Scalable Algorithms for Realistic Real-time Rendering

Vom Fachbereich Informatik der
Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

von

Valentin Fütterling

Datum der wissenschaftlichen Aussprache:

17. Juni 2019

Prüfungskommission/Berichterstatter:

Prof. Dr. Nicolas R. Gauger, TU Kaiserslautern (Vorsitz)
Prof. Dr. Achim Ebert, TU Kaiserslautern
Prof. Dr. Bernd Hamann, UC Davis

Dekan:

Prof. Dr. Stefan Deßloch

D386

Abstract

In computer graphics, realistic rendering of virtual scenes is a computationally complex problem. State-of-the-art rendering technology must become more scalable to meet the performance requirements for demanding real-time applications.

This dissertation is concerned with core algorithms for rendering, focusing on the ray tracing method in particular, to support and saturate recent massively parallel computer systems, i.e., to distribute the complex computations very efficiently among a large number of processing elements. More specifically, the three targeted main contributions are:

1. **Collaboration framework for large-scale distributed memory computers**

   The purpose of the collaboration framework is to enable scalable rendering in real-time on a distributed memory computer. As an infrastructure layer it manages the explicit communication within a network of distributed memory nodes transparently for the rendering application. The research is focused on designing a communication protocol resilient against delays and negligible in overhead, relying exclusively on one-sided and asynchronous data transfers. The hypothesis is that a loosely coupled system like this is able to scale linearly with the number of nodes, which is tested by directly measuring all possible communication-induced delays as well as the overall rendering throughput.

2. **Ray tracing algorithms designed for vector processing**

   Vector processors are to be efficiently utilized for improved ray tracing performance. This requires the basic, scalar traversal algorithm to be reformulated in order to expose a high degree of fine-grained data parallelism. Two approaches are investigated: traversing multiple rays simultaneously, and performing multiple traversal steps at once. Efficiently establishing coherence in a group of rays as well as avoiding sorting of the nodes in a multi-traversal step are the defining research goals.

3. **Multi-threaded schedule and memory management for the ray tracing acceleration structure**

   Construction times of high-quality acceleration structures are to be reduced by improvements to multi-threaded scalability and utilization of vector processors. Research is directed at eliminating the following scalability bottlenecks: dynamic memory growth caused by the primitive splits required for high-quality structures, and top-level hierarchy construction where simple task parallelism is not readily available. Additional research addresses how to expose scatter/gather-free data-parallelism for efficient vector processing.

Together, these contributions form a scalable, high-performance basis for real-time, ray tracing-based rendering, and a prototype path tracing application implemented on top of this basis serves as a demonstration.

The key insight driving this dissertation is that the computational power necessary for realistic light transport for real-time rendering applications demands massively parallel computers, which in turn require highly scalable algorithms. Therefore this dissertation provides important research along the path towards virtual reality.

Zusammenfassung

Im Bereich der Computergrafik ist die realistische Darstellung einer virtuellen Szene mit hohem Rechenaufwand verbunden. Neue, skalierbare Ansätze sind nötig, um den Leistungsbedarf für anspruchsvolle Echtzeit-Anwendungen zu decken.

Das Ziel dieser Arbeit ist die Erforschung und Entwicklung grundlegender Algorithmen für die Bildsynthese, welche die Leistung von großen, parallelen Computersystemen vollständig ausschöpfen können. Die Voraussetzung dafür ist, dass die anspruchsvollen Berechnungen sehr effizient auf eine große Anzahl von Recheneinheiten verteilt werden können. Die drei Hauptbeiträge dieser Arbeit verteilen sich auf die folgenden wichtigen Themen der parallelen Bildsynthese:

- **Kollaborationsframework für skalierbares, verteiltes Rechnen**

  Das Kollaborationsframework verwaltet den expliziten Datenaustausch über ein Hochgeschwindigkeitsnetzwerk, der für die Bildsynthese auf verteilten Rechenknoten benötigt wird. Es dient als transparente Infrastrukturschicht für die Anwendung. Der Fokus liegt auf dem Design eines Kommunikationsprotokolls, welches Verögerungen in der Kommunikation ohne Leistungseinbußen toleriert und einen sehr geringen Rechenaufwand beansprucht. Entsprechend kommt ausschließlich einseitiger und asynchroner Datentransfer zum Einsatz. Die Erwartung an ein solch lose gekoppeltes System ist, dass die Leistung linear mit der Anzahl der Rechenknoten skaliert. Bestätigt wird dies durch die direkte Messung aller, durch Kommunikation induzierter Verzögerungen und des Gesamtdurchsatzes.

- **Design von Ray Tracing Algorithmen für Vektorprozessoren**

  Um Vektorprozessoren effizient zur Leistungssteigerung von Ray Tracing verwenden zu können, muss die Möglichkeit zur Ausnutzung von Datenparallelität vorhanden sein. Zwei Ansätze werden zu diesem Zwecke untersucht: die Traversierung von mehreren Strahlen gleichzeitig, sowie die Durchführung mehrerer Traversierungsschritte in einem. Das Ziel ist es, Gruppen von Strahlen mit hoher Kohärenz zu erzeugen, sowie Sortierung während eines Multi-Traversierungsschrittes zu vermeiden.

- **Koordination von Multi-Threading und Speicherverwaltung für die Ray Tracing Beschleunigungsstruktur**

  Eine Verkürtzung der Konstruktionszeiten von hierarchischen, hoch-qualitativen Beschleunigungsstrukturen wird durch Verbesserung der Skalierbarkeit auf Mehrkernrechnern sowie Nutzung von Vektorprozessoren erreicht. Die folgenden Skalierbarkeitshürden werden dabei beseitigt: Verwaltung von dynamisch anwachsendem Speicher, der durch das Zerteilen von Objekten entsteht, sowie der Mangel an unabhängigen Aufgaben zu Beginn der Konstruktion. Des Weiteren wird ein Ansatz zur Gather/Scatter-freien Nutzung von Datenparallelität für die effiziente Vektorprozessierung während des Konstruktionsprozesses erarbeitet.

Die genannten Beiträge bilden zusammengenommen eine skalierbare, hoch-performante Grundlage für die Ray Tracing basierte Bildsynthese in Echtzeit. Auf dieser Basis wird eine prototypische Anwendung implementiert.

Die grundlegende Motivation für diese Arbeit ist die Erkenntnis, dass der immensen Rechenaufwand, der durch den Foto-realistischen Lichttransport bei Echtzeitanwendungen entsteht, nur von hochparallelen Computern bewältigt werden kann und dies nur, wenn auch entsprechend hochparallele Algorithmen zur Verfügung stehen. Aus diesem Grund leistet diese Arbeit einen wichtigen Beitrag auf dem Weg zur virtuellen Realität.

Acknowledgements

I would like to deeply thank both my advisors, Achim Ebert and Bernd Hamann, for offering their time and guidance whenever needed, and to support me in the successful completion of my dissertation. Also, I would like to thank Nicolas Gauger for readily agreeing to serve as chair of my Ph.D. defense.

I gratefully acknowledge the opportunity to participate in the International Research Training Group (IRTG) 2057, funded by the German Research Foundation (DFG) and organized by Jan Aurich and Benjamin Kirsch, which has further strengthened my interdisciplinary background.

I would like to express my appreciation towards Carsten Benthin for valuable discussions and open ears, and towards Chuck Lingle and his team for making my internship at Intel a great experience.

With enormous gratitude I look back on many years of unwavering support from the Fraunhofer ITWM, in particular from Franz-Josef Pfreundt and Carsten Lojewski, who have very significantly and positively influenced my career, and without whom this dissertation would not have been possible. Also, I thankfully acknowledge the Fraunhofer ITWM for funding this dissertation.

I would like to extend my gratitude to my family and friends to whom I say: thank you so much for everything and for carrying with me the burden of this dissertation, we have made it to the finish line!

Contents

For Anna

Chapter 1

Introduction

Rendering is central to the field of computer graphics. It concerns the computation of two-dimensional images from digital three-dimensional scenes that are projected onto a projection plane using the concept of a virtual camera, analogous to photography. Photo-realistic rendering takes a physically-based approach to model and simulate the flow of light in order to compute the appearance of the objects in the scene, often with the goal to create an image indistinguishable from a hypothetical photograph taken of the scene as if it existed in reality. The defining property for photo-realistic rendering is that global illumination is accounted for, i.e., light incident at a given point arriving not only from a light source directly but also from reflection, transmission and scattering by the environment. This indirect light contributes profoundly to the perceived realism of a synthesized image.

A spectrum of algorithms exist that compute global illumination or specific terms thereof with varying degrees of accuracy and efficiency. In order to establish paths of light between points most of these algorithms rely on ray tracing, a method to find intersections between a ray given by origin and direction and any scene object. While the realism achievable by many of the ray-tracing based global illumination algorithms on today's commodity computing devices is astonishing, the immense amount of computation often required for a single image may result in minutes to hours of rendering time.

This is sufficient for production rendering of movies and product visualization, even though faster rendering always increases productivity. However, for interactive applications such as games, computer-aided design or scientific exploration, where real-time rendering allows only a few millisecond time budget per image, significantly more computational power is mandatory. Nevertheless, the demand for photo-realistic real-time rendering is growing fast, driven especially by virtual and augmented reality applications aiming to deliver immersive experiences.

Computational complex problems give rise to high-performance computing (HPC) and supercomputers. Supercomputers are fast because they are specialized in parallel computing, i.e, performing many calculations simultaneously. For algorithms to run efficiently on such a system they must be specifically designed for parallel, scalable execution.

Since the diminishing of processor frequency scaling in the early years of the century traits of supercomputers such as vector- and multiprocessing have been absorbed by and adapted to mainstream computer architecture. Simultaneously, supercomputer design has pivoted from specialized processors to clusters of mainstream processors

interconnected by specialized high-performance networks. As this convergence continues the difference between high-performance and mainstream computing may be hypothesized to become just one of scale: the supercomputer of today is the mainstream device of tomorrow.

Hence, designs of scalable rendering algorithms supporting real-time global illumination on massively parallel high-performance computers today will enable desktops and mobile devices in the future as well, as soon as advances in manufacturing technology allow to integrate the same amount of parallelism within an appropriate area footprint and power envelope.

1.1 Research Goals and Results

This dissertation presents research focused on parallel algorithm design for ray tracing and rendering in HPC environments. In order to grow ray tracing performance with the rapidly expanding capacity for parallel computation in computer systems the scalability of the algorithms and their implementations is considered key. In particular, distributed computing, vector processing and multi-threading are applied, in combination, to ray traversal, generation of acceleration data structures and the overall rendering process. The respective current state of the art is found to be insufficient to enable optimal utilization of parallel units on the scale of current and future HPC systems.

Throughout the dissertation it becomes clear that scalability is mandatory to master the computational complexity associated with real-time photo-realistic rendering, and that highly specialized techniques are necessary in order to make the best possible use of the parallel computational resources. The newly introduced algorithms enable massively parallel ray tracing and the corresponding implementations are demonstrated by experiment to be the best-in-class. The algorithms act as building blocks, enabling applications to take advantage of massively parallel systems for real-time global illumination computation much more efficiently compared to previous approaches. Specific contributions are made in:

1. **Collaboration framework for large-scale distributed memory computers**

   The purpose of the collaboration framework is to enable scalable rendering in real-time on a distributed memory computer. As an infrastructure layer it manages the explicit communication within a network of distributed memory nodes transparently for the rendering application. The research is focused on designing a communication protocol resilient against delays and negligible in overhead, relying exclusively on one-sided and asynchronous data transfers. The hypothesis is that a loosely coupled system like this is able to scale linearly with the number of nodes, which is tested by directly measuring all possible communication-induced delays as well as the overall rendering throughput.

2. **Core algorithms designed for vector processing**

   Vector processors are to be efficiently utilized for improved ray tracing performance. This requires the basic, scalar traversal algorithm to be reformulated in order to expose a high degree of fine -grained data parallelism. Two approaches are investigated: traversing multiple rays simultaneously, and performing multiple traversal steps at once. Efficiently establishing coherence in

a group of rays as well as avoiding sorting of the nodes in a multi-traversal step are the defining research goals.

3. **Multi-threaded schedule and memory management for the core acceleration structure**

   Construction times of high-quality acceleration structures are to be reduced by improvements to multi-threaded scalability and utilization of vector processors. Research is directed at eliminating the following scalability bottlenecks: dynamic memory growth caused by the primitive splits required for high-quality structures, and top-level hierarchy construction where simple task parallelism is not readily available. Additional research addresses how to expose scatter/gather-free data-parallelism for efficient vector processing.

Based on these contributions, real-time photo-realistic rendering is realized on a 60 node cluster, even for very large and complex scenes. An exemplary application developed on top of the building blocks demonstrates interactive and smooth exploration and editing of such scenes.

Major parts of the research presented in this dissertation has resulted in publications in peer-reviewed journals and conference proceedings: *Efficient ray tracing kernels for modern CPU architectures* [39] is concerned with the traversal of ray groups for efficient vector processing, *Accelerated single ray tracing for wide vector units* [38, 37] proposes a scalable multi-branch ray traversal algorithm and *Parallel Spatial Splits in Bounding Volume Hierarchies* [40] accelerates high-quality acceleration data structure generation with many threads and vector processing. A complete overview of the publications and presentations associated with this dissertation are listed in the appendix: Publications by Valentin Fütterling.

1.2 Structure of the Dissertation

Chapter 2 covers the prerequisites and related work for later chapters, regarding photo-realistic rendering, ray-tracing and parallel computing.

The topic of Chapter 3 is the acceleration of ray tracing with vector processing. Several new algorithms are proposed for ray traversal, i.e., the most time-consuming part of ray tracing, targeting multiple sources of data parallelism, respectively, and supporting increased vector lengths. Measurements confirm that the increased scalability of the introduced techniques translate into significant performance gains, clearly surpassing previous approaches. The chapter combines the research of two peer-reviewed publications [39, 38] and a patent application [37].

Chapter 4 proposes a solution to the time-to-image scalability restrictions imposed by the construction of high-quality bounding volume hierarchies. The new approach employs multi-threading and vector processing very efficiently to generate this central, ubiquitously employed acceleration data structure for ray tracing with the best-known quality. Experimental results show that the the new algorithm enables increased scalability in the construction process, approaching the parallel efficiency of the rendering itself, and thus significantly outperforms the fastest previous approaches of the same high-quality category. The chapter is based on the research already presented in a peer-reviewed publication [40].

In Chapter 5 rendering is brought to large, distributed cluster environments. For the first time, this is achieved by a fully asynchronous communication model between the cluster nodes, leading to unprecedented scaling efficiency of the rendering process. The efficiency is facilitated by a new decentralized, self-balancing algorithm for work distribution which is aware of data locality and thus improves caching behaviour as well. Experimental results demonstrate that the new model is superior to previous approaches and even exhibits super-linear scalability due to caching effects.

Chapter 6 presents an application for interactively exploring and editing large data sets in photo-realistic quality. The application integrates the methods developed throughout this dissertation as essential building blocks which allow it to run across a range of devices, from laptop to supercomputer, with high performance.

Finally, Chapter 7 summarizes the contributions of this dissertation again and provides a final conclusion and outlook.

Chapter 2

Background

The topics covered in this chapter provide the basic background and state-of-the-art for the research agenda in the following chapters. The first section starts with rendering fundamentals and definitions, followed by the second section with a basic introduction to ray tracing, a category of rendering algorithms which simulate light transport by shooting rays. The book by Pharr et al. [88] is recommended as a supplementary resource. Ray tracing performance is a key factor for fast realistic image synthesis. Since this dissertation aims at accelerating ray tracing by designing specialized algorithms for massively parallel computer architectures the third section deals, in a general way, with the available forms of parallelism and their characteristics relevant to efficient programming. The last section analyzes the state-of-the-art in scalable ray tracing algorithm design for parallel architectures and outlines the open challenges addressed by this dissertation's research.

2.1 Photo-realistic Rendering

In computer graphics, rendering describes the process of projecting a three-dimensional scene through a virtual camera onto a two-dimensional viewport, ready for presenting on a display device, see Figure 2.1 for an illustration. Position and orientation of the camera define which part of the scene is visible within the viewport area. Light is emitted from virtual light sources. For every viewport pixel, the object closest to the camera, i.e., *primary visibility*, is determined, possibly occluding other objects further away. The pixel color is derived from the surface properties of the object and the corresponding lighting model, a process called *shading*. Simple shading takes into account only the local surface properties and assumes an ubiquitous light source, for example. In order to produce hard *shadows* or perfect *reflections*, shading requires global information about the scene, i.e., if there exists an occluding object between the shaded sample and a light source or which object is visible in the reflection, respectively. Hence, the computational complexity of the shading operation increases as more realistic effects are considered.

Shading which includes all relevant optical phenomena to produce an image that "looks like a photo" of a real scene is the defining characteristic of *photo-realistic rendering*. In particular, soft shadows, glossy reflection and refraction and *indirect lighting* from non-emissive surfaces contribute greatly to the perceived realism of a rendered image.

The *light transport equation* introduced by Kajiya [64] captures the entirety of geometrical optics, a physically accurate model of electromagnetic radiation in the optical regime. The geometrical optics model neglects the wave and quantum character of

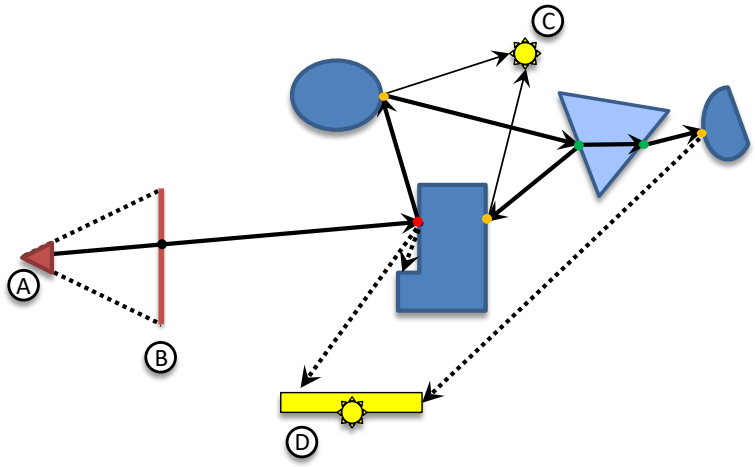

FIGURE 2.1: Photo-realistic rendering based on ray sampling. The example shows the virtual camera (A), the viewport area (B), light sources (point light C and area light D) and scene objects (blue shapes). Primary visibility of the initial ray, defined by the camera origin and a pixel location on the viewport, is indicated by the red dot. Dashed arrows sample the area light, producing soft shadows, thin arrows sample the point light, producing hard shadows, and solid arrows sample indirect lighting. Orange dots indicate diffuse reflections and the two green dots represent perfect reflection and refraction, respectively.

light, since manifestations thereof are rarely visible to the human eye. For a surface point p the light transport equation states the outgoing radiance L_o in direction ω_o:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos \theta_i| d\omega_i. \tag{2.1}$$

The first term, L_e, represents the emission from a light source on the surface at point p in direction ω_o and the second term, an integral over the unit sphere S^2, describes the scattering of incoming light. L_i denotes the radiance incident on p from direction ω_i and the bidirectional scattering distribution function (BSDF) f is a proportionality factor between incoming and outgoing radiance, thus defining the reflective and refractive properties of the surface material. The cosine of the angle θ_i, between the surface normal at point p and the incoming direction ω_i, scales the radiance L_i according to projected area on the surface. An illustration is provided in Figure 2.2.

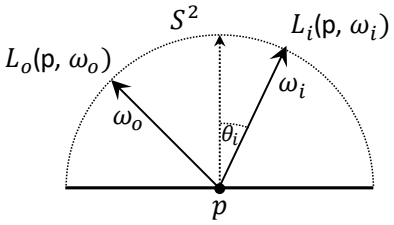

FIGURE 2.2: Light transport equation. A reflective surface is represented by the solid horizontal line and its vertical dashed surface normal; the integration domain is indicated by the dashed semicircle.

Simplifying Equation 2.1, it is assumed that the scene consists entirely of homogeneous media where surfaces define the interfaces between different media. Further, all media are considered to be non-participating, which means that light is not affected by volumetric scattering, absorption or emission. Then, light travels in straight lines between surfaces, and the incoming radiance L_i at surface point p can be related to the outgoing radiance L_o from surface point p', where p' is determined by the *ray tracing function* t:

$$L_i\left(p,\omega\right) = L_o\left(t\left(p,\omega\right),-\omega\right). \tag{2.2}$$

The ray tracing function t finds p' by tracing a ray from point p along ω_i. The first intersection point between the ray and any scene surface then corresponds to p'. Hence, the light transport equation can be rewritten in terms of $L_o \equiv L$:

$$L\left(p,\omega_o\right) = L_e\left(p,\omega_o\right) + \int_{S^2} f\left(p,\omega_o,\omega_i\right) L\left(t\left(p,\omega_i\right),-\omega_i\right) |\cos\theta_i| d\omega_i. \tag{2.3}$$

This form of the light transport equation more clearly indicates how to approach a solution: a surface point p is shaded by adding to the emission term the contributions of all surface points $p_\nu \in \{t\left(p,\omega_\nu\right) : \omega_\nu \in S^2\}$ visible from p. The respective contributions from each p_ν are determined by recursive evaluation of Equation 2.3. A major focus of this dissertation is on efficiently solving the ray tracing function t which establishes the connections between surface points.

Path Integral

The *path integral* formulation allows to write the light transport problem in terms of a simple integral instead of the recursive integral equation shown in Equation 2.3 [101, 73]. This allows the application of general integration techniques and provides an intuitive understanding of light transport as a summation over light paths. A light path can be viewed as a polyline, where the first vertex is located on a light source, the inner vertices are located on the scene surfaces and the last vertex is located on the lens of the virtual camera. Also, within the path integral framework multiple techniques for efficiently calculating specific parts of the rendering integral can be readily combined. The path integral form of light transport is:

$$I_j = \int_\Omega f_j(\bar{p})\mathrm{d}\mu(\bar{p}). \tag{2.4}$$

Here, I_j is the contribution to the jth pixel, Ω is the set of all possible paths, $\mathrm{d}\mu(\bar{p})$ is the path measure and $f_j(\bar{p})$ is the measurement contribution function. The path $\bar{p}_n = p_0 \ldots p_n$ of length n is composed of $n+1$ vertices which are located on the scene surfaces S, including lights and the camera viewport. Then, the set of all possible paths can be limited to the sum over all finite path lengths n and the Cartesian product of S with itself taken $n+1$ times, transforming Equation 2.4 into:

$$I_j = \sum_{n=1}^{\infty} \int_{S^{n+1}} f_j(p_0 \ldots p_n)\mathrm{d}A(p_0)\ldots\mathrm{d}A(p_n). \tag{2.5}$$

The path measure $\mathrm{d}\mu(\bar{p})$ is then revealed to be the product of the area measure on S raised to the power of $n+1$. The measurement function $f_j(\bar{p})$ measures the light

received on the camera lens due to path $\bar{p}$, see also Figure 2.3:

$$f_j(\bar{p}) = L_e(p_0 \to p_1)\, T(\bar{p})\, W_e^j(p_{n-1} \to p_n). \tag{2.6}$$

It is a product of the light $L_e(p_0 \to p_1)$ emitted at the first vertex p_0 into the direction of the second vertex p_1, the *path throughput* $T(\bar{p})$ and the *importance* $W_e^j(p_{n-1} \to p_n)$ assigned to the light arriving on the sensor pixel j at vertex p_n from the direction of vertex p_{n-1}. For rendering an image, the importance can be identified with the filter function applied to the image samples. The throughput describes the fraction of the initially emitted light that is not absorbed or scattered at the inner path vertices and finally arrives at the sensor:

$$T(\bar{p}) = G(p_0 \leftrightarrow p_1)\, \rho_s(p_1) \ldots \rho_s(p_{n-1}) G(p_{n-1} \leftrightarrow p_n). \tag{2.7}$$

It is a product of the BSDFs $\rho_s(p_i)$ at vertices i and the *geometry terms* $G(p_i \leftrightarrow p_j)$ between adjacent path vertices p_i and p_j:

$$G(p_i \leftrightarrow p_j) = V(p_i \leftrightarrow p_j) \frac{|\cos\theta_i||\cos\theta_j|}{||p_i - p_j||^2}. \tag{2.8}$$

The function $V(p_i \leftrightarrow p_j)$ represents binary visibility and is equal to one if there is a clear line of sight between vertices p_i and p_j or zero otherwise. Comparing to Equation 2.3, $|\cos\theta_i|$ is the same term in both equations even though the meaning of the index i differs, and $V(p_i \leftrightarrow p_j)$ can be identified with the ray tracing function $t(p, \omega)$ after the change of integration variables from solid angle $\mathrm{d}\omega$ to area $\mathrm{d}A$. The remaining $|\cos\theta_j|$ term divided by the squared distance between vertices stems from the Jacobian introduced by the variable change.

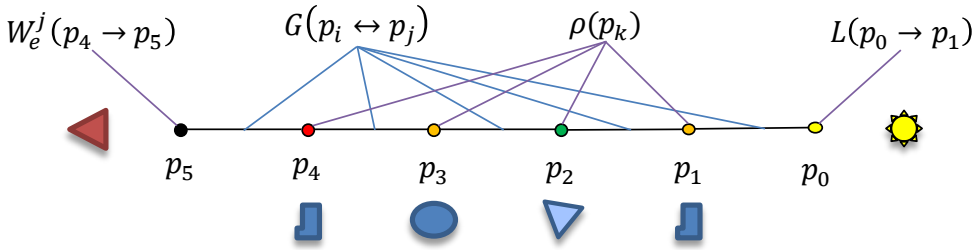

FIGURE 2.3: Path measurement function. The shown example matches one of the paths in Figure 2.1, with each vertex labeled by corresponding color and shape.

A general approach for a numerical solution to the path integral in Equation 2.4 is provided by either Monte Carlo [65] or Markov Chain Monte Carlo [102] integration techniques. A Monte Carlo estimate of an integral $I = \int f(x)\mathrm{d}x$ is given by

$$\langle I \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}, \tag{2.9}$$

where the x_i are drawn from a probability distribution with probability density function (PDF) $p(x)$. The expectation value of the estimate is equal to the accurate solution of the integral: $E[\langle I \rangle] = I$. Increasing the number of samples N reduces the variance of the estimate. Regarding the path integral, the estimate of a single sample

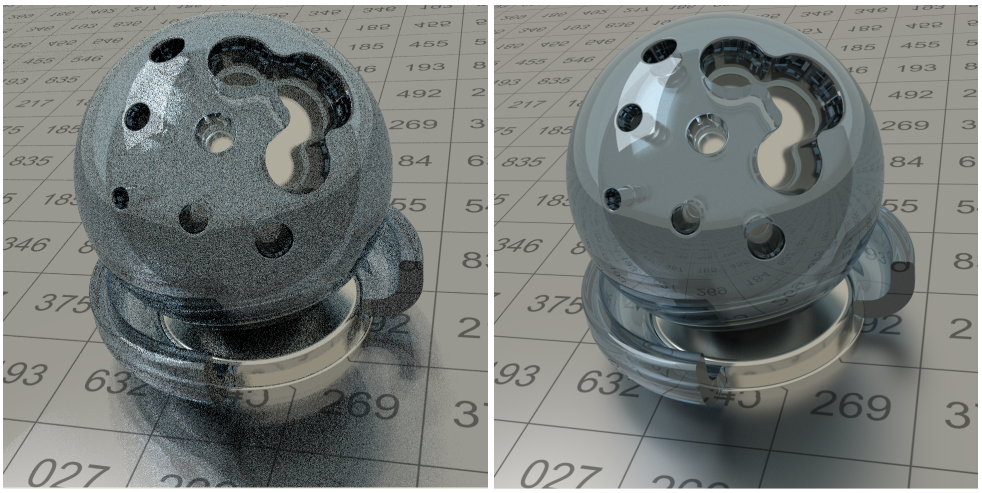

FIGURE 2.4: Two rendered images of a glass sculpture, using four path samples per pixel (spp) on the left and 1024 spp on the right. A low number of samples produces high variance in the Monte Carlo integrand, visible as grainy noise in the left image.

is given by:

$$\langle I_j \rangle = \frac{f_j(\bar{x})}{p(\bar{x})}. \tag{2.10}$$

Averaging the contribution of multiple paths reduces the variance and thus the visible noise in the rendered image, as demonstrated in Figure 2.4. From the perspective of the path integral formulation, many light transport algorithms such as path tracing [65], light tracing [6], bi-directional path tracing [74], vertex connection and merging [45], etc., differ only in their path sampling strategy and the corresponding PDFs. Common to all the path sampling strategies is their reliance on ray tracing for visibility computations.

2.2 Ray Tracing

Given an origin $\mathbf{o}$, a direction $\mathbf{d}$ and a parameter t, a ray $\mathbf{r}$ is defined by the equation

$$\mathbf{r} = \mathbf{o} + t\mathbf{d} \quad t > 0. \tag{2.11}$$

In a scene S, representing a set of objects, the following set of intersections t_i of the ray with these objects may exist:

$$\{t_i : \mathbf{r}(t_i) \cap S \wedge i < j \Leftrightarrow t_i < t_j\} \tag{2.12}$$

The task of a ray tracing algorithm is to find either the first intersection t_0, any one intersection, some or all intersections. Ray tracing for rendering was first introduced as a solution for primary visibility [5]. Rays originating at the camera location are cast into the scene with a different direction for each viewport pixel and the closest object intersected by the corresponding ray determines the color of the pixel. More generally, ray tracing allows to point-wise project any function onto a two-dimensional manifold as long as a ray intersection algorithm for the function exists. This unique flexibility makes ray tracing highly valuable for rendering since a multitude of scene data such as triangular meshes, parametric surfaces, density fields,

particles, implicit functions, etc., can be rendered using the same, unified method. Also, the viewport manifold is flexible in shape, which is useful, for example, in virtual reality applications for correcting lens distortions [90].

A straightforward approach to finding t_0, for example, from the set in Equation 2.12 is to calculate the intersection of the ray with every object in the scene and take the minimum of the results. Quite clearly, for non-trivial scenes this method is not efficient as most objects are likely to be either occluded by other objects or missed altogether.

Hence, an acceleration structure is an intrinsic part of nearly every ray tracing algorithm (an exception is divide-and-conquer ray tracing [1]). This reduces the algorithmic complexity of a search query from $O(N)$ to $O(\log N)$ where N is the number of objects in the scene. The acceleration structure partitions the scene objects into subsets and, given a ray, a *traversal* algorithm quickly determines all *potentially* intersecting subsets, thus greatly reducing the number of required intersection calculations. If the traversal enumerates the subsets in a *front-to-back* manner, i.e., ordered with respect to increasing distance from the ray origin, even occluded subsets may be discarded or *culled* before an intersection test becomes necessary. While acceleration structures greatly reduce intersection calculations, their construction and update prior to traversal and upon transformations of the scene geometry, respectively, produce additional work. Hence, the construction efficiency of an acceleration structure is very important, especially for dynamic scenes.

Generally, acceleration structures either subdivide space or objects into disjoint sets. In practise, all types of acceleration structures are organized as a hierarchy, i.e., as a directed acyclic graph, in order to adapt to nonuniform scene geometry. Space subdivision has the advantage that nodes in the hierarchy reference disjoint volumes. This enables an exact front-to-back traversal and guarantees that intersections are found in order along the ray, i.e., t_0, t_1, t_2, etc. Thus, the traversal algorithm can usually terminate upon the first valid intersection.

In contrast, object subdivision must permit nodes to overlap spatially if the objects themselves or their bounding volumes overlap spatially as well. Hence, the traversal algorithm cannot guarantee an exact front-to-back order and must check all nodes with a ray entry distance smaller than the current intersection for a potentially closer one. On the positive side, object subdivision stores only one reference per object whereas space subdivision might need to store multiple references per object, inflating the depth and/or storage requirements of the hierarchy.

Acceleration structures based on space subdivision include octrees [46], binary space partitioing (BSP) trees [62], kd-trees [109], and grids [41]. Grids are not intrinsically hierarchical, but they are usually embedded into a hierarchical structure to better adapt to nonuniform geometry [86, 72]. Acceleration structures using object partitioning include the bounding interval hierarchy [104] and the bounding volume hierarchy (BVH) [47]. BVHs are the state-of-the-art in ray tracing and integrated into many ray tracing systems due a favourable mix of flexibility, fast traversal and fast construction speed. Thus, this dissertation also emplolys BVHs as the acceleration structure. In the following section, BVHs are discussed in detail.

2.3 Bounding Volume Hierarchies

A BVH is an acyclic graph or tree where each node is associated with a bounding volume. The type of bounding volume most commonly used is a box defined by six axis-aligned planes because it bounds most geometry well and the intersection calculation with a ray is efficient. The bounding volume of the *root* node contains all bounding volumes of the tree's other nodes. An *inner* node bounds all nodes which are part of its sub-tree and a *leaf* node bounds one or multiple objects, e.g., triangles.

Regarding terminology, if a node A is a descendant from node B, A is part of the *sub-tree* of B. Also, if a node A is an immediate descendant from node B, A is a *child* node of B and B is the *parent* node of A. If both nodes A and B share the same parent node C, A and B are *sibling* nodes and belong to the same node *cluster*.

The traversal of a ray starts at the root node of the tree by calculating the intersection with the corresponding bounding box. Upon hit, the child nodes are intersected: missed child nodes are discarded and one hit child node is selected to continue the traversal with as the new parent node. The remaining hit child nodes are saved on a stack for later retrieval. The current traversal branch stops if no child node is intersected or a leaf node is reached. At a leaf node, an intersection test is performed on the contained objects. A successful object intersection shortens the ray to the intersection distance. After a stop, a new parent node is taken from the top of the stack and traversal continues. The traversal is completed once the stack is found empty upon requesting a new node. Alternatives to a stack-based traversal are provided by restart [75] or back-tracking [14] algorithms, which are useful if stack memory is scarce or slow. Important for traversal performance is the order in which a ray visits the hit child nodes in a node cluster. This is discussed further in Section 2.3.

