	vEB	AoS	806.7	TDFS	AoS	5369.4	vEB	AoS	1326.4
COL	AoS	373.4	BFS	AoS	807.7	COL	AoS	5386.1	COL	AoS	1334.8
SWST	AoS	373.8	TBFS	AoS	810.2	SWST	AoS	5394.3	SWST	AoS	1353.1
DFS	AoS	374.2	TDFS	AoS	812.6	DFS	AoS	5394.9	DFS	AoS	1356.4
BFS	SoA32_24	412.9	vEB	SoA32_24	837.4	vEB	SoA32_24	6839.6	BFS	SoA32_24	1877.0
vEB	SoA32_24	417.0	BFS	SoA32_24	845.6	BFS	SoA32_24	6868.8	vEB	SoA32_24	1955.8
COL	SoA32_24	417.1	DFS	SoA32_24	852.7	COL	SoA32_24	6956.5	COL	SoA32_24	1978.8
DFS	SoA32_24	423.4	COL	SoA32_24	852.9	DFS	SoA32_24	6971.4	DFS	SoA32_24	2023.4
vEB	SoA16_8	495.3	COL	SoA16_8	973.1	vEB	SoA16_8	9261.5	BFS	SoA16_8	2837.8
BFS	SoA16_8	497.0	vEB	SoA16_8	981.6	COL	SoA16_8	9515.5	vEB	SoA16_8	2932.9
COL	SoA16_8	506.1	BFS	SoA16_8	988.0	BFS	SoA16_8	9570.8	COL	SoA16_8	2999.3
DFS	SoA16_8	535.7	DFS	SoA16_8	1042.8	DFS	SoA16_8	9663.3	DFS	SoA16_8	3229.9

Table 7.3: Per-scene ranking of BVH and node layout combinations w.r.t. runtime, where nodes are either stored in global memory (GMem) or texture memory (TMem). Runtime (R) in milliseconds is shown as well. The thresholds of TDFS, TBFS, and SWST in global memory are 0.6, 0.3, and 0.5, respectively. In texture memory the thresholds are 0.6, 0.2, and 0.4, respectively.

AoS node layout. It is only marginally faster than the baseline layout combination. Interestingly, there is an increase in the amount of data transferred across all scenes and layout combinations though the transaction size of both the texture cache and the global memory L2 cache is 32B. The only explanation we can provide for a lower traffic size of global memory is superior broadcasting compared to texture memory.

Memory Area Comparison If we compare the runtime of the best layout combinations in texture and global memory, we can observe that using texture memory is approximately 30 – 40% faster. Even some of the slowest layout combinations in texture memory are faster than the best layout combinations in global memory. Our baseline layout with nodes in texture memory also outperforms all layout combinations in global memory by 25% – 38%. The reason why texture memory performs better is not entirely clear. As we have seen in Section 7.2, the average access latency of texture memory is consistently lower than for global memory and for access patterns which cause very high latency in global and shared memory, the texture memory's latency increases only by a comparably small amount. We presume that this property of the texture memory cache, in conjunction with quite possibly other unknown hardware details, let it deal very well with incoherent memory accesses such that it is on average faster than the L2 global memory cache. Contrary to Aila et al. [2012] our path tracer benefited from using texture memory for loading nodes when run on a Fermi GPU (see Schulz et al. [2013]).

Comparison of Best Layout Combination Figure 7.6 and Figure 7.7 show profiling graphs for the baseline layout combined with profiling graphs for the on average best performing TDFS 0.6/AoS layout. For both layouts we included the memory bandwidth of the L2 cache for node access via global memory and memory bandwidth of the texture cache for access via texture memory. Overall we can see that independent of the memory area used for node access all metrics, except for runtime and bandwidth, are essentially identical for both layouts. Lower relative kernel runtimes of TDFS 0.6/AoS are directly related to a higher relative memory bandwidth. This is most pronounced in *San Miguel*, *Hairball*, and *Kitchen*. The relative bandwidth increase for TDFS 0.6/AoS is slightly higher for node access via global memory. It is unclear where this bandwidth improvement comes from, as cache hit rate and load efficiency are the same for both layouts. Only *San Miguel* also clearly shows a slightly increased L2 cache hit rate.

7.6 Conclusion

We have presented a number of different BVH layout schemes and analyzed their performance on tracing ”real-world” distributions of incoherent rays. Two schemes make use of information gathered in a pre-processing pass over the BVH. Our TDFS layout had the best average speedup in global and texture memory. In global memory we have achieved a runtime reduction by 1% – 6%. We gained a 30% – 40% runtime reduction compared to the baseline in global memory when the BVH is stored in texture memory similar to Aila et al. [2012]. But also accessing the baseline in texture memory, an improvement of only 0.5% – 4.0% was observable for the TDFS layout. The common DFS layout performed worst for all node layouts in both memory areas. Excluding layouts using statistics the equally simple BFS layout on average performed best and similar to the TDFS layout.

There are several possibilities for future work. We have not included the performance impact on tracing coherent rays with our analyzed layouts. As optimizing the BVH layout and ray grouping and reordering techniques are orthogonal one might investigate if there is any synergy when using both techniques together. Especially BVH layouts based on node access statistics should benefit from these techniques. Knowing that a node is more likely to be visited can unfold its full potential when the relevant rays are processed together. Generally increasing SIMD efficiency will result in more memory accesses being issued at a time which may be beneficial for our node reordering techniques. The dynamic fetch kernel from Aila et al. [2012] which replaces terminated rays when SIMD efficiency drops below a certain percentage could be used to investigate this. The GK110 [NVIDIA 2016b] features a 48 KB read-only data cache, which is the same as the texture memory cache. It would be interesting to see the effects of a much larger texture cache. As kd-tree nodes are much smaller than BVH nodes more nodes fit into cache lines. Though this should increase cache hits, more nodes get evicted on cache misses. It would be interesting to see which effect outweighs the other and what performance gains can be achieved with different tree layouts.

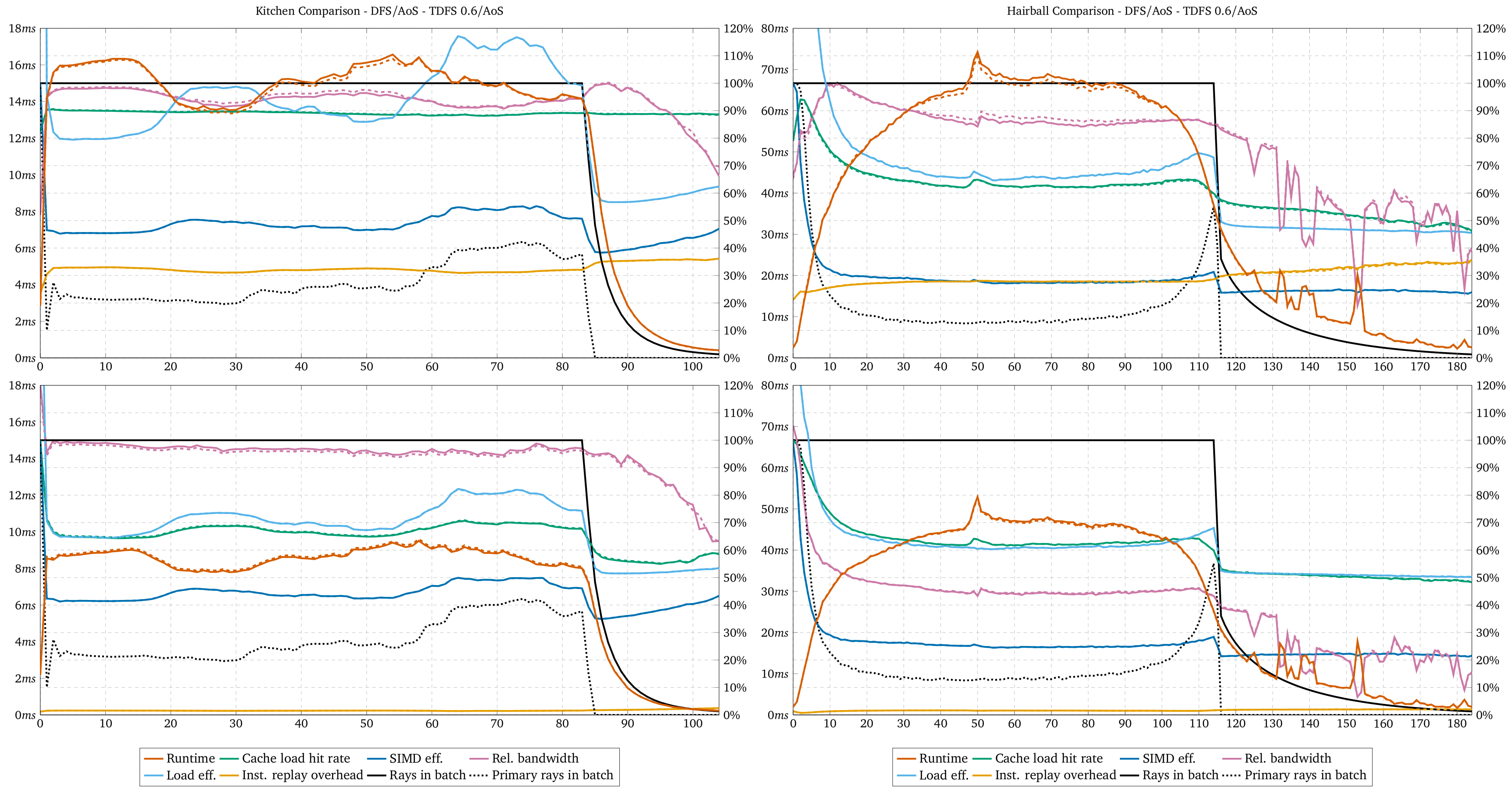


Figure 7.6: Comparison of the trace kernel profiling graphs using the baseline layout (*solid*) and the best performing TDFS 0.6/AoS layout (*dashed*) for the *Kitchen* and *Hairball* scenes. Runtime behavior (*left y axis*) and GPU metrics (*right y axis*) over all rendering loop iterations (*x axis*) are shown. BVH nodes are either stored in global memory (*top*) or texture memory (*bottom*). Branch efficiency is replaced with cache bandwidth, which is normalized w.r.t. the highest bandwidth achieved in an iteration containing incoherent rays.

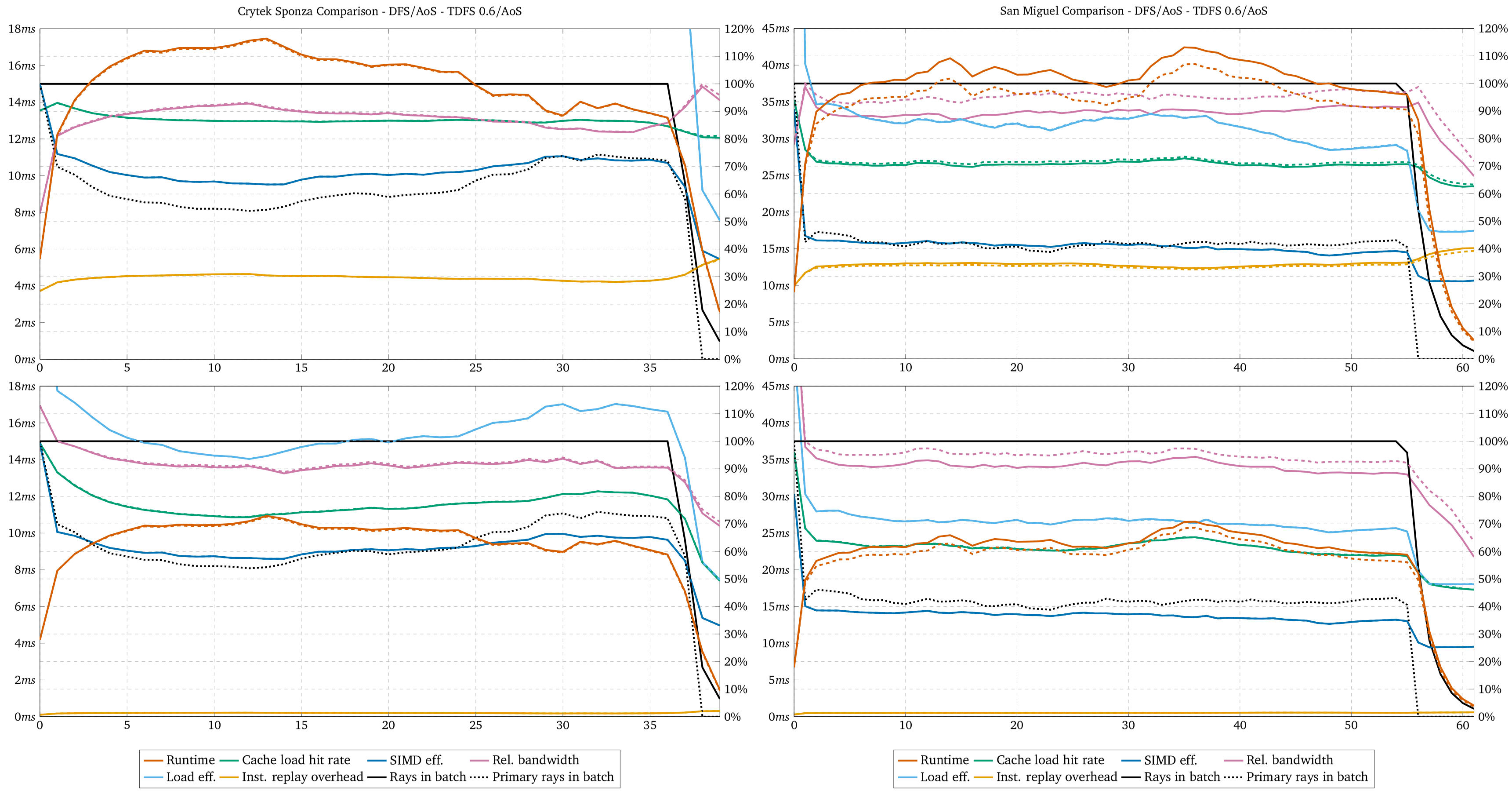


Figure 7.7: Comparison of the trace kernel profiling graphs using the baseline layouts (*solid*) and the best performing TDFS 0.6/AoS layout (*dashed*) for the *Sponza* and *San Miguel* scenes. Runtime behavior (*left y axis*) and GPU metrics (*right y axis*) over all rendering loop iterations (*x axis*) are shown. BVH nodes are either stored in global memory (*top*) or texture memory (*bottom*). Branch efficiency is replaced with cache bandwidth, which is normalized w.r.t. the highest bandwidth achieved in an iteration containing incoherent rays.

Chapter 8

Multi-GPU Out-of-Core Top-Down SAH-based kd-Tree and BVH Construction

Contents

After we analyzed the GPU cache behavior of different BVH tree and node memory layouts to improve BVH traversal performance in the previous chapter, we finally shift our focus on fast construction of high quality BVHs and kd-trees. A tremendous amount of previous work focuses on the efficient construction of ray tracing acceleration structures in a CPU and GPU context. Construction times are on the order of milliseconds for scenes with an appropriate number of primitives. But a common requirement of these approaches is that geometry and the acceleration structure are small enough so that they can completely stay in-core. This renders these approaches unsuitable for applications with a huge amount of geometry, e.g. highly detailed environments in production rendering. Especially GPU-based construction approaches suffer from this issue, as GPU memory typically is around one order of magnitude smaller than system memory. Though out-of-core CPU and GPU rendering has been investigated, there has been less attention on efficient out-of-core acceleration structure construction. Acceleration structures are typically assumed as given. Exceptions are PantaRay [Pantaleoni et al. 2010] and work by Wang et al. [2013], which perform out-of-core SAH-based BVH construction in a hybrid bottom-up top-down approach on CPU and GPU. Both approaches, however, avoid top-down SAH construction in the upper levels. This is understandable as it requires several expensive iterations over the whole geometry. Instead an initial regular clustering is performed in just a few iterations without SAH guidance. The collection of clusters itself is reorganized in a top-level SAH-based BVH, but split decisions are already suboptimal by the pre-clustering.

To the best of our knowledge complete SAH-based out-of-core BVH construction has not been done with GPUs. We argue that this is, however, worth the effort as the higher quality will result in less memory loads in the final application due to better separation of geometry. This is in fact even more important for out-of-core rendering. Though typically larger than BVHs, kd-trees can perform better than the former. But out-of-core kd-tree construction has not been addressed on both, CPUs and GPUs. The main reason for this is their unbounded memory footprint, which is problematic in an out-of-core context.

We show that full SAH-based top-down out-of-core BVH and even kd-tree construction can be done in a way that efficiently exploits the massive parallelism of GPUs. Apart from handling of the huge data itself, the significant overhead introduced by memory copy operations between host and GPU is the biggest challenge that has to be faced. Memory copies drastically reduce the computational throughput and introduce additional synchronization requirements between the two sides. It becomes an even more serious bottleneck when the system is extended to multiple GPUs.

Our contributions are as follows:

- an efficient out-of-core multi-GPU algorithm for BVH and kd-tree construction that allows the memory footprint of the output tree as well as the geometry to exceed graphics memory,
- a construction that applies SAH right from the beginning and does not rely on quality degrading pre-clustering of geometry, and
- an SAH improvement threshold that allows to trade rendering performance for a reduced acceleration structure memory footprint and construction time in a controllable way.

8.1 Related Work

A high quality acceleration structure is essential to achieve good ray tracing performance. As described in Section 2.5 empirically BVHs and kd-trees perform best when constructed using SAH originally introduced by Goldsmith and Salmon [1987] and MacDonald and Booth [1989,1990]. Construction of SAH-based kd-trees and BVHs for ray tracing is a challenging and time consuming task. State-of-the-art algorithms construct trees top-down and try to locally optimize SAH on a per node basis. From Section 2.5.3 we know this construction computes the cost c_{split} of splitting a node n containing $|n|$ triangles into two hypothetical left and right leaf nodes l and r as:

$$c_{split} = c_t + p_l|l|c_i + p_r|r|c_i. \tag{8.1}$$

p_l and p_r are the geometric probabilities of intersecting the tentative left and right node depending on the surface area of their bounds. $|l|$ and $|r|$ are the number of triangles in the left and right leaf. In case of ordinary BVHs $|l| + |r| = |n|$ holds, while in case of kd-trees the sum can be bigger than $|n|$ due to triangle splitting. c_t and c_i are the ray traversal implementation dependent constants. If c_{split} is smaller than the cost $c_{leaf} = c_i \cdot |n|$ for creating a leaf node for n the node is split. Our choices for the traversal constants can be found in the evaluation section (Section 8.5). Wald and Havran [2006] and Wald et al. [2007] introduced $O(n\log(n))$ CPU algorithms for SAH based kd-tree and BVH construction. We give an overview over the most relevant related work on parallel GPU approaches for these algorithms and out-of-core acceleration structure construction.

8.1.1 kd-Trees

Zhou et al. [2008] proposed the first GPU kd-tree construction algorithm, which achieved realtime build times for small scenes. It applies a hybrid construction strategy that uses cheap spatial median splits in the upper levels and expensive SAH splits as soon as a node contains at most 64 triangles. The poor splitting decisions made in most of the top part of the tree cannot be corrected or compensated in any way in the last 1 to 6 levels. As a result the negative effect of spatial median splitting on tree quality increases with scene size. Wu et al. [2011] proposed an efficient GPU implementation and Choi et al. [2010] proposed an efficient multi-core implementation of full SAH construction. Full SAH algorithms involve sorting the complete data several times for each dimension.

Popov et al. [2006], proposed binned kd-tree construction which does not require sorting. This approach uses a discrete amount of equidistant split planes to sample the SAH cost function at certain points. It allows for much faster implementations with negligible tree quality deterioration. Danilewski et al. [2010] presented an efficient single-GPU implementation of binned SAH kd-tree construction. All steps are implemented in five different variations/stages. Each stage is optimized for a distinct amount of geometry in a node and number of such nodes in a tree level. Only one stage is computed at a time. Thus, nodes which are classified for a different stage than the current one are scheduled for later processing. Scheduling details and overhead are not discussed, but the authors state their implementation is faster than the lower quality hybrid construction from Zhou et al. [2008].

8.1.2 BVHs

First efficient GPU algorithms for BVH construction were proposed by Lauterbach et al. [2009]. They presented three approaches with different trade-offs between tree quality and construction time. The fastest algorithm called linear BVH (LBVH) first assigns Morton codes to triangles. Then the triangles are sorted according to their codes using efficient parallel radix sort. The whole BVH can then be extracted from the sorted Morton codes by interpreting them as coordinates in an octree. This simple construction roughly corresponds to a spatial median split which results in poor tree quality, but is fast to compute. The second algorithm is a parallel approach for full binned-SAH BVH construction. Tree quality is high but construction is much slower, especially since the approach taken lacks sufficient parallelism in the upper levels. To strike a balance, they propose a third algorithm, that is a hybrid of the former two. The upper levels are constructed according to the highly parallel first algorithm while the remaining levels expose enough parallelism to be efficiently constructed according to the second one. As a result the output tree is of lower quality than full SAH as it suffers from the same problems as Zhou et al.'s approach. Exact SAH values are omitted but the authors report tracing times close to full SAH for the hybrid algorithm and up to 7 times higher for LBVH. Pantaleoni and Luebke [2010] and Garanzha et al. [2011] proposed much faster implementations for LBVH and the hybrid algorithm called HLBVH which allow realtime rebuilds for scenes with up to 2 million triangles. A key change in the hybrid algorithm is, that LBVH is used to build the lower levels of the tree first. The roots of the subtrees themselves are then used for binned top-down SAH BVH construction. Thus the expensive part of the algorithm is performed on much less input elements and tree quality is improved in the important upper levels. The authors state a tree quality which is about half way between LBVH and full SAH.

8.1.3 Out-of-Core construction

The discussed GPU-based BVH and kd-tree construction techniques require both static scene geometry and transient data to fit into GPU memory. There is only little work on out-of-core ray tracing acceleration structure construction, especially in the context of GPUs. Wald et al. [2001a] roughly outlined an hypothetical out-of-core algorithm for kd-tree (called BSP-tree in the paper) construction involving several compute nodes with CPUs. The exact construction strategy is, however, unstated.

Baert et al. [2013] proposed an out-of-core CPU algorithm for regular voxelization and bottom-up sparse voxel octree construction of extremely large triangle meshes. They manage to be roughly as fast as an unoptimized in-core solution for in-core datasets with just 1 GB of available memory by exploiting the relationship between Morton codes and octrees. Their concepts are not applicable to top-down SAH-based kd-tree or BVH construction.

Pantaleoni et al. [2010] proposed an out-of-core two-level BVH construction algorithm for complex scenes which runs entirely on the CPU. First, the scene geometry is divided into a regular 3D grid of buckets. Afterwards geometry buckets are merged or split into chunks with respect to a specified target chunk size. For each chunk of geometry a separate SAH based BVH is constructed. Then, the chunks themselves are organized into a single top-level SAH-based BVH. Finally the tree of a chunk is decomposed into a set of smaller treelets (bricks) and stored on disk. Wang et al. [2013] presented a combined preprocessing and rendering approach for many lights rendering of out-of-core scenes that uses GPUs in all steps. Acceleration structure construction is very similar to [Pantaleoni et al. 2010]. The main difference is how initial chunks are determined. Each primitive is associated with a Morton code. Then the list of primitives is sorted with respect to their codes and partitioned into chunks of specified target size. The resulting spatial clustering should be at least similar to [Pantaleoni et al. 2010]. Instead of an ordinary SAH-based BVH a higher quality SBVH [Stich et al. 2009] is constructed for each chunk. SBVHs allow to also adaptively apply spatial splits during construction if beneficial. No efficient SBVH GPU implementation has been presented to date. Thus it is unfortunate, that the authors have omitted any details of their implementation. Again chunks themselves are organized in a single top-level BVH. Both methods can be described as a bottom-up top-down approach. This is not possible with kd-trees as placement of the root splitting plane is a global decision, that affects all following steps due to triangle splits. Further both methods lead to reduced acceleration structure quality as the applied initial chunking enforces a spatial median like distribution of triangles.

Finally, Hou et al. [2011] presented semi out-of-core extensions to the GPU kd-tree construction approach from Zhou et al. [2008] and the hybrid BVH construction approach from Lauterbach et al. [2009]. Hereby only the size of the tree is allowed to exceed graphics memory. Scene geometry has to completely fit into memory. As an extension to the two algorithms, their approach also inherits the inferior quality in the upper tree levels these algorithms suffer from. They propose a *partial*-BFS order that processes as many nodes in parallel in a BFS manner as memory allows. In every iteration as many descendants of the previous batch as fit into memory are processed. Thus if not all nodes of a tree level fit into memory the algorithm gradually transform into a DFS-BFS traversal. The algorithm degenerates to full BFS processing for small scenes. Special care is taken to reduce the amount of memory resulting from triangle duplicates in kd-tree construction. They propose the commonly used technique already introduced by Havran and Bittner [2002] to store a *primitive reference* for every triangle that consists of its bounding box and a reference to it. When a triangle is split only its primitive reference is split into two new references with tightly fitted bounding boxes. No geometry has to be duplicated. Additionally, they store a 32-bit code with each reference that allows to efficiently reconstruct the up to 9 vertices of the polygon resulting from clipping the referenced triangle against the box. It is unclear what the authors need the polygon vertices for as the original triangle together with the tight bounding box of the clipped triangle is already sufficient. Thus their format is actually inferior in terms of memory-footprint reduction compared to state-of-the-art.

There is no work on full top-down SAH-based out-of-core GPU BVH construction. Work on out-of-core kd-tree construction with any strategy is not available neither for GPU nor CPU.

8.2 Motivation and Assumptions

In the previous section we could see that previous work on out-of-core construction traded tree quality for simplicity and construction speed. Full top-down SAH-based constructed trees have their benefits especially in an out-of-core context. Applying SAH right from the beginning of construction leads to better separation of geometry and reduces the amount of geometry and nodes that have to be loaded. As discussed in Section 2.5.7, Aila et al. [2013] noted that in general greedy top-down SAH-based construction algorithms produce trees that are faster to trace than predicted by their SAM cost. The trees also can perform better than trees constructed with other construction strategies that produce lower SAM cost. The authors introduced the end-point-overlap (EPO) metric, which in combination with SAM can better predict performance of scalar (and almost of SIMD) ray tracing. EPO describes the expected extra traversal cost due to node overlap. While we observed in Chapter 5 and Chapter 6 that in some cases the combined SAM-EPO predictor was not sufficient, our results mostly supported its validity. These observations allow to conclude that SAM alone much more accurately predicts trace performance of kd-trees than of BVHs, as kd-trees have no node overlap and thus zero EPO cost. On the other hand, bottom-up kd-tree construction is not possible and there is no alternative to top-down SAH-based construction when quality is desired.

Assumptions Our approach builds on several assumptions. First of all it is only out-of-core w.r.t. GPU memory, i.e., the memory footprint of geometry, any transient data, as well as the final tree are allowed to exceed available graphics memory. We have a host-side memory cache that is used for swapping data out of and into the GPUs. The host cache itself has no size limit, i.e., no explicit swapping to hard disk occurs. If it gets full nonetheless, the operating system is assumed to take care of swapping. There is no limit on the allowed number of GPUs.

8.3 Construction

Precise sweep SAH-based construction in the sense of Wald et al. for kd-trees [Wald and Havran 2006] and BVHs [Wald et al. 2007] relies on sorting input geometry several times. In an out-of-core multi-GPU computing context this is prohibitively expensive and difficult

job type	#chunks	#nodes	multi-GPU?
multi	> 1	1	yes
single	1	≥ 1	no

Table 8.1: Overview of the number of chunks and nodes a multi- or single-job contains and whether multi-GPU processing is possible.

to implement. Thus our base method of choice is binned construction [Popov et al. 2006], [Wald 2007]. We refer to Section 2.5.4 for algorithm details.

A key observation made by Popov et al. [2006] is that binning could also be performed independently on arbitrary subsets of geometry. The final binning result would be obtained by combining the results of the subsets. This renders binning highly suitable for a multi-GPU approach and forms the basis of our out-of-core algorithms. We now first discuss our BVH construction algorithm and proceed by describing the kd-tree algorithm as a modification to the BVH algorithm.

8.3.1 BVH Construction

As we allow geometry to exceed GPU memory we unavoidably have to partition it into chunks that can be managed by a GPU. A single GPU performs binning and partitioning on a single *chunk* of geometry at a time. A chunk can only contain up to N_C triangles. N_C depends on the maximum amount of memory available on the GPUs and the maximum amount of memory required during execution. Peak memory consumption is reached in the triangle partitioning step, which requires a single GPU to be able to store its input and output primitives. As partitioning does not produce duplicates we have N_C input plus N_C output triangles. Thus the upper bound of N_C is chosen such that the input primitives consume at most 50% of the maximum available memory. In practice we use much lower thresholds of 10% for several reasons. First, we need memory for auxiliary data such as bin counters and bin bounds. Further, we want several chunks to be present in a GPU at the same time. This allows to asynchronously load more chunks while a chunk is processed. Geometry chunks are logically grouped in either so called *single-* or *multi-jobs* which are scheduled by the host for processing. A multi-job contains the geometry chunks of a single node, that is too large to completely fit into a single GPU. Single-jobs contain a single chunk that aggregates the complete geometry of one or more nodes such that it still fits into GPU memory. Thus multi-jobs can be processed by several GPUs, while single-jobs are processed by a single GPU. The criteria for multi- and single-jobs are summarized in Table 8.1.

The algorithm is initialized by partitioning the geometry into chunks, which are all grouped together in a multi-job for the root node. Upon execution of a multi-job, chunks are distributed to several GPUs to perform binning on them in parallel. Then, the intermediate binning results are copied to one GPU which combines them to compute the global binning result and determine the approximately best split plane. This introduces a single point of synchronization within multi-jobs which is unavoidable if we aim for full SAH top-down construction. The best binning axis and split position is sent to the other GPUs, so that they can perform the actual distribution of triangles into the left and right subtree for every chunk. If all triangles of a chunk are put exclusively into the left or the right sub tree, no copying is needed. We easily identify this case for each chunk by looking at

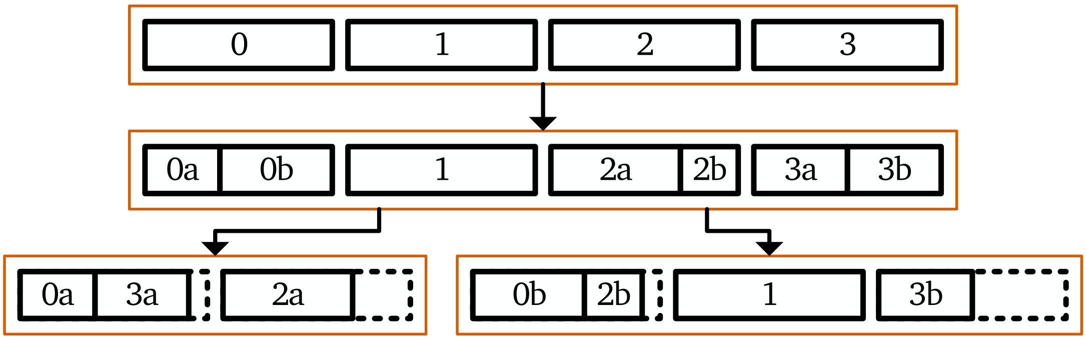


Figure 8.1: Splitting a multi-job with four chunks into two new multi-jobs. The split results in seven chunks. Chunk merging reduces the number of chunks to five. Chunks are grouped in a first-fit manner for computational simplicity. A more optimal strategy would have grouped $2b$ and $3b$ together to increase minimum chunk size for even higher GPU efficiency.

the binning counters of the intermediate per-chunk results. After the splitting step a new multi-job containing the corresponding chunks is created for both new subtrees. Both jobs are completely independent and can be processed in parallel on multiple GPUs.

Repeated splitting of chunks eventually results in chunk sizes which cannot fully occupy a GPU. As a countermeasure we perform a merging step directly after splitting. It merges small chunks into new chunks with at most N_C triangles. Therefore a merging schedule is created. We start with an empty merge group. Then we iterate the list of chunks and merge them into the group as long as the chunk size limit is not exceeded. If a chunk does not fit into the current group, we do not immediately open a new group. Instead we iterate over all groups that have been opened so far to seek for a suitable group in a first-fit manner. If no group could be found, a new merge group is opened. Though we could go for a more optimal chunk grouping algorithm, our simple approach already significantly reduces the number of chunks and increases the efficiency of the algorithm. Multi-job splitting with chunk merging is depicted in Figure 8.1.

But even with chunk merging, subtrees will eventually need only one chunk and contain a small number of triangles. To alleviate this problem the algorithm can decide to merge the data of both resulting subtrees into a single-job if it fits after splitting. This further helps to maintain good GPU utilization. Though the amount of geometry in a single-job stays the same or decreases due to leaf creation, the number of nodes can increase. As nodes cause additional memory overhead because of binning related and other auxiliary data it might be necessary to create two single-jobs. It would be possible to merge different single-jobs if they fit into a single chunk. But because of the independence that is defined between jobs this would require synchronization and is not included in our approach.

8.3.2 Job Scheduling

Scheduling of single- and multi-jobs can be done breadth-first or depth-first. In case of breadth-first, all subtrees on the same depth are processed before commencing to the next depth level. This is done in all parallel implementations except for Hou et al. [2011]. An inherent property of breadth-first construction is that creation of inner nodes is implicitly

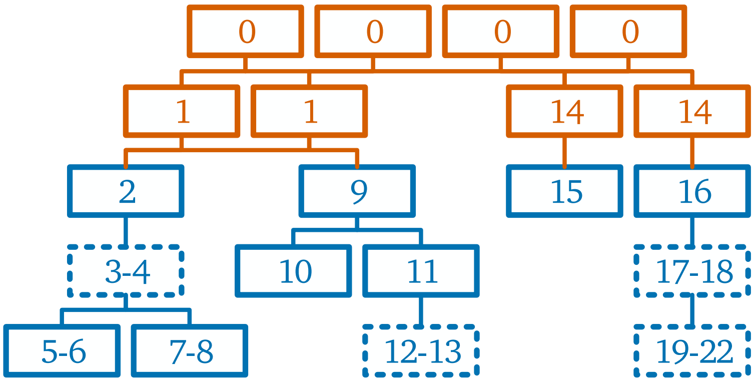


Figure 8.2: Processing order of multi-jobs (red) and single-jobs (blue) for a single GPU and a scene initially consisting of four chunks. Each rectangle depicts a chunk that fits into GPU memory. The numbering indicates different nodes and the processing order of these nodes at the same time. Dashed boxes indicate single-jobs which were small enough to not be split. The second level has the same number of chunks due to chunk merging. The chunk containing nodes 3 and 4 contained too many splits and/or additional node information after processing and had to be split into two single-jobs.

temporally promoted, while creation of leaves happens more at the end of the algorithm. With our target scene sizes, for which the whole geometry does not fit into GPU memory it is highly likely that data has to be swapped to the host cache to process a different subtree on the same level with this strategy. Thus, our method of choice is a *partial*-DFS-BFS order. It aims at first processing a subtree to its end to reduce the amount of transient geometry but also accounts for increasing node parallelism in the lower parts of the tree. Another advantage is that data that just has been split is typically directly processed further, so that swapping of subtrees is less likely. Single-jobs locally construct the acceleration structure in a breadth-first manner. In general this is no problem as, by their very nature, data of single-jobs is already small enough to fit into a single GPU. An example processing order is depicted in Figure 8.2.

The execution flows of a GPU and the scheduler are illustrated in Figure 8.3. Our scheduler works with two separate stacks for single- and multi-jobs. Initially the single-job stack is empty and the multi-job stack contains a job for the tree root. When a GPU is looking for work it first checks if there is a currently active multi-job, as these have highest priority. The reason for this is that the more GPUs participate in multi-job processing, the less swapping occurs. If there is no currently active multi-job or the multi-job has no unprocessed chunks left, we first check the single-job stack and then the multi-job stack for work. If the single-job stack is empty it pops a multi-job from the multi-job stack. Single-jobs that possibly have been created after splitting are put on the single-job stack. If the right subtree created by a multi-job again is a multi-job it is pushed onto the multi-job stack, while if the left subtree is a multi-job it is issued directly.

8.3.3 kd-Tree Construction

The main differences between the kd-tree and BVH algorithm arise from the fact, that we need the triangle geometry in the splitting stage. Unfortunately, as geometry does not

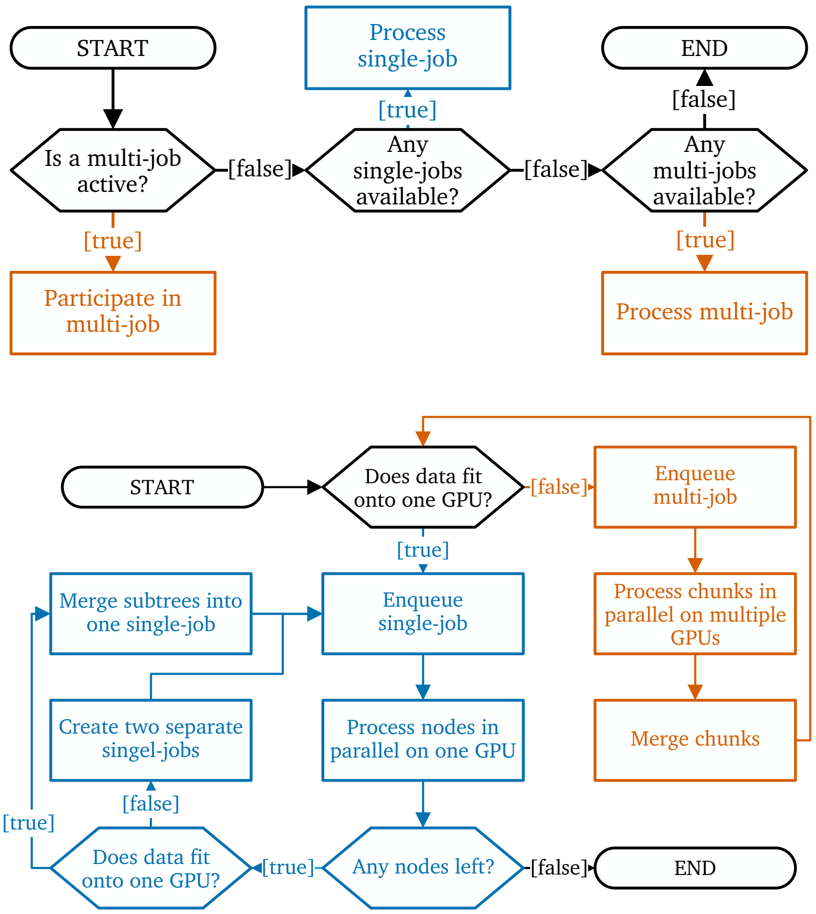


Figure 8.3: Depiction of the execution flow of a GPU looking for work (*top*) and the job creation and processing flow (*bottom*). Multi-job and single-job processing related events are colored orange and blue respectively.

completely fit into graphics memory, we cannot just duplicate primitive references on a split. Referenced geometry must be present, too, to recompute tight bounds. To get any memory savings from the surjective mapping of references to triangles, we would have to keep a list of triangle geometry with each chunk that exactly only contains triangles referenced by the primitive references without duplicates. This requires a chunk scope index per primitive reference. In case the original triangle index cannot be reused because triangles need a unique index for additional properties like normals and a material, this also requires additional storage. After each split all referenced triangles would have to be detected while filtering out duplicates. For simplicity we avoid all this and also duplicate the actual triangle geometry on chunk splitting. But we still do not have to retessellate along splitting planes.