The construction of a BVH can be performed either divisive [47] or agglomerative [114]. Divisive construction starts with a single set containing all objects and divides the set into two new disjoint sets, producing one inner node and two leave nodes. This procedure is repeated, adding inner nodes and leaf nodes to the tree, until a leaf node is deemed small enough, either in size or in the number of objects it contains, to stop the recursive division. Agglomerative construction starts with many sets, e.g., one object per set, and one leaf node per set. The sets are merged recursively until only a single set remains, generating inner nodes and finally the root node.

The strategy for sub-dividing or agglomerating during the build process has a major impact on BVH *quality*. For example, dividing a set of objects into two equally large sets with minimal overlap generates a balanced tree which, for a general tree structure, guarantees the best average query time. However, the varying geometric properties of bounding boxes and objects within a node's sub-tree can lead to a significant discrepancy in terms of traversal computations between two nodes for the average ray even if they contain the same amount of objects. Hence, a strategy that optimizes BVH quality is desired.

The quality of a BVH is synonymous with the average time of a ray query and correlates with the number of nodes visited and the number of objects intersected on average. The quality can be determined by sampling many rays randomly distributed over origins and directions relevant to the scene (i.e., not starting within a wall, not looking away from the scene, etc.). Such a measurement requires a BVH to begin

with and is impractical as a quantity to guide actual BVH construction. The *surface area heuristic* (SAH) provides a model to approximate the traversal cost for a node, a sub-tree or the complete BVH. It can be used to steer subdivision and agglomeration during the build process and is detailed in the following section.

Surface Area Heuristic

The surface area heuristic (SAH) is a cost model for the ray traversal of tree structures. It assigns a cost to a node N based on the following simplifying assumptions:

- Rays are distributed uniformly in origin and direction.
- A ray pierces the bounding box of N unobstructed and its origin is located outside of N.
- The cost of a single traversal step and the cost of a single primitive intersection are known as C_t and C_i, respectively.

Then, the cost assigned to node N after being partitioned into a left child node N_L and a right child node N_R is estimated as:

$$C_{inner}(N \to \{N_L, N_R\}) = C_t + \frac{\mathrm{SA}(N_L)}{\mathrm{SA}(N)} C(N_L) + \frac{\mathrm{SA}(N_R)}{\mathrm{SA}(N)} C(N_R) \tag{2.13}$$

Here, the $C(N_L)$ and $C(N_R)$ correspond to the costs of left and right child nodes, respectively. The functions $\mathrm{SA}()$ calculate the surface area of the corresponding nodes' bounding boxes. The surface area fractions represent the geometric probability of a ray intersecting either of the child nodes N_L and N_R given the existence of an intersection with node N. This formula can be applied recursively to compute $C(N_L)$ and $C(N_R)$ until a leaf node is reached. The cost of a leaf node L is estimated as the number of corresponding primitives $|L|$ times the cost of a single primitive intersection:

$$C_{leaf}(L) = |L| C_i \tag{2.14}$$

Expanding Equation 2.13 in conjunction with Equation 2.14 leads to the total estimated cost of a (sub-)tree with root node T:

$$C_{tree}(T) = \sum_{N \in \text{inner nodes}} \frac{\mathrm{SA}(N)}{\mathrm{SA}(T)} C_t + \sum_{L \in \text{leaf nodes}} \frac{\mathrm{SA}(L)}{\mathrm{SA}(T)} |L| C_i \tag{2.15}$$

Unfortunately, finding the minimum C_{tree} on a global scale is an infeasible problem for practical scenes. However, the SAH formulation is useful for greedy decisions at the scope of a single node as well. In order to evaluate cost of a partition of node N, i.e., $C_{inner}(N \to \{N_L, N_R\})$, it is necessary to know the costs $C(N_L)$ and $C(N_R)$ corresponding to Equation 2.15. For agglomerative construction these costs are readily available, but for divisive construction the costs are impossible to determine at the moment of partitioning because the sub-tree of the child nodes does not yet exist. Hence, the costs for child nodes are approximated by $C_{leaf}(N_L)$ and $C_{leaf}(N_R)$, respectively:

$$C_{inner}(N \to \{N_L, N_R\}) \approx C_t + \frac{\mathrm{SA}(N_L)}{\mathrm{SA}(N)} |N_L| C_i + \frac{\mathrm{SA}(N_R)}{\mathrm{SA}(N)} |N_R| C_i \tag{2.16}$$

Equation 2.16 usually overestimates the true cost of node N, revealed only after its sub-tree is fully constructed, by assuming the child nodes N_L and N_R are not subdivided further.

Despite the initial simplifying assumptions, local instead of global cost estimation, and the approximation in Equation 2.16 the SAH is the basis of all high quality BVH construction algorithms. Methods for approaching the global minimum of the cost $C_{tree}(T)$ such as local tree rotations [69] and treelet restructuring [67] are based on the idea of perturbing an existing hierarchy in order to overcome local minima in the global cost function. However, while minimizing the SAH these methods do not always lead to faster ray queries, but often even deteriorate the actual BVH quality. An analysis [3] has resulted in an extension of the SAH cost model that can account for the discrepancy between SAH cost and the measured cost of a BVH, even though an algorithm for optimal partitioning according to the extended model could not be derived.

Spatial Splits

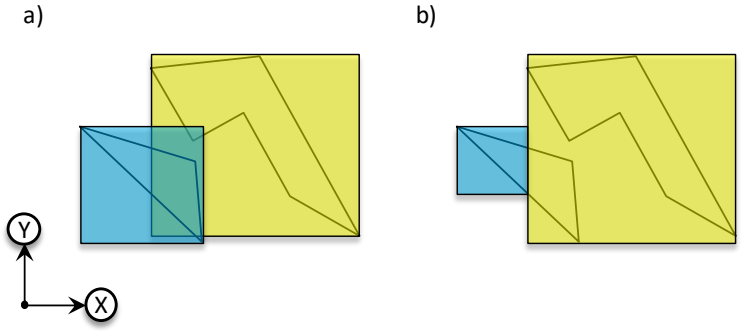

FIGURE 2.5: Two BVH nodes bounding the same objects, once without (a) and once with (b) a spatial split. With a spatial split, the overlap is removed, the total surface area reduced and the number of object references increased.

Object subdivision becomes inefficient for ray tracing if the bounding boxes of primitives overlap significantly even though the actual geometry is disjoint. This occurs especially for larger-than-average primitives and for primitives extending along a direction diagonal with respect to the principal axes.

The approaches of [31] and [25] subdivide the bounding boxes of problematic primitives or the primitives themselves, respectively, before proceeding with the BVH construction. Since the subdivisions are performed speculatively they do not necessarily improve the quality of the final BVH but may even degrade it due to the increased number of primitives.

A different approach [97] integrates the subdivision of primitive bounding boxes into the BVH build process and applies a subdivision only if it reduces the SAH cost $C_{inner}(N \rightarrow \{N_L, N_R\})$ during divisive partitioning of a node N, see Figure 2.5. The corresponding algorithm, referred to as the split BVH (SBVH), is the method known to produce the highest quality BVHs on average. A drawback of the SBVH is that it is computationally demanding and difficult to parallelize, issues which are a major focus of this dissertation and addressed in Chapter 4.

Traversal Order

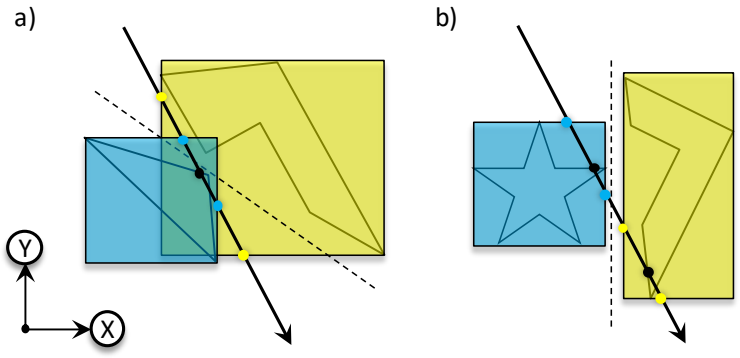

FIGURE 2.6: Traversal order considerations. (a) Two overlapping inner nodes containing non-overlapping geometry and corresponding sub-trees (not shown). The ray pierces the yellow node first but a geometry intersection exists only within the blue node, which is not known at the moment of the traversal order decision. (b) Disjoint nodes: front-to-back traversal clearly follows the blue node first. The intersection within the blue node shortens the ray and allows to skip the yellow node entirely (culling).

A ray that overlaps with multiple child nodes may trace the corresponding sub-trees in any order and always produce the correct final result. However, for maximum performance it is crucial to maintain a front-to-back ordering so that a successful primitive intersection can efficiently cull more distant nodes, see Figure 2.6. If nodes overlap, a strict front-to-back ordering cannot be determined a priori, but a heuristic can be employed to make a good guess.

Commonly implemented heuristics are based on the distance of the ray origin to the node bounding box or on the signs of the ray direction. Both heuristics corresponds to a strict front-to-back ordering only if the ray does not cross a region of overlapping nodes. For the *distance heuristic* the ray is intersected with all bounding boxes and the distance values are sorted to determine the traversal order. The main drawback of the distance heuristic is the computationally expensive sorting step, especially for branching factors larger than two. The *sign heuristic* can be evaluated more efficiently as it does not depend on the dynamics of the traversal. For a binary BVH, the traversal order is chosen based on the axis along which the child nodes overlap the least and on the sign of the corresponding ray direction component. The axis is stored in the node data structure to quickly index the ray sign during traversal.

Just like first-hit traversal, multi-hit traversal performance profits from a front-to-back traversal order. Multi-hit traversal [49, 48] attempts to find the first n closest intersections with primitives, in order, along a ray. This type of ray query is sometimes employed in scientific or technical visualization.

A different case is any-hit traversal, as used for shadow rays for example, which can terminate as soon as a intersection with any primitive is found. For this type of query, a front-to-back order is not important and specialized traversal orders are more efficient [83]. These specialized traversal orders are identical for all rays and can be encoded into a BVH by arranging the nodes in memory accordingly.

2.4 Optimization

Acceleration structures such as the BVH decrease ray query times substantially. However, evaluating the path integral in Equation 2.4 directly can require hundreds of samples, i.e., thousands of ray queries, per pixel until the inherent noise of the Monte Carlo technique falls below a perceptible level. For a full resolution image this translates into Billions of ray queries. Aiming at 60Hz refresh rate for real-time rendering this number approaches one Trillion ray queries per second. A hypothetical ray tracing processor capable of tracing one ray per cycle would have to run at an illusive frequency of around one Terahertz to achieve this.

The best approach for accelerating ray tracing is of course to avoid shooting rays as much as possible. A variety of techniques have been proposed to reduce the number of samples while improving image quality. These techniques can be broadly assigned to two categories: (1) cleverly choosing samples, such as *(multiple) importance sampling*, *quasi Monte Carlo*, *Markov chain Monte Carlo*, *adaptive sampling*, and (2) making the most of the available samples, such as *filtering* and *reconstruction*.

Importance sampling generates new path segments based on a probability distribution closely matching the features of the integrand in Equation 2.3. For example, the contribution of a sample is weighed by the BSDF, so generating a new direction based on the angular distribution of the BSDF weight has the potential to reduce noise compared to a uniformly random direction. However, the integrand is a product of multiple additional terms hidden inside the L_i, so sampling according to a single term like the BSDF does not always approximate the full integrand very well. If other terms are known, e.g., the emission profile of a light sampled directly [95], additional samples can be generated accordingly and combined by *multiple importance sampling* [103].

Quasi Monte Carlo integration [68] replaces random numbers by low discrepancy sequences. A set of low discrepancy numbers is more uniformly distributed within the desired domain compared to a set of random numbers, which tend to form clusters.

Metropolis light transport [102] employs *Markov Chain Monte Carlo* integration. New paths are generated by repeatedly modifying existing ones, leading to sequences of light-carrying paths with high contributions to the final image. Metropolis light transport is especially efficient for complex lighting situations.

Samples with spatial and/or temporal coherence on the image plane, in path space or in texture space can be used for *filtering*. If applied for *reconstruction* the filtering operation modifies a sample based on its neighbours in the respective domain. Reconstruction suppresses noise but introduces bias. In many cases the bias is less distracting to a human observer, however, and thus preferable to noise. Instead of or complementary to reconstruction, a filter operation can be applied to determine the noise level in an area.

Adaptive sampling directs new samples to areas with high noise where they are most efficient in reducing the overall noise of the rendered image. A comprehensive overview of adaptive sampling and reconstruction techniques is provided by [117]. More recently, machine learning approaches have been introduced into this field

with very promising results for real-time rendering [21] and movie production [10]. Also, one sample per pixel global illumination for real-time applications have been proposed [92, 80].

While all the aforementioned methods help to reduce the number of samples required to achieve a desired level of image quality, sometimes significantly so, a vast number of rays remain to be traced regardless. Hence, this dissertation pursues a different but orthogonal strategy for accelerating rendering with ray tracing: parallelism.

2.5 Parallel Computer Architecture

The speed of computer operations are fundamentally limited by the underlying physical processes. These fundamental restrictions and architectural design determine the attainable clock frequency of a processor. Over the last decade frequencies have saturated in a range of $2 - 5GHz$ for mainstream general purpose CPUs. Hence, the only way to increase compute power, i.e., the throughput of instructions, (apart from inventing entirely new physical and / or logical approaches for computation) is to go parallel.

The key issue in parallel computing is scalability: given a computational problem, by how much does the time required to compute the result decrease if the width of a parallel computer increases, i.e., if more processing elements are added. A classical answer to this question is given by Amdahl's law [4] which separates a program into a serial and a parallel part. Ideal scalability, i.e., linear scalability, is possible only for a program with no serial part. This answer is incomplete, however, because (1) it considers only a specific program, not the actual problem to be solved and (2) it neglects any interaction between processing elements. Usually a specific problem may be formulated in different ways, leading to different algorithms, which allow a variety of implementations, resulting in a multitude of programs which all solve the original problem. Thus, scalability of a problem solution strongly depends on the choice of problem formulation, algorithms and implementation. On the hardware side processing elements often share system resources such as memory access with limited capacity so that processing elements may interfere with each other, leading to bottlenecks. Also, parallel programs require communication between processing elements: depending on the provided mechanisms, communication may be performed either in parallel with computation or instead of computation, the first option allowing better scalability. Importantly, a holistic view is required to map a problem to a parallel computer architecture with maximum scalability.

Flynn's taxonomy [34] provides a theoretical analysis of parallel computer organization and distinguishes between *single instruction stream - multiple data stream* (SIMD) and *multiple instruction stream - multiple data stream* (MIMD). Compared to the simplex processor *single instruction stream - single data stream* (SISD) a processor executes a single instruction on multiple data elements at once or multiple independent instructions on multiple data elements, respectively. Modern CPUs (e.g. Intel® Xeon™ Platinum 8180M) combine different techniques belonging to both SIMD and MIMD categories to maximize parallel performance:

- *Instruction-level parallelism* (MIMD) conceptually breaks up the linear instruction stream of a single program into multiple independent instruction streams

if possible. The CPU monitors the linear instruction stream within a certain window during program execution and issues instructions with resolved data dependencies which can then be processed in parallel by multiple execution units (superscalar execution). While instruction-level parallelism is automatic, a programmer may improve performance of a program by reducing data dependencies between instructions and choosing instructions to match the capabilities of the execution units, i.e., a single execution unit may be designed to process only a subset of all instructions.

- *Vector instructions* (SIMD) operate on extended registers with multiple data elements, thus exploiting data-level parallelism. For example, the Intel® AVX instruction set operates on 256 bit wide registers, either as eight doublewords or four quadwords. An *add* instruction with two AVX registers executes in the same time as with a single-element registers but performs up to eight times the work. Vector instructions may be either generated automatically by a compiler from standard source code or inserted explicitly by a programmer as compiler-intrinsic functions.

- *Multi-core* (MIMD) execute multiple threads of a program in parallel within a shared memory space. Physical cores fully replicate all execution units which may be shared among multiple logical cores with their own set of registers to optimize utilization. For example, the Intel® Xeon™ Platinum 8180M has 28 physical cores which are shared by two logical cores each, for a total of 56 cores. Each core executes a separate program thread, which may be either created explicitly by the programmer or automatically by the compiler.

- A *cluster* is composed of a set of processor nodes which are interconnected by a network. Like the multi-core processor, the cluster is also a MIMD architecture but with a distributed memory space. Data is transferred explicitly between different nodes using a network adapter and the latency and bandwidth of the network transfers is significantly reduced compared to shared memory. However, clusters allow to scale to much higher number of nodes compared to the number of cores in a multi-core processor

The different types of parallel processing listed above are discussed in more detail in the following sections.

2.5.1 Instruction Level Parallelism

A program is defined by a sequence of instructions stored in memory. During execution, the instruction pointer identifies the current position within the sequence, i.e., the memory address of the instruction to be executed next. Following the execution of the instruction, the instruction pointer is incremented by the corresponding size in bytes to point to the next consecutive instruction in the sequence. Consider the following example:

```
// Assembler source code for eax = (eax + 3) * (edx + 5)
#address    #instruction    #destination, source operands
4004b2      add             eax, 3
4004b5      add             edx, 5
4004b8      imul            eax, edx
```

The instruction pointer moves through the sequence of addresses from top to bottom and the corresponding instructions get executed one after the other. A superscalar

TABLE 2.1: Overview of vector instruction set extensions available on Intel® CPUs, as documented in [59]. In addition to the regular vector registers AVX-512 includes seven 64-bit mask registers.

Extension name	Width	Registers
SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2	128 bit	8
AVX, AVX2	256 bit	16
AVX-512	512 bit	32 (+7)

architecture is able to process the two add instructions in parallel since they do not have data dependencies with respect to each other. This is made possible by monitoring the instruction stream ahead of the current instruction pointer within a range referred to as the out-of-order execution window. As the name suggests instructions within this window may be executed in any order and in parallel as long as data dependencies are resolved and sufficient hardware resources are available.

While out-of-order execution is transparent to a program, the implementation of an algorithm directly affects the amount of instruction level parallelism a superscalar processor is able to extract and thus the instruction *throughput* achieved by the program. Associated with every instruction is a *latency* and one or multiple *execution ports*. In every clock cycle a single instruction can be issued to each of the processor's execution ports. Hence, in order to achieve a throughput greater than one instruction per cycle a mix of instructions that can be issued to different ports is required. The latency of an instruction determines how many clock cycles must elapse before a consecutive dependent instruction can be issued. Hence, for maximum throughput a sufficient number of independent instructions must be available to keep the execution ports busy in between dependency chains. A more detailed and comprehensive description of the superscalar behaviour of specific processor architectures can be found in [58]. Also, [35] lists latency and execution ports for instructions of many different processors.

2.5.2 Vector Instructions

Vector instructions differ from regular instructions in that they operate on dedicated registers containing multiple data elements. They are commonly defined as extensions to existing scalar instruction sets. Table 2.1 lists a succession of vector instruction set extensions for the x86 instruction set architecture (ISA) available on many Intel® and AMD® CPUs. The SSE* instruction set extensions offer eight 128-bit registers (named *xmm0-7*) and about 300 vector instructions that treat the contents of a register as either 1,2,4 or 8 byte data elements. For example, the instructions *addps xmm0, xmm1* and *paddq xmm0, xmm1* add the two vector registers *xmm0* and *xmm1* as either four 32-bit IEE754 floating point values or two 64-bit integer values, respectively.

AVX and AVX2 supersede the SSE line of instructions, increasing the number of vector registers to 16 and the vector width to 256 bit (named *ymm0-15*). Most original SSE* instructions are promoted to the new vector width and a few new instructions are added, most notably cross-lane permutation and fused multiply-add for floating point data. The lane concept of AVX logically subdivides a *ymm* register into two

adjacent *xmm* registers, each referred to as a lane. Instructions that move data between lanes are called cross-lane operations and are more expensive in terms of CPU cycles compared to their in-lane counterparts. This is mostly a technical restriction affecting some programming choices. AVX also introduces a non-destructive three operant syntax which reduces register pressure, i.e., the number of registers required by a piece of code, and saves move instructions. For example, *addps xmm0, xmm1* becomes *vaddps ymm2, ymm0, ymm1* where the destination operant *ymm2* contains the result while the source operants *ymm0-1* remain unaffected by the operation. A novelty of AVX2 are gather instructions that allow to fetch individual data elements from disjunct memory addresses into the same vector register with a single operation.

The latest instruction set extension is AVX-512 with 32 512-bit vector registers (named *zmm0-31*), seven 64-bit mask registers (named *k1-7*) and a multitude of promoted and new instructions. Instructions are organized in several AVX-512 sub-groups with varying availability on different CPUs. If not noted otherwise, all references to AVX-512 are limited to the AVX-512 Foundation group which is available in all implementations. The lane concept prevails, organizing a *zmm* register into four *xmm* blocks, but with new, more flexible cross-lane operations compared to the original AVX. A mask register combined with a vector instruction allows to select or disable specific vector elements to alter or void the effect of the operation on the respective data. AVX-512 further complements the AVX gather instructions with respective scatter instructions that are able to store data elements from the same vector register to individual disjoint memory locations.

Programming

Writing a program with vector instructions in assembler is straight forward using the corresponding vector instruction mnemonics, of which some examples are printed in the previous paragraphs. However, assembly language is a tedious and error-prone approach for a complex program and humans tend to perform inferior compared to compilers in terms of scheduling of instructions and register allocation.

An active field of research is auto-vectorization of regular scalar source code, such as C/C++. Different approaches exist, using language extensions, preprocessing directives or code analysis. While many of these methods claim ease-of-use and portability, they often produce results with non-optimal performance. The success of auto-vectorization strongly depends on the type and formulation of an algorithm and usually this means the algorithm must be simple and the formulation tuned for the particular auto-vectorization method. In particular, current auto-vectorization methods are not capable of restructuring or inventing new algorithms to optimize support for vectorized implementations. However, this exactly is an important goal of this dissertation.

An alternative to assmebler instructions is provided by compiler intrinsic functions, referred to as *intrinsics*. Intrinsics look like regular function calls with arguments and a return value, however the compiler maps these function calls one-to-one to a single vector instruction. The types of arguments and return value represent vector registers, for example

```
// C/C++ source code
__m256 a,b,c;
c = _mm256_add_ps(a, b);
```

, is equivalent to the previous mnemonic example *vaddps ymm2, ymm0, ymm1*. The compiler remains responsible for instruction scheduling and register allocation at which it excels, while the programmer is able to precisely pick the instructions intended for a particular algorithm. Intrinsics can be easily mixed and matched with regular source code and thus only need to be used where necessary, often only in short sequences.

Data Movement

Movement of data between vector registers and memory as well as within a vector register is a major performance consideration when designing vectorized algorithms. A basic load instruction moves a continuous block of data from a source address into the destination register and vice versa for stores. Hence, in order to assign each vector element the intended datum the data structure must be laid out in memory accordingly.

A classic example is the element-wise multiplication of two lists of complex numbers. If the real and imaginary parts of the complex numbers are stored in separate arrays, loading from the real or imaginary array results in a vector register filled with consecutive real or imaginary values as required. However, if real and imaginary part are stored as tuples in the same array, loading from the array results in a vector register with interleaved real and imaginary values.

The first case corresponds to a structure-of-arrays (SOA) layout, the second to a array-of-structure (AOS) layout. The AOS layout often requires additional instructions to rearrange the loaded data to match the desired vector element. In the worst case each data element needs to be loaded from a separate address and combined with the already loaded data in the destination vector register. Such serialization of memory access can diminish performance gains from vectorized code, leading to poor scalability. SOA layouts are ideal in this respect because a single load places each datum where it is required in the vector register. A problem that can arise with SOA layout is poor data locality if multiple, large arrays, i.e., many objects with multiple components, are required for a computation. In this case a AOSOA layout, combining AOS and SOA where the array size of the SOA part corresponds to a small multiple of the vector size, helps by packing data in chunks that are cache friendly and conveniently accessible by vectorized code.

Branching

Conditional branches in the instruction stream can lead to divergence in vectorized code. The outcome of a vector comparison may be either identical for all elements, i.e., all true or all false, or mixed, i.e., some true and some false. If all vector elements agree the case is identical to scalar code and there is no divergence. However, if there is disagreement the many results must be mapped to a single binary decision, i.e., take the branch or not. Such a mapping could be, for example: take the branch if at least one vector element does, do not take the branch if at least one vector element does not, take the branch if most vector elements do.

Translated to AVX2 code, the mappings could be expressed as follows:

```
// C/C++ source code
__m256 a,b,c; int m;
```

```
// m[i] = a[i] < b[i] ? 1 : 0;
m = _mm256_movemask_ps(_mm256_cmplt_ps(a, b));

// take branch if at least one element does
if(m != 0) { ... }

// do not take branch if at least one (out of 8) element does not
if(m != 0xff) { ... }

// take branch if most (out of 8) vector elements do
if(_mm_popcnt_u32(m) > 4) { ... }

// iterate over active elements
while(m != 0) {
    int index = _tzcnt_u32(m);
    ...
    m = _blsr_u32(m);
}
```

The vector comparison instruction either sets all bits of an element or none in the destination register. The subsequent *movemask* instruction concatenates the most significant bits, i.e., the sign bits of the vector elements into a compact bit mask. The *popcnt* instruction counts the number of set bits, i.e., active elements. In additions to the mappings, the *while* loop demonstrates how to iterate over active elements and obtain the corresponding element index. The *tzcnt* and *blsr* instructions count the number of zero bits until the first set bit starting from the least significant bit and clear the first set bit, respectively.

Following the decision with mixed comparison results about which code path to take, the inactive vector elements which do not agree with the decision continue to execute (meaningless) instructions on their data. This is acceptable, as long as the data is eventually discarded and intermediate results are not used for memory accesses. Some care must be taken with floating point operations not to generate denormal or not-a-number values by accident in the inactive vector elements because such value can severely affect instruction performance.

Alternatively, modification of inactive vector elements can be prohibited by using *masking*. A bit mask generated from the comparison indicates which elements should be affected by a successive instruction and which should be passed through unmodified.

For AVX2 and below, the mask vector resulting from a vector comparison can not be used to prevent modification of selected vector elements directly, but it can be subsequently applied to combine modified and unmodified elements corresponding to destination and source register of an instruction, for example by using *blend*:

```
// C/C++ source code
// c = a < b ? a + b : a
__m256 a,b,c,m;
m = _mm256_cmplt_ps(a, b)
c = _mm256_add_ps(a, b);
c = _mm256_vblend_ps(a, c, m);
```

For AVX-512, the masking functionality has been extended to include dedicated mask registers and masked instruction support. The code snippet above can be transformed into:

```
// C/C++ source code
__m512 a,b,c; __mmask16 m;
```

```
m = _mm512_cmplt_ps(a, b)
c = _mm256_add_ps(a, m, a, b);
```

The mask register *m* allocates a single bit per vector element to store the result of the comparison. The subsequent addition only updates elements in *c* with the sum of *a* and *b* if the corresponding bit in *m* is one, otherwise *a* is passed through. The masking implementation in AVX-512 saves instructions and also applies to memory accesses, i.e., inactive elements are neither read from nor written to memory during a masked vector instruction with memory operands.

2.5.3 Multi-core

A multi-core processor supports the execution of multiple threads of a program in parallel. The threads run asynchronously with respect to each other and execute independent tasks within a shared memory address space. As long as data is read-only the threads can access the same memory region in parallel without conflicts. As soon as data is modified within a memory region that is accessed by multiple threads synchronization is required to avoid data inconsistencies.

Commonly used synchronization primitives include the *barrier* and the *lock*. A barrier ensures that a group of threads, upon finishing execution of the barrier, have passed a well defined point within the program such that all expected reads and writes to a memory region prior to this point have been completed. A lock guards a defined region of memory and gives access to a single thread only at a time. Other threads must wait or retry at a later time if they require access to a locked memory region. Waiting is performed either by a busy loop repeatedly checking the memory state of the lock or by yielding to the operating system which suspends the thread until the lock becomes available. Barriers and locks impact scalability of a program by blocking threads while waiting on other threads to catch up or to release a lock.

Non-blocking synchronization is possible with atomic operations. Atomic operations load an operand from memory, perform the designated operation, e.g., a conditional swap or an addition, and write the result back to memory, all within a single indivisible transaction. This ensures a sequential ordering of operations among threads performing an operation on the same variable in parallel. Atomic operations are used to implement barriers and locks and can be used to implement custom specialized synchronization primitives, see Chapter 5 for examples. On processors with a shared last level cache for all cores atomic operations usually take a few CPU cycles only. Scalability issues can arise in case of congestion, i.e., if many threads operate on the same cache line with a high frequency. Independent of the atomicity of operations, writes from alternating threads to a shared cache line can significantly increase memory traffic even though different memory addresses are accessed, a situation known as *false sharing*. The reason is that memory coherence is established on cache line granularity and writes by one thread invalidate a cache line for all other threads which then must be reloaded.

Sharing of data, either true or false, becomes more critical in the case of non-uniform memory access (NUMA), i.e., if for a specific core some memory regions are more expensive to access than others. For example, in a multiprocessor system each processor has a set of physically close memory banks which are favourable to access in terms of latency and memory bandwidth compared to memory banks associated with another processor. Thus, private memory regions should be allocated local to

the processor a thread is running on and shared memory usage reduced to a minimum.

Ideally, the scalability of a parallel program running on a multi-core processor is expected to be linear in the number of cores. Otherwise, either synchronization, false sharing or contention of shared resources like memory bandwidth or cache capacity are likely responsible for performance discrepancies. Another factor may contribute to sub-linear scalability: dynamic clock frequency adjustments. Processors with only a few busy cores can sustain higher clock frequencies compared to full utilization while remaining within power and thermal specifications. For example, a program might be processed at 3GHz while using only a single core and at 2GHz if all available cores are utilized.

Also, in order to increase instruction level parallelism and thus utilization of execution units physical cores are often subdivided into multiple logical cores, usually two or four. However, two threads running on the same physical core cannot be expected to be as fast as if they ran on separate physical cores. Depending on how well a single thread already utilizes the execution units of the physical core, two threads can roughly achieve a scalability factor between 1.0 and 1.5.

2.5.4 Cluster

A cluster is a supercomputer that consists of a set of processor *nodes* interconnected by a network such as Ethernet or InfiniBand. Similar to multi-core processors a cluster can be categorized as a MIMD architecture where multiple instruction streams are processed in parallel. The main difference is the distributed memory organization: each node has local memory with a local address space instead of shared resources. Data exchange between nodes is performed by network adapters which copy the data as instructed from a remote memory location to a node's local memory. A node's processor can then access the local copy of the remote data. Due to limited network bandwidth and increased latency access to remote memory is usually an order of magnitude slower compared to local memory.

The prevalent programming model for clusters is based on message passing. Both the sender and receiver nodes initiate a transfer on their respective local side. The sender node copies the data in a dedicated staging area from where the sender network adapter reads the data and sends it over the network to the receiver network adapter. The receiver network adapter writes the data to a dedicated staging area from where the receiver processor can copy the data to the desired location in local memory. The advantage of the message passing model is that it is easy and save to use. However, application performance is negatively impacted by the copying overhead and *two-sided* nature of the transfer: both sender and receiver processors are actively involved and need to synchronize. Also, algorithm design based on a synchronous communication paradigm is encouraged, resulting in algorithms with inherent scalability limitations.

An alternative approach is known as *remote direct memory access* (RDMA). Here, data is transferred directly between two nodes' memories by the respective network adapters' DMA controllers without intermediate copies to staging areas, a technique known as *zero-copy*. Also, RDMA operation are *one-sided*, i.e., the communication is initialized by the local node only and the data transfer does not interfere with either

node's processor during the transfer. This allows to design algorithms which overlap computation with communication. The communication performed by the network adapters is asynchronous with respect to both the local and remote node. The result is negligible cost of communication in terms of CPU cycles enabling highly scalable algorithms.

2.6 Parallel Ray Tracing

The computational complexity for generating ray-traced images, especially when including global illumination, motivates the exploitation of parallel computation resources. Rendering algorithms solving the path integral based on ray tracing are highly amenable to parallelization since different samples are independent of each other. Hence, ray tracing is often categorized as an embarrassingly parallel algorithm.

However, different techniques for improving performance introduce new dependencies between samples by sharing data or exploiting data parallelism. Data sharing can be beneficial either implicitly by increasing cache hit rates or explicitly by reusing intermediate results. For example, reuse is effective for feature reconstruction via filtering, adaptive sample placement and caching of intermediate steps in bi-directional approaches. Data parallel sample evaluation dictates a shared control flow between a group of samples. Efficiently managing divergence in the data-driven control flow of different samples is a challenging problem. Also, exposing parallel computation within a single sample is possible but difficult to turn into performance gains.

Further, an important part of scalable parallel rendering is the load balancing, i.e., distributing the parallel work among the parallel units, and aggregation of partial results into the final image, here referred to as the *distribution framework*. Load balancing should be as fine-grained as necessary in order to avoid execution stalls but negligible in computation so that the actual rendering is not obstructed. The common approach for MIMD architectures is to divide the image plane into independent tiles to determine the granularity of the load balancing. For shared memory multi-core processors tile distribution is efficiently implemented using simple atomic counters but for clusters load balancing is more involved because of the network bottleneck and a generally larger number of parallel units, i.e, nodes.

In the sections below, the previous work related to the research reported in this dissertation is examined and connected to the new contributions described in the later chapters. First, the important concept of *ray coherence* is defined, followed by vectorized traversal for single rays and ray groups, parallel BVH construction and distribution frameworks for clusters.