As in the BVH case, peak memory consumption is reached in the triangle splitting step. As potentially all triangles might get split we have N_C input plus up to $2N_C$ output triangles. Thus N_C is chosen such that the input primitives consume at most 33% of the maximum available memory. Despite the higher memory overhead caused by the geometry, auxiliary data is lower for the kd-tree. Thus, we again used the same 10% threshold as for BVHs in practice.

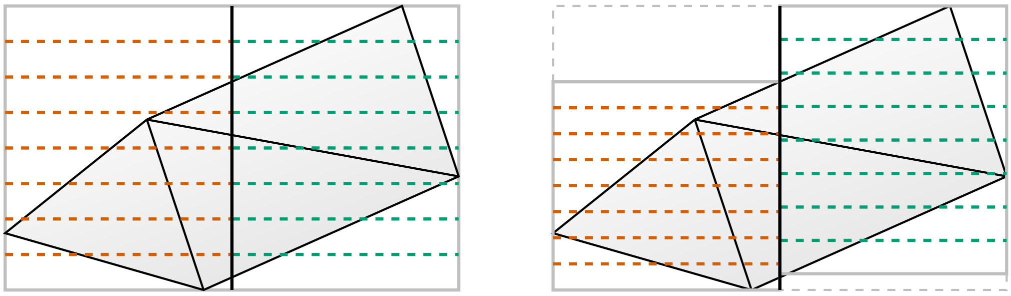


Figure 8.4: Example of localized binning with eight bins for a scene with three triangles which already has been split once (black split plane). Conventional binning (*left*) constructs equidistant split planes (red and green for left and right node) w.r.t. node bounds. Two planes for the left node have all geometry on one side. Localized binning (*right*) uses the bounds of the clipped geometry of a node for split plane construction. Compared to conventional binning, for a fixed number of bins this allows for a more effective sampling of the candidate cost function.

As already discussed, we conclude from the results of Aila et al. [2013] that SAH much more accurately predicts trace performance of kd-trees than of BVHs. Thus, it is more worthwhile to find minimal SAH splits. Our choice of using an approximating binning approach is disadvantageous in this regard. As a countermeasure we propose to construct bins for binning from the tight bounds of the geometry in a node rather than from node bounds. Node bounds are not guaranteed to be tight for space partitioning algorithms. Figure 8.4 depicts this *localized* binning approach. Localized binning guarantees that all split planes produce a real partition whereas conventional binning can produce planes where all geometry is only on one side and thus effectively wastes cost function samples. This can also be seen in the example figure. A comparison of localized binning with conventional binning is provided in Section 8.5.5.

Depth-first job scheduling is even more crucial for kd-tree than for BVH construction. Breadth-first scheduling would result in a rapidly increasing amount of transient geometry from triangle splitting. Creation of leaves should be favored as it decreases the amount of transient geometry.

8.3.4 Improvement Threshold

Finding ways to reduce the number of nodes in the output tree is desirable for several reasons. Less nodes means faster construction and, more importantly in an out-of-core context, smaller tree sizes. For kd-trees this also means less duplicates, which again is a benefit in an out-of-core context. As the implementation dependent traversal cost constant c_t has a huge impact on tree size, Danilewski et al. [2010] proposed to treat it as a parameter that allows to trade quality for construction performance. They showed results for relative costs ranging from $c_t = 1.3$ to $c_t = 16$. But even 1.3 is already relatively high for a kd-tree. Relative traversal cost of kd-trees is usually around 0.5 and around 1 for BVHs. As an alternative we introduce an SAH improvement threshold τ. We only perform a node split if the best split manages to decrease the node's SAH cost by at least a factor of τ. That is we perform a split if the following inequality is true:

$$1 - \frac{c_{split}}{c_{leaf}} > \tau. \tag{8.2}$$

The motivation behind this is the observation that if a split manages to decrease SAH by p, the actual trace performance gain usually is lower than p. Rearranging the split criterion for c_{leaf} gives

$$c_{leaf} > \frac{c_{split}}{(1-\tau)} = c_{split}\left(1 + \frac{\tau}{1-\tau}\right). \tag{8.3}$$

Thus, we can see that τ effectively increases the whole split candidate cost by $\frac{\tau}{1-\tau}$ which is not equivalent to simply increasing c_t. This is a more rigorous approach than the one proposed by Danilewski et al., as it allows us to use the correct implementation dependent costs. Thresholds of as small as 5% have a huge impact on tree depth and triangle duplication. The resulting effects on construction and ray tracing performance are shown in Section 8.5.6.

8.4 Implementation

We used Nvidia's CUDA Version 8.0 [1] to implement our proposed algorithm. The binning and splitting steps are implemented in different kernels and are also implemented differently for multi- and single-jobs. We use a fixed number of 64 bins for BVH construction and 128 bins for kd-tree construction. Empirically larger bin counts only increased construction times without significant SAH improvements. Thus, we optimized our implementations for these bin counts. We first describe details of the BVH implementation and then the differences of the kd-tree implementation.

8.4.1 BVH Implementation

A chunk of a multi-job consists of millions of primitive references that all belong to the same node. This offers a tremendous amount of primitive parallelism. A straight forward way to do this would be to create enough blocks for all three binning dimension to bin all N chunk primitives. This results in $N_B = 3\left\lceil \frac{N}{S_B} \right\rceil$ blocks with a block size of S_B. For each block we would have to store a set of 64 bins where each bin stores a primitive counter and bin bounds. Using a 4 byte counter and 4 bytes per bounds coordinate component results in 28 bytes per bin. For our smallest evaluation dataset *Powerplant* with 12.7M primitives, which easily fits into a chunk, and a block size of 256 this results in 148830 blocks which in total require about 250MB of auxiliary binning data. As several chunks have to be present for much larger datasets this is too much and the multi-job binning kernel have to keep the amount of needed device memory for intermediate data as low as possible. Therefore, we let binning operate in a persistent-blocks manner [Aila and Laine 2009]. A chunk is equally partitioned among the N_{SM} streaming multiprocessors of a GPU. This is done by letting at least as many blocks as there are multiprocessors *iterate* over the primitives and collect results in shared memory. Each thread in a block computes the destination bin for its own primitive and atomically updates the bin bounds and the count in shared memory. As CUDA only defines *atomicMin* and *atomicMax* operations for integral

---

[1] `https://developer.nvidia.com/cuda-downloads`

class	n	processed by	remarks
Large	$n > 2048$	#SMs blocks / node / dim.	geometry parallelism
Medium	$256 < n \leq 2048$	block / node / dim.	node parallelism
Small	$32 < n \leq 256$	warp / node / dim.	implicit SIMD synchr.
Tiny	$16 < n \leq 32$	warp / node	exact SAH
Micro	$8 < n \leq 16$	half warp / node	exact SAH
Nano	$n \leq 8$	quarter warp / node	exact SAH

Table 8.2: The different node classes along with their triangle limits and parallelization for BVH single-job processing. Each class is processed by a dedicated kernel. The kd-tree implementation uses a similar classification.

data types, triangle bounds are stored in order preserving integer format [Terdiman 2000]. After a block iterated over all its triangles it atomically combines its results with the results from other blocks in global memory. For code simplicity a block only performs binning in a single binning dimension. Thus, a set of blocks is generated for each dimension and we only require a single set of bins per dimension. This essentially nullifies the amount of needed auxiliary memory to a constant 5.25KB with our 64 bins per dimension. The resulting minimum number of blocks for multi-job binning is $B_{MJ} = 3 \cdot N_{SM}$. Depending on the chosen block size and kernel resource usage integer multiples of B_{MJ} have to be used to maximize occupancy. After binning, results of all chunks are combined and the best split plane is determined with parallel prefix sum and reduction.

The next step is distribution of primitive references to the new children. We explicitly dedicate blocks responsible for putting primitives only to the left or only to the right side. Each block compacts its triangles and writes them to a position specified by atomic counters for both children. The resulting number of blocks for splitting is $S_{MJ} = 2 \cdot \left\lceil \frac{N_{chunkprimitives}}{S_B} \right\rceil$.

In the spirit of the GPU kd-tree construction approach from Danilewski et al. [2010] a single-job uses different specialized kernels for binning and partitioning depending on the number of primitives in a node. This is to adapt to the shift from primitive parallelism to node parallelism by adapting the mapping of threads to primitives and nodes. Danilewski et al. [2010] execute their different specializations in stages. Each stage works on the complete set of nodes. When threads responsible for a node detect that a node has the wrong primitive count they immediately return, causing unnecessary overhead. Stages which map nodes to warps are especially inefficient because they loose effective occupancy when warps in a block partially return if they have the wrong node primitive count.

To avoid these problems we analyze the current set of nodes to classify each node w.r.t. its number of primitives into six different classes. For each class a compact list of node IDs is extracted which is then processed by the corresponding specialized implementation. We use a block size of 256 for all binning and partition specializations. The different classes along with their triangle limits and parallelization are depicted in Table 8.2. *Large*-nodes still offer enough geometry parallelism that they essentially can be processed by several multiprocessors like a chunk of a multi-job but with a set of blocks for every node. With *Medium*-nodes parallelism slowly changes to node parallelism, where a single block per dimension performs binning on a node in a couple of iterations over the geometry in a node. With *Small*-nodes effective occupancy would start to decrease as there would be less primitives than threads in a block which causes warps in a block not to be assigned to any primitives. Thus, *Small*-nodes switch to mapping warps in a block to nodes and let warps iterate over the triangles of a node for binning in a persistent-warps manner. We also take advantage of the implicit synchronization of threads in a warp by omitting block synchronization primitives which greatly increases performance. With less than 32 primitives per node the *Small*-node approach starts to suffer from decreasing SIMD efficiency. The last three node classes *Tiny*, *Micro*, and *Nano* account for this by partitioning a warp into sub warps which process their own node, keeping more lanes busy. Similar to Danilewski et al. [2010] for kd-trees we noticed that initialization of all bins and performing scans on the bin data dominates computation time for such small node primitive counts. This is even more severe for BVHs as BVH bins store 3.5 times as much information as kd-tree bins. Thus, like Danilewski et al. [2010] we switch to exact SAH computation for such small nodes by letting each thread iterate over all primitives. It also turned out to be beneficial to directly handle all three candidate dimensions.

With additional effort it should be possible to introduce more node classes for more differentiated levels of parallelism.

8.4.2 kd-Tree Implementation

Implementation of multi- and single-job processing is analogous to the BVH case with the different node classes. The major difference is that we have to count enter and exit events of primitive references which is simpler in terms of computation and memory consumption than growing of bin bounds in BVH construction. The primitive distribution step is more involved, as we also actually have to split triangles to compute clipped primitive bounds. A straightforward implementation requires a couple of nested conditional statements with computational complexity in their bodies which causes poor SIMD efficiency due to high thread divergence and also resulted in high register usage which additionally reduced occupancy. Empirically most primitives in a node do not straddle the split plane. According to Wald and Havran [2006] “for reasonable scenes [1], there will be (at most) $O(\sqrt{N})$ triangles overlapping” the split plane. Thus, our approach of choice is to exploit this by splitting primitive distribution into two phases. The first phase copies all primitive references and triangles to their respective side of the split plane including references which have to be split. Duplicate primitives are explicitly compactly stored at the ends of the arrays of each side. Without the primitive splitting code this kernel is essentially a memory copy kernel which has high occupancy due to its simplicity. Now in the second phase the highly divergent and inefficient primitive bounds splitting kernel is only executed on the few compacted duplicate primitives. Performance of splitting increased by almost one order of magnitude with this approach compared to the branching version.

8.4.3 Out-of-Core Work and Data Management

CUDA kernels are grouped in a so called *task*. Data dependencies of kernels are registered with the task. Multi-jobs have tasks for binning, combination of binning results and chunk splitting. A whole single-job is mapped to one task.

A GPU device processes at most two tasks at a time. One task executes its kernels while the other resolves its data dependencies. Devices in need of work register at a task scheduler. A good scheduling strategy aims at reducing host-to-GPU, GPU-to-host and GPU-to-GPU memory transactions. At the same time GPUs have to be kept busy as much as possible. Our chosen task scheduling strategy for a requesting device is as follows: The scheduler iterates over all available tasks and determines for each task the most suitable device. The primary deciding factor for suitability is the number of already resolved data dependencies. The second factor is the number of tasks currently processed by a device. A requesting device is only assigned tasks it is the most suitable device for. Thus, we intentionally assign no task if there are other more suitable devices for the available tasks. If a task is equally suitable for all devices, it is assigned to the requesting device. Though this seems subpar at first, it proved to be a good strategy as it trades some idle time for a significant reduction in memory transfers.

GPU memory allocation functions in CUDA cause overhead and, even worse, synchronize the GPU. To avoid these issues we allocate a large self managed GPU memory pool for each GPU on startup. Allocations are performed in a first fit manner and are evicted to system memory with an LRU strategy. Data dependencies belonging to the two tasks a device can process at a time are protected from eviction. In the rare case, that no memory can be allocated due to external fragmentation a defragmentation step is performed.

When resolving dependencies, data that resides in other GPUs' memory pools is directly asynchronously copied via peer-to-peer GPU copy. This avoids expensive round trips through system memory. For transactions from or to system memory to be asynchronous, CUDA requires the involved system memory to be page-locked. Allocation of page-locked memory has a much higher overhead than GPU memory allocation and also causes synchronization. Again we avoid these issues by allocating a huge self managed memory pool of page-locked memory on startup.

8.5 Evaluation

We evaluated the proposed construction algorithm with regard to scaling behavior and performance for four different scenes of increasing size. All experiments were performed on a system running Ubuntu 16.04 which is equipped with two Intel Xeon E5-2687W v3 deca-core CPUs, 128GB of RAM, and eight NVIDIA Geforce GTX 980 GPU cards with 4GB of RAM each (NVIDIA driver version 375.66). The underlying threading machinery of our implementations used Intel's Threading Building Blocks (TBB) version 2017 Update 6[2].

We used four scenes of increasing memory footprint ranging from 12.7M triangles (438MB) to 940M triangles (31.5GB) (see Figure 8.5). The smallest scene, *Powerplant*, can be considered an upper bound for in-core GPU algorithms. As already mentioned in Section 8.4 all BVH construction tests used 64 bins and kd-tree construction tests were performed with 128 bins. All kd-tree tests where preformed with localized binning (see Section 8.3.3), except for the evaluation of localized binning itself in Section 8.5.5. The effects of localized binning, hybrid construction, and the SAH improvement threshold on tree quality and actual traversal cost were evaluated using an in-core multi-core CPU path tracer.

For SAH-based kd-tree construction we used the constants $(c_t, c_i) = (1.3, 1.0)$ though $c_t = 0.75$ corresponds to the actual relative traversal cost of our CPU path tracer. $c_t = 0.75$ resulted in too excessive triangle splitting for the larger scenes. For BVH construction we used $(c_t, c_i) = (1.2, 1.0)$. Following the approach from Section 5.4.2 we computed the average number of traversal steps $\overline{n}_s$ and primitive intersection tests $\overline{n}_p$ performed by our path tracer to get platform and ray traversal implementation independent ray tracing

---

[2] `https://www.threadingbuildingblocks.org/`

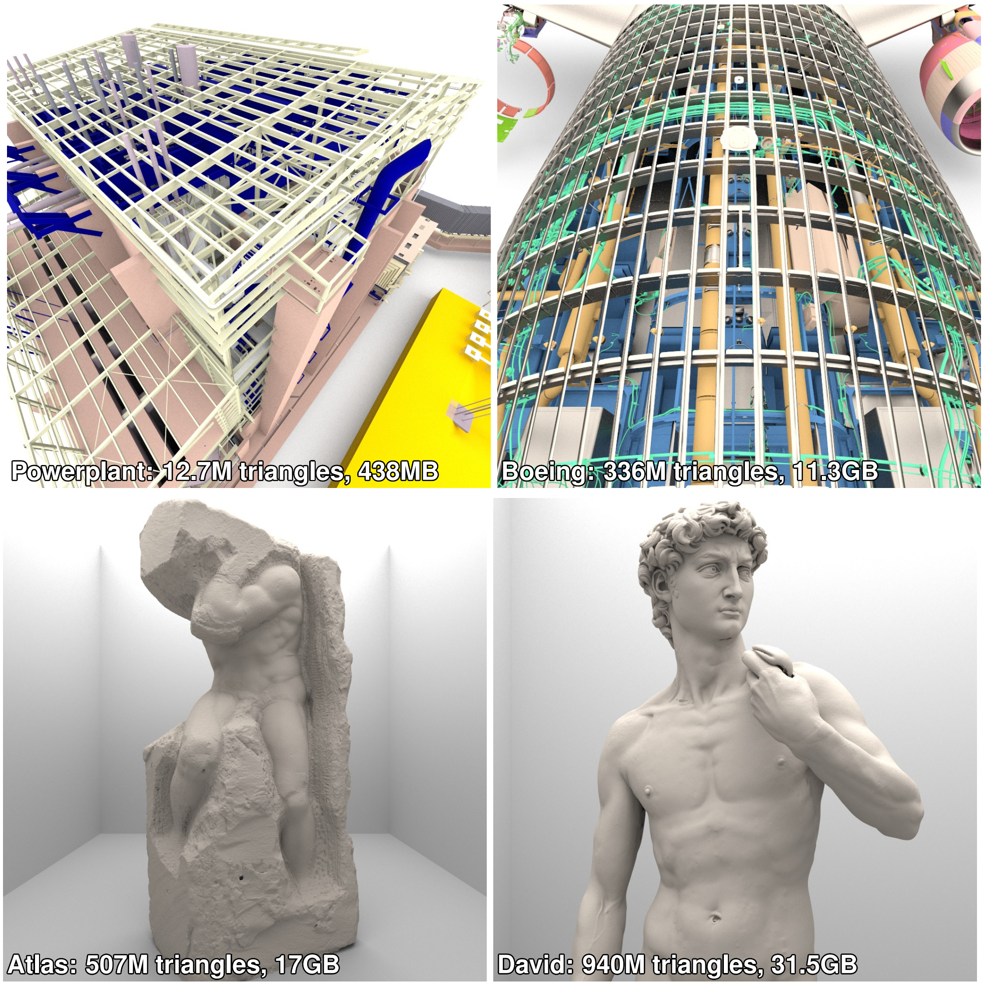


Figure 8.5: Test scenes used for our experiments. The depicted camera views were also used for measurement of traversal cost.

performance measurements. Combined with the SAH cost constants (c_t, c_i) this lets us compute the average measured traversal cost $m = \overline{n}_s c_t + \overline{n}_p c_i$. In case of BVH traversal we used the constants $(c_t, c_i) = (1.2, 1.0)$ we also used for construction. For kd-trees we used our actual $c_t = 0.75$ of our path tracer.

8.5.1 Peak System Memory Footprint

For BVH construction we need to store primitive references, which consist of a bounding box and a primitive id. Using 32-bit values this requires 28 bytes per primitive. For kd-tree construction we additionally need to store the triangle primitives themselves. This results in 64 bytes per primitive for kd-tree construction. Table 8.3 lists the peak amount of system memory required to store the initial set of primitives for BVH and kd-tree construction when using no GPU. Adding GPUs reduced this peak amount by the amount of graphics memory each GPU provided. In our setup this meant a 4GB reduction per GPU. Using all eight GPUs only kd-tree construction for *Atlas* and *David* required additional system memory. kd-tree construction can cause the footprint to grow slightly larger in the upper

Builder	Powerplant	Boeing	Atlas	David
BVH	339MB	8.8GB	13.2GB	24GB
kd-tree	775MB	20GB	30.2GB	56GB

Table 8.3: Peak amount of system memory required to store the initial set of primitives for BVH and kd-tree construction when using no GPU. Each added GPU reduces this amount by the available amount of graphics memory.

levels due to triangle splitting. As described in Section 8.3.2 we apply a partial-DFS-BFS job scheduling order, which not only reduces this temporary footprint growth but also causes the amount of transient primitives and primitive references to decrease rapidly.

8.5.2 Comparison with Optimized CPU Implementations

To better judge the efficiency of our GPU implementations we first compare performance with in-core CPU builders. Our test system has enough system memory for the CPU algorithms to work on the larger scenes. NVIDIA Optix[3] is a GPU ray tracing middleware which also provides CPU and GPU-based BVH construction solutions. As of version 4.1, Optix has no GPU solution for out-of-core full top-down SAH-based BVH construction. Further, the construction constants for SAH-based construction are not exposed. For these reasons we do not include a comparison with Optix.

For BVH construction we compare against the highly optimized high quality BVH builder from Intel's Embree[4] library. Embree provides a set of highly optimized ray tracing kernels and BVH builders which exploit the SIMD capabilities of CPUs. Some aspects make comparison difficult, though. Embree uses a lower count of 32 bins for construction. Further, it constructs multi-branching BVHs (MBVH) with branching factor 8 for our hardware setup. While MBVH construction uses binary construction during construction we cannot directly apply its used traversal constant c_t. The BVHs constructed for our test scenes with Embree have an average of four primitives per leaf. Empirically this corresponds to $c_t = 2.4$ for pure binary BVH construction which we used for our GPU implementation for comparison with Embree.

For kd-tree construction we intended to compare with the freely available implementation of the highly parallel construction algorithm from Choi et al. [2010]. Their implementation used $c_t = 0.75$ for construction which we changed to $c_t = 1.3$ for comparability. Unfortunately, we had to artificially limit the maximum depth of their implementation because it did not seem to stop construction on its own for our smallest scene, *Powerplant*. Choi et al. [2010] themselves evaluated their algorithm with a maximum depth of 8, which is only enough for very small scenes with a primitive count on the order of 10^3 primitives. As our GPU implementation produced a tree with depth 58 for *Powerplant* we used this number as the maximum depth for the implementation of Choi et al.. The resulting construction time of *Powerplant* was 20 minutes. This is already slower than the *serial* implementation from Wald and Havran [2006] which according to the authors is "poorly optimized". Thus, we decided to use our own parallel implementation, which lets threads cooperatively process larger nodes or construct whole subtrees on their own

---

[3] `https://developer.nvidia.com/optix`

[4] `https://embree.github.io/`

	BVH(s)			kd-tree(s)		
Scene	GPU	Embree	Rel.	GPU	CPU	Rel.
Powerplant	0.7	0.6	-14.3%	2.7	7.2	+166.7%
Boeing	27.5	17.5	-36.4%	101.0	237.7	+135.3%
Atlas	36.0	26.8	-25.6%	110.5	275.9	+149.7%
David	77.6	190.0	+144.8%	228.4	508.8	+122.8%

Table 8.4: Timings for BVH and kd-tree construction of our GPU implementations on a single GTX 980 with Embree and our CPU implementation for kd-tree construction on a deca-core processor with 20 threads. The relative time difference of GPU and CPU implementations is shown as well. See Section 8.5.2 for more details on the test setup and comments on comparability.

#GPUs	1	2	3	4	5	6	7	8
BVH (ms)	891	625	551	429	356	356	358	364
kd-tree (ms)	2691	1696	1301	1079	965	943	919	770

Table 8.5: GPU construction time scaling for *Powerplant* with increasing number of GPUs.

if they are small enough.

Experiments were conducted with a single GTX 980 GPU and one deca-core CPU with 20 threads. Results of the GPU and CPU implementations including relative time differences are shown in Table 8.4. Though a bit slower for BVH construction, our single GPU can roughly compete with Embree on a deca-core processor for the first three scenes. For reasons unknown to us Embree shows a drastic increase in construction time for *David* and clearly falls behind our GPU solution. For kd-tree construction our GPU solution is 2 to 3 times faster than our CPU solution. Our implementation did not use vectorization. With proper vectorization we expect our CPU solution to be at least on par with our GPU solution given our hardware setup. Our CPU solution is also about two orders of magnitude faster than the implementation from Choi et al. [2010]. We suspect the reason for this enormous discrepancy must be a bug or weakness in their implementation related to the size of the *Powerplant* scene, as they used smaller scenes and the aforementioned maximum tree depth of 8 for their evaluation.

8.5.3 Multi-GPU Scaling

For multi-GPU scaling behavior we measured construction time and speedup efficiency of our construction algorithms with one to eight GPUs. Speedup efficiency $E_n = \frac{S_n}{n}$ for n processors is defined as the ratio of speedup $S_n = \frac{T_1}{T_n}$, where T_n is the construction time for n GPUs (Kumar et al. [1994]). A speedup efficiency of 1 means perfect linear speedup, whereas an efficiency of 0.5 means the speedup corresponds to the expected perfect speedup for half as many processors. As *Powerplant* is an in-core dataset we had to artificially split it into n chunks for n GPUs. Results for BVH and kd-tree construction are depicted in Figure 8.6 for all scenes. Results for *Powerplant* are also shown separately in Table 8.5. Timings also include geometry load time. We also provide differentiated

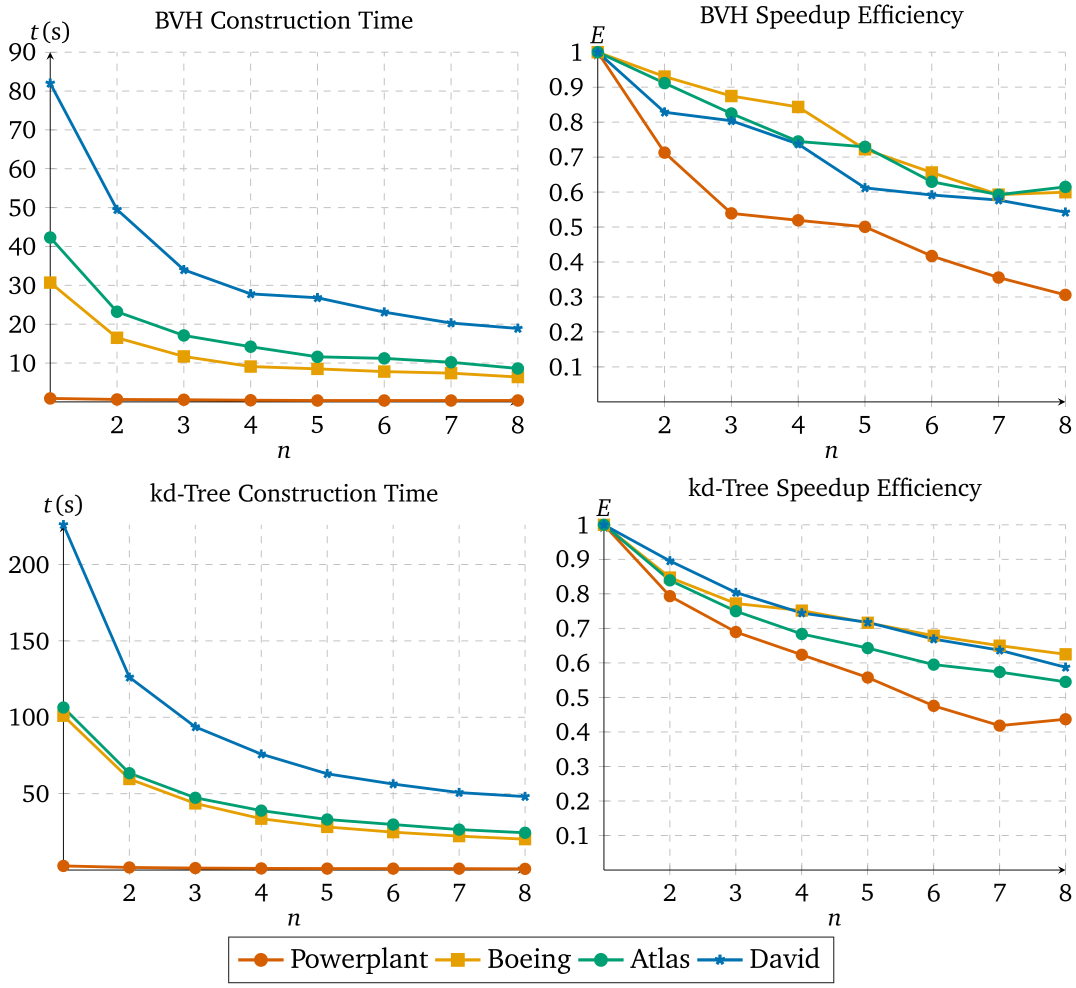


Figure 8.6: Scaling of construction time with the number of GPUs (*left*) and speedup efficiency (*right*) for BVH (*top*) and kd-tree construction (*bottom*).

plots for single- and multi-job processing time, and speedup efficiency for BVH and kd-tree construction in Figure 8.7 and Figure 8.8. As *Powerplant* is initially split into n chunks with n GPUs there are no multi-jobs when using one GPU. Thus multi-job speedup efficiency is w.r.t. two GPUs for this scene.

Powerplant clearly shows by far the lowest speedup efficiency for BVH and kd-tree construction. It is the only scene where efficiency falls below 50%. Though the speedup of BVH single-job processing is superlinear and similar to the other scenes for kd-tree construction the introduction of multi-jobs from artificial chunk generation causes large extra overhead.

For BVH construction with two GPUs speedup efficiency is above 90% for *Boeing* and *Atlas*, and above 80% for *David*. Even with eight GPUs efficiency stays above 50% for these three scenes. From the job processing time plots we can see that BVH single-job efficiency is above 90% for the three smaller scenes and still above 80% for *David* with eight GPUs. Thus, as expected the loss in global speedup efficiency comes from lower multi-job scaling, which falls to less than 40% efficiency. While single-job time initially is higher than multi-job time the higher single-job efficiency manages to achieve lower

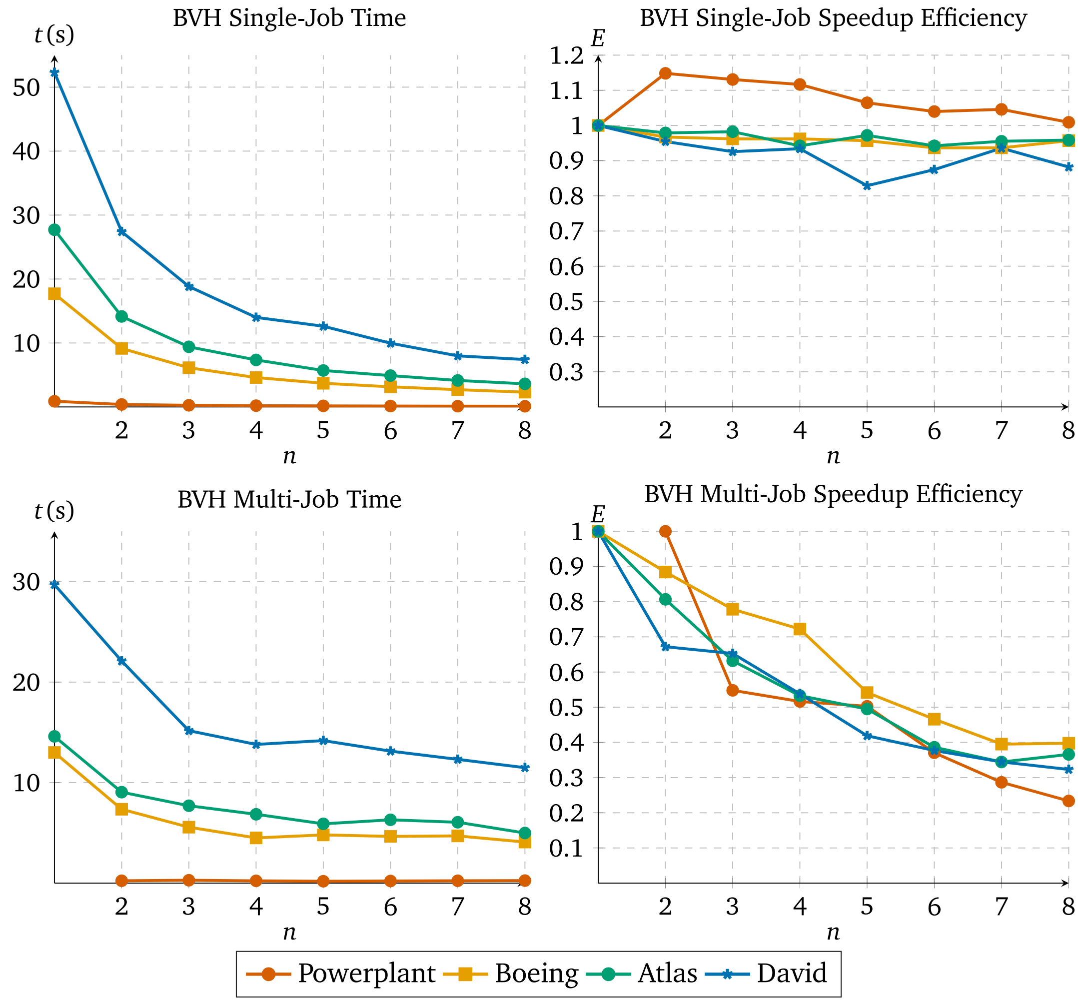


Figure 8.7: BVH scaling of job processing time with the number of GPUs (*left*) and speedup efficiency (*right*) for single- (*top*) and multi-jobs(*bottom*). Due to its size *Powerplant* does not have multi-jobs with 1 GPU and speedup efficiency is w.r.t. two GPUs in this case.

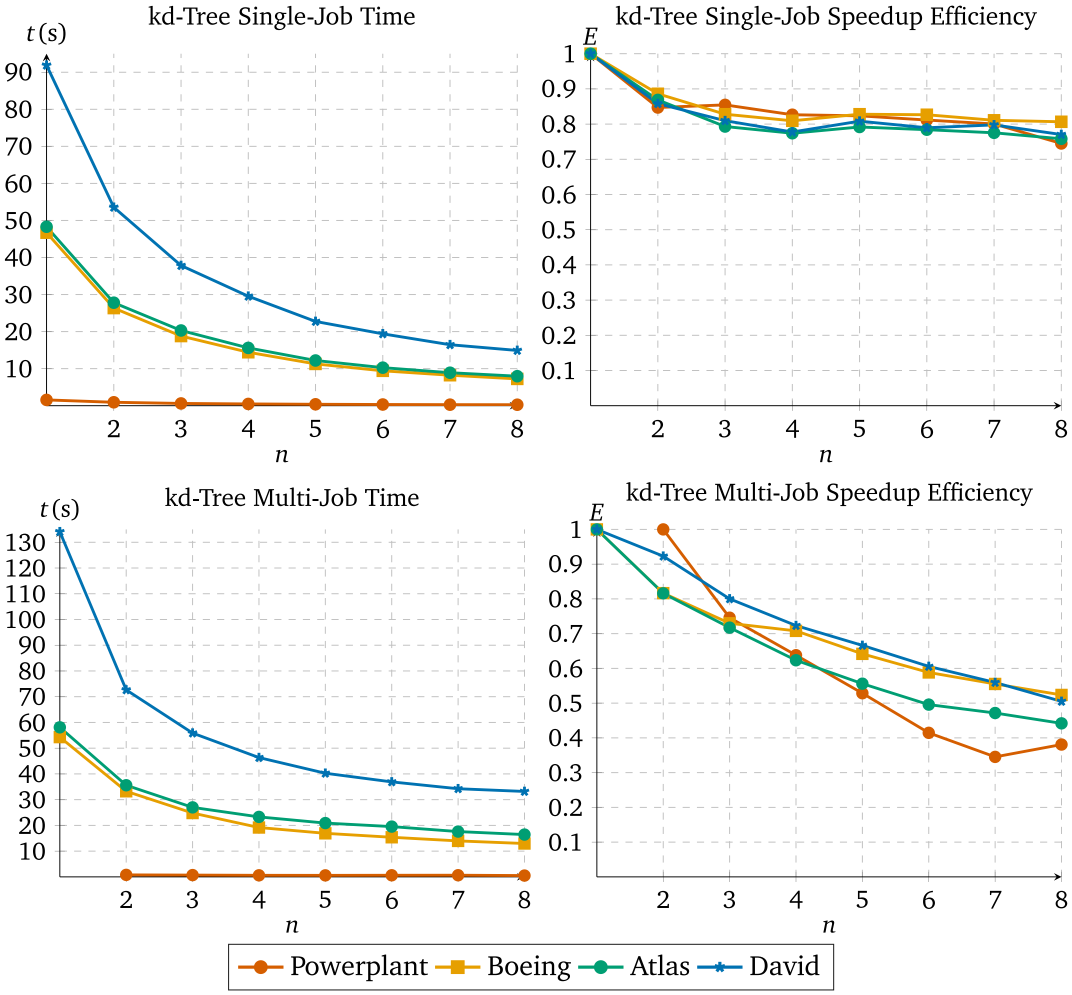


Figure 8.8: Kd-tree scaling of job processing time with the number of GPUs (*left*) and speedup efficiency (*right*) for single- (*top*) and multi-jobs(*bottom*). Due to its size *Powerplant* does not have multi-jobs with 1 GPU and speedup efficiency is w.r.t. two GPUs in this case.

Scene		Powerplant	Boeing	Atlas	David
SAM	Full	55.0	130.7	38.4	29.8
	Hybrid	62.3	126.0	37.3	30.8
	Rel.	+13.3%	-3.6%	-2.9%	+3.4%
m	Full	101.3	186.6	44.4	42.9
	Hybrid	115.0	194.3	45.0	44.8
	Rel.	+13.4%	+4.1%	+1.4%	+4.4%
Build(s)	Full	0.9	29.4	41.5	84.6
	Hybrid	1.6	33.0	78.0	112.1

Table 8.6: Timings, tree quality (SAM), and average measured traversal cost m for our full top-down construction and our implementation of the hybrid construction algorithm proposed by Pantaleoni et al. [2010].

processing time than multi-jobs starting from four GPUs.

Results of kd-tree construction show similar total speed up efficiency. Kd-tree construction has slightly lower single-job efficiency with 70% to 80% and slightly higher multi-job efficiency with 40% to 50%. Contrary to BVH construction multi-job processing time is higher from the beginning.

8.5.4 Tree Quality Comparison with Hybrid Construction Approach

To compare quality of our results with more simple to implement hybrid techniques we implemented the out-of-core BVH construction approach from Pantaleoni et al. [2010]. The input of their algorithm consists of microgrids, a collection of micropolygons. To stay faithful to their implementation, we converted our scenes into microgrids by regular recursive clustering of triangles into microgrids of at most 256 triangles. The limit for bucket aggregation is the same 64KB of microgrid references, while the threshold for further subdivision of overfull buckets is 4M triangles. Suitable aggregate clusters of buckets are issued to available GPUs as single-jobs to construct corresponding sub-BVHs. For top-level tree construction exact sub-BVH SAM costs are used. We omit the treelet construction step as we do not need it for our renderer. Results are shown in Table 8.6. Though efficiency of our implementation did not have highest priority, construction time is shown as well. An optimized implementation should achieve lower construction times.

For *Powerplant* we observed a 13% higher SAM and traversal cost with hybrid construction. For *Boeing*, hybrid construction actually achieves a 4% smaller SAM cost, but traversal is 4% slower. *Atlas* and *David* more or less have similar SAM and traversal costs to the full top-down approach. The reason for this is, that though these scenes have the highest triangle count they expose a much more uniform tessellation and simpler shape. This makes it easier to construct well performing trees. In fact we also constructed a much simpler spatial median split BVH for *David* and *Atlas*. The resulting SAM and traversal costs were practically identical to full top-down and hybrid construction.

Scene	SAM			m		
	N	L	Rel.	N	L	Rel.
Powerplant	101.7	93.5	-8.1%	109	97.4	-10.6%
Boeing 777	159.3	142.8	-10.4%	166.7	127.3	-23.6%
Atlas	173.5	103.3	-40.6%	133.9	78.2	-41.6%
David	190.5	79.7	-58.2%	551.6	78.7	-85.7%

Table 8.7: SAM and average measured traversal cost m of our test scenes for kd-tree construction with normal (N) and localized binning (L). Relative differences of the costs are shown as well.