2.6.1 Ray Coherence

The coherence of a ray set can be defined as the tendency of the individual rays to follow the same traversal path through an acceleration structure, e.g. the BVH, and to intersect the same primitives. Perfectly coherent rays perform identical operations on the same data throughout the entire traversal routine.

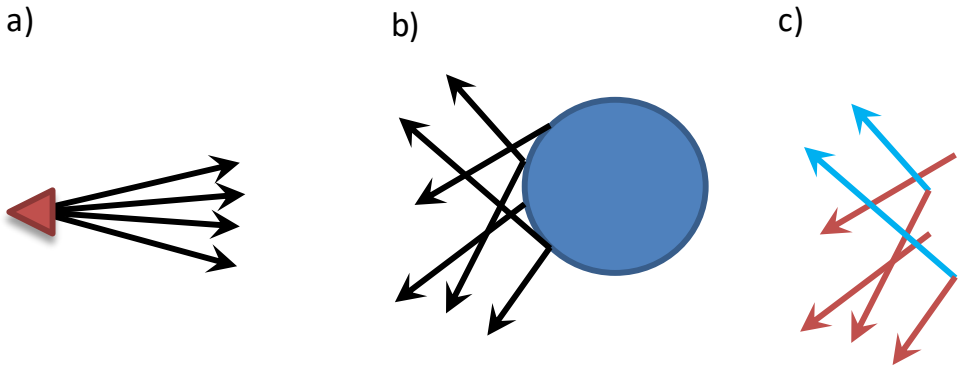

FIGURE 2.7: The two approaches to ray coherence. (a) Structured coherence: the algorithm knows that all rays in a group have high coherence. The example shows primary rays originating at the camera. (b) Unstructured coherence: The algorithm cannot make any assumptions about the coherence of a group of rays. The example shows rays originating from diffuse reflections. (c) Sub-groups of unstructured rays may have strong coherence which a traversal algorithm can take advantage of.

Primary visibility queries, for example, produce well *structured coherent* ray sets with a common origin and tightly bundled ray directions along the camera orientation that often share a common traversal path down to the leaf nodes, see Figure 2.7a.

In contrast, the rays spawned by diffuse global illumination calculations can originate anywhere on the scene surface and have their directions randomly distributed, leading to a divergence of the rays' traversal paths already at the upper levels of the hierarchy, see Figure 2.7b. However, even for ray sets without structured coherence groups of rays might end up following a shared path partially or completely during traversal, i.e., exhibit *unstructured coherence*, as indicated in Figure 2.7c.

2.6.2 Vectorized Traversal

A prerequisite for vectorization is availability of data parallelism in the traversal routine. By structuring the traversal algorithm in different ways vector elements can be either filled with multiple nodes or multiple rays or a mixture thereof. For a benefit in case of multiple rays the existence of either structured or unstructured coherence among the rays is mandatory. In the following, previous work related to the different approaches towards data parallelism in BVH traversal is introduced.

Single Rays

For traversal of a single ray through a binary BVH the exposed amount of data parallelism is scarce since a single traversal step requires only little computation and consecutive steps have sequential dependence. A multi-branch BVH [32, 24, 107] reduces the depth of a binary hierarchy by removing intermediate nodes to make it possible in a single traversal step to test multiple child nodes. In addition to increased memory access coherence and fewer traversal steps overall this approach enables data-parallel bounding box intersection tests using vector instructions. Computation time spent during a traversal step is shifted from the intersection test to stack operation and child ordering.

The 4-ary BVH has been combined with the sign heuristic either by storing the intermediate binary BVH [24] or by a local look-up table stored within the nodes [32]. The corresponding results demonstrate a significant performance gain over the binary BVH, the kd-tree and the bounding interval hierarchy [104]. The 16-ary BVH paired with the distance heuristic has been evaluated on a simulator for a 16 element wide vector instruction set extension [107]. The comparison to a binary BVH packet traversal yields only minor performance gains.

The current multi-branch traversal methods do not scale well towards higher branching factors, i.e., vector widths, primarily because of the following limitations: (1) node ordering and stack operation do not saturate vector registers and requires increasingly more instructions per traversal step while (2) the gain of higher branching factors diminishes because the reduction in total traversal steps is only logarithmic at best. Chapter 3 describes new techniques for more scalable multi-branch traversal by addressing limitations (1) and (2).

Ray Packets

A ray packet [112] is a set of rays which are traversed simultaneously in a data-parallel manner using vector instructions where each ray is mapped to a single vector element. Because of the shared control flow rays within a packet should be as coherent as possible, i.e., follow the same traversal path, in order to achieve a high vector utilization. In case of divergence only a subset of the vector elements corresponding to active rays perform useful calculations. For tightly bundled rays such as camera rays or shadow rays towards small light sources identical paths are the common case and a significant speed-up is observed for packet tracing.

Structured coherence allows to generate ray packet proxies such as frustums, intervals and corner rays [19] which can be used to cull child nodes conservatively for the entire packet, reducing the number of executed ray-bounding box intersections.

Packet proxies are especially efficient if the number of coherent rays is large: the *DynBVH* traversal [108] for the binary BVH supports arbitrarily sized packets. Such a *large packet* is composed of multiple smaller packets matching the native vector width. The algorithm employs arithmetic culling and speculative decent to accelerate packet traversal by algorithmic means in addition to vector scaling.

The formidable results of the DynBVH applied to primary and shadow rays have motivated assembly strategies for packets of rays originating from general global illumination effects [20]. Good results have been demonstrated for specular reflection and refraction.

As a ray packet descents deeper down into the BVH coherence may degrade as more and more rays may become inactive, i.e., diverge from the current control flow. Combining packet and single ray traversal [12] allows to switch between the two modes depending on packet utilization. A threshold on the number of active rays in the packet determines when it becomes more efficient to proceed with individual traversal.

Ray packets have been demonstrated to perform very well on binary BVHs. Following the development of multi-branch traversal for single rays the question has

been raised whether ray packets can be integrated with high performance within the same structure as well [107]. Chapter 3 introduces a very fast packet traversal algorithm that can be combined with single ray traversal for multi-branch BVHs featuring high branching factors.

Ray Streams

The efficiency of packet traversal has motivated algorithms that attempt to extract coherent sub-sets out of incoherent ray sets, i.e, organizing unstructured coherence into structured coherence. Stream filtering and related methods [84, 113] intersect a group of active rays with the current node in a breadth-first manner. Rays missing the current node are taken out of the stream upon decent, extracting coherency implicitly during tree traversal. While this approach promises high vector utilization, it requires expensive gather and scatter operations or sorting which are detrimental to traversal performance. Also, saturating the vector unit necessitates one ray per vector element. If only a few rays are active, e.g., in the lower levels of the BVH, or the size of the stream is not a multiple of the vector width computation in wasted.

Improving the effective vector utilization is possible by combining ray streams with the multi-branch BVH [100]. A single ray is used to saturate a 4-element wide vector register while traversing a 4-ary BVH. The stream approach helps to amortize the costs among the rays related to node ordering and accessing and setting up the node data and, at the leaves, primitive data. Like previous streaming algorithms, a major drawback is that a common traversal order is enforced for every ray in the stream. This reduces the effectiveness of early culling as some rays might visit nodes in reverse order.

The dynamic ray stream traversal (DRST) [11] relaxes the requirement of a single traversal order: in the case of 4-ary BVH each ray can follow approximately the same traversal order that would result from individual traversal with the distance heuristic. Despite the high performance reported for DRST, two major flaws are apparent in the algorithm design:

- First, the flexible traversal order results in a large number of bins leading to fragmentation of the ray stream and increased book-keeping overhead. Accordingly, the method achieves its full potential only for large ray streams.
- Second, despite the flexibility, the number of permitted traversal order permutations is still limited to 8 (2!*2!*2!) out of the possible 24 (4!) for a cluster of four child nodes.

Both of these issues are addressed in a new stream traversal algorithm described in Chapter 3.

2.6.3 Bounding Volume Hierarchy Construction

An acceleration structure such as the BVH is essential for efficient ray tracing. For applications that interact with dynamic and massive scenes a fast BVH construction method is key to reduce the time-to-image. Hence, parallelization of BVH construction is of particular interest in order to increase build performance.

A variety of construction algorithms exist that can be categorized as divisive top-down and agglomerative bottom-up types. The linear BVH (LBVH) [76] is one of the

fastest implementations regarding construction speed and part of the agglomerative family. Its efficiency originates from a linear time complexity and straightforward parallelizability where the primitives are sorted into an implicit octree-like structure defined by the Morton space-filling curve, followed by a simple merging procedure to construct the hierarchy of bounding boxes. However, due to the predetermined structure this method does not adapt to the scene geometry and produces BVHs of low quality, resulting in inflated ray tracing times. Various extensions aim at improving LBVH quality, such as approximate agglomerative clustering [50], post-process optimization [67, 16] or hybrid strategies [85, 43]. However, the quality remains inferior to divisive construction based on the surface area heuristic (SAH) [47, 51, 3].

Augmenting the SAH-based algorithm with the option to split primitives, if cost effective, leads to the split BVH (SBVH) [97, 91] which produces the highest quality BVHs of all known methods. The drawback of the divisive algorithms is the increased time complexity $O(N \log N)$ over the agglomerative approaches based on the LBVH and the increased difficulty for scalable parallelization. In particular, efficient memory management for primitive splits becomes an issue in the presence of multiple threads. While parallelization schemes have been proposed for BVH construction without splits [105, 106, 17], an scalable solution for the SBVH is not yet available.

Also missing from the research so far is the utilization of data parallelism by application of vector instructions to accelerate performance critical parts of the construction algorithms. Chapter 4 develops a parallelization framework for high-quality BVH and SBVH construction addressing previous scalability bottlenecks for multi-threaded execution and integrating support for vector instructions.

2.6.4 Cluster

Massively parallel rendering using supercomputers such as clusters has broad application in a variety of disciplines. Existing rendering approaches and systems can be categorized by considering the main aspect underlying their design, including: rendering algorithms (e.g., ray tracing [22, 115], volume rendering [66]); latency (e.g., real-time rendering [61], off-line rendering [89]), data management (e.g., distributed rendering [78, 89], replicated rendering [29]), and target architecture (e.g., cluster [110], shared-memory [96]). The computational cost involved in achieving the desired rendering quality and scale is rapidly increasing, and research must consider both hardware and software to achieve the necessary rendering efficiency. An overview of research challenges in parallel rendering is provided by [13].

An essential part of every parallel rendering stack is a scalable communication layer, i.e., a *distribution framework*, that provides a transparent interface between parallel hardware architecture and rendering algorithms for distribution of rendering tasks and aggregation of results. Distribution frameworks for distributed memory architectures are traditionally designed with a message passing paradigm in mind [36] where two communicating nodes actively send and receive messages. For multi-core nodes a hybrid approach combining message passing with shared memory communication between cores of the same node has been found to reduce the overhead of message passing implementations [55].

A partitioned global address space (PGAS) provides a fundamentally different approach to distributed memory communication, where nodes can directly access the memory of their peers without synchronization or involvement of the remote node [44]. This approach has the potential to scale more efficiently compared to message passing if the algorithms themselves are designed to be asynchronous and one-sided. The research in Chapter 5 brings the PGAS paradigm to the domain of cluster-based rendering upon which a distribution framework for real-time path tracing is designed.

An early approach for interactive ray tracing performed on a cluster [110] demonstrates good scalability for a small number of nodes with 14 cores in total. Frames may overlap for improved load balancing, and tiles are preferably assigned to the same node when transitioning from frame to frame, to take advantage of data locality. The centralized tile distribution step, actively performed by the display node, does not cause a bottleneck in the experimental set-up due to the small number of rendering nodes used.

The method discussed by DeMarle et al. [29] recognizes the centralization bottleneck arising in a larger cluster environment. They propose to use a decentralized work-stealing strategy for load balancing by randomly selecting victims among the participating rendering nodes. An additional benefit of their method is the implicit data locality – previously obtained explicitly – since tiles remain with the same node when processing frames, unless tiles are stolen. The steals are scheduled at the end of a frame, which may not always be frequent enough to achieve optimal load balance. A 64 core set-up is used for the results but no exact numbers regarding scalability are provided.

Cosenza et al. [23] improve the work-stealing design by allowing steals to be performed throughout a frame, on demand. Their approach generates a predictive cost map by rendering an approximation of the current view on the display node's GPU; all tiles are distributed, at the start of a frame, among the rendering nodes considering balanced cost. Imbalances arising from the initially static tile assignment during rendering are counteracted by work stealing. The results show parallel efficiencies between 83% and 98% scaling up to 84 cores.

A similar approach is described in [98], which relies on frame-to-frame coherence to generate a predictive cost map based on a previous frame's load distribution. This method is demonstrated to be sufficiently accurate for simple ray tracing work loads and a medium-sized cluster totalling 240 cores, achieving parallel efficiencies between 92.9% and 95.9%. Communication is completely avoided throughout a frame, which is greatly beneficial for cluster environments without high-performance interconnects.

The approach covered in [61] performs centralized tile assignment, similarly to the method presented in [110]; however, the targeted cluster environment is more modern and significantly larger with multi-core nodes and InfiniBand network. It is proposed to use the InfiniBand network's RDMA capabilities for efficient one-sided communication, but the described implementation is not able to perform one-sided operations due to the underlying communication library's limitations. The results are obtained on a 512 core cluster and show sub-linear scalability although exact numbers are not provided.

In this dissertation, the goal is to research and design a distribution framework capable of achieving linear scalability by using, in contrast to all previous work, a fully asynchronous, one-sided PGAS approach. Also, the targeted system totalling 1200 cores is considerably larger compared to previous results which allows a more accurate evaluation of scalability. Chapter 5 describes the new distribution framework and the corresponding algorithms and techniques in detail.

2.7 Conclusion

Photo-realistic rendering is a tremendously complex computation. A major part of this complexity stems from frequent global visibility queries that must take into account the entirety of a scene. The visibility queries are processed by the ray tracing algorithm; despite efficient acceleration structures such as the BVH and a multitude of techniques to reduce the amount of queries, sequential computers are not fast enough to meet the computational demand of real-time photo-realistic rendering. Hence, this dissertation's research pursues parallel algorithms to allow the problem to scale to as many parallel computational units as necessary.

Chapter 3

Vectorized Bounding Volume Hierarchy Traversal

The traversal of a ray through a BVH quickly reduces the scene geometry to a small set of potentially intersecting primitives. The traversal operation accounts for a large share in the overall ray tracing time, thus acceleration of the ray traversal has a significant effect on ray tracing performance. Accordingly, the research focus in this chapter is on efficient, vectorized traversal algorithms exploiting data parallelism. Two known techniques, the multi-banch BVH and ray sets, allow to expose data parallelism for ray traversal. Section 3.1 introduces a novel traversal algorithm for tracing single rays through wide multi-branch BVHs. Section 3.2 and 3.3 introduce novel traversal algorithms for ray sets of unstructured coherence, i.e., stream traversal, and for ray sets of structured coherence, i.e., packet traversal, respectively. This chapter is based on two publications by Fuetterling et al. [39, 38].

3.1 Multi-branch Traversal

Traversal algorithms for single rays generally can be divided into separate phases, which are: ray setup, inner node traversal, leaf intersection, and the stack pop. *Ray setup* performs pre-computation to facilitate efficient execution of the remaining phases. During *inner node traversal* the ray descents down the BVH until it misses all children of an inner node or encounters a leaf. In the first case, traversal directly proceeds to the stack pop; in the second case, *intersection with the leaf's primitives* is performed first, reducing the maximum ray distance t_{far} to the closest primitive intersection (if any). The *stack pop* takes the top node from the stack (if a node exists) and determines whether the corresponding ray entry distance is still within t_{far}. If this is not the case, the next node is taken from the stack. After the stack pop ray traversal continues with inner node traversal.

A multi-branch tree, compared to a binary tree, allows every inner node of the hierarchy to have more than two child nodes. For ray tracing, the benefit of a multi-branch BVH becomes evident in the context of vectorized traversal algorithms. One costly part of the inner node traversal is the intersection test of the ray with the child nodes' bounding boxes. Since the floating point computation can be performed over multiple bounding boxes in a data-parallel manner an opportunity for vector instructions is created. The vector utilization is optimal if the number of child nodes, i.e., the branching factor, is equal to the vector width because every vector processing element is tasked with one ray-box intersection test.

However, the multi-branch approach affects other parts of the traversal algorithm as

well. Most notably, determination of the traversal order, i.e., the order at which intersected child nodes are subsequently traversed, becomes more difficult. Efficiently finding the first intersecting primitive along a ray necessitates to descent into the hierarchy in a front-to-back manner, i.e., to always follow the child which is closest to the ray origin. Thus, a sorting step must be added to the multi-branch inner node traversal whereas a simple comparison is sufficient in the binary case. The complexity of the sorting step increases with the number of child nodes n as $\mathcal{O}(n \log n)$ and counteracts the performance benefit obtained from intersection test vectorization.

The stack push is another part of the inner node traversal that becomes more complex in a multi-branch scenario. After sorting the child nodes according to intersection distance all intersected nodes, except the closest, must be placed in reverse order onto the stack. This requires looping through a list of intersected child nodes and placing them on the stack one-by-one, resulting in a $\mathcal{O}(n)$ complexity.

Considering the scalability of such a multi-branch traversal algorithm, the ideal $\mathcal{O}(1)$ complexity for the vectorized intersection test is counteracted by traversal order determination and stack push with $\mathcal{O}(n \log n)$ and $\mathcal{O}(n)$ complexities, respectively. Hence, increasing n leads to a performance gain up to a n_{max}, after which performance starts to decline with respect to the $\mathcal{O}(\log n)$ binary traversal of a multi-branch node and eventually turns into a performance loss.

A new traversal algorithm for wide vector units WIVE introduced in the following solves the complexity problem, yielding $\mathcal{O}(1)$ for the entire inner node traversal. Instead of ordering child nodes by ray distance WIVE utilizes precomputed front-to-back traversal orders based on the split axes of the BVH construction and ray signs.

3.1.1 Multi-branch Traversal Order

Sorting child nodes by intersection distance from the ray origin is not the only efficient method to determine a good traversal order [79]. In fact, the distance order does not guarantee perfect front-to-back traversal, as demonstrated by the following example: if two child nodes overlap and the ray pierces through the overlapping volume, a ray-primitive intersection in the sub-tree of the far child node might be closer than a primitive intersection in the sub-tree of the near child node (Figure 2.6). An order decision resulting in prefect front-to-back traversal is not possible without investigating all overlapping child node sub-trees first. Hence, a heuristic must be used to generate an approximation.

The sign heuristic is a competitive alternative to the distance heuristic. A simple example illustrates the essential idea: two non-overlapping child nodes' bounding boxes are separated along the x-axis; intuitively a ray with a positive sign in the x-direction should always choose the child node with the smaller x-coordinates first. The separating axis and the corresponding order of a child node pair can be precomputed and stored with the node data structure and retrieved during traversal. If no separating axis exists, i.e., the child nodes overlap, the axis with the shortest projected overlap can be chosen. Compared to the distance heuristic, the sign heuristic produces an identical order in case of a separating axis and a possibly different order otherwise, depending on the actual ray and child node geometry.

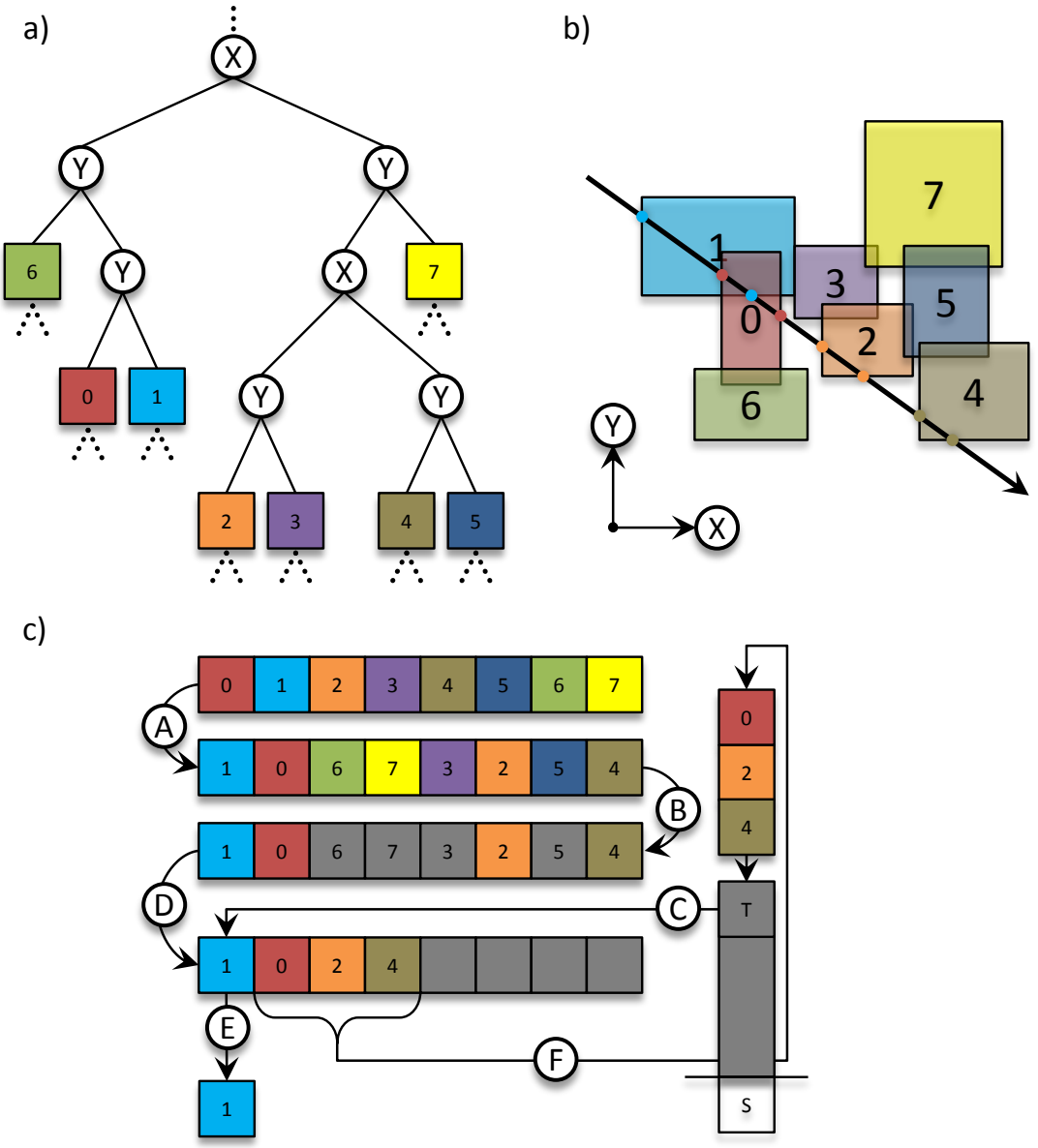

FIGURE 3.1: WIVE single ray algorithm for wide vector units. (a) A treelet embedded in a larger binary BVH. Collapsing the treelet yields a 8-ary BVH node cluster (colored squares). The inner nodes of the treelet (disks) are labeled according to the split axis used during binary BVH construction. (b) Bounding boxes of the 8-ary BVH node cluster from (a) in a 2D, xy-coordinate system. A ray with positive x and negative y sign is shown, with marked entry and exit points on the edges of bounding boxes. (c) Ordered traversal for the ray in (b). The initial order of the nodes is based on the order in memory, which is chosen to conform with positive ray signs. Step A performs the node permutation for the ray, which can be deduced from (a) by flipping the children of the Y nodes due to the negative y sign. Step B performs the intersection test resulting in a mask that is applied in step D for compressing the valid nodes into a continuous array. Step C loads the current stack top element which takes the first slot in the compressed result if no valid node exists. The first element of this array is extracted in step E to be the next node to be traversed and the remaining part is stored in order atop the stack in the final step F.

The sign heuristic has been applied to binary BVHs initially and later has been extended very specifically to a 4-ary BVH [32, 25]. For the WIVE algorithm a general extension to arbitrary branching factors is required, as illustrated in Figure 3.1. A multi-branch BVH is usually derived from a binary BVH, which is either generated on-the-fly for temporary storage or fully constructed a priori. Starting from the root, the binary BVH is divided into treelets according to the desired branching factor of the multi-branch BVH. An exemplary treelet is shown in Figure 3.1a. Inner nodes and intermediate leaves are represented as black circles labeled by axis of separation and colored squares labeled by memory order, respectively. The treelet's intermediate leaves form a child node cluster whereas the treelet's inner nodes are eventually removed. The inner nodes' axes of separation are used for the derivation of the front-to-back traversal order according to the sign heuristic. In the example, taking on the perspective of a ray with only positive directional components, following a branch to the left yields the near child node and following a branch to the right yields the far child node. Accordingly, the branches are swapped if a ray has a negative directional component along the axis of the branch label. In Figure 3.1b the depicted ray has a positive component in the X direction and a negative components in the Y direction. By swapping all branches in 3.1a with a Y label the traversal order as defined by the sign heuristic is retrieved, which is (1,0,6,7,3,2,5,4), see also the second row in 3.1c. Omitting all child nodes which are not intersected, this order is equivalent to the order in which the ray penetrates the child node bounding boxes and intuitively appears to be a good front-to-back approximation.

In general, a d-dimensional ray amounts to 2^d possible sign combinations, which result in up to 2^d unique traversal orders per node cluster as determined by the sign heuristic. In a n-ary BVH a cluster is composed of up to n nodes, so that a traversal order may be represented by a vector of up to n indices, where each index must be at least $\log_2 n$ bits in size. This *permutation vector* specifies how to re-arrange the nodes with respect to the base order, i.e., the order in which the nodes are laid out in memory. Thus, by pre-computing permutation vectors for all 2^d sign combinations and storing them in the cluster data structure, the traversal order for a particular ray is retrieved by concatenating its directional sign bits to form an d-bit index into the table of permutation vectors.

3.1.2 Algorithm

LISTING 3.1: Main traversal function for WIVE.

```
1  def traverseRay(node, ray)
     stack ← {stop}
3    while(true)
       if(node.isInner())
5        (elems, num) ← traverseCluster(node.cluster, ray, stack.top().getNode())
         node ← elems[0].getNode()
7        stack.push(elems[1:num], num−1)
       else if(node.isLeaf())
9        if(intersectLeaf(node, ray))
           stack.cull(ray.tfar)
11       node ← stack.pop().getNode()
       else
13       break
```

In the following the WIVE algorithm is described on a high-level with references to Listing 3.1 (using line numbers) and Figure 3.1c. The implementation details of the essential parts are deferred to Section 3.1.3.

The *ray* and the initial *node*, usually the BVH's root node, are passed as arguments to the traversal function *traverseRay* (line 1). The initialization of the node *stack* (line 2) with a terminating *stop* element is followed by the traversal loop (lines 3-12). At the beginning of an iteration the type of *node* is determined by testing if it is an inner node (line 4) or a leaf (line 7), i.e., references a node cluster or a primitive cluster, respectively (line 4). In the first case, the *traverseCluster* function returns the next *node* and a sorted list of elements referencing further intersected nodes and the corresponding entry distances (line 5). This list is pushed to the stack (line 6). In the second case, the ray is intersected with the primitives (line 8). If a valid intersection exists, the the maximum distance t_{far} of the ray is reduced accordingly and the stack is culled by keeping only elements with a node entry distance closer compared to the updated t_{far} (line 10). The compression step is optional, i.e., the algorithm functions correctly even when it is omitted, but performance is increased by removing all node references from the stack fully occluded by the latest primitive intersection, avoiding unnecessary further traversal iterations. To continue the traversal the next node is taken from the stack (line 11). If the *node* is neither inner node nor leaf (line 12) it must be the *stop* element from the bottom of the stack, indicating that all leaf nodes potentially containing the closest ray-primitive intersection have been exhausted, leading to the termination of the traversal routine (line 13). The closest primitive intersection, if it exists, is stored in the *ray* structure.

The key innovations of the WIVE algorithm lie in the *traverseCluster* function and the stack push. These parts of the algorithm are illustrated in Figure 3.1c, continuing the example defined by the node cluster and the ray shown in Figure 3.1b and the corresponding hierarchical representation of the node cluster in Figure 3.1a. The procedure begins with a vector containing the nodes of the cluster in the order in which they are are laid out in linear memory. In (A) the permutation vector selected by the ray's directional signs, i.e., (1,0,6,7,3,2,5,4) in the example, is applied to the node vector, re-arranging the nodes into the desired front-to-back order. The intersection test (B) produces a bit mask that selects only the nodes with a valid ray-bounding box intersection (colored) whereas the remaining nodes are discarded (gray). To continue the traversal, the closest node must be extracted and the remaining active nodes must be stored to the stack. If no node with a valid intersection exists, the topmost element of the stack must fill the the slot of the next active node. In order to take care of this special case, the topmost stack element T is loaded into the first slot of a separate vector (C), also used as the destination vector for the following compaction operation (D). The compaction operation concatenates all nodes marked as active by the bit mask from the intersection test by removing the inactive nodes in between and writing the continuous list of active nodes to the destination vector (D). If the list is not empty it overwrites the previously loaded stack element. In either case, reading the first element of the vector correctly extracts the closest node for the next traversal iteration (E) and the remaining nodes are pushed in front-to-back order onto the stack (F) with a single store operation. Eventually, the *stop* element S terminates the traversal.

In the case of single-instruction support for the permutation and compression vector operations, this algorithm has a time complexity of $\mathcal{O}(1)$ for both the child node ordering and stack push operations compared to the typical $\mathcal{O}(n \log n)$ complexity for sorting n active nodes and $\mathcal{O}(n)$ complexity for pushing them onto the stack.

The vectors and the formatting of the data they contain, i.e., nodes and stack elements, are still lacking a precise definition. The exact mapping of the WIVE algorithm to binary vector registers will be described next.

3.1.3 Implementation

The WIVE algorithm allows two variants of efficiently vectorized implementations. In the *full width* variant, the width of the vector registers, i.e., the number of vector elements, is equal to the branching factor of the BVH, whereas in the *half width* variant, the width is twice the branching factor. This gives WIVE a broad applicability because it can be tuned to different kinds of applications and architectures. For instance, an AVX2 implementation with a vector width of eight elements may use a 4-ary or 8-ary BVH whereas a AVX-512 implementation with 16 elements may choose between a 8-ary and 16-ary BVH for full vector utilization, assuming single precision floating point data for computation. The two implementation variants are discussed in the following two sections, exemplified for an 8-ary BVH.

The implementation descriptions do not assume a particular vector instruction set, but a certain set of operations must be supported to achieve high performance. The corresponding AVX2 and AVX-512 instructions [59] are annotated to each operation for the full width and half width implementations, respectively. Key operations for WIVE that may not be available for all instruction sets are *permutation* and *compaction*:

- *vpermq a, b, c*: Copy 32/64-bit elements from *c* selected by the lower three bit of the 32/64-bit elements in *b* to the corresponding positions in *a*.
- *vpcompressq a{k}, b*: Select 32/64 bit elements in *b* using mask *k* and compress the selected elements to form a continuous array aligned to the low element in *a*.

For example, AVX2 does not have a compaction operation but it can be emulated efficiently by a permute operation and a look-up table as explained in the last part of the following section.

Full Width

The data structure for an 8-ary BVH node cluster laid out to support a full width WIVE implementation is depicted in 3.2a. The bounding box data comes first, each node defined by three $[min, max]$ intervals along the principal axes. A structure-of-array layout organizes the x_{min}, x_{max}, y_{min}, y_{max}, z_{min} and z_{max} bounding box planes into arrays, such that the ith element corresponds to node i. Each element is stored in a four byte single precision floating point format for a total of 32 bytes per array.

The six arrays of bounding box planes are followed by the offset array n. Relating to node i, n^i contains an inner node flag and a memory offset pointing to a child node cluster if the flag is set or to a primitive cluster otherwise.

The permutation vectors are stored in the last part of the data structure in an array of eight 32-bit elements. The elements of the permutation vectors, the permutation indices, are 3 bit in size for an 8-ary BVH and the total of eight permutation vectors are stored in an interleaved manner, so that there is exactly one permutation index per permutation vector per 32 bit of memory. The following formula defines the

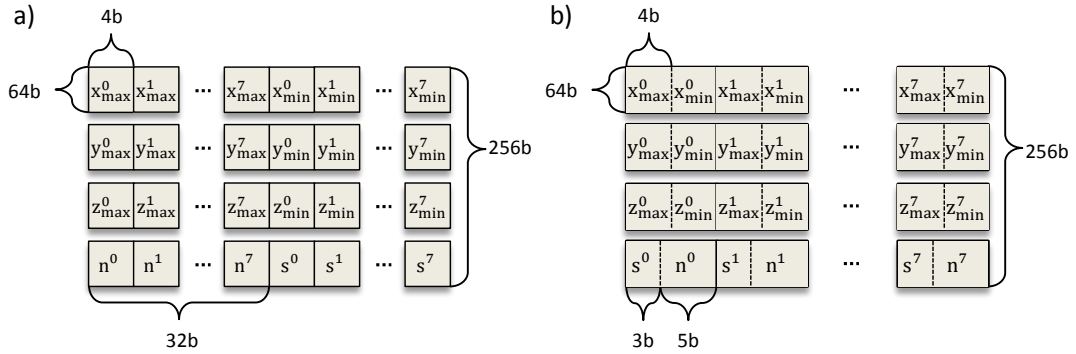

FIGURE 3.2: Data layout of a 8-ary BVH node cluster with a total size of 256 bytes. The nodes' bounding boxes are stored as separate x-, y- and z-vectors with alternating max/min coordinates. A fourth vector contains five bytes for child offset and flags (n) and three bytes for permutation indices (s) per node.

three bits allocated to permutation index (k, i), where k is the 3-bit directional sign octant and i the position to place the node referenced by the permutation index:

$$(k, i) \rightarrow [i * 32 + k * 3 : i * 32 + (k + 1) * 3] .$$

Further, for a valid permutation vector $(k, i) \neq (k, j)$ must hold, i.e., there is exactly one reference to each node.

LISTING 3.2: Core traversal function for WIVE, full width approach. All local variables are vectors with the exception of *mask* and *num*. The {} operator performs a broadcast of scalar values.

```
1  def traverseCluster(cluster, ray, stacknode)
     (bx_min, bx_max, by_min, by_max, bz_min, bz_max, n, s) ←cluster.load()
3    if(ray.sign.x) (bx_min, bx_max) ←(bx_max, bx_min) // Switch min/max bounding planes
     if(ray.sign.y) (by_min, by_max) ←(by_max, by_min) // to conform with ray direction.
5    if(ray.sign.z) (bz_min, bz_max) ←(bz_max, bz_min)
     tx_min ←(bx_min − {ray.org.x}) ∗ {ray.idir.x} // Compute ray t intervals between
7    ty_min ←(by_min − {ray.org.y}) ∗ {ray.idir.y} // min/max bounding planes.
     tz_min ←(bz_min − {ray.org.z}) ∗ {ray.idir.z}
9    tx_max ←(bx_max − {ray.org.x}) ∗ {ray.idir.x}
     ty_max ←(by_max − {ray.org.y}) ∗ {ray.idir.y}
11   tz_max ←(bz_max − {ray.org.z}) ∗ {ray.idir.z}
     t_min ←max(tx_min, ty_min, tz_min, {r.tnear}) // Find interval overlap over all
13   t_max ←min(tx_max, ty_max, tz_max, {r.tfar}) // dimensions and along ray segment.
     index ← shift(s, {ray.sign.xyz}) // Extract permutation vector.
15   t_min ←permute(t_min, index) // Apply permutation vector to arrange t
     t_max ←permute(t_max, index) // intervals and node data n in traversal order.
17   n ← permute(n, index)
     mask ← compare(t_min, t_max) // Compute intersection mask.
19   elems ← (compact(mask, n, stacknode), compact(mask, t_min)) // Stack elements.
     num ← countBits(mask) // Number of intersected child nodes.
21   return (elems, num)
```

The pseudo code for the half width flavor of the *traverseCluster* function left unspecified in Listing 3.1 is printed in Listing 3.2 and discussed in the following with references to line numbers.

Initially, in line 2, the arrays making up the node cluster data structure in Figure 3.2a are loaded into vector registers (*vmovaps,vmovdqa*). The minimum and maximum bounding box planes bx_{min}, by_{min}, bz_{min} and bx_{max}, by_{max}, bz_{max}, respectively, are kept in separate registers as indicated by Figure 3.3a, with superscripts indicating the corresponding node index. It is instructive to contrast this layout to Figure

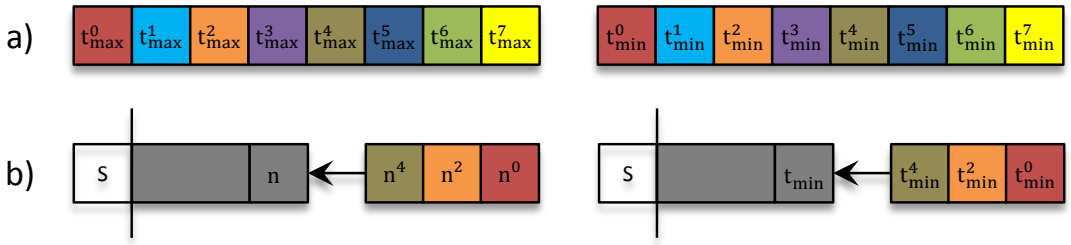

FIGURE 3.3: (a) Full width register layout for bounding box intersection. Entry (t_{min}) and exit (t_{max}) distances are kept in seperate registers with one node per 4-byte vector element. (b) Stack layout. Stack elements are 4 bytes in size with seperate stacks for child offset and entry distance (t_{min}). The stop elements S mark the bottoms of the stacks. Different colors indicate different nodes.

3.4a for the half width approach.

In the next step, from lines 3-5, minimum and maximum bounding box planes are exchanged for every dimension where the ray has a negative directional coordinate using a conditional move operation (*vblendvps*). This transforms the bounding box such that minimum and maximum values are consistent with the ray's point of view. The conditions can be evaluated before traversal begins and readily kept in registers since they depends only on the immutable signs of the ray direction.

After rearranging the bounding box planes, the distances between the ray origin and each plane are computed between lines 6-11 (*vsubps,vmulps*). The multiplication with the inverse ray direction *ray.idir*, precomputed right before traversal, replaces a more costly division by the actual ray direction. Rearranging e.g. $(bx_{min} - \{ray.org.x\}) * \{ray.idir.x\}$ into $bx_{min} * \{ray.idir.x\} - \{ray.org.x * ray.idir.x\}$ allows to perform the computation in a single multiply-add operation (*vfmsub*ps*) which may be faster on some architectures but also can lead to numerical precision issues [99].

Entry and exit distances of of the ray with respect to the bounding box are found in lines 12-13 by determining the maximum across all minimum distances t_{min} and the minimum across all maximum distances t_{max}, respectively (*vminps,vmaxps*). If t_{min} and t_{max} define a valid segment, i.e., $t_{min} \leq t_{max}$, the ray pierces the bounding box. The box segment is further reduced by clipping t_{min} and t_{max} to the valid segment of the ray defined by t_{near} and t_{far}.

Before the comparison $t_{min} \leq t_{max}$ is performed, however, the traversal order is established by permutation. In line 14, a shift of the vector register containing the set of interleaved permutation vectors aligns the permutation indices associated with the ray's directional signs to the lower three bits of every vector element (*vpsrlvd*), effectively selecting one out of the eight precomputed permutation vectors. The shift value *ray.sign.xyz* is a concatenation of the three sign bits and thus constant throughout traversal of a single ray.

The resulting vector *index* is applied to t_{min}, t_{max} and the n vector containing the offset array in lines 15 - 17, thus rearranging the order of the node data with respect to the memory layout to conform to the traversal order (*vpermps, vpermd*).

The comparison deciding on the ray-bounding box intersections follows in line 18, producing a *mask* with a set bits for vector elements corresponding to an intersected node (*vcmpps*).

Line 19 implements the compaction of intersected nodes illustrated in Figure 3.1c, steps C and D. Since the intersection data is split across two vector registers, the bounding box entry distance t_{min} and the child cluster offset n, a *compact* operation is applied to each. The operation takes up to three arguments, (1) the *mask* produced in line 18 to select the elements part of the resulting compact vector, (2) the source vector, and optionally, (3) a fill vector to provide explicit values for elements of the result vector where no compact values exist (the default is "0"). A fill vector containing the *stacktop* in the first element (or in every element) is used for the compaction of n, ensuring that the cluster offset for the next traversal iteration is available in the compact result even if no node intersections exists. For t_{min} the value from the stack is not required because it is only used for stack culling after successful primitive intersection, not within a traversal iteration.

The rather specialized *compact* operation is available as a single instruction (*vcompressps, vpcompressq*) in AVX-512 but not for other instruction sets, AVX2 being one example. Hence, a sequence of more generic instructions are presented as a replacement in Listing 3.3.

LISTING 3.3: Replacement for *vcompressps, vpcompressq* instructions.

```
1  const LUT[256], shiftVecor
   def compact(mask, n, stacknode)
3    index ← rightShift({LUT[mask]}, shiftVector)
     elems ← permute(n, index)
5    return blend(elems, stacknode, index)
```

The look-up table *LUT* stores 256 permutation vectors, one for each 8-bit *mask* value, implementing every possible compaction operation for a vector width of eight elements. In order to reduce the memory footprint of the look-up table the permutation vectors are compressed to 32-bit integers and expanded to full vectors after selection using a broadcast (*vpbroadcastd*) and variable bit shift to the right (*vpsrld*). The magnitude s of the shift, stored in the $shiftVector$, is proportional to the vector element index $i \in [0, 7]$, i.e., $s_i = 4 * i$, aligning a different 4-bit slice to the beginning of each element of the resulting *index* vector. The lower three bits of every *index* element are used by the permutation operation *vpermd*, whereas the fourth bit is set to zero everywhere with one exception: for a zero *mask*, bit 31 of the look-up table value, i.e., the fourth bit in s_7, is set. If no child node intersection exists, this results in a set sign bit of the first element of the *index* vector, which is further used as the selector in a sign-based *blend* operation *vblendvps*, placing the *stacknode* into the first element of the result vector, according to Figure 3.1c, step C.

Back to Listing 3.2, the number *num* of intersected nodes is determined by counting the set *mask* bits in line 20. The *traverseCluster* function ends by returning the number *num* together with the corresponding lists of node offsets and entry distances in line 21.

Following the *traverseCluster* function, the algorithm continues in Listing 3.1, line 6: the cluster offset for the next iteration is extracted from the *elems* list, i.e., from the first element of the corresponding vector register (*movq*). The stack push in line 7 is implemented with two masked store operations (*vmovups, vmovdqu32*), one for

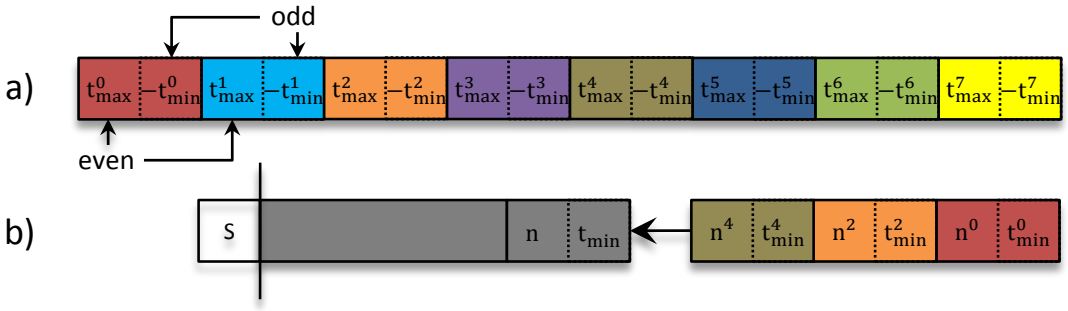

FIGURE 3.4: (a) Half width register layout for bounding box intersection. Entry (t_{min}) and exit (t_{max}) distances are computed simultaneously within 8-byte lanes for each node, which requires t_{min} to be treated as negative. (b) Stack layout. Stack elements are 8 byte in size with interleaved child offset and entry distance (t_{min}). The stop element *S* indicates the bottom of the stack. Different colors indicate different nodes.

each stack as illustrated in Figure 3.3b. Since the sorted nodes must be put on the stack in reverse order to be retrieved as intended the stack grows towards smaller memory addresses. The mask is necessary to prevent existing stack values from being overwritten by the inactive vector elements remaining after the compaction (the grayed-out elements in Figure 3.1). The mask is obtained by comparing the node register either to the original fill vector used during the compaction operation (*vpcmpeqd*) or to zero (*vptestnmd*). After the stack push, the WIVE inner traversal iteration is concluded.

On a final note, if the compaction by look-up table is used, a change to the look-up table allows to get rid of the *mask*: the permutation vectors can be precomputed such that the first active element is moved to the first element as usual but the remaining active elements are densely aligned to the end of the vector register. In this case no *mask* is necessary because no existing stack elements can get overwritten during a regular store.

Half Width

The half width variant of the WIVE algorithm utilizes twice the number of vector elements compared to the full width variant by interleaving the majority of the computations for which separate instructions would be required otherwise. Therefore, in this section, the two adjacent 32-bit vector elements aliased with a single 64-bit element are referred to as lanes and the lower and upper 32-bit elements as even and uneven elements, respectively, see Figure 3.4a. The memory layout of an 8-ary BVH cluster is illustrated in Figure 3.2b, with a standard 256-byte footprint corresponding to four 64-byte vectors. The nodes' bounding boxes are stored in three separate arrays, one for every axis, with alternating maximum and minimum planes. The fourth array encodes the permutation vectors and the node data, which includes a flag to indicate an inner node or a leaf, the corresponding child cluster or primitive cluster offset, a mask to identify valid nodes in a child cluster or the number of primitive clusters within a leaf. Permutation indices are three bit in size to reference one of the eight nodes, and the eight permutation vectors are compressed into three bytes per node.

Listing 3.4 shows the *traverseCluster* function which is described in detail in the following paragraphs, referencing the corresponding line numbers.

LISTING 3.4: Core traversal function for WIVE, half width approach. All local variables are vectors with the exception of *mask* and *num*. The {} operator performs a broadcast of scalar values.

```
1  def traverseCluster(cluster, ray, stacknode)
     (bx, by, bz, ns) ← cluster.load()
3    if(ray.sign.x) bx ← swapEvenOdd(bx) // Switch min/max bounding planes
     if(ray.sign.y) by ← swapEvenOdd(by) // to conform with ray direction.
5    if(ray.sign.z) bz ← swapEvenOdd(bz)
     tx ← (bx − {ray.org.x}) ∗ {−ray.idir.x, ray.idir.x} // Compute ray t intervals
7    ty ← (by − {ray.org.y}) ∗ {−ray.idir.y, ray.idir.y} // between
     tz ← (bz − {ray.org.z}) ∗ {−ray.idir.z, ray.idir.z} // min/max bounding planes.
9    t ← min(tx, ty, tz, {−r.tnear, r.tfar}) // Common interval.
     index ← shift(ns, {ray.sign.xyz}) // Permutation vector.
11   t ← permute(t, index) // Arrange and combine t intervals and node data.
     nt ← bitwiseSelect(n, tmin, selectMask)
13   nt ← permute(nt, index)
     tmax ← swapEvenOdd(t)
15   tmin ← flipSignsOdd(t)
     mask ← compare(tmin, tmax) // Compute intersection mask.
17   elems ← compact(mask, nt, stacknode) // Stack elements.
     num ← countBits(mask) // Number of intersected child nodes.
19   return (elems, num)
```

In line 2 the three bounding box arrays for the $x-$,$y-$ and $z-$axes are loaded into vector registers (*vmovps*), so that the max/min pairs align with the lanes, see Figure 3.4a. Depending on the sign of the rays directional components the corresponding max/min values are be swapped within lanes in lines 3-5 to conform with the ray's point of view. The swaps are performed efficiently with masked 64-bit rotate operations (*vprolq*), where the mask for every axis is assumed to have been precomputed during the ray setup phase. After swapping, in lines 6-9, the ray's entry and exit distance calculations are performed for all eight nodes in parallel, computing t_{max} and $-t_{min}$ in even and odd lanes, respectively. The calculations correspond to the following four equations:

$$\begin{aligned} t^{n,i}_{max} &= (b^{n,i}_{max} - o^i) * d^i \\ t^{n}_{max} &= \min_{i=x,y,z} t^{n,i}_{max} \\ -t^{n,i}_{min} &= (b^{n,i}_{min} - o^i) * (-d^i) \\ -t^{n}_{min} &= \min_{i=x,y,z} -t^{n,i}_{min} \end{aligned} \tag{3.1}$$

Here, i and n denote the axis and the lane, respectively, o^i is a component of the ray origin, d^i is the inverse component of the ray direction, and $b^{n,i}$ represent the minimum and maximum bounding box planes after the initial swap. Both o^i and d^i are constant throughout traversal, and the sign of the $d^{n,i} = (-1)^n d^i$ vector can be adjusted during the ray setup phase to alternate between d^i and $-d^i$.

The t^n_{max} and $-t^n_{min}$ values are further clipped to the active segment of the ray defined by t_{near} and t_{far} (*vminps*), and the final result is laid out in the vector register as illustrated in Figure 3.4a.

Next, the results are arranged according to the front-to-back traversal order stored in the node cluster. In line 10, the appropriate permutation vector is extracted from

the fourth array shown in Figure 3.2b by bit-shifting with the concatenated sign bits of the ray's directional components, i.e., a precomputed three-bit value, to align the permutation vector's components to the lower three bits of the vector register's 64-bit elements (*vprolvq*). After the permutation step (*vpermq*) in line 11, the $\boxed{t_{max} \mid -t_{min}}$ pairs are ordered such that the first node to be traversed corresponds to the last active node lane in the register.

The same permutation is applied to the stack element candidates *nt* in line 13 after composition in line 12. The stack elements combines the node offsets and corresponding t_{min} values according to the formatting indicated by Figure 3.4b. The *bitwiseSelect* function allows to select, for every bit, between the two input operands n and t_{min}, based on the *selectMask* (*vpternlogq*). n and t_{min} are aligned to the beginning and end of the odd and even vector lanes, respectively. The node offset allocated in the data layout in Figure 3.2b is 40 bits in size. If all bits are to be used the lower eight bits of t_{min} may be modified. This does not compromise the correctness of the stack culling later on but may lead to a minor reduction in its efficiency due to a more conservative test. The sign bit of t_{min} may be undefined at this point and is set to zero upon retrieval from the stack since t_{min} is positive by definition.

The intersection test is completed by comparing t_{max} and t_{min} to retrieve the active mask, which requires the values to be in separate registers aligned to the odd elements. This requirement is met via a 64-bit rotate operation (*vprolq*) in line 14 to form $\boxed{-t_{min} \mid t_{max}}$ pairs and a sign flip with an exclusive or operation (*vpxorq*) in line 15 to obtain $\boxed{t_{max} \mid t_{min}}$ pairs. Since $t_{min} \geq 0$ always holds, the predicate of the test $t_{min} \leq t_{max}$ in line 16 can be determined correctly with integer arithmetic by re-interpreting the floating-point patterns of the pairs as 64-bit signed integers (*vpcmpq*). The resulting mask is used to for the following compaction operation in line 17 (*vpcompressq*) together with the topmost stack element as the default value if no intersection exists.

Leaf Clusters and Stack Culling

Once the inner node traversal reaches a leaf, the $intersectLeaf$ function in Listing 3.1, line 9, is executed. The node offset no longer references a child node cluster but a list of primitive clusters instead. For the experiments in the following section a primitive cluster packs up to four triangles, which has been found to result in the best performance regardless of the vector width. Larger clusters increase vector utilization at the cost of performing more intersection tests and increased bandwidth demand. Smaller clusters have the inverse effect. The optimal balance depends on multiple factors such as hardware architecture and traversal algorithm. An overview over optimized and vectorized triangle intersection algorithms can be found in [82, 94, 9, 70, 116].

Once an actual intersection is found among the triangle clusters the maximum ray distance t_{far} is updated accordingly. In this case, nodes on the stack with $t_{far} < t_{min}$ are removed by the *cull* function in line 10. This pruning procedure is efficiently implemented by loading as many stack elements as possible into a vector register, starting from the stack bottom, performing the comparison and compacting and storing the remaining valid elements back onto the stack. The implementation of the comparison and compaction operations for the stack culling is analogous to the inner node traversal.

Eventually, once the ray has exhaustively traversed the hierarchy, the *node* represents no longer a node cluster or a primitive cluster but the terminal stack element. The terminal stack element initiates the stop of the traversal loop in line 13 and the completion of the *traverseRay* function.

3.1.4 Results

The WIVE algorithm is evaluated by generating performance data based on AVX2 and AVX-512 implementations on the dual-socket Intel® Xeon™ E5 2680v3 with 2.5GHz (HW) and the Intel® Xeon Phi™ 7250 with 1.4GHz (KNL), respectively. The results are compared with those obtained from Embree 2.15.0 [111], the leading high-performance ray tracing library for CPUs. In order to ensure comparability of performance data, the WIVE code has been integrated into the open source Embree benchmark suite Protoray [30], which by default offers Embree and Nvidia® OptiX™ [87] kernels. A comparison to the GPU-leading OptiX ray tracing library is outside the scope of this dissertation, however results from the Protoray benchmark have been published elsewhere [33]. Embree constructs a native 8-ary BVH using SAH-based centroid binning [106], which is directly converted to WIVE's native data layout retaining the exact same topology. Spatial splits are disabled to ensure better comparability of the results shown here with results obtained with other methods. In order to generate the permutation indices for the sign heuristic the Embree code has been modified to annotate each node cluster with the original split hierarchy. Triangle intersection is served by the same Möller-Trumbore [82] implementation and triangle data structure as used in Embree. Therefore, the traversal algorithms are solely responsible for the observed performance differences. The performance evaluation is based on five scenes consisting of between 5.7M and 37.5M triangles. On the KNL, these benchmarks are processed by all 272 threads, with all data allocated in the high-bandwidth memory segment. The on-chip mesh network is configured in quadrant mode. On the HW all 48 threads are active.

TABLE 3.1: Traversal statistics for sign and distance ordering based on Embree's SAH-binned 8-ary BVH. The columns *I(nner)Nodes*, *Leaves* and *Tri(angle)s* list the per-ray average numbers of inner nodes visited, leaves intersected, and triangles intersected, respectively. The SAH cost for each scene is also broken down by *I(nner)Nodes* and *Leaves*. The rendered images are shown in Figure 3.5.

	Sign			Distance			SAH	
	INodes	Leaves	Tris	INodes	Leaves	Tris	INodes	Leaves
MAZDA	14.1	3.6	4.1	14.1	3.6	4.1	4.20	2.23
SAN MIGUEL	21.2	4.5	5.4	21.2	4.2	5.1	4.01	1.90
ART DECO	11.1	2.3	2.9	11.0	2.2	2.8	4.59	2.74
POWERPLANT	20.5	5.6	9.3	20.3	5.6	9.2	5.78	4.21
VILLA	17.4	4.6	5.5	17.4	4.5	5.4	20.7	15.6

The key comparison between distance and sign heuristics determines how well they approximate a front-to-back traversal order to maximize node culling, see Table 3.1. The three per-ray average indicators (inner nodes visited, leaves intersected, and triangles intersected) are very close to being equal across the scenes, with a slight bias towards the distance heuristic. A notable discrepancy is observed for SANMIGUEL,

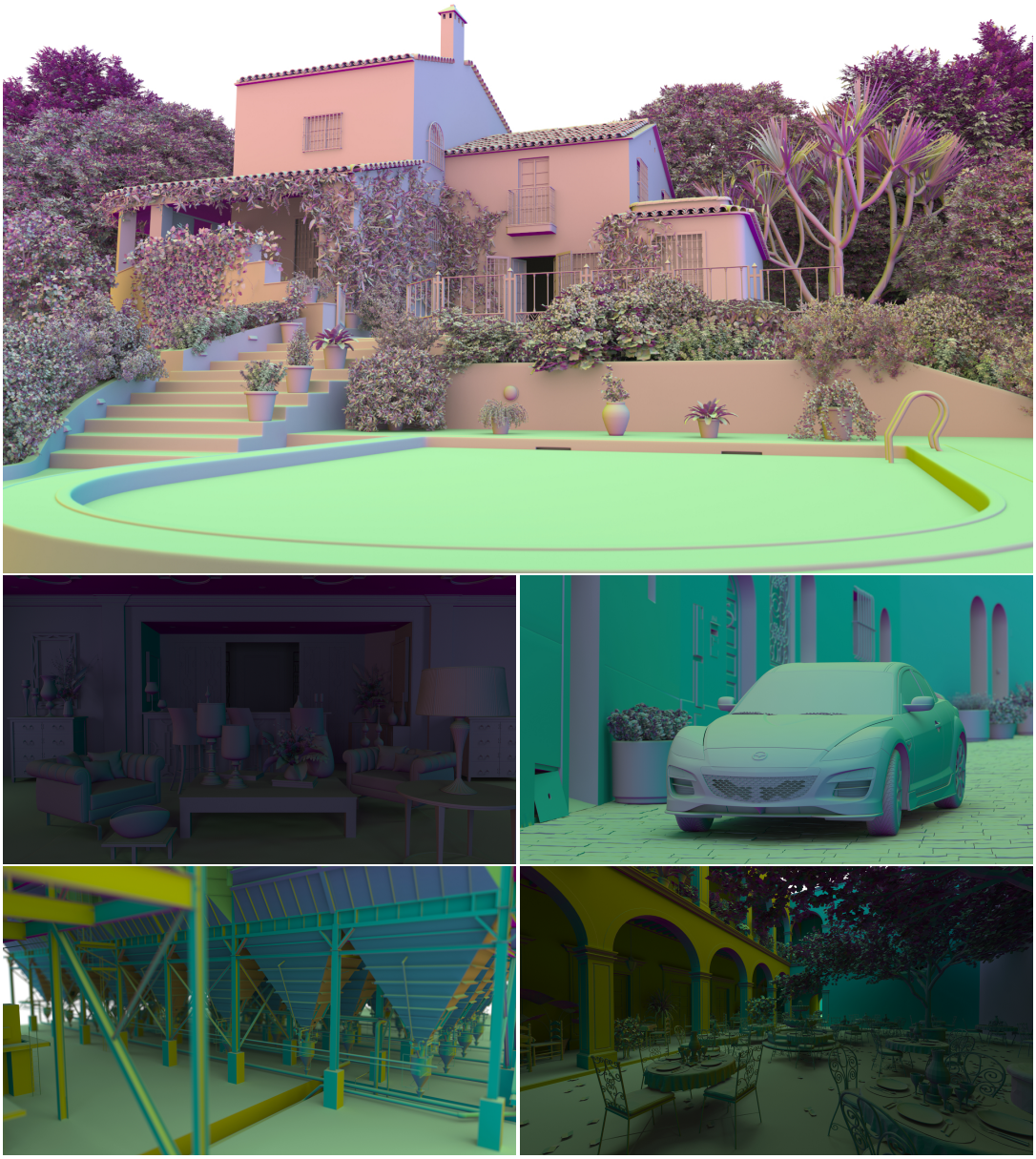

FIGURE 3.5: Pictures of benchmark scenes rendered with the WIVE algorithm using path tracing. The shading color is based on the surface normal. From left to right and top to bottom, and with triangle counts: VILLA (37.5M), ARTDECO (10.7M) and MAZDA (5.7M), all courtesy of Evermotion, POWERPLANT (12.8M), courtesy of University of North Carolina, SAN MIGUEL (10.5), courtesy of Guillermo M. Leal Llaguno.

where the number of intersected leaves and triangles is up 6-7% for the sign heuristic, which is attribute to the high degree of overlap of the alpha-textured leaf triangles. In such a setting, the distance heuristic can be more precise as it considers the actual intersection point of the ray. For completeness Table 3.1 also lists the surface area heuristic (SAH) cost [47, 107] associated with each of the generated 8-ary BVHs.

TABLE 3.2: Performance in million-rays per second (MRays/s) for the sign-based WIVE algorithm and Embree based on AVX2 and AVX-512 implementations. Rendering is performed at a resolution of 3840×2160 pixels using diffuse path tracing with up to eight bounces. The rendered images are shown in Figure 3.5. Colors are based on surface normals and the shading cost is included in the results, accounting for 8-12% of the rendering time.

	MAZDA	SAN MIGUEL	ART DECO	POWERPLANT	VILLA
# triangles[M]	5.7	10.5	10.7	12.8	37.5
AVX2					
WIVE	**74.0**	**46.0**	**97.2**	**57.4**	**48.8**
Embree	70.9	43.0	93.7	51.9	46.2
WIVE[+%]	**4**	**7**	**4**	**11**	**6**
AVX-512					
WIVE	**126.7**	**73.1**	**165.0**	**85.4**	**87.4**
Embree	110.0	63.2	143.4	68.4	76.3
WIVE[+%]	**15**	**16**	**15**	**25**	**15**

Table 3.2 provides performance data measured in million-rays per second (MRay/s) for a basic diffuse path tracer with up to eight bounces per sample. When comparing the AVX-512 implementations of the sign-based WIVE traversal to the distance-based Embree algorithm, a sizeable speed-up of between 15% and 25% is observed across all scenes. The increased efficiency can only originate from the traversal phase since all other parts share the same implementation. This implies that the reduced code complexity due to our novel algorithm is the only significant differentiating factor. Variance in memory access patterns due to slight differences in traversal order between the two heuristics is negligible, which follows from the nearly identical data listed in Table 3.1. The WIVE algorithm is especially advantageous when rays frequently overlap with more than three children during a traversal step, e.g., in the POWERPLANT scene. The resulting performance data are shown in Table 3.3. In this case, the distance heuristic requires increasingly expensive sorting and stack operations while the WIVE algorithm's execution is *independent* of the number of active children.

TABLE 3.3: Distribution of numbers regarding valid child node intersections (in percent) for a single traversal step.

	0	1	2	3	>3
MAZDA	24	39	22	9	6
SAN MIGUEL	29	35	19	9	8
ART DECO	30	38	17	8	7
POWERPLANT	40	24	14	10	12
VILLA	25	36	20	11	8

Performance comparison of the AVX2-based implementations of WIVE and Embree demonstrates that WIVE is faster also for the "BVH branching factor equals vector width" variant, albeit with a smaller relative difference of between 4% and 11%. Since Embree also uses an interleaved slab test for AVX-512 and a regular slab test for AVX2, the only notable differences between the WIVE variants is the compact operation and the double stack. The compaction operation is emulated due to the lack of hardware support by a sequence of four instruction including a memory access into a sizable table. The double stack is intrinsic to the algorithm and requires an additional compaction operations and an additional store to memory.

3.1.5 Summary

The introduction of the multi-branching bounding volume hierarchy has led to the last major performance gain in single ray traversal, by utilizing vector instructions for bounding box tests. This section has continued and completed the formulation of an innovative, fully vectorized BVH traversal by introducing the WIVE algorithm. The efficiency gain obtained by WIVE is made possible by transforming node ordering and stack-push operations from conditional scalar execution paths to constant-time vector operations, making them ideal for current and future massively parallel microarchitectures. The algorithm's performance is demonstrated by experiment with an implementation for the AVX-512 instruction set. The performance data document that WIVE outperform the industry-leading ray tracing library Embree by between 15% and 25% on an Intel®Xeon Phi™ CPU. A corresponding AVX2 implementation on an Intel® Xeon™ CPU has yielded a speed-up compared to Embree by between 4% and 11% despite incomplete instruction support.

3.2 Stream Traversal

Stream traversal conceptually aggregates multiple traversal requests into a single set of rays. A sufficiently large set is traversed through the bounding volume hierarchy as a whole, in every traversal iteration testing all active rays against the current node and partitioning the set according to the intersection results. Hence, the ray sets provide inherent data parallelism exploitable for vectorization and superscalar execution of independent instructions.

A further major motivation for stream traversal is optimizing the memory access behavior compared to basic single ray traversal. Both memory bandwidth and access latency for node data fetches are amortized among multiple rays, thus relieving the memory subsystem and reducing execution stalls from cache misses. Necessarily, the memory footprint of the ray data is required to comfortably fit into the cache hierarchy.

A prerequisite for the efficiency of stream traversal is coherence among the rays in a set. Stream traversal allows to capitalize on *unstructured ray coherence* as defined in Section 2.6.1. At the root node all rays are considered coherent by definition and subsequent traversal iterations automatically partitions the initial set into smaller subsets of stronger coherence, i.e., of longer common traversal path. At the leaf nodes typically only a few fully coherent rays remain of the initial set. Hence, every traversal iteration operates on the complete subset of rays with identical coherence. Thus, if there is any coherence among rays in a set, stream traversal uncovers it and

puts it to use by filling vector units and amortizing memory access to node data.

A fundamental challenge for stream traversal is maintaining a front-to-back traversal order per ray for efficient node culling, in particular for multi-branch bounding volume hierarchies. Previous work, for example [100], followed the node with the smallest average distance over all rays first, leading to increased number of traversal steps and primitive intersection per ray on average compared to single ray traversal. A simple illustration of the problem is to imagine two rays with opposing directions where one ray follows its front-to-back order, forcing the other ray to take a back-to-front order.

The algorithm proposed by Barringer et al. [11], dynamic ray stream traversal (DRST), lifts the restrictions on one global traversal order. Every ray is allowed to choose between a subset of all possible traversal orders in every traversal iteration based on the distance heuristic. As a negative consequence, the ray sets become more fragmented as shared traversal paths become shorter. Another issue of DRST is the computational complexity imposed by the dynamic traversal order. Especially when only a few rays remain in a subset, which is a common occurrence, the cost of the overhead becomes high per ray. DRST works around this issue by switching to single ray traversal once the number of rays falls under a parameter threshold, accepting that the switching logic itself adds further overhead.

The stream traversal algorithm introduced in the following, named ordered ray stream traversal (ORST), addresses the limitations of DRST. A different approach to ordering removes computational complexity and prohibits fragmentation during traversal. As a result, ORST is able to maintain larger coherent ray sets compared to DRST while being close in performance to dedicated single ray traversal even if only a single ray remains in a set.

3.2.1 Ordered Traversal

The ordering mechanism described in the following implements the sign-based order heuristic discussed in Section 3.1.1 for a 4-ary BVH using a two-stage look-up table. A previous approach [24], combining the sign-based heuristic with a 4-ary BVH, produces an implementation with higher computational overhead and reduced flexibility because only a subset of the 24 possible order arrangements for a four-node cluster are supported, i.e., those eight that can be mapped to a balanced binary BVH sub-tree. Compared to the technique introduced in Section 3.1.1 the mechanism described here requires less memory per node and is better suited for stream evaluation; the traversal order however is exactly the same.