8.5.5 Localized Binning

A comparison of tree quality and traversal cost for kd-trees constructed with normal and localized binning is provided in Table 8.7. Measured traversal costs correlate more or less well with these SAM cost improvements. We achieve 8.1% and 40.6% smaller SAM costs for *Powerplant* and *Boeing* with slightly higher traversal cost reductions of 10.6% and 41.6%. SAM cost improvements of 10.4% and 58.2% for *Boeing* and *David* are less correlated with the much higher traversal cost reductions of 23.6% and 85.7%. Our results show that localized binning can yield huge SAM cost improvements for binned kd-tree construction without significant implementation overhead. This allows to use lower bin counts, which reduces resource overhead.

8.5.6 SAH Improvement Threshold

We evaluated the effects of applying the improvement threshold described in Section 8.3.4 to kd-tree and BVH construction. Therefore we made several construction runs with thresholds ranging from $\tau = 0\%$ to $\tau = 20\%$ in 5% steps for all scenes. For each threshold we measured construction time, SAM cost, traversal cost m and tree size. For kd-trees we additionally measured the number of primitive duplicates. A BVH node is 64 bytes in size. 48 bytes are needed for a pair of children bounds, 8 bytes for a pair of child pointers, and 8 bytes for a pair of triangle counts in case a child is a leaf. A kd-tree node is 16 bytes in size. 4 bytes are used for the split plane, 4 bytes for encoding the node type (inner node or leaf) and split dimension/triangle count, and 8 bytes for a pair of child pointers. Construction was performed using a single GTX 980 GPU. Results are shown in Table 8.8. Figure 8.9 additionally gives relative differences of the various measured quantities to the case $\tau = 0\%$.

Results are mixed. In case of BVH construction we were able to push τ to 10% without noticing significant changes in traversal cost when not considering *Boeing*. For *Atlas* and *David* τ can even be pushed to $\tau = 20\%$ without major traversal cost reductions. At the same time their BVHs are more than 30% smaller at this threshold. *Boeing* shows a much more dramatic development. Already with $\tau = 5\%$ SAM cost more than doubles. With only a 9% increase traversal cost does not follow this SAM cost explosion. Still this increase is already higher than for the other scenes. BVH analysis showed that construction created a leaf with 16K triangle primitives with almost 1% of the scene bounds surface area which has a high contribution to the SAM cost. As not all rays visited this leaf the average traversal cost increase is not as high. Already for $\tau = 0\%$ there is a leaf with

	Powerplant kd-tree					Powerplant BVH			
τ(%)	Build(s)	SAM	m	Dupl.	Size(MB)	Build(ms)	SAM	m	Size(MB)
0	2.7	93.5	97.4	537%	387	891	55.0	101.3	331
5	2.2	97.8	100.5	418%	304	880	55.4	102.0	304
10	1.8	102.1	106.4	325%	275	846	58.5	105.1	275
15	1.4	111.0	119.4	243%	247	840	60.8	106.8	247
20	1.1	138.0	137.2	161%	175	760	69.6	112.0	175

	Boeing 777 kd-tree					Boeing 777 BVH			
τ(%)	Build(s)	SAM	m	Dupl.	Size(GB)	Build(s)	SAM	m	Size(GB)
0	101.0	142.8	127.3	560%	9.0	30.6	130.7	186.6	7.8
5	86.5	147.6	132.3	440%	6.2	29.5	279.8	203.2	7.1
10	73.6	155.4	140.8	340%	4.0	28.1	441.2	356.4	6.4
15	65.1	166.9	153.1	252%	2.4	27.3	1.1k	2.8k	5.6
20	56.4	194.0	176.1	175%	1.3	27.0	9.4k	6.7k	4.5

	Atlas kd-tree					Atlas BVH			
τ(%)	Build(s)	SAM	m	Dupl.	Size(GB)	Build(s)	SAM	m	Size(GB)
0	110.5	103.3	78.2	267%	11.1	41.5	38.4	44.4	14.7
5	93.5	105.9	81.3	215%	7.9	39.6	39.2	45.0	13.8
10	79.4	111.6	89.1	173%	5.5	39.5	39.2	45.0	13.0
15	70.2	123.1	106.6	137%	3.7	39.4	39.3	45.1	11.6
20	62.1	135.0	120.4	103%	2.3	37.2	40.2	45.7	9.6

	David kd-tree					David BVH			
τ(%)	Build(s)	SAM	m	Dupl.	Size(GB)	Build(s)	SAM	m	Size(GB)
0	228.4	79.7	78.7	271%	20.8	84.6	29.8	42.9	27.1
5	189.2	82.2	85.4	219%	14.9	84.0	30.4	43.2	25.4
10	162.6	88.0	90.0	176%	10.4	82.3	30.4	43.2	23.7
15	142.4	94.5	100.2	138%	6.9	79.5	30.4	43.3	21.3
20	127.1	104.1	110.9	105%	4.3	77.1	31.3	43.7	17.5

Table 8.8: Effects of the improvement threshold on kd-tree and BVH construction time, tree quality, tree size and measured traversal cost m for all test scenes. The relative amount of triangle duplicates is included for kd-tree construction.

2K triangle primitives with 0.1% scene bounds surface area which hints at BVH construction having some trouble in finding good partition candidates for this scene. This may be caused by the fact that we inserted the Boeing scene into an environmental light box which is made from 12 triangle primitives. This increased the non-uniformity in triangle sizes. An SBVH (see Stich et al. [2009]) would have performed much better in this case. At $\tau = 10\%$ also traversal cost explodes. Along with these incoherent observations construction time and more so tree size decreases steadily but slowly with increasing thresholds for all scenes. Thus, while our improvement threshold seems to be able to trade some traversal performance for acceptable reductions in tree size, it is unclear if it is applicable in general considering the result from *Boeing*. A larger number of out-of-core scenes would be required to make a reliable statement.

In case of kd-tree construction all measurements are more sensitive to the improvement threshold. At $\tau = 5\%$ SAM and traversal cost on average increased by 5% for all scenes. At the same time tree size is reduced by 25% to 30% and the number of duplicates by at least 20%. At $\tau = 10\%$ SAM and traversal cost is already increased by at least 10%

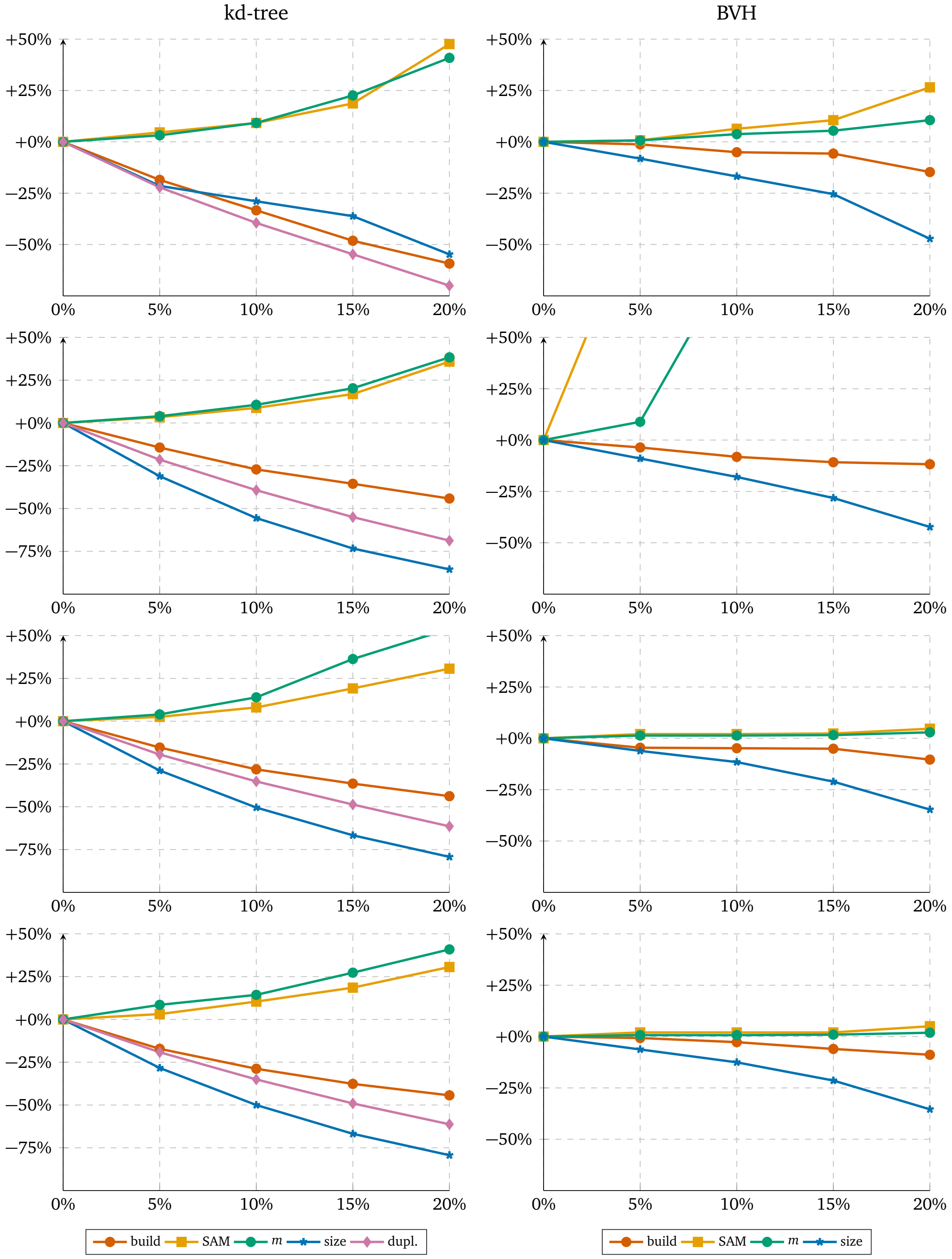


Figure 8.9: Relative differences of the quantities from Table 8.8 to the case of an improvement threshold of 0% for kd-tree (left column) and BVH (right column) construction. The x-axis depicts the improvement threshold, while the y-axis depicts the relative quantity difference. From the top row to the bottom row the scenes are *Powerplant*, *Boeing*, *Atlas*, and *David*.

and increases slightly superlinearly for higher τ. Except for *Powerplant* tree size is halved and duplicates reduced by at least 30%. While larger τ significantly reduce tree size and the number of duplicates, performance decreases steadily. Cost explosions for *Boeing* do not occur as spatial splits can better deal with non-uniform geometry. Thus, for kd-trees the improvement threshold more predictably allows to trade traversal performance for much smaller tree sizes and duplicate counts. Thresholds smaller than 5% should result in very minor traversal cost increases with already noticeable size reductions.

8.6 Summary and Discussion

We presented two approaches for full SAH-based top-down parallel out-of-core BVH and kd-tree construction that also scale with multiple GPUs. The performance of our BVH construction implementation with one GPU was comparable to the highly optimized and vectorized BVH builder of the Embree library running on a deca-core processor. Using binned construction allowed us to easily parallelize construction in the upper tree levels across multiple GPUs. As soon as subtrees are small enough to be completely processed by a single GPU, the sheer amount of such subtrees still allowed us to occupy several GPUs. We showed, that it is possible to construct high quality kd-trees even with the binned approach at relatively low bin counts. Key to this is our localized binning approach.

Traversal cost of the kd-tree was lower than with a BVH for the more complex shaped Powerplant and Boeing scenes. Performance was inferior for Atlas and David. Both scenes have a lot of empty space. Thus, most certainly this was due to the better empty space cut-off capabilities of the BVH. Implementing empty space cut-off techniques as for example described in Wald and Havran [2006] probably would have shifted odds again in favor of kd-trees. We also used a higher traversal cost constant for kd-tree construction to prevent excessive splitting. Thus, measured traversal cost would be even lower with the lower constant. At the same time the tree memory footprint would be much higher and should be larger than the BVH memory for all scenes. To reduce this overhead we proposed an SAH improvement threshold for tree construction. A threshold of up to 10% allowed to reduce tree size by more than 50% and duplicates by at least 30%. At the same time, rendering performance also decreased by at least 5%. Application of the threshold to BVH construction displayed a much stronger scene dependence. Results ranged from almost no performance reduction to gradual degradation, or sudden severe degradation.

We also compared tree quality with a hybrid construction approach. Traversal costs were 13% and 4% higher for Powerplant and Boeing with hybrid construction. For Atlas and David the full top-down and the hybrid approach essentially produced the same result as a simple top-down spatial median split construction.

8.6.1 Future Work

A future avenue would be to combine our BVH and kd-tree algorithms to construct an out-of-core SBVH algorithm (see Stich et al. [2009]). This lets us take advantage of the smaller memory footprint and construction time of BVHs, and quality increasing splits of kd-trees. Presumably, application of the improvement threshold would also give better results, similar to the kd-tree. A possibility to reduce memory overhead from duplicates during construction was sketched in Section 8.3.3. Another direction would be to find an efficient out-of-core algorithm for the BVH construction approach from Karras and Aila

[2013] (see Section 2.5.5). They perform local BVH optimizations on a fast to construct low quality BVH in parallel. The resulting BVHs have almost the same traversal performance as a full SAH-based top-down construction but can be constructed much faster. Mapping this algorithm to huge datasets should pose a difficult challenge when the BVH does not fit into GPU memory. It would also be interesting to see if the local tree optimizations are still effective for much larger trees.

Chapter 9

Final Summary and Discussion

Contents

In this dissertation we presented several contributions on bounding volume hierarchies (BVH) and kd-trees in the context of ray tracing. We now proceed with a summary and discussion of the relevant chapters, and conclude with discussions of overall open problems and future work in the final chapter.

9.1 Summary

The overarching problem of chapters four to six was the construction of BVHs with higher quality. Higher quality was meant in the sense of BVHs with a measurable lower average number of node and primitive intersection tests. A good indicator for hierarchy quality is the surface area metric (SAM). Construction strategies based on the surface heuristic (SAH) are able to build BVHs with lower SAM cost than other strategies.

SAM and SAH both use a geometric conditional probability for intersecting the bounds of a node. In Chapter 4 we attempted to improve this probability function in order to improve SAM and SAH, as well. Our function definition is based on the observation that the conditional probability of intersecting a convex body A with a random ray given that we intersected another convex body B which contains A depends on the ray direction. This probability can expose strong directional variance. In contrast to the conventional probability function p_c used for SAM and SAH our alternative function p_a included this directional dependence in its derivation. Except for certain types of bounding volumes such as spheres or cubes, where p_a directly simplifies to p_c, our probability cannot be evaluated analytically. We identified cases where p_a degenerates to the conventional probability. In general, we found that p_a and p_c did not have a large absolute difference in our experiments. Ultimately, this also explained why SAH-based construction with p_a resulted in essentially the same BVHs than construction with p_c.

Additionally, we noted that for random rays the outer body is more likely to be intersected from some directions than from others, which effectively changes the probability distribution of random ray directions. We were able to show that p_a simplifies to p_c when this effective non-uniform ray direction distribution is factored into the derivation of p_a. This revealed an alternative derivation for the conventional probability p_c which allowed to interpret p_c as accounting for directional variation. This property of p_c has been unrecognized so far and might partially explain the lasting success of the surface area heuristic and metric.

Aila et al. [2013] found that SAM is less accurate in predicting the traversal performance of BVHs then predicting the traversal performance of kd-trees, as it does not sufficiently capture the extra BVH traversal costs caused by node overlap. BVH construction has to minimize SAM and EPO cost of a BVH to be sure that measured traversal cost is reduced as well. In Chapter 5 we attempted to improve traversal performance by constructing BVHs with lower SAM and EPO cost. To achieve this, we examined three construction algorithms that use the SAM cost of temporarily constructed SAH-built BVHs to guide BVH construction. This was motivated by research from Aila et al. [2013] which has shown that greedy top-down SAH builders construct BVHs with superior traversal performance despite the fact that the resulting SAM costs are higher than those created by more sophisticated builders. This comes from the innate property of the top-down builder to implicitly minimize the EPO metric. This property allowed us to implicitly guide construction to choose candidates, which produce lower EPO, as the temporary BVHs have also been constructed with the greedy top-down construction strategy. Our constructed BVHs gave a significant increase in trace performance. We made several observations where temporary BVH construction with spatial splits clearly built BVHs with the lowest SAM and EPO cost, but other construction algorithms gave higher measured traversal performance. These observations suggest, that the SAM-EPO predictor is not sufficient. While our approach is not suitable for real-time BVH construction, we have shown that the proposed algorithm has subquadratic computational complexity in the number of primitives, which still renders it usable in practical applications.

Additionally, we have shown that the uncentered Pearson correlation is more suitable for computation of α values for the SAM-EPO ray tracing performance predictor than the centered Pearson correlation as used by Aila et al. [2013]. The centered Pearson correlation violates the proportionality assumption between predicted and measured traversal performance. Though the SAM-EPO predictor was initially designed for diffuse rays our improvement in the forecasting abilities also increased its relevance for primary rays.

In Chapter 6 we attempted to increase the quality of BVHs constructed with bottom-up approaches. The highest quality bottom-up approach is agglomerative clustering, a greedy bottom-up algorithm from Walter et al. [2008]. This algorithm is guided by the intuition that always clustering nodes which give the cluster bounds with the lowest surface area should be beneficial w.r.t. the SAM cost of the resulting tree. We presented a fully SAM-driven approach for agglomerative clustering. For this, we developed a SAM cost function for the cost of tracing a forest of BVHs. From this function we derived two clustering strategies. The first strategy clusters nodes where the cluster has lowest SAM cost. The second strategy clusters nodes which give the largest reduction in the forest cost. Both strategies allow to naturally create leaves as a clustering decision, which was not possible with the original approach in a meaningful way. While our first strategy proved to be detrimental for tree quality, our second strategy on average produced better BVHs than the original algorithm. This is overshadowed by the fact that the original algorithm and our clustering based algorithms performed worse than standard top-down plane sweeping in two thirds of the test scenes. Most of these scenes suffered from severe performance degradations. Aila et al. [2013] made similar observations for the original algorithm. We found that the bottom-up approach in general creates larger EPO costs. The greedy bottom-up process favors the creation of inner nodes where the children are closer together, risking overlap. Also, clustering decisions in lower levels do not take into account consequences regarding overlap in the upper tree levels. Greedy top-down sweep construction tries to create inner nodes where the children have small bounds, but by construction are also separated. As we have seen in Chapter 5 the greedy top-down algorithm from Popov et al. [2009] also suffered from high overlap as it allowed any kind of object split as long as the SAH cost is smaller.

Chapter 7 moved the focus to traversal performance with parallel hardware. We investigated how different BVH tree and node memory layouts in different memory areas impacted the ray tracing performance when tracing incoherent rays of a GPU path tracer. We also optimized the BVH layout using information gathered in a pre-processing pass by applying a number of different BVH reordering techniques. Depending on the memory area chosen for the BVH nodes and scene complexity, we achieved moderate speedups. The overall observation we made is that node layout had the largest impact on performance. In our extended data collection (see Schulz et al. [2013]) we could see that this is true for different GPU architectures, and that the best layout also can differ between architectures. Similar to Aila et al. [2012] we also observed that accessing the BVH via texture memory results in a significant performance boost. When assessing nodes via texture memory node layout was still important, but the chosen tree layout had a smaller influence on performance.