Figure 3.6 illustrates the new look-up-table-based implementation. A *perm* value, stored in the BVH node layout discussed later (Figure 3.7), is used in conjunction with the ray *sign mask* to obtain an *order index* from the two-dimensional order look-up table (orderLUT). The ray *sign mask* contains the signs of the ray components in the lowest 3 bits, and the *order index* represents one of the 24 possible traversal orders. The *order index* together with the *active node mask* forms a second index pair to obtain the actual *order* from the two-dimensional compact look-up table (compactLUT). The *order* contains 2-bit indices to the child nodes in a front-to-back order, where the 2 lowest bits are occupied by the most distant node. The *active node mask* maps the lowest 4 bits to the child nodes according to the memory position and has

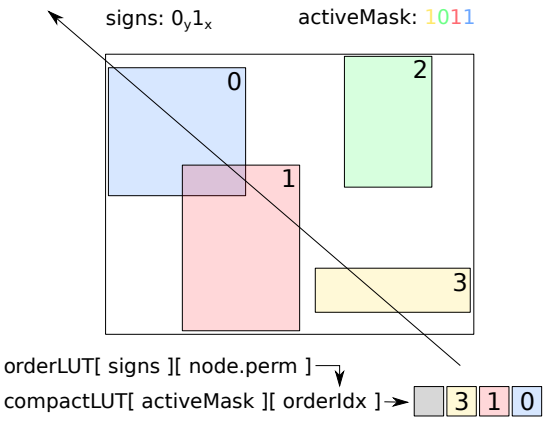

FIGURE 3.6: Visualization of the traversal order look-up mechanism. The colored boxes represent child nodes which are labeled according to memory order. A ray with a negative x sign and a positive y sign intersects child nodes 3, 1 and 0, and the corresponding color-coded *active mask* is shown. The geometrical relations of the child nodes are captured in the parent node's *perm* field. The *perm* value together with the ray signs produces an *order index* from the order look-up table which may represent any permutation of the child nodes. The *order index* together with the *active mask* are used for a second look-up to retrieve a sorted, compact list of intersected child nodes.

a set bit for every intersected node. As a result, the indices obtained from the *order* are already compacted to reference only child nodes that are actually intersected, so that the traversal can directly proceed without additional checks. The complex logic implemented by the two look-ups is very efficient and would require significantly more instructions otherwise.

The remainder of this section explains the details of the *perm* index structure. The *perm* index is constructed as a hierarchy of three subdivisions to partition the four child nodes. There are exactly two unique hierarchy types: One is the balanced type, where the first subdivision creates two sets of two nodes, and the second is the unbalanced type, where the first subdivision creates one set of one node and one set of three nodes. All possible subdivision variations of the balanced type are symmetric and can be accounted for by adjusting the relative position of the child nodes in memory accordingly. For the unbalanced type four unique topologies exist. Thus the first part of the *perm* index identifies one of the five possible topologies. The second part defines the axes for each of the three subdivisions. The three possible axes (x,y,z) result in $3 * 3 * 3 = 27$ variants for each topology. Thus the *perm* index is computed as:

$$axis_{1^{st}split} + axis_{2^{nd}split} * 3 + axis_{3^{rd}split} * 9 + topologyId * 27$$

Since a ray can have $2^3 = 8$ possible sign combinations, eight sets of the topology variants are required, resulting in a orderLUT size of $5 * 27 * 8 = 1080$ bytes which corresponds to 17×64 byte cache lines. The compactLUT has a size of $24 * 16 = 384$ bytes (24 possible traversal orders and 16 possible active mask combinations), so that a total of 26 cache lines is reserved for the look-up tables. If a 4-ary BVH is derived from a binary BVH, the original binary BVH traversal order is maintained by transfering the axis values (defining the axis along which the two children of a binary BVH node overlap the least) and setting the $topologyID$ to the balanced

type. Consider a binary BVH treelet with a root node, two intermediate nodes and four child nodes, which is flattened into a single 4-ary BVH node. Then $axis_{1^{st}split}$ receives the axis value of the root node, and $axis_{2^{nd}split}$ and $axis_{3^{rd}split}$ receive the axis values of the left and right intermediate nodes, respectively. The four child nodes are arranged in memory according to the traversal order of a ray with only positive direction components. Example code for the look-up table generation is provided in [39].

3.2.2 Algorithm

The key feature of the new algorithm is the traversal order look-up mechanism presented in Section 3.2.1. The order produced by this method is exactly the same for all rays with identical signs. Thus, sorting a stream into the octants generated by the possible sign combinations elegantly groups the rays by common traversal order, resulting in up to eight disjoint sets which are then processed successively. The rays included in a stream are tracked by a list of ray indices (short integers). During node intersection a new list for each of the four child nodes is created which contains the indices of all the rays intersecting the respective child node. The four lists are pushed to four seperate stacks mapped to the four nodes, and the algorithm proceeds with the stack entry corresponding to the node chosen by ordered traversal. Pseudo code is listed in Algorithm 1, and details are discussed below, referencing line numbers.

At the start, the ray indices of the stream are pushed onto one of eight ray index stacks depending on the ray sign, each stack corresponding to one of the octants (lines 5-9). A ray stack is also called a lane, and the first four lanes are simultaneously mapped to the four child nodes during traversal. Then, a task is generated for each octant that is not empty and pushed onto the task stack (lines 11-16). The core loop begins with popping a task (line 20), checks whether the current node is an inner node (line 21) and, if not, jumps to the leaf intersection (line 46). Otherwise, the corresponding rays are intersected with the current child nodes in parallel (lines 25-32). The variable *rayStart4* aliases with the first four lanes of the *rayStart* stack indices. The ray index is copied to the first four lanes associated with the child nodes (lines 28-30), and *rayStart4* is increased for every lane that matches a successful child node intersection (line 31). Since *msk4* has all bits set if the corresponding node is intersected, the value is subtracted from *rayStart4* in order to increase the stack indices by one. After all active rays have been tested, the traversal order is obtained from the look-up tables, where the bits of the active ray mask are set for all lanes with at least one child node intersection (lines 33-36). Finally, a new task is generated for all active child nodes and pushed onto the task stack in back-to-front order (lines 39-44). The algorithm is simple and efficient, maximizes coherence and enables high vector utilization.

The memory layout of the 4-ary BVH structure is shown in Figure 3.7. Conceptually, a BVH node is considered separate from its geometric bounding box. The nodes are always grouped in clusters with capacities of two or four nodes (never three due to 16 byte alignment for vectorized memory access). Every cluster is adjacent in memory to four bounding boxes in SOA format (96 bytes) which are located in front of the cluster. With this definition a node occupies only eight bytes, with a five byte index to the child node cluster, one byte for the active child mask, one byte for the node's relative index within its own node cluster, and one byte for traversal order look-up. In case of a leaf, the active child mask is set to zero, the five byte index points to a

Algorithm 1 Ordered ray stream traversal Vector symbols are postfixed with a 4, as are functions operating on vector data. For the definition of functions see Algorithm 2.

```
1: rayStack[8][]
2: rayStart[8] ← 0
3: sP ← 0
4: {Sort ray indices onto stacks based on sign octant.}
5: for rayID ← 0 to |R| do
6:    signs ← Sign4(R(rayID))
7:    rayStack[signs][rayStart[signs]] ← rayID
8:    rayStart[signs] ← rayStart[signs] + 1
9: end for
10: {Generate task for each octant and push to task stack.}
11: for i ← 0 to 8 do
12:    if rayStart[i] > 0 then
13:       taskStack[sP] ← (root, rayStart[i], i, i)
14:       sP ← sP + 1
15:    end if
16: end for
17: {Main traversal loop.}
18: while sP > 0 do
19:    sP ← sP − 1
20:    (node, numRays, lane, sign) ← taskStack[sP]
21:    if node is inner node then
22:       rayStart[lane] ← rayStart[lane] − numRays
23:       numActive4 ← rayStart4
24:       {Intersection of active rays with current child nodes and push of intersecting rays'
          indices to corresponding stacks.}
25:       for r ← 0 to numRays do
26:          rayID ← rayStack[lane][rayStart[lane] + r]
27:          msk4 ← IsectBox4x1(node, R(rayID))
28:          for i ← 0 to 4 do
29:             rayStack[i][rayStart[i]] ← rayID
30:          end for
31:          rayStart4 ← rayStart4 − msk4
32:       end for
33:       activeMsk ←Sign4(rayStart4 > numActive4)
34:       numActive4 ← rayStart4 − numActive4
35:       orderIdx ← orderLUT[sign][node.perm]
36:       order[] ← compactLUT[orderIdx][activeMsk]
37:       cnt ← CountBits(activeMsk)
38:       {Generate task for each non-empty child node and push to task stack in order.}
39:       for i ← 0 to cnt do
40:          o ← order[i]
41:          n ← root + node.child + o
42:          taskStack[sP] ← (n, numActive[o], o, sign)
43:          sP ← sP + 1
44:       end for
45:    else
46:       Leaf intersection
47:    end if
48: end while
```

list of primitives and the number of primitives is saved in the look-up byte. This somewhat flexible structure allows to save about 5% of memory compared to the standard 128 byte layout [24, 32] and also performs slightly faster during traversal.

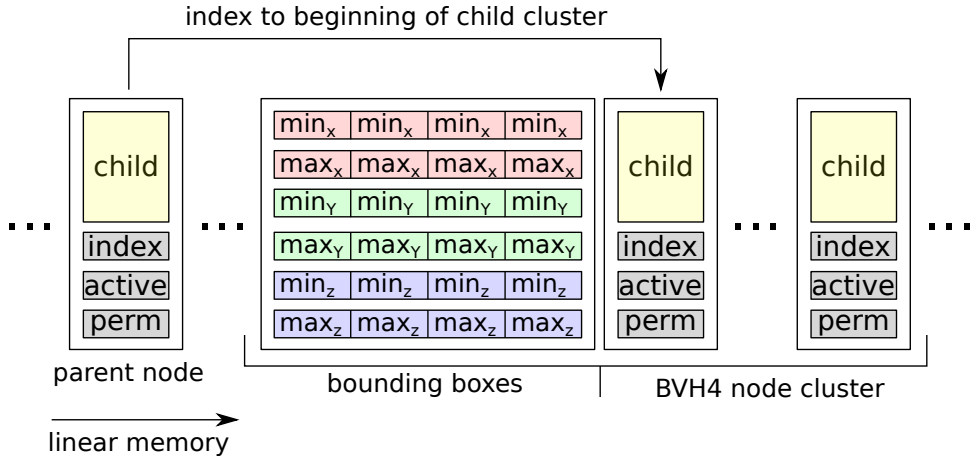

FIGURE 3.7: Memory layout of the 4-ary BVH. A node contains four records: *child* is an index to a cluster of child nodes, *index* is the relative position of a node within its own cluster, *active* is a bit mask that has a set bit for every active child node, and *perm* encodes the geometric configuration of the child nodes which is used in the ordered traversal look-up. If the node is a leaf, *child* indexes a list of primitives and *perm* holds the primitive count. Every node cluster is directly preceded by a SOA structure of four bounding boxes independent of the cluster size to allow vectorized access.

3.2.3 Results

The new *ordered* ray stream traversal (ORST) is compared to the previously fastest algorithm named *dynamic* ray stream traversal (DRST) [11] using the original code provided by the authors. BVHs are generated by the SBVH algorithm described later in Chapter 4 with identical topologies but different binary formats for ORST and DRST, corresponding to the respective optimized node layout. The same triangle intersection test implementation is used for both algorithms. The benchmarks are performed for six different scenes, the SPONZA, FAIRY, DRAGON, HAIRBALL, POWERPLANT and the R8. Fixed camera samples along a fly-through path have been generated for every scene in order to capture the full scene complexity and avoid view point specific variations. The hardware platform is a dual socket Intel® Xeon™ E5-2680v3 Haswell (24 cores / 48 threads total at 2.5GHz) which allows DRST to use AVX2 while the more widely available AVX instruction set is sufficient for the ORST implementation. In the last part of this section, the look-up table sign heuristic described in Section 3.2.1 is compared to the distance heuristic approximated by DRST and commonly used by other single and packet traversal algorithms.

The benchmark results are presented in Table 3.4. All frames are rendered in 17 iterations, where each iteration consists of a single primary ray per pixel spawning 16 diffuse rays from its scene intersection point. The diffuse rays are distributed randomly over the hemisphere around the surface normal according to the Lambertian reflectance model. They can bounce up to four times and are terminated earlier only if they hit the background. In total up to $17 \times 16 \times 4 = 1088$ rays are traced per

FIGURE 3.8: Pictures of benchmark scenes rendered with the ORST and CLPT algorithms using path tracing. The shading includes materials and textures, if available. From left to right and top to bottom, and with triangle counts: FAIRY (174K), courtesy of the University of Utah, HAIRBALL (2.9M), courtesy of NVIDIA Research, SPONZA (66K), courtesy of Marko Dabrovic, and DRAGON (871K), courtesy of Stanford University.

pixel and frame and care is taken that the rays are exactly the same for all competing algorithms. The primary ray traversal is performed using a packet algorithm (CLPT, see Section 3.3.1) with a tile size of 8×8 pixels and all diffuse rays corresponding to the same tile (up to 1024) are placed in a single stream for the first bounce. Each bounce then produces a new stream that is equal or smaller in size compared to the previous stream. An optimized single ray implementation (SR), also based on the traversal look-up mechanism, is provided for reference. The measurements include random number and ray generation, traversal, intersection and shading (with textures). Both ORST and DRST are AVX/AVX2 implementations that always intersect two rays simultaneously with four nodes or four triangles. DRST switches to single ray traversal once the stream size drops below a certain threshold which is optimally set to 16, whereas for ORST a switch to single ray traversal is not required to increase overall performance. ORST is able to outperform DRST by 37% on average with respect to the total run time. If only ray traversal and intersection is considered, ORST is about 51% faster. One notable result is the complex HAIRBALL scene where DRST offers no advantage over single ray traversal, while ORST can maintain a 31% lead. The throughput increase from DRST to ORST can be attributed to lower overhead

TABLE 3.4: Results for diffuse rays. Pictures of the test scenes are presented in Figures 3.5, 3.8 and 3.9. The measurements are taken during a full camera fly-through with the total number of frames listed below. The performance numbers include the entire rendering process, while *ray queries* lists the percentage of the total time spent in traversal and intersection as measured for single rays (SR). For each frame 17 primary samples and up to 272 diffuse rays with a maximum bounce depth of 4 are generated per pixel at a resolution of 1280×1024. Speed-up percentages are given once for the total rendering process, and once only for the combined traversal and intersection part.

	SPONZA	DRAGON	FAIRY	R8	HAIRBALL	POWERPLANT
# triangles	66k	871k	174k	795k	2.9M	12.8M
# frames	20	13	15	20	10	13
Ray queries	76%	84%	73%	81%	91%	86%
	Mray/s	Mray/s	Mray/s	Mray/s	Mray/s	Mray/s
SR (SSE4)	75	73	72	75	27	74
ORST (AVX)	**117**	**117**	**115**	**118**	**35**	**119**
DRST (AVX2)	84	86	85	85	27	84
	Speed-up	Speed-up	Speed-up	Speed-up	Speed-up	Speed-up
Total	39%	35%	35%	39%	31%	41%
Traversal	**59%**	**47%**	**61%**	**53%**	**35%**	**53%**

(node ordering, stack operations, lane management) and reduced stream fragmentation. The difference in overhead is already apparent from the single ray threshold (0 vs 16) for which either algorithm achieves its maximum performance. Thus the implementation of the ORST traversal logic is nearly as efficient as for pure single ray traversal.

Table 3.5 presents a direct measurement of the stream fragmentation by averaging the stream size maintained by each algorithm throughout traversal and intersection, which can be compared to the average initial stream size. The data is broken down by bounce number to demonstrate the effect of increasingly chaotic ray distributions. For DRST the switch to single ray traversal is disabled to allow a direct comparison to ORST, otherwise the results for the effective stream size would be significantly lower. The numbers show clearly that ORST can always maintain larger streams for both traversal and intersection, up to 19% during the first bounce for the R8 scene. On average, ORST requires only 85% of the traversal steps and 95% of the triangle intersections performed by DRST. Once a stream is sorted into octants by ray direction, the corresponding sub-sets never fragment and two rays that have an overlapping traversal path are guaranteed to perform the common steps and triangle intersections together. This property is unique to ORST and explains the higher efficiency for coherence extraction compared to DRST. For higher bounces the advantage decreases slightly due to overall decreased ray coherence.

Table 3.6 compares the distance heuristic and sign heuristic for traversal ordering according to the number of resulting traversal steps and intersection tests for single ray traversal. Both heuristics perform about the same on average for intersection and traversal, with a slight preference for the distance heuristic. Due to the fast look-up mechanism single ray traversal with the sign ordering is about 1% faster compared to the distance ordering in this particular implementation, though ultimately this

TABLE 3.5: Performance counts for diffuse rays corresponding to the benchmarks presented in Table 3.4. For each bounce the average effective stream size (number of active rays in a stream) during traversal and intersection is listed. The stream size for ORST is presented in absolute numbers, while for DRST the results are expressed relative to ORST. In addition the average initial stream size is given. The switch to single ray traversal for DRST is disabled.

	SPONZA	DRAGON	FAIRY	R8	HAIRBALL	POWERPLANT
	avg. rays node / leaf	avg. rays node / leaf	avg. rays node / leaf	avg. rays node / leaf	avg. rays node / leaf	avg. rays node / leaf
Bounce 1						
Initial	1024	1007	1024	1021	984	1000
ORST	27.6 / 8.69	17.3 / 6.19	23.5 / 8.61	35.3 / 14.7	8.84 / 4.03	38.1 / 13.6
DRST	88% / 93%	85% / 88%	85% / 87%	84% / 84%	86% / 88%	87% / 88%
Bounce 2						
Initial	1009	294	617	599	705	603
ORST	8.06 / 3.15	3.29 / 1.41	6.29 / 2.93	6.62 / 2.90	2.94 / 1.55	9.25 / 3.93
DRST	85% / 91%	88% / 93%	86% / 87%	82% / 86%	84% / 90%	88% / 89%
Bounce 3						
Initial	991	131	376	556	621	497
ORST	5.02 / 2.17	2.19 / 1.16	3.93 / 2.08	4.13 / 1.88	2.13 / 1.21	5.13 / 2.35
DRST	86% / 93%	90% / 96%	88% / 89%	85% / 90%	86% / 94%	89% / 91%
Bounce 4						
Initial	968	78	258	552	553	440
ORST	4.09 / 1.90	1.87 / 1.10	3.16 / 1.84	3.38 / 1.63	1.86 / 1.11	3.85 / 1.83
DRST	87% / 93%	93% / 97%	88% / 90%	86% / 92%	88% / 95%	90% / 93%

TABLE 3.6: Performance counters for the single ray traversal order produced by the distance heuristic and the sign heuristic corresponding to the benchmarks presented in Table 3.4. The total number of visited nodes and intersected triangles for the distance heuristic relative to the sign heuristic is reported.

	SPONZA	DRAGON	FAIRY	R8	HAIRBALL	POWERPLANT
Single ray						
# nodes	99.9%	100.0%	99.8%	100.0%	100.3%	99.8%
# triangles	103.9%	99.8%	99.7%	100.0%	100.0%	98.9%

depends on the specifics of the rendering framework and in general both methods appear to be an equally good choice. The major strength of the sign heuristic is the synergy with ORST where it helps to reduce stream fragmentation without compromising the traversal order quality for individual rays compared to the distance heuristic.

3.2.4 Summary

The introduced ORST algorithm is a novel ray stream traversal algorithm for processing traversal requests of ray groups with unstructured coherence. The algorithm is tailored towards 4-ary BVHs and vector widths of four to eight elements. A front-to-back traversal order based on the ray directional signs avoids dynamic fragmentation of the ray stream which enables higher coherence extraction and reduces complexity compared to previous approaches. The experimental results show that the sign-based order is on par with the distance-based alternative in terms of node

culling. An average traversal speed-up of 51% compared to the best performing stream algorithm DRST has been measured in the experiments.

3.3 Packet Traversal

The ray stream traversal introduced in the previous section benefits from unstructured coherence in ray sets and, if none exists, degrades gracefully to single ray traversal performance. For ray sets with *structured coherence* such as primary rays or shadow rays probing a common light source, for example, *packet traversal* allows even more aggressive vectorization and amortization techniques.

A very efficient packet traversal algorithm for the binary BVH is described by Wald et al. [108], where a single search ray out of the packet is intersected with the current BVH node, speculatively descending the entire packet on hit. Otherwise, a conservative frustum or interval test attempts to reject the node completely. Only then the individual rays are tested and upon the first intersection (if any) the search ray is updated and the traversal continues down the hierarchy. The implementation combines *speculative early hit*, *conservative early miss*, *ordered traversal*, *active ray tracking* and a *packet test of last resort* in a single unified traversal step.

The contribution of this section is the extension of the original packet traversal algorithm to support multi-branch BVHs. This allows mixing of packets with ray streams and single rays using the same acceleration structures, so that the most suitable technique can be chosen during runtime depending on the workload, for example small or large ray sets with or without structured coherence. In addition, multi-branch BVHs reduce the number of traversal steps and thus promise further performance gains for packets.

The basis for the novel multi-branch packet traversal is formed by the node ordering and stack modification techniques developed in the previous two sections for streams and single rays. Accordingly, two slightly different variants of the algorithm are described in the following, based on the ORST look-up table mechanism and the WIVE mechanism, respectively. Further, a reworked *deferred* packet test of last resort ensures that the same culling efficiency of the binary packet traversal is maintained for multi-branch BVHs as well.

3.3.1 Coherent Large Packet Traversal

This section describes the coherent large packet traversal (CLPT) based on the look-up table mechanism for node ordering introduced previously for ORST, thus targeting a 4-ary BVH. In the following, the terms *packet* and *large packet* are distinguished with respect to the number of rays contained and the corresponding memory layout. A *packet* exactly fits the vector length, for example containing eight rays for AVX, and organizes its rays into a SoA layout. A *large packet* comprises several packets organized in an AoS layout. The idea behind grouping rays in sets larger than the vector size is to allow conservative rejection and speculative acceptance of a node for the entire large packet, aggressively taking advantage of the assumed high coherence among the rays. The pseudo code of the complete algorithm is given in Algorithm 2, and the various techniques are discussed in detail below.

Algorithm 2 **Coherent large packet traversal** Vector symbols are postfixed with a 4, as are functions operating on vector data. Individual rays are accessed with R(ray index), SIMD ray packets with R4(packet index). Sign4() concatenates the sign bits of vector elements to a 4-bit integer, IsectBox4x1() intersects four child nodes with one ray, IsectBox1x4() intersects one node with a SIMD packet, FirstSetBit() returns the index of the lowest set bit, BitSet() returns the value of a bit at the given bit index, and CountBits() counts the number of set bits.

```
 1: stack[]
 2: sP ← 0
 3: node ← root
 4: activeRID ← 0
 5: loop
 6:     if node is inner node then
 7:         {Conservative early miss test using interval arithmetic.}
 8:         activeMsk ← IntersectBox4(node, intRay)
 9:         activeMsk ← activeMsk and node.activeMsk
10:         if activeMsk = 0 then
11:             goto line 46
12:         end if
13:         {Speculative early hit test with first active ray.}
14:         (hitMsk, t4) ←IsectBox4x1(node, R(activeRID))
15:         savedRID ← activeRID
16:         orderIdx ← orderLUT[sign][node.perm]
17:         order[] ← compactLUT[orderIdx][activeMsk]
18:         o ← 0
19:         {Packet test of last resort and active ray tracking.}
20:         for cnt ← CountBits(activeMsk)−1 to 0 do
21:             o ← order[cnt]
22:             if BitSet(hitMsk, o) then
23:                 goto line 35
24:             end if
25:             for p ← activeRID/4 to numPackets do
26:                 msk ← IsectBox1x4(node,R4(p))
27:                 if msk ≠ 0 then
28:                     activeRID ← p ∗ 4+FirstSetBit(msk)
29:                     goto line 35
30:                 end if
31:             end for
32:         end for
33:         goto line 46
34:         {Push remaining nodes to stack and continue with the closest according to ordered traversal.}
35:         for i ← 0 to cnt − 1 do
36:             so ← order[i]
37:             n ← root + node.child + so
38:             stack[sP] ← (n, savedRID, t4[so])
39:             sP ← sP + 1
40:         end for
41:         node ← root + node.child + o
42:         continue
43:     else
44:         Leaf intersection
45:     end if
46:     while sP ← sP − 1 ≥ 0 do
47:         (node, activeRID, t) ← stack[sP]
48:         {Early hit pruning.}
49:         if cast2uint(t) < cast2uint(R(activeRid).t) then
50:             goto line 5
51:         end if
52:         {Deferred packet test of last resort and active ray tracking.}
53:         for p ← activeRID/4 to numPackets do
54:             msk ← IsectBox1x4(node, R4(p))
55:             if msk ≠ 0 then
56:                 activeRID ← p ∗ 4+FirstSetBit(msk)
57:                 goto line 5
58:             end if
59:         end for
60:     end while
61:     break
62: end loop
```

Conservative early miss: this test attempts to quickly reject nodes which do not overlap with any rays in the large packet. Algorithm 2 employs interval arithmetic (IA) as discussed in [52] (line 8). While the alternative frustum plane test [8] is less conservative, the increased computational cost does not pay off, especially because IA shares some calculations with the speculative early hit (line 14, see below).

Applying IA to a set of rays generates intervals for the x-,y- and z-coordinates of ray origins and directions for all rays of a large packet to perform a conservative rejection test for nodes outside these intervals. The bounding box intersection test already is an IA operation itself, producing the $[t_{min}, t_{max}]$ interval. By expanding the definition of an origin o^i and inverse direction d^i from points and vectors to the intervals $[\underline{o}^i, \overline{o}^i]$ and $[\underline{d}^i, \overline{d}^i]$, respectively, t_{min} and t_{max} can be computed conservatively for a set of rays with the following changes applied to Equation 3.1:

$$\begin{aligned} t^{n,i}_{max} &= (b^{n,i}_{max} - \underline{o}^i) * \overline{d}^i \\ -t^{n,i}_{min} &= (b^{n,i}_{min} - \overline{o}^i) * (-\underline{d}^i) \end{aligned} \tag{3.2}$$

Here, we assume that $(\underline{d}^i, \overline{d}^i)$ does not contain 0, i.e., all ray directions in the large packet have the same sign combinations. During the set-up phase, a special interval ray "*intRay*" is created with maximum and minimum values for the origin and the direction vector components. Bounding boxes missed by this interval ray will be missed by the large packet and can be ignored.

The IA is evaluated for all four child nodes simultaneously, resulting in a 4 bit active mask which is further superimposed with the active mask stored in the node data structure (Figure 3.7). If the active mask is zero at this point, all active child nodes are missed and traversal can continue with a node from the stack (lines 46-60).

Speculative early hit: the purpose of this test is descending into a child node early if it overlaps with the current active ray (see active ray tracking below), omitting bounding box intersections for the rest of the rays, which are assumed to take the same traversal path. While this speculation can be harmful in the case of divergent rays, for highly coherent rays the performance gains can be huge. The intersection of the active ray is performed for all child nodes in parallel and an early hit mask along with the intersection distance is produced (line 14). The intersection distance is set to a value with all bits equal to one if no intersection exists.

Ordered traversal: as described in Section 3.2.1, the *perm* field of the current node together with the signs of the active ray represent an index into the orderLUT, which yields the *order index* (line 16). The *order index* in conjunction with the *active mask* form another index into the compactLUT, which returns an ordered list of active child node indices (line 17). The list is represented by a single byte and the elements are 2-bit values.

Deferred packet test of last resort: the ordered child nodes are iterated from front to back (lines 20-32). First, the early hit mask of the current child node is checked and if the corresponding bit is set, control flow exits the loop (lines 22-24). Otherwise, the packet test of last resort is executed (lines 25-31). If a new active ray can be found, control flow returns to the early hit path (lines 27-30). If not, the loop continues with the next child node until all have been processed without success (line 33). At this

point, traversal can continue with a node from the stack. The first child node with a ray overlap (if any) becomes the new current node (line 41). The remaining unprocessed child node indices are pushed onto the stack in a back-to-front order, together with the near distance value from the early hit test and the corresponding active ray (lines 35-40). This way, the packet test of last resort is deferred for stacked nodes until the moment of their retrieval (lines 47, 53-59), maximizing culling efficiency.

Active ray tracking: active ray tracking is essential for the early hit test, because if the test fails once it will fail for the entire sub-tree. Instead, a new active ray (if any) is found during the packet test of last resort (line 28). As a bonus, all rays that come before the active ray are guaranteed to miss the current sub-tree and require no further processing. In our implementation, the rays of a large packet are stored continuously in memory in a SOA format befitting the vector width, so that all rays in a packet can be intersected with one node simultaneously (line 26). The active ray is the first ray in memory order, which has an overlap with the current node. The ray index of the active ray is composed from its packet index and its position within the packet (lines 28 and 56).

Early hit pruning: a node popped from the stack requires a deferred packet test of last resort if the early hit test has failed, but also if, in the meantime, the active ray has detected a primitive intersection closer than the hit distance saved earlier. This is important to increase culling efficiency. Since the saved distance value is either a positive real number due to a successful early hit or a value with all bits set to one otherwise, both cases are evaluated correctly by comparing the bit patterns of the saved distance and the active ray distance as unsigned integers (line 49).

The combination of the presented techniques forms an efficient algorithm for large packet traversal of coherent rays in a 4-ary BVH. The overhead of the traversal logic is minimized due to the order and compaction look-up mechanism, and a culling efficiency identical to the original binary BVH traversal is achieved by the combination of the deferred packet test of last resort and early hit pruning. The notable advantages of the new algorithm are full vector utilization during early hit and early miss tests and significantly reduced traversal steps resulting from the shallower hierarchy.

3.3.2 Wide Vector Coherent Large Packet Traversal

This section discusses the idea of augmenting the WIVE single ray traversal approach with interval arithmetic (IA) culling and node ordering for packet tracing.

The recipe of the algorithm includes most of the ingredients of CLPT, as outlined in the previous section, omitting a dedicated speculative early hit test with a single ray. Instead, the first successful packet intersection initiates the speculative descent for the remaining packets, if any, into the corresponding node. Accordingly, the active ray tracking is replaced by first packet tracking for large packets. This is achieved by means of a first packet index (FPI), initialized to the first element in the packet list. Bounding box intersection starts with the FPI packet, and if a valid intersection exists the remaining packets will be assumed to hit the node as well; otherwise, the FPI will be incremented until either the first packet with a valid intersection is found, which will continue traversal, or all packets have been tested, triggering a stack-pop operation. The assumption is, similar to CLPT, that if packets have high coherence

the result of a single packet correctly predicts the behavior of the remaining packets, reducing bounding box tests considerably. Wrong predictions will drag uninvolved packets down the BVH and increase the number of bounding box tests instead.

The change from look-up table to WIVE for node ordering and stack management, combined with the simplified speculative descent, yields a more streamlined and compact variant of CLPT with support for BVHs with large branching factors, named WIVEC.

Listing 3.5 provides pseudo code for the WIVEC algorithm. The input variable *packets* is a list containing one or more ray packets (line 1). The interval ray is calculated (line 3) to enclose all rays within the packets, with maximum and minimum values located in the even and odd lanes for half width traversal or in separate registers for full width traversal, respectively. If the current node points to a node cluster one of the *traverseCluster* functions defined in Listings 3.4 and 3.2 is performed on the interval ray (line 7), returning the sorted list of elements, where the first element is extracted for continued traversal (line 8) and the remaining elements are stored to the stack (line 9). The function is slightly modified (*) in the sense that it returns different stack elements compared to those illustrated in Figure 3.2c. Instead of a direct reference to the child cluster, a reference to the parent node is stored, along with the current FPI. The t_{min} value is not required because the intersection test is performed for the first active packet only (lines 14 to 17) and deferred for the remaining packets until they become the fist active, so there is no need for stack culling. If the current node points to a primitive cluster, primitive intersection is performed (line 11), followed by a stack pop (line 12). In the following loop (line 13) the current node is intersected by the ray packets (line 15) and repeatedly replaced by a new node from the stack until either the first valid intersection is found (line 16) or the bottom of the stack is reached (line 13), i.e., no more stack elements exist and the traversal is completed (line 19). The *current* method (line 15) returns the packet pointed to by the FPI, and the *next* method (line 17) advances the FPI to the next packet. If only a single packet is traversed, lines 14 and 17 can be omitted.

LISTING 3.5: Main traversal function for WIVEC.

```
1  def traversePackets(node, packets)
     stack ← {}
3    ray ← packets.intervalRay()
     do
5    outerLoop:
       if(node.isInner())
7        (elems, num) ← traverseCluster*(node.cluster, ray, stack.top())
         (node, packets.fpi) ← elems[0]
9        stack.push(elems[1:num], num−1)
       else
11       intersectLeaf(node, packets)
         (node, packets.fpi) ← stack.pop()
13     while(node ≠ stack.bottom())
         do
15         if(intersect(node, packets.current())
             goto outerLoop
17       while(packets.next())
         (node, packets.fpi) ← stack.pop()
19   while(node ≠ stack.bottom())
```

3.3.3 Results

The results for CLPT and WIVEC are obtained from the experimental setups familiar from ORST and WIVE evaluation, respectively. The different hardware platforms do not allow a direct comparison, however the two algorithms are not considered to be competing methods since they provide same function for different types of data structures, catering to different application designs. CLPT is evaluated first with a focus on large packets and maximum throughput given abundant coherence, followed by WIVEC processing only single packets to demonstrate performance in situations where only a small set of rays of high coherence is available.

TABLE 3.7: Results for primary rays. Pictures of the test scenes are presented in Figures 3.5, 3.8 and 3.9. The measurements are taken during a full camera fly-through with the total number of frames listed below. The performance numbers include ray generation, traversal, intersection and simple shading. The resolution is 1280×1024 with 1 and 16 samples per pixel (spp).