Finally, in Chapter 8 we focused on the problem of high quality acceleration structures in an out-of-core context. We presented a multi-GPU full top-down SAH-based BVH and kd-tree construction approach for scenes, which are of production rendering size and thus are by far several times larger than GPU memory. Existing GPU approaches for full top-down construction require geometry and the acceleration structure to completely fit on a single GPU. Out-of-core GPU approaches perform hybrid bottom-up top-down construction which suffers from reduced acceleration structure quality in the critical upper levels of the tree. To handle the massive amount of geometry we split it into chunks. Using binned SAH-based construction allowed to bin chunks independently from each other on multiple GPUs, but some inter GPU synchronization was necessary to merge the binning results in upper tree levels. The slight quality loss from binning was compensated with higher bin counts and a localized binning strategy in case of kd-trees. A partial-DFS-BFS node processing order exploited different levels of inter- and intra-GPU parallelism, and aimed at rapidly reducing the transient memory footprint. With a single commodity GPU we achieved comparable performance to a high-end deca-core CPU. Speedup efficiency was above 70% with four GPUs and stayed above 55% with eight GPUs.

9.2 Discussion

In this dissertation we managed to greatly improve BVH quality w.r.t. measurable traversal performance. The major problem with research on BVH quality is that is not possible to determine how close results are to the optimum. Unfortunately, when using SAM as the gold standard for BVH quality the construction of ground truth BVHs, which are optimal w.r.t. SAM, is highly unfeasible. The situation is complicated further by the results from Aila et al. [2013] on the necessity of the EPO metric to more accurately predict BVH traversal performance. Our own results from Chapter 5 and Chapter 6 mostly support the validity of the EPO metric. Constructing SAM-EPO metric optimal ground truth BVHs should be even more unfeasible than constructing SAM optimal BVHs. The scene dependence of the required α value is problematic, too.

Our construction approach in Chapter 5 successfully relied on the implicit EPO reduction of greedy top-down SAH-based plane-sweeping construction which has been observed by Aila et al. [2013] to construct lower SAM-EPO metric cost BVHs. The more general geometric object splits from Popov et al. [2009], which we examined and discussed in Chapter 5, and the bottom-up agglomerative clustering construction strategy from Walter et al. [2008] with and without our SAM-based clustering heuristic in Chapter 6 turned out to be harmful for SAM and EPO. In the light of this, future research should perhaps concentrate on construction strategies, which only allow object partitions that can be associated with separating planes. This has two interesting aspects. First, this might allow to again use SAM alone as the gold standard for quality by simply relying on the implicit EPO reduction of this split type. Secondly, this construction strategy limits the number of different BVHs that can be constructed. Perhaps it is possible to develop a minimum-SAM algorithm akin to the approach from Karras and Aila [2013], which only considers the plane constrained partitions. Hopefully this algorithm might have a lower computational and space complexity. Even in case that the complexities are unpractical, they might be small enough to create ground truth BVHs for larger scenes than Karras and Aila's approach. Being able to create a ground truth BVH for small models such as the *Sibenik* scene with its 80K primitives should already be sufficient to assess if more research on BVH quality is worthwhile. It would also be interesting to see if the standard SBVH algorithm already produces BVHs, which are better than the optimal purely object split-based BVH.

Considering the accuracy of the SAM-EPO predictor the question arises whether it is worthwhile to further improve on the conditional intersection probability function of SAM as we attempted in Chapter 4. Our results indicate that the conventional probability function is already sufficient. For diffuse rays the lowest and highest mean absolute percentage error of the predictor we measured in Chapter 5, Table 5.1 was just 2.5% and 8.4%, respectively. Even in the purely hypothetical case that this error could be completely explained by a better intersection probability function, there is not much room left for improvement.

Our higher quality BVHs should also complement our efforts on improving the cache behavior of tracing incoherent rays on GPUs. The lower average number of traversed nodes should reduce the pressure on GPU memory caches. Regardless of the memory area we used for node access we observed that our best tree and node layout combinations mainly increased performance by improving the achieved cache bandwidth. At the same time the cache hit rate and the ratio of required to loaded data stayed the same, which raises the question what caused the bandwidth increase. Observations like this one and disclosed intricate or exposed GPU properties probably will change with every new GPU generation. This calls for an auto tuning approach, that optimally adapts the tree and node layouts to the current hardware platform. Section 10.7 in the future work chapter provides a sketch for this. Our techniques would clearly benefit from ray reordering techniques to extract ray coherence from ray batches. A more optimal approach would be to actually build a more ray tracing friendly GPU with the architecture properties proposed by Aila and Karras [2010]. Their additionally proposed ray traversal algorithm has a high cache efficiency, which is insensitive to the ray coherence properties of a batch of rays. Unfortunately, such a GPU still has not materialized.

Huge data parallel problems where not all data fits into graphics memory can pose a serious challenge for GPUs if the processing of data chunks takes less time than loading data chunks on the GPU. Higher quality BVH and kd-tree construction mostly comes at the cost of higher construction time from more computations. This can make out-of-core construction of higher quality acceleration structures on GPUs more attractive. We have shown that SAH-based construction already offers enough computational intensity to efficiently overlap computations with onloading and offloading of data which resulted in competing performance of a consumer GPU with a high-end server CPU.

While our RSAH-based construction algorithms are more compute intensive than pure SAH-based construction, they are less suitable for out-of-core GPU construction. Though it is not necessary to explicitly store temporary BVHs, their construction requires additional memory from temporary primitive references. Their memory footprint can exceed the amount of available GPU memory and increase the amount of transient data that has to be managed. Construction on multiple GPUs should involve more synchronization. Working on cheap subtree cost approximations for RSAH-based construction which do not require the construction of temporary BVHs would be desirable in general but might also be readily applied to our out-of-core GPU-based construction approach. A candidate for such an approximation might be an experimental heuristic we present in Section 10.9.

Chapter 10

Future Work

Contents

Chapters 5 to 8 already discussed possible directions for future work on their respective subject matter. We briefly recapitulate the most promising suggested directions in an organized concise manner and also include overall open questions and future work.

10.1 Possible SAM-EPO Metric Insufficiency

For the agglomerative clustering-based algorithms in Chapter 6 and to a much higher degree for the RSSBVH algorithm in Chapter 5 we observed the situation where an algorithm constructs a BVH with lower SAM *and* EPO cost than another algorithm, but gives a higher measured traversal cost. RSSBVH in general performed worse than expected from its achieved SAM and EPO reductions. We concluded, that there must be an unidentified effect which is not captured in the SAM-EPO metric and might be related to the spatial splits applied by RSSBVH. Additional primitive duplicates seem not to be the cause, as we observed only slightly more or even less duplicates for RSSBVH compared to the similarly performing RSBVH algorithm. The agglomerative clustering-based algorithms do not perform spatial splits. Thus, our observations for these algorithms might hint at an additional uncaptured effect, or spatial splits are not the issue. But due to the low severity of violation of these algorithms the observations might also just be random. Finding an explanation or perhaps a new BVH quality metric which can explain the observed discrepancies at least for RSSBVH is an interesting direction for future work.

10.2 Explicit EPO Reduction

Our RSAH-based algorithms rely on the implicit EPO minimization of SAH-based top-down BVH construction to reduce EPO. Our approach allows to directly include EPO into the construction process. We can readily compute EPO of a candidate partition from the temporarily built BVHs combined with the node to split. This would allow us to directly use the SAM-EPO traversal cost predictor to guide construction. This requires prior knowledge of the scene dependent α value needed for the predictor which usually is unknown. According to experiments from Aila et al. [2013] using a fixed α ($\alpha = 0.71$ in their case) can give acceptable predictions. Alternatively, fast and accurate determination of α for unknown scenes is also an interesting problem. While construction should be much more expensive with explicit EPO reduction, it would still be interesting to see the improvement in traversal performance.

10.3 RSAH and the LCV Metric

In addition to the EPO metric, Aila et al. [2013] also proposed the leaf count variability (LCV) metric, which in a convex combination with SAH and EPO can explain SIMD performance of their GPU ray tracing experiments. They found that top-down greedy SAH-based construction also implicitly reduces LCV. This property should be naturally inherited and boosted by our RSAH-based algorithms. Thus, BVHs constructed with our RSAH algorithms might also be specially suited for SIMD traversal. Experimental validation of this conjecture is left for future work.

10.4 Treelet-based BVH Optimization with RBVH

In Section 2.5.5 we introduced the fast and parallel high quality BVH construction algorithm from Karras and Aila [2013], which first constructs a cheap-to-build low quality BVH, which then is post-processed in a fast and parallel optimization step to yield a high quality BVH. The resulting BVHs achieve quality close to greedy top-down SAH-based construction in much less time. The optimization step computes locally optimal (w.r.t. SAM) sub-hierarchies on treelet partitions of the low quality BVH. As their minimum-SAM BVH construction algorithm has $\Omega(\exp n)$ computational and $O(\exp n)$ memory space complexity only small treelet sizes are practical. We suspect that the local optimal solutions probably are not optimal w.r.t. EPO. Though our RBVH algorithm does not construct minimum-SAM BVHs, its computational complexity and space requirements allows for much larger treelets. Combined with its high-quality lower-EPO output it would be interesting to see what can be achieved. Ideally, results have higher quality than standard SAH-based construction in much less time.

10.5 Including Ray Termination into BVH Construction

In Section 2.5.8 we briefly discussed the RTSAH metric from Ize and Hansen [2011] which gives a measure for the expected traversal cost of BVHs and kd-trees with occlusion rays taking into account that rays can terminate during traversal. Without experimental validation the authors claim that their metric is valid for intersection rays as well. They also do not present an RTSAH-motivated construction algorithm. Our ΔSAGGLO and RSAH-based algorithms can easily be modified to include RTSAH. RTSAH can be directly integrated into our forest cost function of ΔSAGGLO and the bottom-up construction allows to conveniently compute RTSAH cost of cluster candidates on-the-fly. We still expect the resulting BVHs from ΔSAGGLO to suffer from high overlap in the upper tree levels. All our RSAH-based algorithms can be directly turned into RTSAH-based algorithms by computing the RTSAH metric on the temporary subtrees to compute RTSAH partition candidate scores. Under the assumption that the RTSAH metric is indeed usable for ordinary intersection rays it would be interesting to see the improvement in traversal performance.

10.6 Predictive Power of the RTSAH Metric

In the light of the previous section it is an interesting question how well RTSAH performs in predicting traversal performance compared to the SAM-EPO metric, especially as RTSAH does not require a scene dependent constant. As we already had our set of constructed BVHs available from our RSAH experiments in Chapter 5 we simply computed RTSAH for them and computed the mean absolute percentage error (MAPE) of the predicted and measured speedups. Ize and Hansen [2011] provided an adapted RTSAH version for closest intersection rays (called *radiance rays* in their publication) which we used. Preliminary results are collected in Table B.1, Appendix B. We also included the conventional SAM and the combined SAM-EPO predictor with α values determined with the uncentered Pearson correlation.

Though there are exceptions, at least in our experiments RTSAH clearly has a higher average MAPE than the conventional SAM, and the SAM-EPO predictor performed best. RTSAH assumes that rays always terminate in leaves. Suspecting this might be too simplifying we tried a simple approximation of the actual termination probability. For this we computed the expected number of intersected leaf primitives as the sum of the two sided primitive areas divided by the leaf bounds surface area and clamped the result to one. For leaves with a single primitive or a tessellated planar surface this actually gives the correct termination probability w.r.t. the ray distribution underlying the SAM. RTSAH with our modified leaf occlusion computation is included as RTSAH+ in Table B.1. RTSAH+ clearly improved on RTSAH but still has higher MAPE than SAM. SAM can be derived from RTSAH by assuming that rays never terminate. In the light of this it is unclear if a more accurate termination probability can improve the result further, or if RTSAH generally is inferior to SAM and RTSAH+ simply interpolates between both metrics. Whereas RTSAH treats leaves like solid blocks, RTSAH+ treats leaves like they are filled with a gas that probabilistically terminates rays. A more accurate approximation should probably at least distinguish between ray termination in- and outside of node overlap. An improved RTSAH metric or completely different termination-based metric is left for future work.

10.7 BVH Tree and Node Layout Auto-Tuning

In Chapter 7 the optimal BVH tree and node layout heavily depended on the chosen GPU architecture. For a ray traversal implementation to always give optimal performance with future hardware a layout auto-tuning approach should be followed. For this a small set of benchmark scenes and benchmark ray sets should be provided by the application. Then, performance of different tree and node layouts in different memory areas would be profiled. Different layouts and memory areas could also be tried for the geometry. This increases the search space further. It should also be beneficial to provide dedicated kernels for intersection/occlusion rays and coherent/incoherent rays which have to be auto-tuned separately. To avoid an exhaustive search the techniques proposed by Weber and Goesele [2016] might be applied.

10.8 Bounding Volume Graph

It would be interesting to investigate if it is possible to adapt the concept of the graph-based acceleration structure from Gribble and Naveros [2013] (see Section 2.4) to BVHs as this might eliminate the need for a traversal stack without having to process nodes several times. Other stackless BVH traversal algorithms either only allow a specific traversal order or have to process BVH nodes several times. Gribble and Naveros [2013] derived the graph from the spatial scene decomposition from a kd-tree by converting leaves into sectors which store lists of adjacent sectors. In case of ambiguity inner nodes of the kd-tree which can resolve the ambiguity are referenced. The spatial decomposition allowed to uniquely identify which node in the graph to process next and guaranteed traversal progress. As the union of all leaf volumes in a BVH is not guaranteed to cover the whole volume of the scene bounds there can be leaves, which are adjacent to uncovered space. Further, leaf nodes can overlap. Both aspects make it difficult to identify which node to process next during traversal. One possibility might be to analyze the octants defined by the bounds of leaves by combing the half spaces of the bounding planes of three different dimensions. Then each octant references the deepest common ancestor node of all leaves the octant overlaps. At traversal time the ray chooses the appropriate octant depending on the ray direction and resolves the closest leaf. Additional logic is required to avoid cycles during traversal. If this sketch combined with some additional modifications indeed results in a working algorithm, it has to be seen if it is competitive due to the possibly excessive handling of ambiguity.

10.9 An Experimental Alternative Surface Area Heuristic

Motivated by the discussions in Section 5.6.2 and Section 6.4 we developed an alternative experimental surface area heuristic for greedy top-down BVH construction. It approximates the subtree cost of partition candidates, is less blind to the immediate cost caused by the two new partition nodes, and at the same time is simple and cheap to evaluate. According to the iterative definition in Equation 2.33 of the SAM cost of a BVH the cost can be split into the summed cost $c_{\mathcal{I}} = c_t \sum_{n \in \mathcal{I}} p_n^{root}$ of all inner nodes $\mathcal{I}$ and the summed cost $c_{\mathcal{L}} = c_i \sum_{n \in \mathcal{L}} p_n^{root} |n|$ of all leaves. When creating a partition candidate we approximate the cost of the subtrees on each partition side by assuming that all primitives will end up in separate leaves. This allows us to directly compute the BVH leaf cost $c_{\mathcal{L}}$ for the left side l and right side r of a partition as an approximation for their subtree cost. Leaving out the common $\frac{1}{A_{root}}$ area factor from the scene bounds in the geometric probability and the c_i constant this results in sum of primitive bounds areas $c_l = \sum_{p \in \mathcal{P}_l} A_p$ for the subtree cost approximation of the left side with its set of primitives $\mathcal{P}_l$. The subtree cost approximation for the right side is analogous. We could add the partition subtree root bounds A_l to c_l to improve the approximation. But this would cause A_l to get lost in c_l though A_l actually contributes to the cost of the BVH as discussed in Section 5.6.2. Thus, we decided to include A_l by multiplying c_l with A_l. Putting everything together our experimental surface area heuristic (XSAH) is defined as

$$c_{split} = A_l \left(\sum_{p \in \mathcal{P}_l} A_p \right) + A_r \left(\sum_{p \in \mathcal{P}_r} A_p \right). \tag{10.1}$$

Only little effort is required to implement XSAH in a greedy SAH-based top-down builder. We only use the heuristic for best split candidate determination. To decide whether to create a leaf node we compute the conventional SAH cost for the best split determined with XSAH and compare against the conventional leaf cost.

We conducted preliminary experiments with top-down sweep construction with XSAH on the scenes from Chapter 5. In Appendix C average results are shown in Table C.1 and separately for each scene in Table C.2 where the algorithm is denoted XBVH. The standard top-down sweeping construction (BBVH) algorithm and RBVH are included for comparison. Except for *Powerplant* and *Rungholt*, which in general proved to be problematic scenes, XBVH always performed better than BBVH. In some scenes the result is also close to RBVH. Considering that XBVH is essentially as simple to compute as BBVH this is a good result. Identifying why it failed in *Powerplant* might result in an improved heuristic. But more scenes should be evaluated to detect more problematic scenarios. It is also interesting to see, how XSAH performs with binned construction and the SBVH algorithm. As XSAH seems to be more sensitive to the size of the bounds of a partition it might be even able to improve the geometric splits from Popov et al. [2009]. Further, it might also be used to improve the quality of temporary subtrees in RSAH-based construction for overall higher quality. As a more or less accurate approximation to the RBVH algorithm XBVH might also be used for treelet-based BVH optimization with much larger treelets.

10.10 Out-of-Core BVH Optimization

Our presented out-of-core GPU BVH construction approach gave competitive construction performance compared with a deca-core processor. The treelet-based in-core construction approach from Karras and Aila [2013] is a promising alternative to top-down SAH based construction. To find an efficient out-of-core algorithm for this approach which can efficiently utilize multiple GPUs is an interesting direction for future work. Mapping this algorithm to huge datasets should pose a difficult challenge when the BVH does not fit into GPU memory. To fit into memory the BVH has to be split into chunks. The optimization is performed bottom-up or top-down in a clear chunk processing order. For a single GPU at least this should allow to precompute a chunk processing schedule which optimizes the asynchronous processing and on- and off-loading of required chunks with a chunk cache. Optimization treelets that cross several chunks increase implementational effort.

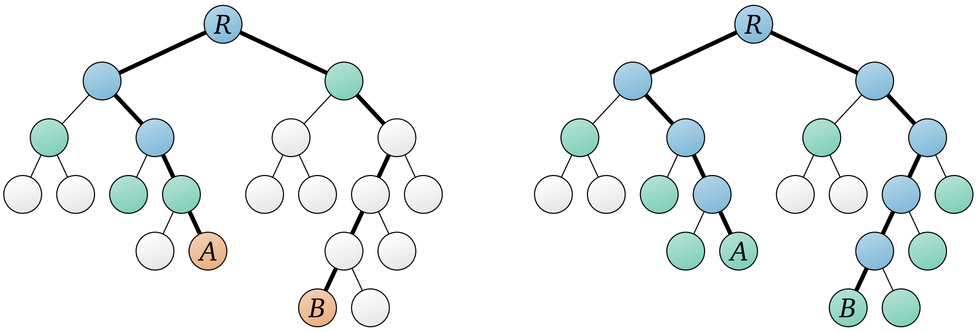


Figure 10.1: Example of a BVH where the two red leaves A and B are assumed to be close together in space and would greatly improve hierarchy quality if merged into a leaf, but at the same time are far apart in the hierarchy. Their closest common ancestor is R. In a corresponding undirected graph the nodes have a distance of nine. *Left:* The treelet size of four is smaller than nine. No treelet starting at R can contain A and B at the same time. Treelet optimization may decrease or increase the node distance. After several iterations the nodes may end up in the same treelet. This should be less likely for large BVHs in out-of-core settings. *Right:* The treelet size must be at least nine to be able to include both nodes. The subtrees of the treelet degenerate into lists in this case.

It would also be interesting to see if the local tree optimizations are still effective for much larger trees. If the distance d between nodes in a corresponding undirected graph is larger than the treelet size n they cannot end up in the same treelet for optimization to resolve unfavorable leaf constellations. This is a problem if both nodes are close together or even overlapping in space, but far away from each other in the tree. Spatial or object median split constructions can easily generate such constellations. Even in case the treelet size is large enough there is no guarantee for both nodes to end up in the treelet. Though several optimization iterations can change the distances between nodes this probably is less likely for tree depths encountered in out-of-core scenes. See Figure 10.1 for an illustration of this problem.