	SPONZA		DRAGON		FAIRY		R8		HAIRBALL		POWERPLANT	
# triangles	66k		871k		174k		795k		2.9M		12.8M	
# frames	4000		2600		3000		4000		2000		2600	
	Mray/s		Mray/s		Mray/s		Mray/s		Mray/s		Mray/s	
	1 spp	16 spp	1 spp	16 spp	1 spp	16 spp	1 spp	16 spp	1 spp	16 spp	1 spp	16 spp
CLPT (AVX)	1192	**1552**	594	**1247**	819	**1282**	883	**1362**	178	**452**	583	**1018**
BVH2 (AVX)	655	736	427	717	493	636	577	752	113	212	286	424
Embree (AVX2)	349	383	293	409	322	385	387	463	110	161	250	307
	Speed-up		Speed-up		Speed-up		Speed-up		Speed-up		Speed-up	
BVH2	1.8×	**2.1**×	1.4×	**1.7**×	1.7×	**2.0**×	1.5×	**1.8**×	1.6×	**2.1**×	2.0×	**2.4**×
Embree	3.4×	**4.1**×	2.0×	**3.1**×	2.5×	**3.3**×	2.3×	**3.0**×	1.6×	**2.8**×	2.3×	**3.3**×

The CLPT results are presented in Table 3.7. The measurements include ray generation, intersection and simple shading (dor product, no secondary rays), which only amounts to a small fraction of the total run time. Embree is configured to use packet traversal (*BVH4Triangle4IntersectorChunkMoellerNoFilter*). The BVH2 entry is a custom, optimized implementation of the original binary BVH algorithm [108], the starting point for the CLPT design. Both algorithms coexist in the same code base and use a tile size of 8×8 pixels. CLPT is able to outperform Embree significantly, from 1.6× for the HAIRBALL at 1 spp to 4.1× in SPONZA at 16 spp. The algorithm aggressively exploits the high coherence in the ray sets, saving many instructions compared to Embree due to conservative early miss, speculative early hit and efficient culling. This can be demonstrated in two ways, either by increasing the pixel samples or by considering, for example, the HAIRBALL and SPONZA results. HAIRBALL is a chaotic model with many small triangles, while SPONZA has a number of large surfaces composed of few triangles so that rays are more likely to hit the same triangle. In both situations rays tend to have higher coherence, which shows in the relative performance increase. The comparison of the HAIRBALL and SPONZA results also brings up a limitation and a corresponding optimization opportunity for CLPT, which shares the performance characteristics of the original binary BVH algorithm: For scenes with very small triangles (compared to the tile size) performance degrades disproportionately compared to Embree due to the overestimating nature of the speculative early hit. In this case a switch from large packets to a finer granularity could be beneficial. Comparing CLPT to its binary BVH ancestor, a speed-up from 1.4× up to 2.4× is observed. For the first time, a 4-ary BVH traversal for primary rays is demonstrated to be faster than the best competing binary BVH

algorithm.

TABLE 3.8: Performance counters for the distance and sign traversal order heuristics for coherent ray packets corresponding to the benchmarks presented in Table 3.7. The total number of visited nodes and intersected triangles for the distance heuristic relative to sign heuristic is reported.

	SPONZA	DRAGON	FAIRY	R8	HAIRBALL	POWERPLANT
Large packet						
# nodes	100.2%	101.2%	100.6%	99.9%	100.1%	99.7%
# triangles	104.9%	102.7%	104.2%	99.9%	100.7%	100.6%

Both implementations perform exactly the same number of triangle intersections because the traversal order produced by the sign heuristic is identical in both cases, as is the culling efficiency due to the deferred packet test of last resort. At the same time, CLPT requires less traversal steps due to the higher branching factor of the 4-ary BVH and also allows to efficiently vectorize the conservative early miss and speculative early hit tests. Table 3.8 compares the distance and sign traversal order heuristics according to the number of resulting traversal steps and intersection tests for CLPT. Intersection tests are slightly reduced in case of the sign heuristic, while traversal steps are about equal compared to the distance heuristic. The CLPT results are similar to the single ray case in Table 3.6.

TABLE 3.9: Performance in million-rays per second (MRays/s) for our WIVE coherent algorithm (WIVEC) and Embree's hybrid traversal based on AVX-512 implementations. The packet size is 4×4 pixels. An image is rendered at a resolution of 3840×2160 pixels using primary rays. The camera perspectives in the scenes correspond to Figure 3.5.

	MAZDA	SAN MIGUEL	ART DECO	POWERPLANT	VILLA
# triangles[M]	5.7	10.5	10.7	12.8	37.5
AVX-512					
WIVEC	**555**	**533**	**796**	**472**	**337**
Culling[%]	73	82	82	78	76
Embree	275	220	409	212	184
WIVEC[+%]	**102**	**142**	**95**	**123**	**83**

The WIVEC algorithm is evaluated next. The performance comparison data for the WIVEC traversal and Embree's hybrid traversal for primary rays is provided in Table 3.9. To obtain a valid comparison WIVEC is executed on a single ray packet at a time, thus exploiting coherence at the same granularity as the hybrid traversal. The results demonstrate a significant and consistent speed-up of between 83% and 142% for WIVEC across all scenes. The culling statistics shown in Table 3.9 indicate that the interval ray avoids between 73% and 82% of all node intersection tests, partly explaining these impressive results. The other important aspect is reduced code complexity resulting from the integrated culling and ordering technique. Since only a single ray packet is processed at a time, conventional culling implementations would pose a significant overhead because amortization over a large packet is not possible. The chosen high image resolution favors frustum culling methods due

to high ray coherence. Less coherence would reduce culling efficiency and speed-up accordingly. However, this is true for packet tracing in general. As an avenue for future work fusing of WIVEC and hybrid traversal could prove beneficial to further accelerate partly coherent ray packets such as those occurring for shadows and specular effects.

3.3.4 Summary

The two algorithms, CLPT and WIVEC, presented in this section accelerate the traversal of packets featuring structured coherence. Speculation and interval arithmetic applied to the entirety of a ray packet scales the efficiency gains beyond the vector width. The novel algorithms build on an original packet traversal algorithm designed for the binary BVH and extend it to support multi-branching BVHs. The main difference between CLPT and WIVEC is the traversal order technique, resulting in compatibility with either ORST or WIVE data strutures, respectively. For primary visibility, the conducted experiments show that CLPT achieves an average speed-up factor of 2.8× over Embree, which is the industry-leading ray tracing library. In the case of WIVEC a speed-ups by between 83% and 142% are observed on an Intel®Xeon Phi™ CPU.

3.4 Conclusion

This chapter has introduced four novel algorithms for ray traversal through bounding volume hierarchies, motivated by the observation that ray traversal performance is fundamental to the design of any efficient ray tracing system. The overarching approach for the design of these algorithms has been parallelization through vectorization, i.e., utilizing the vector capabilities of modern processors to profit the data parallelism exposed through multi-branching BVH and ray coherence techniques. Each of the algorithms, within their respective categories, define the current state-of-the-art regarding traversal performance on CPUs as verified by experiment.

With the WIVE algorithm a fully vectorized formulation and implementation for single rays and wide multi-branching BVHs has been introduced, which completes the partial vectorization of previous approaches and improves upon scalability correspondingly.

The ORST algorithm solves the problem of dynamic fragmentation of ray streams due to front-to-back traversal ordering, increasing effective ray coherence and reducing code complexity.

The pair of WIVEC and CLPT algorithms, based on an original algorithm designed for binary BVHs, accelerate the traversal of ray packets featuring high coherence compatible with multi-branching BVHs.

The manual vectorization exercised during the implementation of the traversal algorithms, besides parallelization, has yielded highly optimized code in general. The procedure shifts the focus from language abstractions towards instruction set architecture and data structures, which, in a feedback-loop with algorithm design, are key to gaining access to the entire computational performance.

Which of the introduced algorithms performs best is application dependent in general. Ray streams and ray packets require the application to commit rays for traversal in sufficiently large groups for efficient execution, preferably with an effort to maximize ray coherence within groups. A design for rendering pipelines facilitating ray streams and packets is ongoing research [77, 2]. For applications traversing only a small number of rays at a time, single ray tracing is most likely the fastest and most convenient option. Sometimes, a modification or permutation of the techniques discussed in this chapter may yield a new algorithm variant best suited for the particular needs of the application.

Looking forward, it is unclear if major leaps in CPU-based ray traversal performance can be expected, given the tight coupling and high degree of optimization of the algorithms and their implementations presented here. A promising avenue for dramatically improving ray traversal performance in the future are special purpose accelerators in hardware, for example co-processors on the CPU die.

FIGURE 3.9: Pictures of benchmark scenes rendered with the ORST and CLPT algorithms using path tracing. The shading includes materials and textures, if available. From top to bottom, and with triangle counts: R8 (795K), and BOEING (300M), courtesy of Boeing Corporation.

Chapter 4

Parallel Bounding Volume Hierarchy Construction

In this chapter a new algorithm for parallel construction of high-quality BVHs is introduced. The quality of a BVH directly relates to the traversal performance achieved by a given traversal algorithm and, thus, it is a key aspect in maximizing ray tracing speed. Another key aspect is the BVH construction time, which is especially important for dynamic applications where parts of the scene geometry change over time, requiring a partial or full rebuild of the BVH in every frame. Constructing a BVH with high quality requires more computation compared to a low-quality BVH so that a trade-off must be made between speed and quality. On opposing ends of the algorithm spectrum, the split BVH (SBVH) [97] produces the best quality hierarchy on average, while the linear BVH (LBVH) [76] features the lowest construction times. Conversely, SBVH construction is notoriously slow while the LBVH yields poor traversal performance. The far inferior construction speed of the original SBVH implementation compared to the LBVH is caused not only by the additional computation but more significantly due to the lack of parallelization. In the following several parallelization techniques are introduced that make possible the derivation of a new massively parallel SBVH algorithm. Initially, Section 4.1 details the SBVH algorithm in its original form to provide context and terminology. Section 4.2 discusses the challenges associated with SBVH parallelization regarding task scheduling, memory management and vectorization and the following Sections 4.3-4.5 present the corresponding solutions, respectively. Finally, Section 4.6 delivers experimental results and Section 4.7 concludes the chapter, which is based on a publication by Fuetterling at al. [40].

4.1 Bounding Volume Hierarchies with Spatial Splits

This section introduces the SBVH algorithm [97] and establishes the terminology used throughout the chapter. The structure of the SBVH algorithm is similar to other divisive BVH builders. Initially, a single set of primitives exists (the parent set) that is partitioned into two smaller sets (the child sets). Partitioning is repeated recursively until the sets are small enough to form leaf sets. Along with every new child set a node is created which is referenced by its parent node and holds the axis-aligned bounding box enclosing all the primitives in the set. Once a set is turned into a leaf the corresponding node (now a leaf node) references the remaining primitives directly. As the SBVH is SAH-based, determining the partitioning with minimum cost for a given set is required before the actual subdivision can be performed. Since finding the exact partitioning with minimum cost is not feasible, an approximate is computed by choosing a small number of samples and selecting the one with the

lowest cost. The sampling is implemented in the form of binning, where the parent bounding box is subdivided into $n + 1$ equally-sized bins by n equidistant axis-aligned planes. The binning is performed for each axis separately.

The SBVH algorithm distinguishes between object binning and spatial binning. During object binning primitives are only considered as point-like elements defined by the center of their bounding box, while spatial binning takes the full size of a primitive into account. Thus, spatial binning requires a primitive to be split if it overlaps one or multiple of the planes. While object binning produces partitionings that have disjoint child sets which may have overlapping bounding boxes, spatial binning leads to partitionings with disjoint bounding boxes but possibly overlapping child sets. Which binning strategy will result in the lowest cost partitioning is dependent on the primitive constellation and cannot be foreseen. Thus, the approach taken by the SBVH is to find the best object partitioning and, if the corresponding child bounding boxes overlap by a certain amount, try to lower the cost further by testing the spatial binning. This is a sensible compromise because splitting is an expensive operation and increases memory consumption, while it is unlikely to improve SAH cost if object binning yields spatially disjoint sets. The total amount of splits during hierarchy construction is bounded by the *split budget* parameter. Once the split budget is consumed spatial binning is disabled. The SBVH algorithm is summarized in the following pseudo code:

```
1:  stack[]
2:  task ← root
3:  loop
4:    loop
5:      leafCost ← CalculateLeafCost(task)
6:      objCost ← BestObj(task)
7:      if task.childBoxes overlap then
8:        spatialCost ← BestSpatial(task)
9:      end if
10:     if leafCost is best then
11:       createLeaf(task)
12:       break
13:     else if objCost is best then
14:       (left, right) ← PartitionObj(task)
15:     else
16:       (left, right) ← PartitionSpatial(task)
17:     end if
18:     createNodes(task)
19:     stack.push(right)
20:     task ← left
21:   end loop
22:   task ← stack.pop()
23:   if task is empty then
24:     break
25:   end if
26: end loop
```

A task contains all the information required to partition the corresponding set of primitives. After partitioning, execution continues with one of the two resulting tasks, denoted left and right, while the other is pushed to the task stack. Once the

task stack is popped in an empty state, hierarchy construction is finished.

4.1.1 Primitive Fragments

Instead of working directly with the primitives (triangles in most cases), proxy elements called *fragments* are used. A fragment stores the axis-aligned bounding box and a reference to the primitive it represents. Thus fragment data is sufficient for the binning process and access to the full primitive structure is only required in the event of splitting. Also splitting does not result in duplication of the primitives, just copies of the corresponding fragments with refitted bounding boxes.

4.1.2 Binning

As mentioned previously, the parent bounding box is sliced into $n+1$ equally sized bins $b_i, i \in [0, n]$, separated by n equidistant axis-aligned planes $p_i, i \in [0, n-1]$. Each bin keeps track of the number and spatial extent of the fragments assigned to it. The *bin index* i corresponding to a particular coordinate c is computed as $i = (c - parent_{min}) \,/\, planeDistance$. In the case of object binning a fragment's bin index is derived from its bounding box centroid. Spatial binning requires two indices, i_{min} and i_{max}, calculated from the minimum and maximum of the fragment's bounding box, respectively. If the indices differ the fragment overlaps all bins $b_i, i \in [i_{min}, i_{max}]$ and requires splitting at every plane $p_i, i \in [i_{min}, i_{max} - 1]$, resulting in $i_{max} - i_{min}$ new fragments. The bounding boxes of the fragments are updated to tightly fit the primitive they represent within their respective bin. After the binning procedure, the SAH cost is evaluated for every pair of child partitions left and right to the planes p_i.

4.1.3 Partitioning

Performing the partitioning resulting from object binning is straightforward: for each fragment the bin index is computed again and compared to the best plane index i_{best}. If the bin index is smaller the fragment is moved to the left, otherwise to the right set. Since the left and right counts are known from the binning, memory offsets can be computed to store the fragments of both sets in a continuous array.

In the case of spatial binning the procedure is slightly different: minimum and maximum indices are computed again and compared to the best plane index i_{best}. If i_{min}/i_{max} is smaller/larger than i_{best} the fragment is moved to the left/right set. Otherwise the fragment intersects the split plane and requires insertion into both sets.

4.2 Parallelization Considerations

The properties of the SBVH algorithm most relevant to parallelization are its classification as a divisive construction algorithm and its dynamically growing working set due to the spatial splitting of objects. The implication of these properties are discussed in the following paragraphs with respect to multi-threaded execution.

Divisive algorithms repeatedly divide the working set into disjoint partitions, each linked to an independent task as defined in the previous section. The tasks can be processed by multiple threads in parallel without synchronization at maximum parallel efficiency.

However, at the start of hierarchy construction, only a single task exists (the root task) and with every subsequent level of subdivision the number of independent tasks doubles. Consequently, a maximum of 2^n tasks are available at level n. If the number of participating threads is high, this initial bottleneck harms scalability considerably since the amount of work is approximately constant at every level. For example, a 256 thread CPU would start using 100% of its resources only from level eight onward, yielding an overall efficiency of only 56% for a hypothetical balanced 16 level BVH.

Removing this bottleneck requires the implementation of shared task parallelism, so that multiple threads can collaborate on the same task [105, 111]. A shared task distributes its fragments among the participating threads which have a separate set of bins each. The threads map their fragments to their local set of bins and a reduction operation at the end of the binning phase accumulates the local bins for the global result and calculates per-thread offsets for the following partitioning phase. Thus, shared tasks introduce synchronization points and additional bookkeeping with detrimental effects on scalability. The impact on performance depends on the ratio of fragments per thread: if it is very high (10000, for example) the impact may be negligible but for low ratios (100, for example) it can be substantial. Hence, per-thread exclusive tasks are preferable when possible.

However, exclusive tasks are not free from scalability pitfalls either due to their variable computational cost. An imbalance occurs at the end of construction if not all threads finish their final task at the exact same time because no further work can be distributed to idling threads. This issue can be minimized by ensuring that tasks with mostly homogeneous, small workloads remain towards the completion of the BVH.

Further, since the generation of new tasks hierarchically depend on the completion of previous tasks, a single large task can produce a global bottleneck: if a thread request a new task but the outstanding number of tasks is zero, then this thread may have to stall until the large task completes and produces two new tasks. This situation is equivalent to the beginning of construction where only the root task exists.

Figure 4.1 gives an illustration of task dependencies and possible bottlenecks in extreme cases. Initially, two threads are about to take on a new task each after completing a shared task (black dot). Thread 1 continues to work on task 1 and the corresponding sub-tree (red circle) and thread 2 starts processing task 2 (green dot). The key question is which task (3 or 5) thread 2 should continue with after task 2 is finished, assuming that the time required for a task is proportional to the number of its fragments and, without loss of generality, that task 5 is larger than task 3. Shown on the right of Figure 4.1, the five possible extreme configurations of task sizes are:

a) Task 1 and its entire sub-tree take less time than task 2. Stall unavoidable.

b) Thread 2 chooses task 5, tasks 1 and 3 including their sub-trees are finished by thread 1 after task 5. No stall.

c) Thread 2 chooses task 3 and its sub-tree while thread 1 gets task 5. No stall but takes more time than b) to finish task 5.

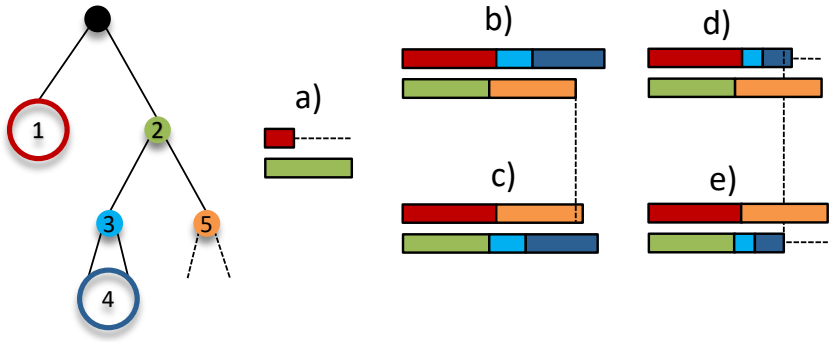

FIGURE 4.1: Example of a task dependency tree. Solid discs mark individual tasks, circles represent the set of tasks corresponding to an entire sub-tree. On the right, five examples illustrate the time lines of the tasks as colored blocks arranged in double rows, matching the corresponding task color on the left. The double rows correspond to two separate threads. The various constellations are explained in the text.

d) Thread 2 chooses task 5, tasks 1 and 3 including their sub-trees are finished by thread 1 after task 5. Short stall.

e) Thread 2 chooses task 3 and its sub-tree while thread 1 gets task 5. Long stall.

Hence, in any of these cases, the best possible outcome is achieved when the larger task 5 is prioritized. This corresponds to minimizing the maximum latency in the task dependency chain.

In addition to task distribution, a second critical component for multi-threaded SBVH construction is efficient memory management. For efficiency reasons, the fragments of a partition are kept in an array of continuous memory instead of, for example, a linked list. Splitting the partition without increasing the number of fragments produces two new partitions that fit again into the original partition's memory as two separate, continuous arrays. In the event of spatial splits, however, the number of fragments increases and the available memory is no longer sufficient to host the two new partitions as continuous arrays. With multiple threads, the original partition can not safely grow its bounds to the left or to the right because other threads may be working on neighbouring partitions. Thus, a straightforward solution would be to dynamically allocate memory for the new partitions. Since splits happen frequently and the number fragments increase recursively, this would lead to excessive allocation which would waste resources and reduce scalability.

In the following, the aforementioned parallelization issues are addressed by new approaches for task parallelism and dynamic memory growth.

4.3 Multi-thread Schedule

The multi-thread schedule proposed in this section introduces dynamic thread pools and prioritized task exchange, two novel strategies for balancing BVH construction based on the previous observations. Dynamic thread pools initially employ shared tasks with the goal to create an equal work distribution among threads but strive to permanently switch to exclusive task execution as soon as possible. For the dynamic

load-balancing of exclusive tasks a lightweight lock-free mechanism is introduced which allows prioritized on-demand sharing of tasks while maintaining the task topology. Both contributions are generally applicable to divisive algorithms given an appropriate task cost estimation.

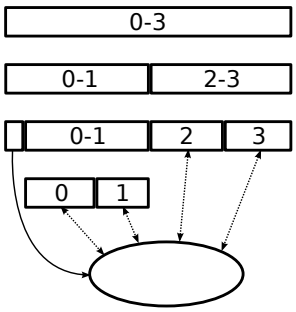

FIGURE 4.2: Visualization of the thread management. The numbers represent thread identifiers. The root task is processed by all threads, and after the subdivision the thread pool is split into two in proportion to the left and right set size. The next subdivision performed by threads 0 and 1 yields one very small and one very large set on the left and on the right, respectively, so the left task is inserted into the global task queue (implemented as a ring buffer) and both threads continue with the right task. Once a thread owns a task exclusively it switches to single-threaded execution. Dynamic load balancing is performed by exchanging tasks on the global task queue as indicated by the dashed arrows.

4.3.1 Dynamic Thread Pools

As mentioned previously, the goal of shared tasks is to allow fully multi-threaded execution from the start of hierarchy construction. At the same time it is desirable to minimize the number of threads processing the same task simultaneously and quickly reach the threshold where every thread can work on a single task exclusively. While the concept of shared tasks is not new itself [106], the proposed novel scheduling mechanism optimizes the above constrain, permanently switching to exclusive tasks as soon as a proper load balancing is established with the help of dynamic thread pools. Dynamic thread pools prevent the inherent risk of a permanent switch, once the number of independent task is equal to the number of threads, that the complexity of the individual tasks may vary widely, to the point where one thread has finished the entire sub-tree belonging to its task, while another thread is still working on the first subdivision, thus stalling the fast thread due to the lack of more tasks. Figure 4.1a gives an example of this kind of bottleneck.

The idea of dynamic thread pools is illustrated in Figure 4.2. For the root task all threads belong to a single pool. After the first subdivision the thread pool of size T is split into two, with the number of threads in each pool proportional to the number of fragments in the respective child tasks, according to the following equation:

$$T_l = \left\lfloor \frac{N_l}{N_l + N_r} T + 0.5 \right\rfloor, T_r = T - T_l,$$

where N_l and N_r are the number of fragments of the left and right child tasks, respectively, and T_l and T_r the number of threads of the corresponding thread pools.

Both pools can now operate independently. This procedure is repeated recursively, and once a thread finds itself to be the only one in the pool it permanently switches to exclusive task execution. If the subdivision of a shared task yields one child task with too few fragments to be assigned even a single thread, the task is inserted into the global queue for exclusive tasks and the entire thread pool continues with the larger child task. As a result, all threads will have tasks with roughly the same number of fragments upon switching from shared to exclusive task execution, creating an initial equalized work distribution such that the demand for dynamic load balancing of exclusive tasks is kept to a minimum and the risk for stalls due to high latency tasks is minimized.

4.3.2 Prioritized Task Exchange

Exclusive tasks are processed by a single thread only, thus avoiding any kind of synchronization. However, dynamic load balancing requires that tasks produced by one thread can be consumed by another. In addition, the task topology should be maintained across thread boundaries so that post-order procedures can be applied to the BVH hierarchy, such as leaf pruning.

In contrast to previous approaches, the novel algorithm does not classify tasks by the number of primitives to push them either to a strictly local stack or to a shared task pool. Instead, tasks are always placed on the local stack and exchange of tasks is achieved with a global task queue storing *task pointers*, which is implemented as a lock-free atomic ring buffer. The *target size* defines the number of tasks that should be available from the task queue at any time, for which the base-two logarithm of the thread count has been determined to be a good value. After subdivision of a task into two child tasks, the thread continues with the child task containing the larger number of fragments, i.e., higher cost estimation, and pushes the remaining child task onto the local stack. The thread checks the number of tasks in the global queue against the target size and inserts a task if necessary. Since the check is not atomic, it may happen that the number of tasks in the queue increases above the target size occasionally. Task insertion is always performed with a pointer to the bottom-most task on the local stack. Upon insertion the task is marked as *non-local*. As soon as post-order traversal of the local stack pops a non-local task, the traversal is terminated and a new task pointer is fetched from the global queue. A place holder containing the task pointer is pushed to the local stack. Once post-order traversal returns to the place holder, the corresponding pointer is used to write a completion notification to the original task on another thread's local stack. If a fetch operation is not successful because the queue is empty, the operation will block until a new task pointer is inserted by another thread. Once all threads have entered the blocking state hierarchy construction is almost finished and the threads are released with a null pointer. In the final step the remaining non-local and place holder tasks on the local stacks are processed until the post-order traversal reaches the root node.

The advantage of this approach compared to others [111] is that, on the one hand, task sharing happens only on demand increasing data locality, and, on the other hand, adapts the task size dynamically for optimal load balancing, with large task at the beginning and small tasks at the end of hierarchy construction.

4.4 Memory Management

In this section, a novel approach for recursively growing fragment buffers based on dynamic pre-allocation with reinjection is introduced. This solution requires no synchronization, retains a small memory footprint and, as a positive side effect, keeps the split budget balanced over the entire hierarchy.

A SBVH implementation requires two types of dynamic memory buffers: the temporary buffers, containing the fragments, need to support creation and shuffling of elements, whereas for the output buffers, holding the BVH nodes and the primitive lists referenced by leaves, it is sufficient to support only creation with the constrain that elements are packed as tightly as possible in memory.

Space allocation for the output elements is implemented by simple atomic counters that are shared among all threads. This is similar to previous approaches for BVH construction without spatial splits. In order to reduce the frequency of atomic operations threads always allocate entire chunks of elements and manage such a chunk with local counters. This mechanism is fast and lock-free, resulting in tightly packed elements where a small amount of fragmentation can occur only in the final chunk of every thread. The size of the output buffers can be conservatively estimated by considering the number of input primitives and the size of the split budget.

The presence of spatial splits complicates the management of the temporary fragment buffers considerably in a multi-threaded environment. The reason is that the fragments need to be partitioned recursively and, due to the primitive splitting, the combined size of the two child sets may be larger than the parent set. Thus, the new memory management needs to be significantly more flexible compared to previous approaches.

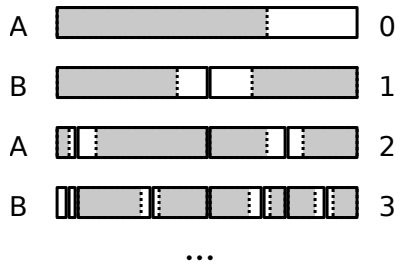

FIGURE 4.3: Visualization of the fragment buffer management. A and B on the left mark the two individual buffers forming the fragment double buffer. The numbers on the right denote the hierarchy level. At the root level buffer A is partially filled with the initial set of fragments (shaded region), while the remaining free space can be consumed by splits. After the first subdivision the child sets are aligned to the left and right borders of buffer B and the free space in the middle is divided proportionally to the set size. This process is repeated recursively on the child sets while alternating buffers A and B.

The key idea is to bind space in the fragment buffer to tasks, and recursively distribute this space among the corresponding child tasks as visualized in Figure 4.3. Initially, the entire fragment buffer is allocated to the root task, where the input fragments reside in the lower part of the buffer and the upper part provides free space for primitive splits. During the partitioning phase the left and right child sets are

created adjacent to the lower and upper boundary, respectively, growing towards the center with the free space in between. The remaining free space is distributed to the left and right tasks in proportion to the size of the respective child sets. Thus, a task always includes the necessary resources for its processing and a thread acquiring one of the tasks can directly access these resources without any additional synchronization. As proposed previously [105], the implementation of the fragment buffer features a double buffering technique, where the parent set resides in one buffer and the child sets are written to the other buffer. After a subdivision source and destination pointers are simply swapped. This way, read/write dependencies that would exist in an in-place approach are eliminated, allowing all fragments to be partitioned in parallel.

As a side effect of the proposed memory management the split budget is distributed evenly among the scene geometry, avoiding the situation where excessive splitting during the early part of the build process can drain the split budget for the later part. However, if a scene demands highly non-uniform split densities, the balanced distribution can be harmful. In this case the split budgets in low density regions go unused while in high density regions insufficient split budgets prevent optimal subdivisions.

To remedy this situation, another mechanism is proposed to reinject unneeded split budgets back into the build process. Upon completion of a leaf task the remaining number of splits are added to the *reserve counter*. The reserve counter is a global state that is managed with atomic operations to allow sharing of the reserve splits among all the threads. However, to reduce frequency of the expensive atomics, each thread caches its reserve budget with a local counter and updates the global state only occasionally. If spatial binning produces a partitioning that exceeds the split budget provided by the corresponding task, a thread acquires the difference from the reserve counters, where the local counter has priority over the global counter. If the reserve budget is insufficient, the algorithm falls back to the best object partitioning. Once the split budget has been secured, the fragment buffer region bound to the current task is not large enough to hold the fragments for both child partitions, so that a new partition needs to be allocated for the smaller of the two child partitions. At the initial allocation of the fragment double buffer, a part of the space is set aside for this purpose, referred to as the *reserve buffer*. Allocations from the reserve buffer are performed with an atomic counter, and once all the reserve space has been used up the remaining tasks can no longer use the reserve mechanism. It would be possible to allocate additional space from system memory as the new buffer does not have to be continuous with respect to the initial buffer, though this would be rarely necessary. Since the per-fragment memory consumption related to the double buffer is marginal (less than 1%), the reserve buffer can be large (e.g. twice the split budget).

Split budget balancing with reinjection combines the advantages of the purely balanced and first-come-first-served principle. Each part of a scene is guaranteed a relative amount of splits, while the unneeded budget can be shifted to high split density regions.

Finally, in order to reduce memory bandwidth demands and overall memory consumption,the fragment double buffer is replaced with a fragment reference double buffer and the actual fragment data is kept in a separate memory region managed by atomic counters in the same way as the output buffers. This is distinct from

previous publications [97, 105]. Since the fragment data structure is 32 bytes in size, whereas a reference occupies only 4 bytes, a total of $2 * 32 - (2 * 4 + 32) = 24$ bytes is saved per fragment. The bandwidth balance is also positive since each task reads its fragments $2 - 3\times$ and writes once. With references this amounts up to $3 * (4 + 32) + 4 = 112$ bytes per fragment while using the fragments directly would result in $3 * 32 + 32 = 128$ bytes. In addition, significantly reducing the size of writes from 32 to 4 bytes has the advantage of reducing DRAM access because, while reads are potentially serviced from the cache, writes need to be flushed to DRAM eventually. The drawback of this approach is increased access latency due to the reference indirection and inhibition of fragment hardware prefetching. However, experimental measurements have shown that for working sets fitting into the L3 cache performance is equal for both buffering schemes, while for working sets larger than L3 a total run time reduction of up to 35% with references is possible.

Interestingly, a very similar technique for recursively growing memory during spatial split partitioning has been developed in parallel [42]. In contrast to the proposition made here reinjection is not supported and the layout of the memory buffer does not keep the free space centered, resulting in unnecessary memory movement. As future work, improve parallelization of the initial phase of partitioning is suggested, which has been addressed in the previous section.

4.5 Vector Processing

The SBVH vector implementation utilizes AVX instructions for all compute intensive parts of the algorithm, specifically for object/spatial binning/partitioning, primitive splitting and SAH calculation. Very important for high vector efficiency is the data organization of the fragments: a good fit is the AoS organization as the bounding box requires six floating point values and one primitive index. By adding one padding element the fragment fits an AVX register exactly:

x_{min}	y_{min}	z_{min}	idx	x_{max}	y_{max}	z_{max}	pad

An SoA layout could avoid the superfluous padding element but would otherwise complicate the binning and partitioning computation considerably. One of the most common operations during binning is the union of two bounding boxes. With the previous data structure this would require unpacking and a minimum/maximum instruction on the lower/upper part. In order to calculate the union of two fragments with a single instruction, it is proposed to use the convention to store the negatives of the minimum values:

$-x_{min}$	$-y_{min}$	$-z_{min}$	idx	x_{max}	y_{max}	z_{max}	pad

This way, a single maximum instruction is sufficient, operating directly on the data structure without transformation.

In the following, the high level vector design of the binning/partitioning kernels and separately primitive splitting are described, which has not been discussed in literature before. Further implementation details are revealed by the source code provided by Fuetterling et al. [40].

4.5.1 Binning and Partitioning

Both binning and partitioning require the calculation of bin indices as described in sections 4.1.2 and 4.1.3. Depending on the bin index of a fragment, the binning kernel updates the count and bounding box of the appropriate bin while the partitioning kernel appends the fragment index either to the left or right child partition. Thus, operating on multiple fragments in parallel demands partly serialized scattered memory accesses. Since no hardware support is available for this kind of scattering mechanism, it is not obvious how to implement it efficiently in software. In fact, the initial attempts have barely improved performance upon the scalar code at all. Previous work has struggled with this problem as well [105], opting to utilize vector instructions inefficiently to parallelize over bins instead of fragments. Through experimentation the following efficient design pattern has been determined to work well for both binning and partitioning multiple fragments in parallel.

The basic idea is to divide the body of the loop over all fragments into a vectorized part for the bin index and a scalar part for the bin update. By interleaving the vectorized part for iteration $i + 1$ and the scalar part for iteration i, both parts can be processed in parallel as they utilize different execution ports of the CPU. Moving the first iteration of the vectorized part and the last iteration of the scalar part out of the loop yields an efficient implementation illustrated in the following snippet:

```
1: vector part start
2: for i = start to end do
3:     vector part i + 1
4:     scalar part i
5: end for
6: scalar part end
```

For both object and spatial binning two fragments are processed along all three axes simultaneously, utilizing six out of the eight vector elements. In this case this is faster than working with eight fragments, because the higher utilization would not compensate for the additional data shuffle overhead. For the object partitioning, however, only a single axis is of interest, so here the best approach is to process eight fragments in parallel. Spatial partitioning only operates on one fragment at a time because the more complex control flow diminishes the advantage of multiple elements.

4.5.2 Primitive Splitting

The primitives considered here are triangles, so primitive splitting requires a triangle-plane intersection test. Given an axis-aligned plane, the triangle-plane intersection is computed by choosing the two edges of the triangle overlapping the plane and calculating the corresponding line-plane intersection points. Processing the edges can be performed in parallel utilizing two vector elements of an AVX register. In order to profit from the remaining elements, multiple triangle-plane intersections are necessary.

The first option is to intersect one triangle with one plane in each dimension, filling only six of the eight vector elements. Further elements are wasted because a triangle is not very likely to overlap binning planes in all three dimensions simultaneously. The second option is to test four different triangles with four different planes. While

this approach guarantees high utilization, the overhead of gathering the data from many scattered locations would be quite significant.

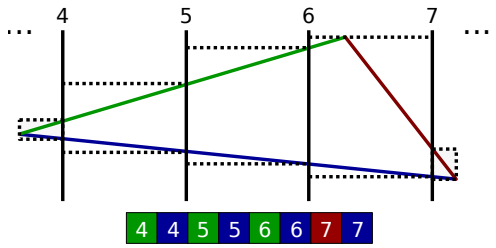

FIGURE 4.4: Visualization of the triangle splitting. Numbers denote planes, colors denote triangle edges. The squares represent elements of a vector register and are colored and numerated according to the edge-plane combination they process. The dashed lines indicate the tight bounding boxes of the triangle within the respective bin.

A good balance is achieved by performing intersection of one triangle with four consecutive planes along the same axis, as illustrated in Figure 4.4. This has the advantage of keeping data access coherent and allowing all vector elements to be utilized. Obviously, if the number of planes a triangle overlaps is not a multiple of four the effective AVX register utilization is reduced.

4.6 Results

The evaluation of the new SBVH algorithm focuses on three aspects: overall performance, parallel efficiency and the vectorization advantage. The overall performance is measured by constructing BVHs for several test scenes and comparing the timings to the parallel SBVH implementation of Embree [111] (version 2.7.1), a high-performance ray tracing library developed by Intel®. Both implementations are configured to perform binning along all axes with 32 object bins and 16 spatial bins, with the split budget set to 100% of the number of input triangles. Included in the timings are all computations required to obtain a ray tracing ready BVH, in particular the root bounding box calculation and triangle processing for accelerated ray-triangle intersection. Also, both implementations output a 4-ary BVH which does not alter the SBVH algorithm except for the node layout. The parallel efficiency is evaluated by analyzing build times for varying thread counts and for varying scene sizes. Finally, the performance advantage is measured which is achieved through vectorization of the binning/partitioning kernels and triangle-plane intersection implementations. For all experiments the hardware platform is a dual socket Intel® Xeon™ E5-2680v3 Haswell (24 cores / 48 threads total at 2.5GHz).

Overall Performance

The build performance is tested for six of the scenes used for the ray traversal benchmarks in the previous chapter, i.e., Figures 3.5, 3.8 and 3.9. The results are presented in Table 4.1. The new SBVH implementation demonstrates a significant speed-up over Embree for all scenes, ranging from $66k$ to $300M$ triangles in size. Especially for the smaller scenes below $1M$ triangles the new algorithm is between $5-7\times$ faster. As demonstrated below, this is influenced to a large extent by the scalability of the two implementations.

TABLE 4.1: Overall performance with 48 threads for several scenes depicted in Figures 3.5, 3.8 and 3.9, comparing the new SBVH implementation and Embree. The *splits* row indicates the increase in triangle count due to splitting for the new implementation.

	SPONZA	FAIRY	R8	POWERPLANT	HAIRBALL	BOEING
# triangles	66k	174k	795k	12.8M	2.9M	300M
Splits	30%	17%	10%	16%	89%	10%
Dual socket Intel Xeon E5-2680v3, 32 object bins and 16 spatial bins						
New	**3.6 ms**	**7.8 ms**	**22.9 ms**	**537 ms**	**351 ms**	**12.6 s**
Embree	26.3 ms	42.7 ms	153.8 ms	2724 ms	1266 ms	101.2 s
Speed-up	**7.3**×	**5.5**×	**6.7**×	**5.1**×	**3.6**×	**8.0**×
Dual socket Intel Xeon E5-2680v3, 32 object bins only						
New	**1.0 ms**	**2.5 ms**	**11.4 ms**	**55.5 ms**	**326 ms**	**10.4 s**
Embree	4.5 ms	9.4 ms	28.7 ms	83.8 ms	416 ms	14.3 s
Speed-up	**4.5**×	**3.8**×	**2.5**×	**1.5**×	**1.3**×	**1.4**×

Also, for extremely large scenes such as the BOEING performance is high with respect to Embree. This observation is attributed in one part to the reduced parallel efficiency of Embree measured for large scenes (Figure 4.6) and in one part to our bandwidth conserving reference scheme, as the highest speed-up of about 35%, relative to double buffering the fragments directly, has been observed for the BOEING.

Compared to the performance achieved by LBVH based builders on a Nvidia® GeForce™ GTX Titan GPU for a moderately sized scene such as FAIRY, the new high-quality SBVH implementation lies within the reported range of $2 - 9ms$ [67].

Parallel Efficiency

The parallel efficiency of the new SBVH implementation is analyzed in two ways, once by keeping the primitive count fixed and scaling the number of threads, and once by scaling the primitive count with all of the 48 threads active. Since the test platform has only 24 cores but 48 threads the core count is multiplied by 1.3 if hyper-threading (HT) is enabled. This multiplier has been determined experimentally by comparing performance for one thread and for two threads pinned to a single core.

Figure 4.5 depicts the scaling factor as a function of thread count for all the test scenes, together with the ideal curve. Up to about 10 threads (or 8 cores + HT) all scenes exhibit ideal scaling. After this point parallel efficiency diverges from the ideal curve and the graphs separate into two bundles. The smaller scenes including HAIRBALL scale up to 26×, while POWERPLANT and BOEING achieve around 20× for all threads active. This behavior indicates that our SBVH implementation is memory bandwidth limited since all the small scenes fit (almost) entirely into the large L3 cache.

The situation becomes clearer by analyzing Figure 4.6. Here the number of triangles is dynamically scaled at a fixed thread count of 48. At 10k triangles the problem size is too small for the new algorithm to scale above 15× (total run-time is about $0.7ms$).

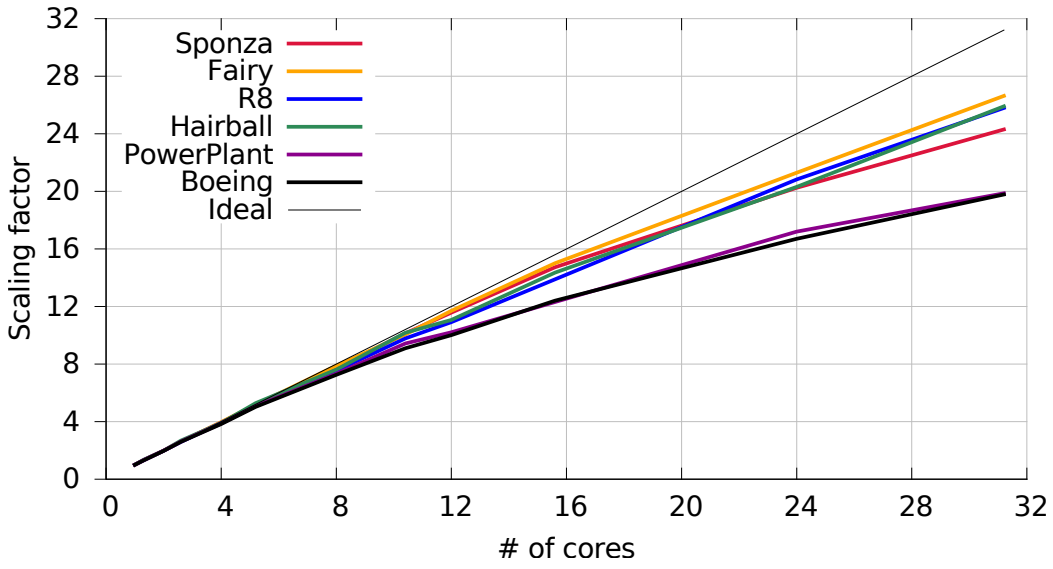

FIGURE 4.5: Scaling factor as a function of core count with respect to the performance of a single thread. Results for all the test scenes from Figure 4.1 are provided. If hyper-threading is enabled (two threads per core), the core count is multiplied by 1.3.

The plateau of highest parallel efficiency (around 26×) is reached with slightly less than 100*k* triangles and extends until about 2*M*. After that scalability rapidly decreases towards a steady state of 20×. This cliff is where the L3 cache looses its effectiveness, which fits with the data from Figure 4.5.

The scalability of Embree exhibits a different behavior. For small triangle counts parallel efficiency is significantly worse compared to the new implementation, but improves for larger triangle counts until catching up at about 750*k* triangles. From there, however, scaling continues up to the ideal of 32×. Contrary to the new algorithm there is no cliff once the scene size exceeds the L3 cache. This indicates that Embree is not limited by memory bandwidth constrains, but rather by computation and/or memory and thread management.

In order to illustrate the load balancing characteristic of the parallelization framework (Sections 4.3.1 and 4.3.2), Figure 4.7 shows the exchange events on the global task queue for the construction of the POWERPLANT scene. For the largest part insert and remove events are very sparse, with only about 10% (300 total / 6 per thread) exchanges until 90% of the BVH is completed. For the last 10% of BVH construction the event rate increases exponentially due to the continued decrease in average number of fragments per task. Hence, the load balancing works as intended: the dynamic thread pool mechanism leaves each thread with a similar initial task size upon switching from shared to exclusive task execution, reducing the demand for task exchange. Only when the tasks become small at the end of BVH construction, fine grained load balancing takes over to keep all threads busy.

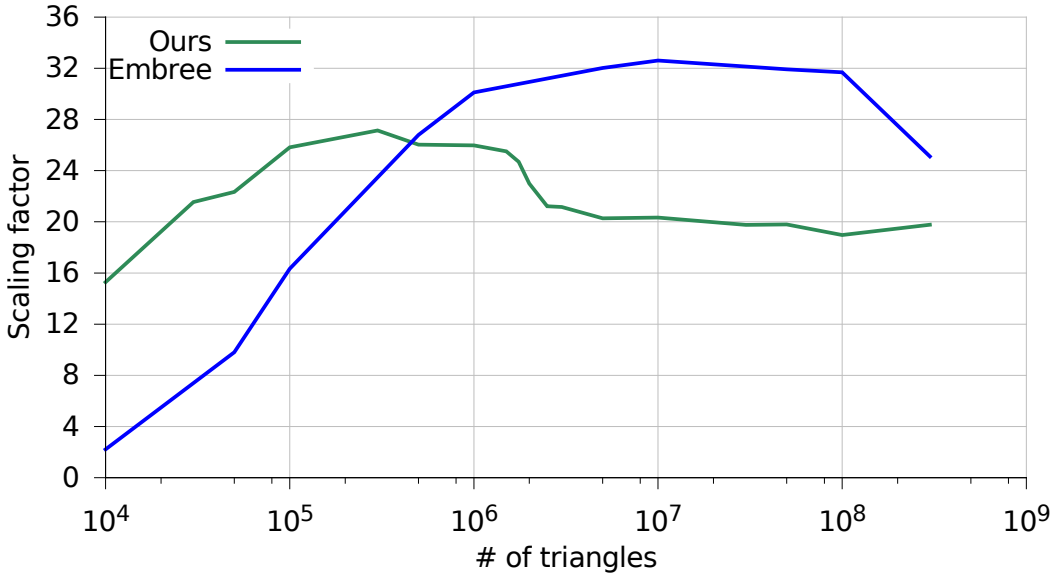

FIGURE 4.6: Scaling factor (1 vs. 48 threads) as a function of triangle count based on the BOEING scene. Results for both Embree and our implementation are provided.

Vectorization Advantage

For the evaluation of the AVX implementations described in Section 4.5 the results are divided in triangle intersection test and binning/partitioning kernels for both spatial and object variants. The speed-ups reported in Table 4.2 are relative to a scalar implementation for either the intersection test or the kernels, respectively, and include the full build process. For the kernels, the AVX version improves between 20% to 60% upon the scalar variant. The spread depends on the ratio of object to spatial binning, since for spatial binning most of the time is usually spent in triangle intersection and not in the binning itself. For triangle intersection, the results vary considerably from scene to scene, from a significant 3.6× for HAIRBALL to a mediocre 1.1× for BOEING. This is in line with expectations since the HAIRBALL geometry is predestined for excessive splitting while the BOEING and also the R8 have high object/spatial ratios.

TABLE 4.2: Speed-up due to vectorization with respect to scalar code. The results for the binning/partitioning kernels and the triangle-plane intersection test are reported separately. For the intersection test the average utilization of the vector registers is indicated (4 would be 100%).

	Kernels	Intersection	
	Speed-up	Speed-up	Utilization
SPONZA	1.3×	2.6×	2.3
FAIRY	1.4×	1.8×	1.6
R8	1.6×	1.2×	1.2
HAIRBALL	1.2×	3.6×	2.8
POWERPLANT	1.4×	1.4×	2.7
BOEING	1.5×	1.1×	1.6

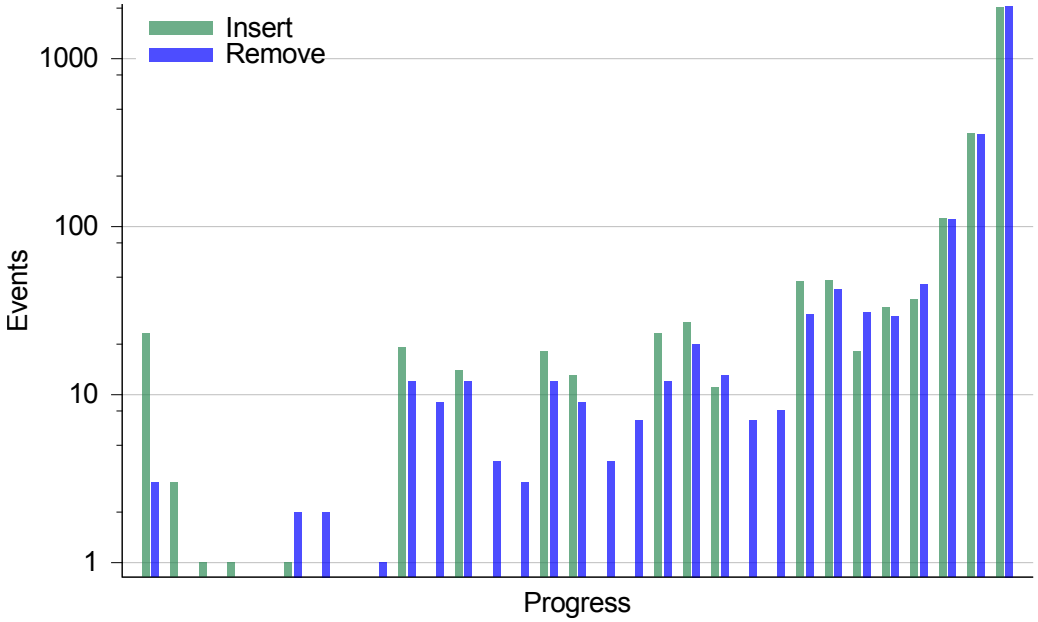

FIGURE 4.7: Exchange events on the global task queue (logarithmic scale) as a function of progress bins for the construction of the POWERPLANT scene with 48 threads. Progress is measured as the number of finished nodes at the occurrence of a particular event. The last bin accounts for 72% of all events.

Noting that an AVX register has eight elements that theoretically allow an 8× speed-up, the question is if a more efficient vectorization compared to the one presented here is possible. Since the binning kernels are very compact, this would most likely require hardware assistance for the gather/scatter operations. However, the triangle intersection leaves some room for further improvement since not all the possible strategies have been explored to efficiently saturate the vector registers.

4.7 Conclusion

In this chapter an efficient parallelization approach for BVH construction with spatial splits, i.e., the SBVH, has been introduced. The scalability of the new algorithm brings together low construction times *and* high quality hierarchies, where previously these two optimization targets have been strongly opposing each other.

For dynamic applications this makes possible the real-time processing of up to one Million triangles in SBVH quality for the first time. Very large scenes with hundreds of millions of triangles also profit tremendously with almost an order of magnitude speed-up over previous approaches, potentially accelerating the workflow for CAD modelling distinctly, for example. The high scalability of the new approach is based on three contributions:

1. **Multi-thread schedule**

   The schedule results from an analysis of the dependency chain produced by BVH construction workloads and strives to minimize the use of shared tasks

as well as latency bottlenecks among exclusive tasks. The schedule's specialization allows light-weight and lock-free load-balancing compared to a generic task-based parallelization scheme.

2. **Dynamic memory management**

   A synchronization-free strategy for managing recursively growing memory buffers due to primitive splitting in a multi-threaded context by employing a heuristic for distributing spare memory among tasks. The technique is applicable to divisive algorithms in general given a suitable heuristic.

3. **Vector processing**

   The introduced vectorized fragment processing and triangle splitting accelerates the computationally intense parts of SBVH construction while avoiding expensive gather / scatter memory access.

The experimental measurements show that the proposed SBVH implementation outperforms the best available alternative by 3-8x and rivals the speed of fast low-quality BVH builders. The results also indicate that for larger scenes system memory bandwidth limits performance despite the suggested optimization regarding fragment references. Hence, future work could focus on data compression for the BVH construction process which would likely require hardware assistance.

Chapter 5

Asynchronous Framework for Distributed Computing

In the previous chapters, vectorization and multi-threading have been exploited to maximize the rendering performance of a single shared memory computer. However, the scalability of vectors and threads is naturally limited by hardware architectural design, since shared, synchronized resources eventually form bottlenecks in the system. The computational demand of real-time photo realistic rendering by far exceeds the performance provided by any single shared memory system to date.

In contrast, a distributed system replicates many self-contained units with loose coupling. One example of such a massively parallel system is a cluster with physically distributed memory. Cluster nodes and their cores can operate on local memory with low latency and high bandwidth, since memory coherence with respect to other nodes and their cores is not required. A high-performance network serves for explicit communication between nodes. Distributed memory architectures are highly scalable as performance of individual nodes is not degraded by adding more nodes. In practise, software limits the scalability of a cluster application. Efficient algorithm design for distributed architectures is challenging due to the delayed and explicit communication between nodes.

Parallel processing for a rendering task is possible by subdivision of the image plane into multiple, independent tiles. A load-balancing algorithm must distribute tiles among cores and combine partial rendering results to complete the final image. An ideal load-balancing algorithm would generate exactly one tile per core, with all cores finishing their computations at the same time. Unfortunately, such an ideal load-balancing algorithm would assume that the execution time for a tile is known a priori and that communication in the system is instant. One must keep in mind that tile execution time depends both on the work load and a specific core's performance. Further, a tile cannot be sized to match a desired work load exactly as a single sample represent a finite piece of work that is indivisible. A lower bound on tile size is required to keep the ratio *acquisition computation / rendering computation* minimal in practice. Imbalance in work distribution and latency in communication lead to nodes and cores being idle, with idle phases increasing super-linearly with core count. Various aspects of this problem have been addressed in the literature as discussed in Chapter 2.6.4. All of he previously published approaches, however, fall below the curve of linear scalability from a certain number of nodes onward. Based on review of the literature and experimental analysis, the following shortcomings are likely responsible for the observed sub-linear scalability:

- **Synchronization**: synchronization points that exist between different nodes,

and, within a single node between different cores, can lead to significant accumulated delays, even for fine-grained imbalances.

- **Two-sided communication**: communication over the network requires active process involvement on the remote node. This requirement introduces a dependency between the program states of the communication partners, causing increasing delays with increasing number of nodes.
- **Centralization**: centralized operations between a single process and the entire cluster, for example, tile assignment, prohibit parallel execution by multiple nodes and may result in blocking.

Another source of inefficiency stems from the implementation of communication libraries that can drain a significant amount of system resources for processing and copying of message data.

Here, a new approach is proposed based on one-sided and asynchronous communication performed in a partitioned global address space (PGAS) via remote direct memory access (DMA) that can tolerate and absorb delays and latencies. This approach makes possible the design of a delay- and latency-resilient distribution framework for tile-based parallel rendering with the following properties:

- **Asynchronous behavior**: the framework operates fully asynchronously, almost all of the time, and it guarantees program correctness via weak synchronization, ensuring that delays occur only in exceptional cases.
- **One-sided design**: The new algorithms adhere to a one-sided design paradigm, where a node fully controls its own progress by communicating exclusively with one-sided operations that do not depend on program states of other nodes.
- **Decentralized approach**: the *tile conquest* strategy is introduced for enabling localized tile assignment that is independent of the number of nodes.

The framework is implemented using a light-weight zero-copy PGAS communication library as foundation, leading to negligible computational overhead.

5.1 Distributed System

In high-performance computing a distributed system typically consists of processors interconnected by a high-speed network, where a processor has access to local memory only and communication with remote processors is managed by a network interface controller. This arrangement allows distributed systems to scale, whereas shared memory system scalability is severely limited by memory access serialization. In fact, as the number of cores increases, memory hierarchies of processors are organized more like a distributed system (for example, recent AMD® Zen and Intel® Skylake-SP CPUs). In most cases memory is provided as a single shared address space but physically, memory is distributed leading to non-uniform memory access (NUMA) characteristics. Several processors exist that expose separate address spaces, for example, the Cell Broadband Engine [63], the Single-chip Cloud Computer [54] or the KiloCore [18]. As the number of cores continues to grow and monolithic chip size reaches physical limits, giving rise to novel multi-chip designs, distributed organization can be expected to become prevalent for many-processor systems and many-core architectures.

5.1.1 Abstract Partitioned Global Address Space Machine

For the purpose of broader applicability and generality, the subsequently introduced algorithms are formulated for a class of distributed systems, defined as the abstract PGAS machine, as illustrated in Figure 5.1. All system architectures that are compatible with this abstract PGAS machine are targeted by the algorithms.

The machine consists of compute nodes, each having several compute cores and a local memory segment directly accessible by all cores. A network connects the compute nodes to form a cluster. Further, each compute node contains a direct memory access (DMA) controller that can invoke data transfer between a node's own local memory segment and any remote memory segment, i.e., a remote node's memory segment. Hence, the union of all memory segments in the cluster form a PGAS. DMA operations can be initiated by any core using the local DMA controller while permitting asynchronous execution with respect to the initiating core or any other core. Memory access of a compute core to local memory is assumed to be significantly faster, in terms of bandwidth and latency, when compared to remote DMA data transfers. The inter-connection topology of the network is preferably simple, such as a grid or torus, exposing highly non-uniform memory access behavior with respect to different remote segments. Hence, The performance of the memory access depends on the distance, i.e., the number of hops, between the communicating nodes.

One of the compute nodes acts as designated display node, and it is connected to an output device, for example, a display or video stream. In addition, it manages input commands. The display node initiates the rendering of frames and composes partial results obtained from all other compute nodes into a complete image before sending it to the output device. Parallelization of the rendering process is achieved by subdividing the image plane into rectangular tiles and assigning the independent work packages among the cluster's nodes and cores.

5.1.2 Distribution Framework

Scheduling tile processing and movement of the associated data among compute nodes and cores in a cluster is performed by an algorithmic layer referred to as the *distribution framework*. It acts as the interface between a distribute memory system, i.e., the abstract PGAS machine, and a tiled renderer. The basic algorithm follows this procedure: A compute core acquires a tile, transfers the tile data to local memory, updates the tile data and transfers the result to the display node. The display node collects processed tiles and generates the composed, final image using the desired output format. The performance of a distribution framework can be characterized by two metrics, for example: (1) percentage of compute time spent on rendering, i.e., throughput, and (2) overall time between initiation and completion of a frame, i.e., latency.

An ideal distribution framework would employ all computational resources for the rendering problem with a perfectly balanced work load. As a consequence, communication would have to exhibit zero latency with zero computational overhead to allow instantaneous, dynamic balancing. In addition, tiles would have to be infinitesimally small or sized adaptively to organize an identical amount of computation per tile so that all cores finish a frame at the same time.

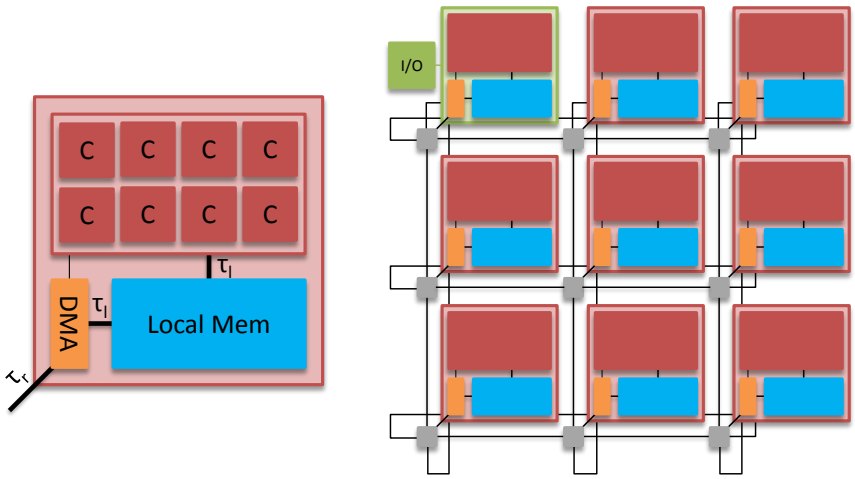

FIGURE 5.1: Distributed memory system with two levels of parallelism. Left: The internal layout of a compute node has three blocks, an execute block with several compute cores (C), a local memory block and a communication block (DMA). Compute cores and DMA have direct access to the local memory within a local shared address space with latency τ_l; only the DMA has direct access to remote memory with latency $\tau_r \gg \tau_l$, depending on the remote location and network congestion. Remote memory segments are accessible through a global partitioned address space (PGAS). Right: A cluster of nine compute nodes, with each node attached to a router (gray) and a torus inter-connection among routers. One designated compute node (green) acts as display node managing user input and image output (I/O).

Unfortunately, communication has a latency τ_r and tile assignment and data transfers between memory segments delays computation, especially for compute nodes having larger distances. Latency can also result from inefficient software design, potentially introducing synchronization points and dependencies on remote nodes. For example, a central entity responsible for distributing tiles to compute cores could become congested when attempting to feed a large number of recipients simultaneously, leaving resources idle.

While small tile sizes are beneficial for balanced workload across cores, tile assignment has a cost, in terms of computation and communication, which must be weighted against the rendering cost of the tile. Maximizing computational throughout in practice requires to maintain a minimum tile size.

Static load balancing avoids the limitations of the network by scheduling all tiles at the beginning of a frame. However, knowing rendering cost of a tile a priori is not possible. Frame-to-frame coherence can be used to estimate required computation times of images to be generated by considering the observed computation times of already computed images. However, generating computation time estimates and re-sizing tiles to equalize workload is expensive and not exact.

5.1.3 Experimental Realization

In the realization of the abstract PGAS machine for experimentation, compute nodes become multi-core shared memory processors interconnected by an InfiniBand network [56, 57]. Distributed memory parallelism is implemented according to the GASPI specification [44] using the GPI-2 library [60]. In compliance with the design of the abstract PGAS machine, GASPI views the distributed system memory as a partitioned global address space and provides one-sided DMA operations to transfer data between shared memory partitions. One-sided DMA communication is natively supported by the InfiniBand hardware. The cores of a processor are programmed using threads. The GASPI specification allows all threads of a processor to post DMA operations concurrently, which are executed asynchronously by the network adapter.

5.2 Tile Conquest

The tile conquest (TC) algorithm introduced in this section decentralizes load-balancing for distributed, tiled rendering. Scalability and data locality are the primary design objectives for this algorithm in order to utilize a large number of loosely coupled compute nodes for efficient parallel rendering.

The TC algorithm assigns locations in the image domain to compute nodes, with each node to process the image neighborhood of its assigned location. Given the compute density $\rho(\boldsymbol{x})$ of the rendering operation across the image domain, the load is perfectly balanced when

$$f_i \int_{A_i} \rho(\boldsymbol{x}) \, \mathrm{d}A = f_j \int_{A_j} \rho(\boldsymbol{x}) \, \mathrm{d}A \qquad i, j \in \mathbb{N}_0 \cap [0, N-1] \,, \tag{5.1}$$

where N is the number of nodes with associated image areas A_i. The factors f_i normalize differences in compute power between heterogeneous compute nodes; they are not needed for a homogeneous set-up. The variable $\rho(\boldsymbol{x})$ is likely to change between frames and, with data locality in mind, re-balancing is achieved by minimally shifting the borders between adjacent areas to keep Equation 5.1 satisfied.

In practice, the image domain is discretized by rectangular tiles representing indivisible work packages for the load-balancing process. It is not desirable to solve Equation 5.1 explicitly. Instead, the proposed implementation approximates the solution implicitly using prioritized work-stealing between neighboring compute nodes. The priority of a tile with respect to a compute node is defined by a distance metric. The Euclidean distance between node location and tile centroid is used. A "virtual network" connects each compute node with its closest neighbors for work-stealing operations, allowing a matching simple network topology (for example, a grid) to operate with the same efficiency compared to a more complex arrangement (for example, a fat tree). Initially, tiles are distributed among the compute nodes according to smallest distance. During rendering a compute node processes its local tiles in order of increasing distance. Once a compute node has processed its local tiles it "steals" from its direct neighbors, prioritizing neighbors according the number of remote tiles still to be processed. Tiles are stolen from the selected neighbor (the "victim") in order of increasing distance, relative to the compute node's position. A compute node transitions to the next frame when no more unprocessed tiles exist in

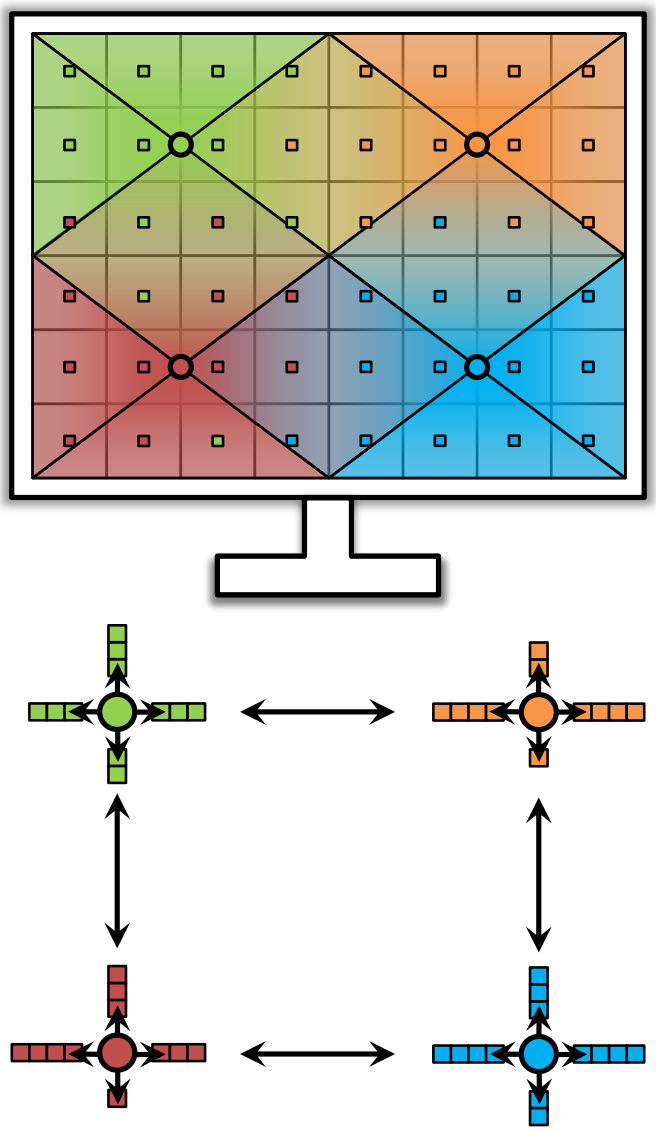

FIGURE 5.2: Tile conquest (TC) algorithm. Top: Image domain positions of four nodes shown as colored disks, with centers of tiles shown as squares. Color indicates to which node a tile currently belongs to. The black diagonals crossing a node separate the image into quadrants. Depending on the quadrant a tile falls into, it is assigned corresponding queues. Tiles are sorted with respect to node distance. Bottom: The queues for the four nodes are shown, with arrows indicating neighbor connections for work-stealing. A local node processes tiles in an "inside-out manner," while neighboring nodes remove tiles from the opposite direction.