Appendix A

RSAH-based Construction Complexity

A.1 Naïve Sweep-Sweep Construction Complexity

In Section 5.2.1 we stated that an RSAH algorithm based on the common naïve sweep approach which sorts in every step has a complexity of $O\left(N^2 \log^2 N\right)$. In this appendix we provide a derivation for this result. The naïve top down SAH-based plane-sweeping construction algorithm implementation according to MacDonald and Booth [1990] sorts all node primitives every time it determines the best SAH-based split. This results in an overall complexity of $O\left(N \log^2 N\right)$. For a node with N primitives a sweep-sweep RSAH algorithm based on this implementation first has to sort its primitives in $O(N \log N)$ and then construct $2(N-1)$ temporary BVHs for its $N-1$ candidate partitions. This results in the following recurrence relation:

$$T(N) = N \log N + 2\left(\sum_{i=1}^{N-1} i \log^2 i\right) + 2T\left(\frac{N}{2}\right)$$

Using the upper bound $\sum_{i=1}^{N-1} i \log^2 i < N^2 \log^2 N$ we get:

$$\begin{aligned}
T(N) &< N \log N + 2N^2 \log^2 N + 2T\left(\frac{N}{2}\right) \\
&< N \log N + 2N^2 \log^2 N \\
&\quad + 2\left(\frac{N}{2} \log \frac{N}{2} + 2\left(\frac{N}{2}\right)^2 \log^2 \frac{N}{2} + 2T\left(\frac{N}{4}\right)\right) \\
&< 2 \sum_{i=0}^{\log N} 2^i \left(\frac{N}{2^i} \log \frac{N}{2^i} + \left(\frac{N}{2^i}\right)^2 \log^2 \frac{N}{2^i}\right)
\end{aligned}$$

At this point we drop the lower order addend and assume $N = 2^n, n \in \mathbb{N}$:

$$\begin{aligned}
T(N) &= 2\sum_{i=0}^{n} 2^i \left(\frac{N}{2^i}\right)^2 \log^2 \frac{2^n}{2^i} = 2N^2 \sum_{i=0}^{n} \frac{(n-i)^2}{2^i} \\
&= 2N^2 \left(n^2 \sum_{i=0}^{n} \frac{1}{2^i} - 2n \sum_{i=0}^{n} \frac{i}{2^i} + \sum_{i=0}^{n} \frac{i^2}{2^i}\right) \qquad \text{(A.1)} \\
&\rightarrow O\left(N^2 n^2\right) = O\left(N^2 \log^2 N\right)
\end{aligned}$$

A.2 Binning-Binning Construction Complexity

In this appendix, we derive the result from Equation 5.7, Section 5.2 in detail.

$$\begin{aligned}
T(N) =&2\left(\sum_{i=1}^{B} i\frac{N}{B}\log\left(i\frac{N}{B}\right)\right)+2T\left(\frac{N}{2}\right)\\
=&2\left(\sum_{i=1}^{B} i\frac{N}{B}\log\left(i\frac{N}{B}\right)\right)+\\
&4\left(\sum_{i=1}^{B} i\frac{N}{2B}\log\left(i\frac{N}{2B}\right)\right)+4T\left(\frac{N}{4}\right)\\
=&2\sum_{i=0}^{\log N} 2^i\left(\sum_{j=1}^{B} j\frac{N}{2^iB}\log\left(j\frac{N}{2^iB}\right)\right)\\
=&2\sum_{i=0}^{\log N}\frac{N}{B}\left(\sum_{j=1}^{B} j\log\left(j\frac{N}{2^iB}\right)\right)\\
=&2\sum_{i=0}^{\log N}\frac{N}{B}\left(\log H(B)+B\log N-iB-B\log B\right)
\end{aligned}$$

Here we use the upper bound $\log H(B) < B^2 \log B$.

$$\begin{aligned}
T(N) <&2\sum_{i=0}^{\log N}\frac{N}{B}\left(B^2\log B+B\log N-iB-B\log B\right)\\
=&2\sum_{i=0}^{\log N}\left(BN\log B+N\log N-iN-N\log B\right)\\
=&2\big(N\log^2 N+BN\log(B)\log(N)-\\
&O(N\log N)-O(\log^2 N)\big)\\
\in&O(N\log^2 N).
\end{aligned}\tag{A.2}$$

Appendix B

RTSAH Metric Speedup Prediction Experiments

	Average prediction MAPE (%)							
	Primary rays				Diffuse rays			
Scene	RTSAH	RTSAH+	SAM	SAM-EPO	RTSAH	RTSAH+	SAM	SAM-EPO
Babylon	32.3	24.4	14.6	**9.4**	27.5	19.8	10.0	**6.2**
Bubs	4.4	**3.7**	5.7	5.7	7.0	6.2	**4.6**	**4.6**
Conference	15.0	14.2	9.0	**8.9**	13.4	12.8	4.3	**2.5**
Epic	27.4	25.0	22.5	**10.9**	24.0	21.4	18.9	**8.4**
Fairy	**7.4**	**7.4**	7.9	8.0	5.1	5.0	4.5	**3.4**
Hairball	32.4	27.0	18.0	**13.2**	22.9	17.6	8.8	**2.8**
Powerplant	16.3	11.7	15.2	**4.1**	12.6	8.1	8.5	**3.4**
Rungholt	**2.2**	2.3	2.6	2.6	**2.0**	2.1	2.6	2.6
San Miguel	25.0	23.5	20.0	**9.0**	23.4	22.8	13.2	**7.8**
Sibenik	15.2	12.3	11.2	**3.5**	8.3	5.8	4.7	**3.1**
Soda	13.0	12.1	12.2	**9.7**	15.5	14.6	8.5	**4.6**
Sponza	22.2	19.3	16.7	**12.4**	18.7	16.3	9.8	**5.9**
Average	17.7	15.2	13.0	**8.1**	15.0	12.7	8.2	**4.6**

Table B.1: Preliminary results of the mean absolute percentage error (MAPE) of speedup prediction with the RTSAH, SAM, and SAM-EPO metrics. RTSAH results are computed from the BVHs obtained in Chapter 5. α values for the SAM-EPO metric correspond to the ones obtained with the uncentered Pearson correlation in Table 5.1. RTSAH+ uses a more accurate leaf termination probability described in Section 10.6.

Appendix C

Experimental Alternative Surface Area Heuristic Experiments

	Avg. (Min/Max) reduction (%)					
			Primary rays		Diffuse rays	
Algorithm	SAM	EPO	*p*	*m*	*p*	*m*
RBVH	-11.7 (-2.7/-33.3)	-22.2 (-1.4/-65.3)	-13.3 (-2.7/-33.3)	-10.4 (+0.9/-28.3)	-12.7 (-2.7/-33.3)	-8.9 (+0.4/-23.7)
XBVH	-7.4 (+0.8/-32.5)	-17.7 (+3.4/-64.4)	-9.2 (+1.3/-32.5)	-7.6 (+3.0/-22.3)	-8.5 (+1.1/-32.5)	-6.0 (+3.4/-19.7)

Table C.1: Average, minimum, and maximum reduction of SAM, EPO, as well as predicted (p) and measured (m) traversal cost of primary and diffuse rays over all preliminary results in Table C.2 relative to BBVH as baseline for RBVH and our experimental BVH builder (see Section 10.9) based on XSAH (XBVH).

					Primary rays		Diffuse rays	
Scene	Builder	Time	SAM	EPO	p	m	p	m
Babylon	RBVH	250.2	49.2	12.9	29.9	51.1	35.3	54.2
	XBVH	5.3	50.2	12.6	30.3	47.8	35.9	54.3
	BBVH	3.8	53.7	15.1	33.2	51.9	39.0	58.2
Bubs	RBVH	1500.2	16.2	2.9	16.2	32.0	16.2	36.5
	XBVH	24.2	16.3	3.0	16.3	34.6	16.3	38.4
	BBVH	16.5	24.2	8.4	24.2	44.5	24.2	47.9
Conference	RBVH	126.0	38.6	7.1	24.5	32.4	30.1	41.8
	XBVH	3.2	41.8	7.6	26.5	35.0	32.6	45.1
	BBVH	2.1	46.4	9.8	30.0	39.5	36.6	49.5
Epic	RBVH	237.5	19.5	6.0	10.6	69.2	11.5	69.2
	XBVH	4.2	19.5	6.4	10.9	70.2	11.7	72.4
	BBVH	3.3	21.3	7.1	12.0	72.6	12.8	73.1
Fairy	RBVH	111.0	31.5	3.0	8.7	40.0	12.2	47.2
	XBVH	2.1	31.9	2.7	8.5	41.0	12.2	47.2
	BBVH	1.4	33.4	3.4	9.4	43.4	13.1	49.3
Hairball	RBVH	1685.2	454.0	36.7	90.7	153.3	152.8	150.1
	XBVH	26.9	465.9	37.5	93.0	160.5	156.7	152.9
	BBVH	23.8	466.4	37.8	93.3	158.8	157.0	153.0
Powerplant	RBVH	96.7	41.1	13.0	27.9	60.8	33.1	60.3
	BBVH	2.0	43.9	13.2	29.5	62.9	35.2	63.2
	XBVH	2.5	44.3	13.6	29.9	64.8	35.6	65.3
Rungholt	BBVH	54.3	109.9	3.4	109.9	45.6	109.9	48.5
	XBVH	62.4	109.5	3.4	109.5	45.6	109.5	48.6
	RBVH	2432.7	105.5	2.7	105.5	46.0	105.5	48.7
San Miguel	RBVH	8985.2	17.3	7.5	13.1	95.8	14.5	95.0
	XBVH	167.6	18.4	8.3	14.0	94.6	15.5	98.0
	BBVH	130.5	20.3	10.2	16.0	109.5	17.4	104.9
Sibenik	RBVH	29.0	48.8	4.2	21.1	51.2	31.7	52.1
	XBVH	0.7	50.9	4.2	21.9	54.6	33.0	53.4
	BBVH	0.5	53.6	5.0	23.4	57.2	35.0	54.8
Soda	RBVH	1451.7	66.2	10.2	40.9	42.6	46.3	49.0
	XBVH	26.8	74.9	11.7	46.4	44.8	52.5	49.3
	BBVH	21.3	77.9	13.7	49.0	48.8	55.2	55.0
Sponza	RBVH	133.4	70.9	7.8	23.7	63.5	31.8	67.2
	XBVH	2.8	75.8	8.9	25.7	66.7	34.3	70.5
	BBVH	2.1	83.1	12.9	30.6	82.1	39.7	83.2

Table C.2: Preliminary results for our experimental BVH builder (see Section 10.9) based on XSAH (XBVH) and the scenes also used in Chapter 5. Results for RBVH and the baseline BBVH are included as well. p is the EPO-based measure for BVH performance (Equation 5.3) and m the average measured traversal cost (Equation 5.13). For each scene builders are sorted from smallest to largest m of diffuse rays.

(Co-)Authored Publications

Dominik Wodniok and Michael Goesele. Construction of Bounding Volume Hierarchies with SAH Cost Approximation on Temporary Subtrees. *Computers & Graphics*, 2017.

Dominik Wodniok and Michael Goesele. Recursive SAH-based Bounding Volume Hierarchy Construction. *Proceedings of Graphics Interface*, Victoria, BC, Canada, 2016.

Sven Widmer, **Dominik Wodniok**, Daniel Thul, Stefan Guthe, and Michael Goesele. Decoupled Space and Time Sampling of Motion and Defocus Blur for Unified Rendering of Transparent and Opaque Objects. *Proceedings of Pacific Graphics*, Okinawa, Japan, 2016.

Dominik Wodniok, André Schulz, Sven Widmer, and Michael Goesele. Analysis of cache behavior and performance of different BVH memory layouts for tracing incoherent rays. *Proceedings of the 13th Eurographics Symposium on Parallel Graphics and Visualization*, Girona, Spain, 2013.

André Schulz, **Dominik Wodniok**, Sven Widmer, and Michael Goesele. Extended Data Collection: Analysis of cache behavior and performance of different BVH memory layouts for tracing incoherent rays. *Technical report,* TU Darmstadt, 2013.

Sven Widmer, **Dominik Wodniok**, Nicolas Weber, and Michael Goesele. Fast dynamic memory allocator for massively parallel architectures. *Proceedings of the 6th Workshop on GPGPU,* Houston, TX, USA, 2013.

Bibliography

[Aila and Karras 2010] Aila, T. and T. Karras (2010). "Architecture Considerations for Tracing Incoherent Rays". In: *Proceedings of the Conference on High Performance Graphics*, pp. 113–122.

[Aila and Laine 2009] Aila, T. and S. Laine (2009). "Understanding the efficiency of ray traversal on GPUs". In: *Proceedings of the Conference on High Performance Graphics*, pp. 145–149.

[Aila et al. 2012] Aila, T., S. Laine, and T. Karras (2012). *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum*. NVIDIA Technical Report NVR-2012-02. NVIDIA Corporation.

[Aila et al. 2013] Aila, T., T. Karras, and S. Laine (2013). "On Quality Metrics of Bounding Volume Hierarchies". In: *Proceedings of the 5th High-Performance Graphics Conference*, pp. 101–107.

[Amanatides and Woo 1987] Amanatides, J. and A. Woo (1987). "A Fast Voxel Traversal Algorithm for Ray Tracing". In: *EG 1987-Technical Papers*, pp. 3–10.

[Antwerpen 2011] Antwerpen, D. van (2011). "Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU". In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pp. 41–50.

[Appel 1968] Appel, A. (1968). "Some Techniques for Shading Machine Renderings of Solids". In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, pp. 37–45.

[Arvo and Kirk 1989] Arvo, J. and D. Kirk (1989). "A Survey of Ray Tracing Acceleration Techniques". In: *An Introduction to Ray Tracing*. Academic Press Ltd., pp. 201–262.

[Ashikhmin and Shirley 2000] Ashikhmin, M. and P. Shirley (2000). "An anisotropic phong BRDF model". In: *Journal of graphics tools* 5.2, pp. 25–32.

[Baert et al. 2013] Baert, J., A. Lagae, and P. Dutré (2013). "Out-of-core construction of sparse voxel octrees". In: *Proceedings of the 5th High-Performance Graphics Conference*, pp. 27–32.

[Barber et al. 1996] Barber, C. B., D. P. Dobkin, and H. Huhdanpaa (1996). “The Quickhull Algorithm for Convex Hulls”. In: *ACM Trans. Math. Softw.* 22.4, pp. 469–483.

[Bender et al. 2002] Bender, M. A., E. D. Demaine, and M. Farach-Colton (2002). “Efficient Tree Layout in a Multilevel Memory Hierarchy”. In: *European Symposium on Algorithms*, pp. 165–173.

[Benthin et al. 2012] Benthin, C., I. Wald, S. Woop, M. Ernst, and W. R. Mark (2012). “Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.9, pp. 1438–1448.

[Bittner et al. 2015] Bittner, J., M. Hapala, and V. Havran (2015). “Incremental BVH construction for ray tracing”. In: *Computers & Graphics* 47, pp. 135–144.

[Boulos et al. 2007] Boulos, S., D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald (2007). “Packet-based whitted and distribution ray tracing”. In: *Proceedings of Graphics Interface 2007*, pp. 177–184.

[Boulos et al. 2008] Boulos, S., I. Wald, and C. Benthin (2008). “Adaptive ray packet reordering”. In: *IEEE Symposium on Interactive Ray Tracing, 2008.* Pp. 131–138.

[Cazals and Sbert 1997] Cazals, F. and M. Sbert (1997). *Some Integral Geometry Tools to Estimate the Complexity of 3D Scenes*. Research Report RR-3204. INRIA.

[Choi et al. 2010] Choi, B., R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart (2010). “Parallel SAH k-D tree construction”. In: *Proceedings of the Conference on High Performance Graphics*, pp. 77–86.

[Dammertz et al. 2008] Dammertz, H., J. Hanika, and A. Keller (2008). “Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays”. In: *Computer Graphics Forum* 27.4, pp. 1225–1233.

[Danilewski et al. 2010] Danilewski, P., S. Popov, and P. Slusallek (2010). *Binned SAH Kd-Tree Construction on a GPU*. Tech. rep. Saarland University.

[Dutre et al. 2006] Dutre, P., K. Bala, P. Bekaert, and P. Shirley (2006). *Advanced Global Illumination*. AK Peters Ltd.

[Emde Boas 1975] Emde Boas, P. van (1975). “Preserving order in a forest in less than logarithmic time”. In: *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pp. 75–84.

[Ernst and Greiner 2008] Ernst, M. and G. Greiner (2008). “Multi bounding volume hierarchies”. In: *IEEE Symposium on Interactive Ray Tracing, 2008.* Pp. 35–40.

[Es and İşler 2007] Es, A. and V. İşler (2007). “Accelerated Regular Grid Traversals Using Extended Anisotropic Chessboard Distance Fields on a Parallel Stream Processor”. In: *J. Parallel Distrib. Comput.* 67.11, pp. 1201–1217.

[Euler 1758] Euler, L. (1758). “Elementa doctrinae solidorum”. In: *Novi commentarii Academiae Scientiarum Imperialis Petropolitanae* 4, pp. 109–140.

[Fabianowski et al. 2009] Fabianowski, B., C. Fowler, and J. Dingliana (2009). “A Cost Metric for Scene-Interior Ray Origins”. In: *Eurographics 2009 - Short Papers*, pp. 49–52.

[Flynn 1972] Flynn, M. J. (1972). “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* 21.9, pp. 948–960.

[Fujimoto et al. 1986] Fujimoto, A., T. Tanaka, and K. Iwata (1986). “ARTS: Accelerated Ray-Tracing System”. In: *IEEE Computer Graphics and Applications* 6, pp. 16–26.

[Ganestam and Doggett 2016] Ganestam, P. and M. Doggett (2016). “SAH guided spatial split partitioning for fast BVH construction”. In: *Computer Graphics Forum* 35.2, pp. 285–293.

[Ganestam et al. 2015] Ganestam, P., R. Barringer, M. Doggett, and T. Akenine-Möller (2015). “Bonsai: rapid bounding volume hierarchy generation using mini trees”. In: *Journal of Computer Graphics Techniques* 4.3, pp. 23–42.

[Garanzha and Loop 2010] Garanzha, K. and C. T. Loop (2010). “Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing”. In: *Computer Graphics Forum* 29.2, pp. 289–298.

[Garanzha et al. 2011] Garanzha, K., J. Pantaleoni, and D. McAllister (2011). “Simpler and Faster HLBVH with Work Queues”. In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pp. 59–64.

[Gärtner 1999] Gärtner, B. (1999). “Fast and Robust Smallest Enclosing Balls”. In: *Proceedings of the 7th Annual European Symposium on Algorithms*, pp. 325–338.

[Gil and Itai 1999] Gil, J. and A. Itai (1999). “How to Pack Trees”. In: *Journal of Algorithms* 32.2, pp. 108–132.

[Goldsmith and Salmon 1987] Goldsmith, J. and J. Salmon (1987). “Automatic creation of object hierarchies for ray tracing”. In: *IEEE Computer Graphics and Applications* 7.5, pp. 14–20.

[Gottschalk et al. 1996] Gottschalk, S., M. C. Lin, and D. Manocha (1996). “OBBTree: A hierarchical structure for rapid interference detection”. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’96, pp. 171–180.

[Gribble and Naveros 2013] Gribble, C. and A. Naveros (2013). “GPU Ray Tracing with Rayforce”. In: *ACM SIGGRAPH 2013 Posters*, 98:1–98:1.

[Gribble et al. 2014] Gribble, C., A. Naveros, and E. Kerzner (2014). “Multi-Hit Ray Traversal”. In: *Journal of Computer Graphics Techniques* 3.1, pp. 1–17.

[Gu et al. 2013] Gu, Y., Y. He, K. Fatahalian, and G. Blelloch (2013). “Efficient BVH construction via approximate agglomerative clustering”. In: *Proceedings of the 5th High-Performance Graphics Conference*, pp. 81–88.

[Gunther et al. 2007] Gunther, J., S. Popov, H.-P. Seidel, and P. Slusallek (2007). “Realtime Ray Tracing on GPU with BVH-based Packet Traversal”. In: *2007 IEEE Symposium on Interactive Ray Tracing*, pp. 113–118.

[Hapala et al. 2011] Hapala, M., O. Karlík, and V. Havran (2011). “When It Makes Sense to Use Uniform Grids for Ray Tracing”. In: *Proceedings of WSCG’2011, communication papers*, pp. 193–200.

[Havran 1999] Havran, V. (1999). “Analysis of Cache Sensitive Representation for Binary Space Partitioning Trees”. In: *Informatica (Slovenia)* 23, pp. 203–210.

[Havran 2000] Havran, V. (2000). “Heuristic Ray Shooting Algorithms”. Ph.D. Thesis. Czech Technical University in Prague.

[Havran and Bittner 2002] Havran, V. and J. Bittner (2002). “On improving kd-trees for ray shooting”. In: *Journal of WSCG*, pp. 209–217.

[Havran et al. 2006] Havran, V., R. Herzog, and H.-P. Seidel (2006). “On the Fast Construction of Spatial Hierarchies for Ray Tracing”. In: *2006 IEEE Symposium on Interactive Ray Tracing*, pp. 71–80.

[Heckbert 1982] Heckbert, P. (1982). “Color Image Quantization for Frame Buffer Display”. In: *SIGGRAPH Computer Graphics* 16.3, pp. 297–307.

[Hermann et al. 2008] Hermann, E., F. Faure, and B. Raffin (2008). “Ray-traced collision detection for deformable bodies”. In: *GRAPP 2008-3rd International Conference on Computer Graphics Theory and Applications*, pp. 293–299.

[Hou et al. 2011] Hou, Q., X. Sun, K. Zhou, C. Lauterbach, and D. Manocha (2011). “Memory-Scalable GPU Spatial Hierarchy Construction”. In: *IEEE Transactions on Visualization and Computer Graphics* 17.4, pp. 466–474.

[Hulst 1981] Hulst, H. C. Van de (1981). *Light Scattering By Small Particles*. Dover Publications.

[Intel 2017] Intel (2017). *Intel Intrinsics Guide*. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`. [accessed: 2017.06.21].

[Ize and Hansen 2011] Ize, T. and C. Hansen (2011). “RTSAH Traversal Order for Occlusion Rays”. In: *Computer Graphics Forum* 30.2, pp. 297–305.

[Jakob 2010] Jakob, W. (2010). *Mitsuba renderer*. `http://www.mitsuba-renderer.org`. [accessed: 2016.07.19].

[Jensen 1996] Jensen, H. W. (1996). “Global Illumination Using Photon Maps”. In: *Proceedings of the Eurographics Workshop on Rendering Techniques ’96*, pp. 21–30.

[Kajiya 1986] Kajiya, J. T. (1986). “The Rendering Equation”. In: *SIGGRAPH ’86*, pp. 143–150.

[Karras 2012] Karras, T. (2012). “Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees”. In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, pp. 33–37.

[Karras and Aila 2013] Karras, T. and T. Aila (2013). “Fast Parallel Construction of High-Quality Bounding Volume Hierarchies”. In: *Proceedings of the 5th High-Performance Graphics Conference*, pp. 89–99.

[Kay and Kajiya 1986] Kay, T. L. and J. T. Kajiya (1986). “Ray Tracing Complex Scenes”. In: *SIGGRAPH Comput. Graph.* 20.4, pp. 269–278.

[Kendall and Moran 1963] Kendall, M. G. and P. A. P. Moran (1963). *Geometrical probability*. Griffin London.

[Kim et al. 2010] Kim, T.-J., B. Moon, D. Kim, and S.-E. Yoon (2010). “RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.2, pp. 273–286.

[Klosowski et al. 1998] Klosowski, J. T., M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan (1998). “Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs”. In: *IEEE Transactions on Visualization and Computer Graphics* 4.1, pp. 21–36.

[Koopman 1956] Koopman, B. (1956). “The Theory of Search. II. Target Detection”. In: *Operations Research* 4.5, pp. 508–509.

[Krokstad et al. 1968] Krokstad, A., S Strom, and S. Sørsdal (1968). “Calculating the acoustical room response by the use of a ray tracing technique”. In: *Journal of Sound and Vibration* 8.1, pp. 118–125.

[Kumar et al. 1994] Kumar, V., A. Grama, A. Gupta, and G. Karypis (1994). *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Redwood City.

[Lafortune and Willems 1993] Lafortune, E. P. and Y. D. Willems (1993). “Bi-directional path tracing”. In: *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics ’93)*, pp. 145–153.

[Larsson 2008] Larsson, T. (2008). “Fast and Tight Fitting Bounding Spheres”. In: *SIGRAD 2008. The Annual SIGRAD Conference Special Theme: Interaction; November 27-28; 2008 Stockholm; Sweden*, pp. 27–30.

[Lauterbach et al. 2009] Lauterbach, C., M. Garland, S. Sengupta, D. Luebke, and D. Manocha (2009). “Fast BVH Construction on GPUs”. In: *Computer Graphics Forum* 28, pp. 375–384.

[Lehericey et al. 2013] Lehericey, F., V. Gouranton, and B. Arnaldi (2013). “New iterative ray-traced collision detection algorithm for gpu architectures”. In: *Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology*, pp. 215–218.

[MacDonald and Booth 1989] MacDonald, D. and K. Booth (1989). “Heuristics for ray tracing using space subdivision”. In: *Proceedings of Graphics Interface ’89*, pp. 152–163.

[MacDonald and Booth 1990] MacDonald, D. and K. Booth (1990). “Heuristics for Ray Tracing Using Space Subdivision”. In: *The Visual Computer* 6, pp. 153–166.

[Mansson et al. 2007] Mansson, E., J. Munkberg, and T. Akenine-Moller (2007). “Deep Coherent Ray Tracing”. In: *2007 IEEE Symposium on Interactive Ray Tracing*, pp. 79–85.

[McGuire 2011] McGuire, M. (2011). *Computer Graphics Archive*. `http://graphics.cs.williams.edu/data`. [accessed: 2016.08.02].

[Mehta and Sahni 2004] Mehta, D. P. and S. Sahni (2004). *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)* Chapman & Hall/CRC.

[Meister and Bittner 2016] Meister, D. and J. Bittner (2016). “Parallel BVH Construction Using K-means Clustering”. In: *Vis. Comput.* 32.6-8, pp. 977–987.

[Möller and Trumbore 1997] Möller, T. and B. Trumbore (1997). “Fast, Minimum Storage Ray-triangle Intersection”. In: *Journal of Graphics Tools* 2.1, pp. 21–28.

[Moon et al. 2010] Moon, B., Y. Byun, T.-J. Kim, P. Claudio, H.-S. Kim, Y.-J. Ban, S. W. Nam, and S.-E. Yoon (2010). “Cache-oblivious ray reordering”. In: *ACM Transactions on Graphics (TOG)* 29.3, 28:1–28:10.

[Müller and Fellner 1999] Müller, G. and D. W. Fellner (1999). “Hybrid scene structuring with application to ray tracing”. In: *Proceedings of the International Conference on Visual Computing (ICVC’99)*, pp. 19–26.

[NVIDIA 2016a] NVIDIA (2016a). *CUDA Compute Unified Device Architecture*. `www.nvidia.com/object/cuda_home_new.html`.

[NVIDIA 2016b] NVIDIA (2016b). *Kepler GK110 Whitepaper*. `www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[NVIDIA 2017a] NVIDIA (2017a). *CUDA C Programming Guide*. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. [accessed: 2017.09.22].

[NVIDIA 2017b] NVIDIA (2017b). *NVIDIA Nsight Visual Studio Edition 5.4 User Guide*. `docs.nvidia.com/gameworks/index.html#developertools/desktop/nsight/analysis/report/cudaexperiments/kernellevel/memorystatisticstexture.htm`. [accessed: 2017.09.12].

[Naveros 2016] Naveros, A. (2016). *Rayforce - High-Performance Raytracing Engine*. `http://rayforce.survice.com/`. [accessed: 2016.11.02].

[Navratil et al. 2007] Navratil, P. A., D. S. Fussell, C. Lin, and W. R. Mark (2007). “Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization”. In: *2007 IEEE Symposium on Interactive Ray Tracing*, pp. 95–104.

[Nicodemus 1965] Nicodemus, F. (1965). “Directional Reflectance and Emissivity of an Opaque Surface”. In: *Applied Optics* 4.7, pp. 767–775.

[Ooi et al. 1987] Ooi, B. C., K. J. McDonell, and R. Sacks-Davis (1987). “Spatial KD-tree: An Indexing Mechanism for Spatial Databases”. In: *Proceedings of the IEEE COMPSAC Conference*.

[Pantaleoni and Luebke 2010] Pantaleoni, J. and D. Luebke (2010). “HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry”. In: *Proceedings of the Conference on High Performance Graphics*, pp. 87–95.

[Pantaleoni et al. 2010] Pantaleoni, J., L. Fascione, M. Hill, and T. Aila (2010). “PantaRay: fast ray-traced occlusion caching of massive scenes”. In: *ACM Transactions on Graphics (TOG)* 29.4, 37:1–37:10.

[Pharr et al. 1997] Pharr, M., C. Kolb, R. Gershbein, and P. Hanrahan (1997). “Rendering complex scenes with memory-coherent ray tracing”. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’97, pp. 101–108.

[Pharr et al. 2010] Pharr, M., W. Jakob, and G. Humphreys (2010). *Physically based rendering: From theory to implementation*. 2nd ed. Morgan Kaufmann.

[Pharr et al. 2016] Pharr, M., W. Jakob, and G. Humphreys (2016). *Physically based rendering: From theory to implementation*. 3rd ed. Morgan Kaufmann.

[Plunkett and Bailey 1985] Plunkett, D. and M. Bailey (1985). “The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed”. In: *IEEE Computer Graphics and Applications* 5, pp. 52–60.

[Popov et al. 2006] Popov, S., J. Günther, H.-P. Seidel, and P. Slusallek (2006). “Experiences with Streaming Construction of SAH KD-Trees”. In: *2006 IEEE Symposium on Interactive Ray Tracing*, pp. 89–94.

[Popov et al. 2007] Popov, S., J. Günther, H.-P. Seidel, and P. Slusallek (2007). “Stackless KD-Tree Traversal for High Performance GPU Ray Tracing”. In: *Computer Graphics Forum* 26.3, pp. 415–424.

[Popov et al. 2009] Popov, S., I. Georgiev, R. Dimov, and P. Slusallek (2009). “Object partitioning considered harmful: space subdivision for BVHs”. In: *Proceedings of the Conference on High Performance Graphics 2009*, pp. 15–22.

[Prokop 1999] Prokop, H. (1999). “Cache-Oblivious Algorithms”. MA thesis. Massachusetts Institute of Technology.

[Purcell et al. 2002] Purcell, T. J., I. Buck, W. R. Mark, and P. Hanrahan (2002). “Ray tracing on programmable graphics hardware”. In: *ACM Transactions on Graphics (TOG)* 21.3, pp. 703–712.

[Reddy and Rubin 1978] Reddy, D. R. and S. M. Rubin (1978). *Representation of Three-Dimensional Objects*. Tech. rep. CMU-CS-78-113. Carnegie-Mellon University.

[Ritter 1990] Ritter, J. (1990). “An Efficient Bounding Sphere”. In: *Graphics Gems*. Academic Press Professional, Inc., pp. 301–303.

[Rubin and Whitted 1980] Rubin, S. M. and T. Whitted (1980). “A 3-dimensional Representation for Fast Rendering of Complex Scenes”. In: *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 110–116.

[Santaló 1976] Santaló, L. A. (1976). *Integral geometry and geometric probability*. Addison-Wesley Publishing Company.

[Schulz et al. 2013] Schulz, A., S. Widmer, D. Wodniok, and M. Goesele (2013). *Extended Data Collection: Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays*. Tech. rep. 13rp003-GRIS. Fraunhofer IGD, TU Darmstadt.

[Shilane et al. 2004] Shilane, P., P. Min, M. Kazhdan, and T. Funkhouser (2004). “The Princeton Shape Benchmark”. In: *Proceedings Shape Modeling Applications*, pp. 167–178.

[Solomon 1978] Solomon, H. (1978). *Geometric probability*. SIAM.

[Sramek and Kaufman 2000] Sramek, M. and A. Kaufman (2000). “Fast Ray-Tracing of Rectilinear Volume Data Using Distance Transforms”. In: *IEEE Transactions on Visualization and Computer Graphics* 6.3, pp. 236–252.

[Stich et al. 2009] Stich, M., H. Friedrich, and A. Dietrich (2009). “Spatial splits in bounding volume hierarchies”. In: *Proceedings of the Conference on High Performance Graphics 2009*, pp. 7–13.

[Stone 1975] Stone, L. (1975). *Theory of Optimal Search*. New York: Academic Press, pp. 27–28.

[Terdiman 2000] Terdiman, P. (2000). *Radix Sort Revisited*. `http://codercorner.com/RadixSortRevisited.htm`.

[Veach and Guibas 1997] Veach, E. and L. J. Guibas (1997). “Metropolis light transport”. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 65–76.

[Veach and Guibas 1995] Veach, E. and L. Guibas (1995). "Bidirectional estimators for light transport". In: *Photorealistic Rendering Techniques*. Springer, pp. 145–167.

[Vinkler et al. 2017] Vinkler, M., J. Bittner, and V. Havran (2017). "Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction". In: *Proceedings of High Performance Graphics*, 9:1–9:8.

[Vogiannou et al. 2010] Vogiannou, A., K. Moustakas, D. Tzovaras, and M. G. Strintzis (2010). "Enhancing Bounding Volumes using Support Plane Mappings for Collision Detection". In: *Computer Graphics Forum* 29.5, pp. 1595–1604.

[Wächter and Keller 2006] Wächter, C. and A. Keller (2006). "Instant ray tracing: The bounding interval hierarchy." In: *Rendering Techniques* 2006, pp. 139–149.

[Wald 2007] Wald, I. (2007). "On fast Construction of SAH-based Bounding Volume Hierarchies". In: *2007 IEEE Symposium on Interactive Ray Tracing*, pp. 33–40.

[Wald and Havran 2006] Wald, I. and V. Havran (2006). "On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)". In: *2006 IEEE Symposium on Interactive Ray Tracing*, pp. 61–69.

[Wald et al. 2001a] Wald, I., P. Slusallek, C. Benthin, and M. Wagner (2001a). "Interactive Distributed Ray Tracing of Highly Complex Models". In: *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pp. 277–288.

[Wald et al. 2001b] Wald, I., P. Slusallek, C. Benthin, and M. Wagner (2001b). "Interactive Rendering with Coherent Ray Tracing". In: *Computer graphics forum* 20.3, pp. 153–165.

[Wald et al. 2007] Wald, I., S. Boulos, and P. Shirley (2007). "Ray tracing deformable scenes using dynamic bounding volume hierarchies". In: *ACM TOG* 26, p. 6.

[Wald et al. 2008] Wald, I., C. Benthin, and S. Boulos (2008). "Getting rid of packets - Efficient SIMD single-ray traversal using multi-branching BVHs". In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pp. 49–57.

[Walter et al. 2007] Walter, B., S. R. Marschner, H. Li, and K. E. Torrance (2007). "Microfacet Models for Refraction through Rough Surfaces". In: *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pp. 195–206.

[Walter et al. 2008] Walter, B., K. Bala, M. Kulkarni, and K. Pingali (2008). "Fast agglomerative clustering for rendering". In: *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008.* Pp. 81–86.

[Wang et al. 2013] Wang, R., Y. Huo, Y. Yuan, K. Zhou, W. Hua, and H. Bao (2013). "GPU-based Out-of-Core Many-Lights Rendering". In: *ACM Transactions on Graphics (TOG)* 32.6.

[Weber 2013] Weber, N. (2013). “Construction of Ray-Tracing Acceleration Structures in an Out-of-Core Multi-GPU Environment”. MA thesis. TU Darmstadt. URL: `http://tubiblio.ulb.tu-darmstadt.de/83044/`.

[Weber and Goesele 2016] Weber, N. and M. Goesele (2016). “Adaptive GPU Array Layout Auto-Tuning”. In: *Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications*, pp. 21–28.

[Weghorst et al. 1984] Weghorst, H., G. Hooper, and D. P. Greenberg (1984). “Improved Computational Methods for Ray Tracing”. In: *ACM Trans. Graph.* 3.1, pp. 52–69.

[Wodniok and Goesele 2016] Wodniok, D. and M. Goesele (2016). “Recursive SAH-based Bounding Volume Hierarchy Construction”. In: *Proceedings of Graphics Interface 2016*, pp. 101–107.

[Wodniok and Goesele 2017] Wodniok, D. and M. Goesele (2017). “Construction of bounding volume hierarchies with SAH cost approximation on temporary subtrees”. In: *Computers & Graphics* 62, pp. 41–52.

[Wodniok et al. 2013] Wodniok, D., A. Schulz, S. Widmer, and M. Goesele (2013). “Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays”. In: *Proceedings of the 13th Eurographics Symposium on Parallel Graphics and Visualization*, pp. 57–64.

[Wong et al. 2010] Wong, H., M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos (2010). “Demystifying GPU microarchitecture through microbenchmarking”. In: *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pp. 235–246.

[Wu et al. 2011] Wu, Z., F. Zhao, and X. Liu (2011). “SAH KD-tree construction on GPU”. In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pp. 71–78.

[Yin and Li 2014] Yin, M. and S. Li (2014). “Fast BVH construction and refit for ray tracing of dynamic scenes”. In: *Multimedia tools and applications* 72, pp. 1823–1839.

[Yoon and Lindstrom 2007] Yoon, S.-E. and P. Lindstrom (2007). “Random-Accessible Compressed Triangle Meshes”. In: *IEEE Transactions on Visualization and Computer Graphics* 13.6, pp. 1536–1543.

[Yoon and Manocha 2006] Yoon, S.-E. and D. Manocha (2006). “Cache-Efficient Layouts of Bounding Volume Hierarchies”. In: *Computer Graphics Forum* 25.3, pp. 507–516.

[Zhou et al. 2008] Zhou, K., Q. Hou, R. Wang, and B. Guo (2008). “Real-time KD-tree construction on graphics hardware”. In: *ACM Transactions on Graphics (TOG)* 27.5, 126:1–126:11.

Lebenslauf[1]

1994 – 2000	Gesamtschule Georg-Christoph-Lichtenberg Schule Ober-Ramstadt
2000 – 2003	Gymnasiale Oberstufe Georg-Christoph-Lichtenberg Schule Ober-Ramstadt
2003 – 2010	Studium der Informatik Technische Universität Darmstadt
2010	Abschluss: Diplom Informatiker Diplomarbeit: „Realtime GPU-Raycasting of Volume Data with Smooth Splines on Tetrahedral Partitions“ Referenten: Prof. Dr.-Ing. Michael Goesele, Thomas Kalbe
2010 – 2017	Wissenschaftlicher Mitarbeiter Graphics, Capture, and Massively Parallel Computing (GCC) Technische Universität Darmstadt
Seit 2018	3D Graphics Engineer GritWorld GmbH Frankfurt

Ehrenwörtliche Erklärung[2]

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades „Doktor-Ingenieur“ mit dem Titel *„Higher Performance Traversal and Construction of Tree-Based Raytracing Acceleration Structures“* selbständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen.

Darmstadt, den 23. Juli 2018

Dominik Maximilian Wodniok

---

[1] Gemäß § 8 Abs. 1 der Promotionsordnung der Technischen Universität Darmstadt

[2] Gemäß § 9 Abs. 1 der Promotionsordnung der Technischen Universität Darmstadt