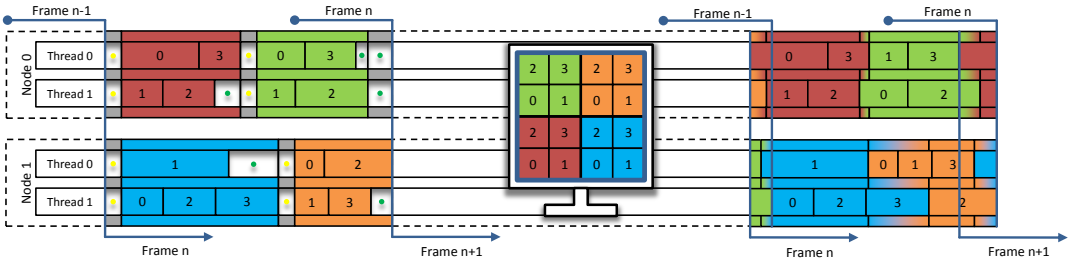

FIGURE 5.3: Benefit of the introduced fully asynchronous distribution approach. This example shows two compute nodes supporting four cores and the screen (middle) partitioned into four macro tiles, where each macro tile is subdivided into four micro tiles. Tiles are mapped to specific nodes and cores for a single frame n. The standard approach (left) produces idle times shown as empty spaces, being a consequence of latencies (yellow dots) and imbalances (green dots). The asynchronous system (right) completely eliminates idle times by processing overlapping macro tiles and seamlessly processing subsequent frames, leading to near-perfect scalability. One-sided communication and the tile conquest algorithm ensure that latencies and imbalances remain extremely small and fully hidden with increasing number of nodes.

its neighborhood.

In other words, a compute node with a low rendering load, relative to its neighborhood, conquers tiles from adjacent areas and increases its rendering "territory". If the balance shifts, the neighborhood node will reclaim the lost tiles. The dynamic re-sizing of rendering territories results in a self-balancing load distribution while keeping tiles localized around their corresponding compute nodes in the image domain. Tiles owned by a compute node tend to be clustered in both space and time, thereby improving the locality of data access patterns and reducing network traffic.

Restricting tile exchange to the local neighborhood promotes data locality and makes possible the use of simplified, specialized network architectures. In the proposed implementation tiles can only do one hop per frame, i.e., to directly adjacent neighbours. If there is a drastic change in workload in a compute node's territory between frames, there will be a possibility that the local neighborhood is insufficient in re-establishing a global balance instantaneously. Multiple iterations of frames might be necessary until all territories are properly resized. This situation reflects the limitation of the algorithm, its "theoretical failure", since the system looses its balance and leaves resources idle. However, under the typical assumption of frame-to-frame coherence and only a moderate workload change between frames, this failure mode is not reached and the system remains fully balanced. The amount of frame-to-frame coherence also affects the locality of data access.

Tile conquest has similarities to diffusion-based load balancing [53]. In fact, the new approach using nodes with an adaptive area of influence can be understood (in a dual sense) as being equivalent to an approach using tiles diffusing (along $-\nabla\rho(\boldsymbol{x})$) towards underutilized nodes. However, while tiles move in an unrestricted manner in the diffusion model, the TC algorithm keeps them localized, i.e., in a node's image domain neighborhood. Compared to previous work-stealing approaches [28], stealing is allowed to be done only between neighboring nodes instead of choosing

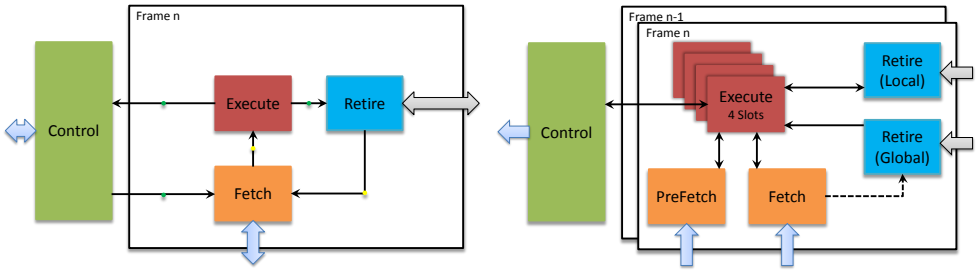

FIGURE 5.4: Distribution framework units and transitions for a single compute node. The bold arrows represent network communication, pointing towards the nodes (local and/or remote) actively involved. The color encodes the data flow direction, towards the local node (light blue) and towards a remote node (light gray). The standard approach (left) in addition indicates latencies (yellow dots) and imbalances (green dots) that lead to delays during transitions. The asynchronous system (right) avoids delays by quadrupling resources for the execute unit, segmenting fetch and retire units into two parts and again duplicating all units except control for frame overlap. All communication is one-sided only (in contrast to standard approach) and triggered by the local node, except for command messages to the control unit which are initiated by the display node.

a victim randomly. While this restriction is intrinsic to the proposed implementation of Equation 5.1, it also makes more informed steals possible, i.e., steals with a higher success rate, since the state of the neighbor nodes can be monitored easily. Finally, the implementation of the TC algorithm is different from previous load-balancing schemes as it relies solely on one-sided communication, discussed next.

The implementation of the tile conquest algorithm for the abstract PGAS machine is based on two central data structures: tile headers and tile header queues. The data structures reside in the partitioned global address space where they can be accessed freely by any node using one-sided atomic operations and asynchronous data transfers executed by the node's DMA controller. Tile headers contain the PGAS location of the tile data, i.e., the pixel data, and meta information about the tile, as needed. Each tile header is unique in the sense that only a single copy exists in one local memory segment, making the corresponding node the unambiguous owner of the tile. Tile headers are located in one of four sorted tile header queues per node, depending on their centroids, see Figure 5.2. The queues extend in the intrinsic coordinate directions (to the left, to the right, upwards and downwards) relative to a node's location, separating space into quadrants. Access to a queue is controlled by an atomic variable having two fields for start and end indices defining the range of valid tile headers. To obtain a tile header, a local node increases the start index, acquiring tiles closest to its location, and a remote neighbor node decreases the end index, acquiring tiles most distant from the local node's location. Thief nodes prioritize queues depending on their relative positions to a victim node in order to receive tiles close to their own locations.

Once a node has completed the processing of a frame $n-1$, it broadcasts the number of available tiles to its neighbors, thereby enabling steals for frame n. The node keeps processing its local tiles, taking headers from the local header queues for frame n for rendering, and inserts the finished headers into the local header queues for frame $n+1$, using an opportunistic insertion-sort method.

As soon as a node's local tiles are depleted (and possibly sooner though this is not implemented), the node starts stealing; it steals from its neighborhood, using information from previous steals and regular update notifications from its neighbors to select a victim and a corresponding queue. If the steal succeeds, i.e., the selected queue has remaining tiles, then the header data will be transferred, followed by the tile data. If the steal does not succeed, the procedure will be repeated until no more tiles are available in the neighborhood and the node transitions to frame $n + 1$. Finished remotely obtained headers are modified to reflect the new memory location of the tile data, and are inserted into the appropriate local queue.

5.3 Asynchronous Tile Processing

With tile conquest, the previous section has introduced an algorithm for tile assignment among compute nodes of the abstract PGAS machine. This section is concerned with the details of asynchronous transfer of tiles and their asynchronous processing on nodes with multiple compute cores. Instead of TQ, other tile assignment strategies could be plugged into the distribution framework described here to benefit from its asynchronous properties.

Since acquisition of a tile over the network is expensive, mainly due to τ_r (Figure 5.1), the two-level approach from Ize et al. [61] is adopted by clustering several micro tiles to form macro tiles. Nodes exchange tiles on macro tile granularity, while the corresponding micro tiles are distributed among the compute cores using significantly less expensive work-stealing through shared memory with latency τ_l. Figure 5.3 illustrates the fundamental idea of an asynchronous distribution framework for the two-level tile approach. Idle gaps in the execution pipelines are filled by enabling overlapped execution of multiple micro tiles, macro tiles and frames.

Figure 5.4 (left) shows a basic distribution framework in terms of functional units and transitions between units on a compute node. The terminology adopted in the following is reminiscent of CPU micro architecture to highlight structural similarities. The control unit receives commands from the display node and coordinates the compute cores accordingly. In case of a render command, a transition to the fetch unit occurs where a macro tile is acquired and made available to the execution unit. Within the execution unit, the micro tiles of the current macro tile are rendered by multiple cores in parallel and, upon macro tile completion, the result is processed in the retire unit and sent to the display node, followed by a new fetch/execute/retire cycle. The following causes lead to the observed gaps in Figure 5.3 (left):

- Remote latency in the fetch unit
- Imbalance among cores in the execution unit
- Data transfer latency in the retire unit
- Imbalance among nodes and cores at the end of a frame
- False dependencies between nodes

An asynchronous framework design allows to hide latencies, absorb imbalances and remove dependencies. The listed issues have been partially addressed in previous work but never in a unified and complete way. Previously published results deviate

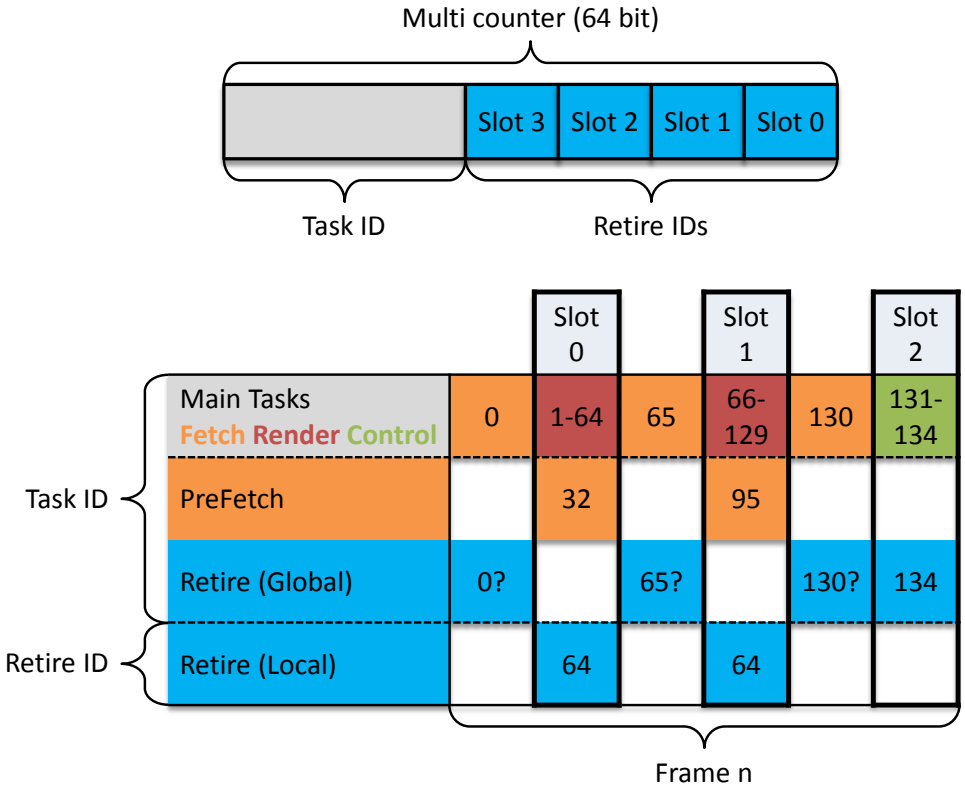

FIGURE 5.5: Compute node task system. Task ID and retire ID define the unit(s) that a core executes. Both IDs are obtained with a single atomic increment of a shared multi-counter. The task ID counter is reset at the start of a frame, while the slot-specific retire ID counters are reset after retirement of the corresponding macro tile. The main tasks are (1) fetching a new macro tile; (2) rendering a micro tile; or (3) returning to the control unit. Additional tasks are PreFetch or Retire (Global). The retire ID triggers Retire (Local) for a slot once equal to the number of corresponding micro tiles (all micro tiles done). The exemplary numbers show frame n with two macro tiles (64 micro tiles each) and four cores, matching the control tasks. The Retire (Global) is optional ('?') except for the final task 134, completing the frame.

from ideal scalability, even for small numbers of nodes.

Here, the the basic framework is extended by increasing the capacity of its data structures to hold multiple tiles/frames in flight and splitting the fetch and retire units at latency-prone points to allow temporal separation of execution, as shown in Figure 5.4 on the the right. In addition, false dependencies are avoided by relying on one-sided network operations exclusively. The resulting behavior is illustrated in Figure 5.3 on the the right. The motivation for this design is provided in the following sections by discussing the details of the individual units. Coordination among cores is performed by the task system, illustrated in Figure 5.5. Particular tasks are "execute tasks" and "execute sub-tasks" that are related to and used synonymous with macro tiles and micro tiles, respectively.

Control Unit

The control unit coordinates the cores according to the commands received from the display node. In particular, it initializes new frames and verifies the completion of earlier frames. The first core to enter the control unit after completion of its local frame in the execution unit is assigned the master role – see Figure 5.5 (frame n+1, core drawing task 131); subsequent cores (tasks 132–134) may need to stall on frame initialization waiting for the master. The master in turn may need to wait for completion of frame n-1 before proceeding to frame n+1 initialization, since the allocated resources suffice for two frames in the pipeline only, see Figure 5.4.

Pre-fetch Unit

The macro tile distribution logic, including tile conquest, is part of the pre-fetch unit. The tile header for the next macro tile is determined either from the local queues or via a remote steal operation and, in the second case, the DMA transfer of the tile header is initiated. The pre-fetch task aliases with an execution sub-task determined by the pre-fetch distance, as measured via the last sub-task number of the current macro tile. In Figure 5.5, the pre-fetch distance for tasks 32 and 95 is $64 - 32 = 32$ and $129 - 95 = 34$, respectively, as the pre-fetch distance may be adjusted dynamically on a macro tile basis. A core for which the pre-fetch task is triggered first transitions to the pre-fetch unit and, after completion, back to the execution unit to process the corresponding execution sub-task.

Fetch Unit

The macro tile header delivered by the pre-fetch unit is processed by the fetch unit to set up a new execution task in one of the four execution slots, see Figure 5.4. If all slots are occupied, the unit has to stall until one becomes available. If the tile header originates from a local queue, the render target is set to the immediately available tile data buffer for accumulation. Otherwise, the unit performs the following actions: (1) It stalls if the tile header transfer is still in progress; (2) it initiates the DMA transfer of the tile data from the source PGAS location indicated by the tile header; (3) it sets the render target for the macro tile to an intermediate buffer that is later combined with the tile data in the retire units. Once the execution task is ready, the task counter (determining the number of available tasks) is increased according to the number of micro tiles, i.e., execution sub-tasks, plus one to account for the next fetch task. In Figure 5.5, fetch task 0 increases the task counter to 65, and fetch task 65 increases the task counter to 130 etc. Finally, a transition occurs either to the retire (global) unit, if vacant, or back to the execution unit, otherwise, see Figure 5.4.

Execution Unit

The execution unit can hold up to four macro tiles in execution slots. Each micro tile spawns a corresponding execution sub-task, and each of the sub-tasks may be processed by any of the available cores in parallel. Even though a new execution slot is filled only when all existing sub-tasks have been acquired, it is necessary to use multiple slots since acquired sub-tasks complete out-of-order, depending on work load. When a core has finished an execution sub-task and requests a new task, it increments the task ID and the corresponding slot's retire ID of the multi-counter, see Figure 5.5. This approach makes it possible to track the completion of the macro tile. The core to finish the last micro tile transitions to the retire (local) unit. While the

retirement of finished macro tiles is fully asynchronous, it is still possible for stalls to occur in the execution unit in case the fetch unit cannot deliver new macro tiles in time.

Retire (Local)

In the retire (local) unit, a completed macro tile is detached from its execution slot; the corresponding tile header is moved to the retirement buffer. If the macro tile has local origin, the accumulated result of the rendering is available and the DMA transfer of pixel data to the display node is initiated.

Retire (Global)

The retire (global) unit clears macro tile headers from the retirement buffer and places them in the tile header queues for the next frame $n + 1$. Clearing involves these steps: (1) for tiles with remote origin, accumulation of rendering results from the intermediate buffer with the remote tile data when available, and initiation of the DMA transfer of pixel data to the display node; (2) checking display node DMA transfers for completeness. Once a tile header is cleared, it is insertion-sorted into the appropriate tile header queue in frame $n + 1$. Since tiles may already have been taken from the tile header queue, via a local pre-fetch in frame $n + 1$, perfect sorting is not guaranteed (but not required either). The retire (global) unit is non-blocking. A core will be able to transition from the fetch unit if the retire (global) unit is not occupied by another core from a previous fetch, for example. This relaxed transition is indicated by the question mark shown in Figure 5.5. However, the core that draws the last control task – task 134 in Figure 5.5 – will stall and repeatedly enter the retire (global) unit until the retire buffer is depleted, indicating the completion of frame n within the node's scope.

5.4 Results

The new algorithms are evaluated on a 60-node cluster, each node being an Intel® Xeon™ E5-2680v2 dual socket machine interconnected by QDR InfiniBand. An additional Intel® Xeon™ E5-2695v3 dual socket machine serves as display node, with a monitor and input devices directly attached. The rendering module performs single ray traversal on a 4-wide bounding volume hierarchy (as detailed in Chapter 3) to compute 8-bounce diffuse path tracing with one sample per pixel. The display resolution is 960×540 pixel, and macro and micro tile sizes are 32×32 and 4×4 pixels, respectively. This rather small resolution (a quarter of the common HD resolution) has been chosen to put an emphasis on strong scaling behavior. The results are obtained from prerecorded virtual camera flights through three benchmark scenes, namely the SAN MIGUEL, POWERPLANT and BOEING scenes (Figures 3.5, 3.8 and 3.9). In addition to a TQ evaluation, the distribution framework is tested using a centralized macro tile assignment strategy, done similarly in other approaches like the ones discussed in [61, 110], based on work-stealing from the display node (MA). Due to the asynchronous, one-sided optimization the MA implementation has better scaling behavior compared to previously published results. Two variants of MA, one where all nodes are synchronized between frames (MA-F) and one where all cores of a node are synchronized between macro tiles (MA-T), artificially prohibit frame and tile overlaps in the rendering pipeline to demonstrate the corresponding

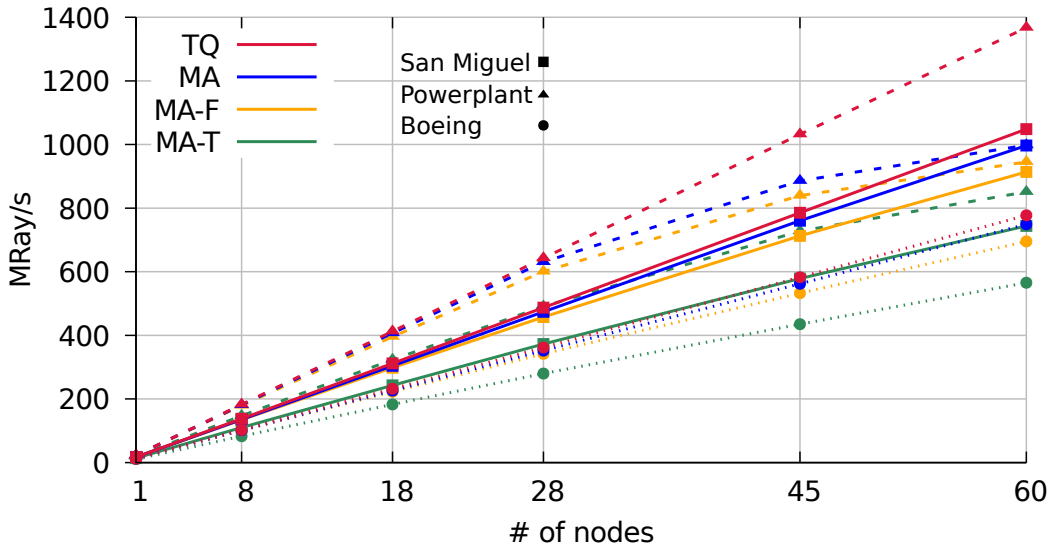

FIGURE 5.6: Rendering throughput in million-rays per second for distribution frameworks TQ, MA, MA with frame synchronization (MA-F) and MA with tile synchronization (MA-T).

impact on performance.

Since the test cluster is configured for normal batch execution, various management and monitoring processes compete for CPU time with the rendering application. Suspension of any rendering thread can stall the entire cluster when it remains inactive for a time longer than twice the time required for a frame, which can be on the order of a few milliseconds. In order to minimize the probability for such an event to occur we do not use hyper-threads, thereby trading a significant increase in throughput for reduced volatility of the results.

Figure 5.6 provides plots of the measured throughput in million-rays per second (MRay/s). For all three scenes TQ is the most efficient algorithm, with an advantage of 3.7%-13.7% over the second best at 60 nodes. MA and MA-F perform well but increasingly deviate from TQ performance with increasing node count. The large gap for the POWERPLANT scene is a consequence of network saturation as discussed below. The benefit of overlapping tile execution becomes evident when considering MA-T performance as it trails the remaining results by a significant margin, reaching only 62.2% to 72.8% of TQ throughput.

Figure 5.7 provides insight into scalability for the SAN MIGUEL scene. Since POWERPLANT and BOEING results are quite similar they are omitted in the graph. Instead, results are added for another distribution framework for real-time ray tracing (TAMM) [98]. TAMM computes a static load distribution for a frame to be generated next using predictions based on statistics of previously rendered frames, combined with frame overlap. Similarly to TQ, this strategy avoids centralized communication – in this case any communication – during rendering. Different from the results measured here, Whitted [115] ray tracing and not path tracing has been used as the rendering method, which allows more accurate predictions by a static load balancer.

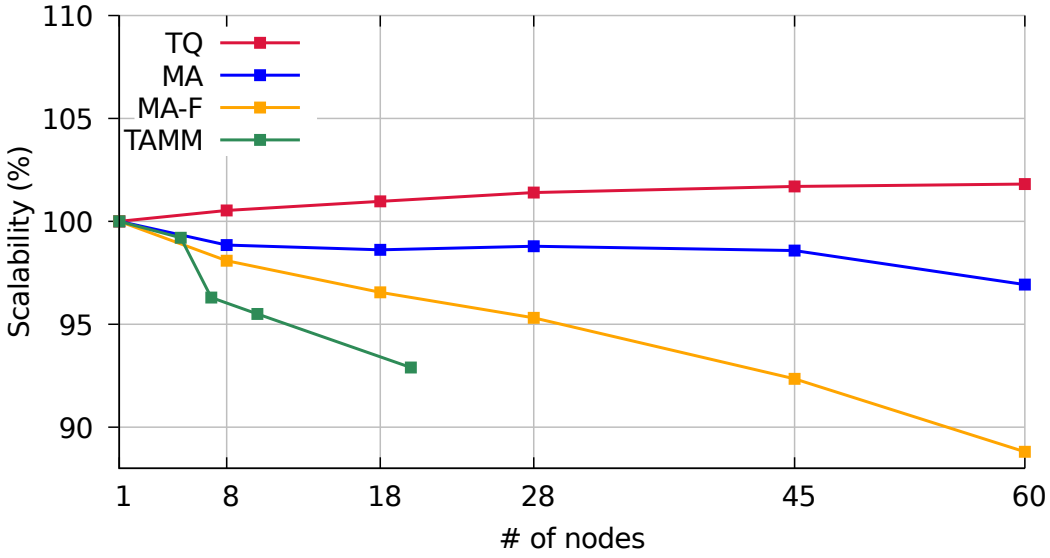

FIGURE 5.7: Scalability measured as throughput per node relative to a single-node set-up for San Miguel scene. TAMM cites results from [98]. Linear scaling corresponds to 100%.

Compared to its competitors, TQ's scaling behavior is qualitatively different. While the throughput per node decreases for MA, MA-F and TAMM with increasing node count, the TQ strategy allows nodes to gain additional throughput as their number grows, resulting in super-linear scalability for TQ, which is explained by increasing cache efficiency. Adding more nodes effectively shrinks the working set of an individual node, and TQ's temporal stability of load balancing make possible the reuse of cached data between frames. This effect is significant despite the complex and poorly localized memory access patterns intrinsic to path tracing. TAMM also reports super-linear scaling behavior for its ray tracing core. However, in contrast to TQ, TAMM's overall rendering process remains well below linear scalability and trails the baseline MA and MA-F algorithms.

TABLE 5.1: TQ rendering scalability for a sphere entirely contained in cache. (Corrections applied to original result to account for additional computation due to tile exchange)

Node Number	8	18	28	45	60
Performance[%]	99.2	99.3	99	98.8	98.9
With Correction[%]	99.9	99.8	99.9	99.9	99.8

In order to isolate the caching contribution from the fundamental scalability of TQ, the following synthetic benchmark scenario is used: a triangulated sphere, with a memory footprint within per-core L2 cache size, is rendered from within. Linear scalability behavior can be expected for this uniformly demanding rendering task but no super-linear cache effects. Linear scalability is confirmed by the results provided in Table 5.1. The small "corrections" applied to the measured values account for the additional computations required for exchanging ∼1 tile/node/frame when

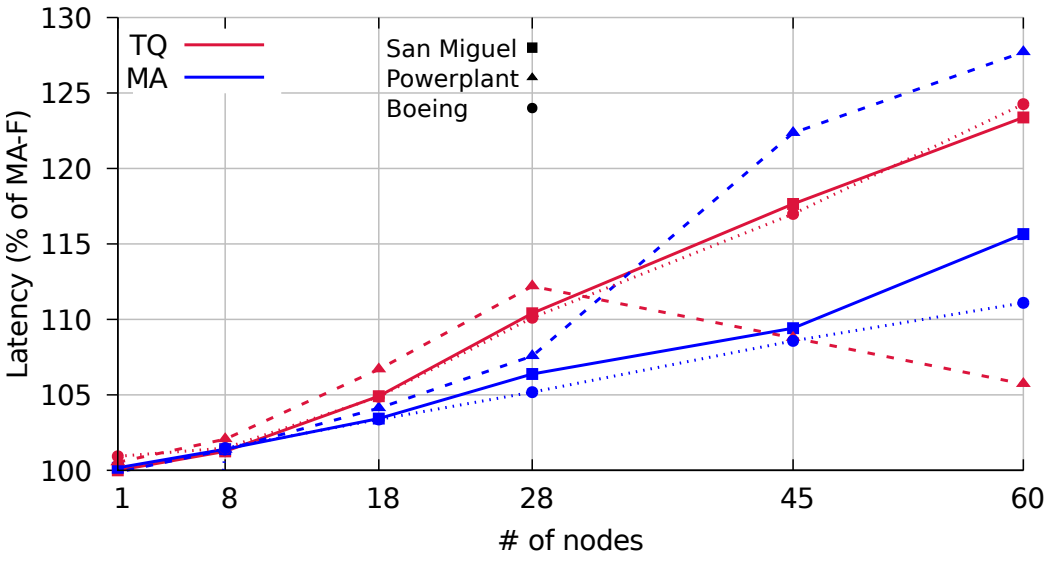

FIGURE 5.8: Latency from initial rendering command to display output for TQ and MA (frame overlap) relative to MA-F (no frame overlap).

stepping from one to multiple nodes.

Figure 5.8 shows overall latency of a frame, i.e., the time needed from an initial rendering command until the completion of a frame in the display node's memory. This time is different from the inverse frame rate due to frame overlap, which allows for two incomplete frames to be present in the rendering pipeline. The results are normalized by MA-F latency which has frame overlap disabled. The relative latency of TQ increases linearly with increasing node number, indicating a higher degree of frame overlap. MA's increase is also linear but exhibits a smaller slope. TQ utilizes overlapping more aggressively to reach maximum throughput. The results for the POWERPLANT scene are somewhat different, where both MA and MA-F are network-limited for more than 28 nodes – while TQ continues to scale.

Network limitation becomes evident in Figure 5.9, showing the consumed bandwidth of the algorithms in gigabyte per second (GB/s) for the POWERPLANT scene. MA reaches the maximum display node bandwidth of 3.4GB/s at 60 nodes. Scalability already starts to degrade at 18 nodes and above: not only the bandwidth but also the message rate approaches the limits of the display node network link. The resulting network congestion introduces transfer delays that propagate into the rendering pipeline and lead to idling cores that wait, for example, for the next tile header to arrive. The bandwidth consumed by TQ is significantly reduced compared to MA and better distributed due to the direct neighbor communication.

Figure 5.10 summarizes overall idle time of the cores during a benchmark run in parts per million of total execution time. Idle times occur due to unresolved dependencies, i.e., data from the network or progress of other cores. Idle times increase linearly with increasing node count in most cases, shown as logarithmic curves. Deviations are a result of network congestion since data latency can no longer be fully

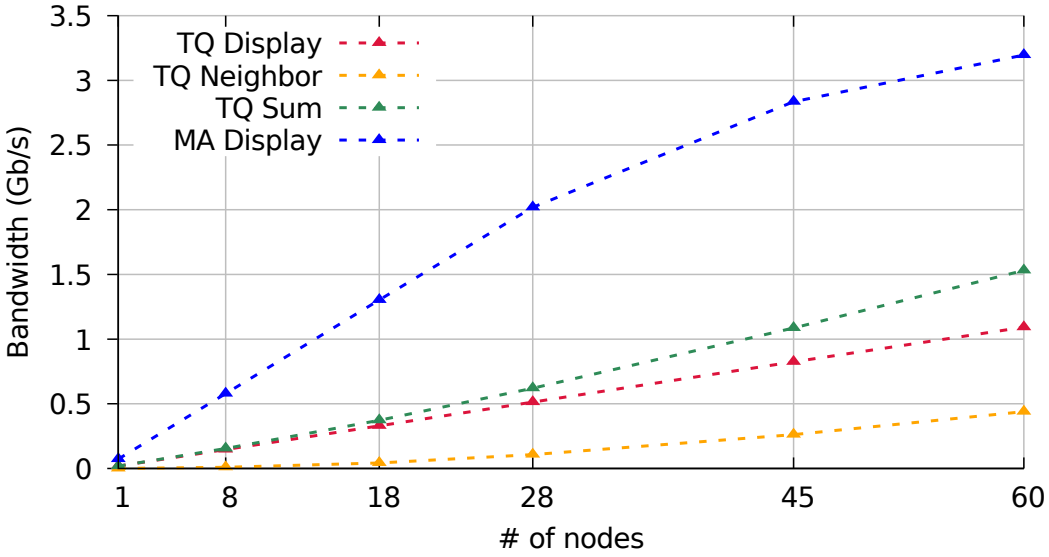

FIGURE 5.9: Bandwidth (gigabyte per second, GB/s) consumed by TQ and MA for communication between display and neighbor nodes.

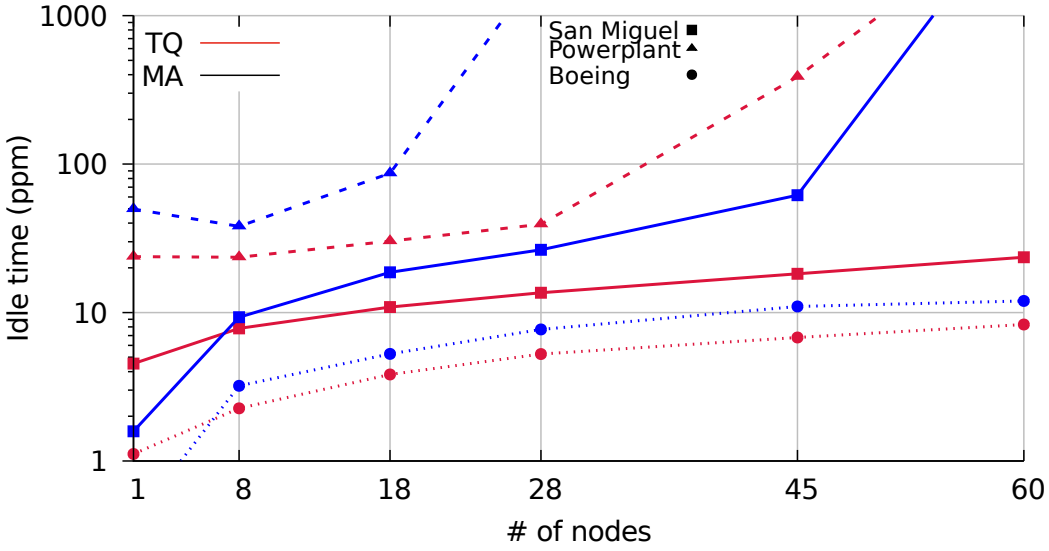

FIGURE 5.10: Accumulated idle time in parts per million relative to total run time for TQ and MA.

hidden by the distribution framework. TQ idle time is significantly below the corresponding MA times and exhibits a smaller slope. For the MA framework and POWERPLANT scene, idle times rise considerably, when going from 18 to 28 nodes, and even TQ is affected slightly for 28 nodes and above due to message rate congestion.

The results document that tile conquest can optimally scale rendering throughput for large distributed systems with many nodes. Super-linear behavior emerges when data sets no longer fit into a single node's cache. Network traffic is reduced considerably and better distributed among network resources, allowing TQ to continue to scale while performances of competing distribution frameworks saturate. The negative impact on frame latency is exhibited by all methods making use of frame overlap. Mitigation is possible by allowing tiles to adapt in size for a homogeneous, uniform work distribution, which is an important avenue for future research.

TQ draws its strength from frame coherence and localized communication; however, this can become a disadvantage for drastic changes in rendering load. Prompt and precise adaption to rendering load could be accomplished, for example, through dynamic association of nodes with screen-space. As a consequence, the currently static grid would become a dynamically deforming grid based on screen-space load distribution. This aspect is another possible future work direction, complementing an adaptive tile strategy. Another remaining challenge concerns dynamic, distributed scene management on distributed systems that could potentially benefit from an approach similar to tile conquest.

5.5 Conclusion

With a focus on scalability and real-time application this chapter has introduced a distribution framework for parallel tile-based rendering. It is designed to optimally adapt to the constrains of a large distributed system with a partitioned global address space. The scalability of the new approach more closely reaches optimality compared to previously published results and, for the first time, demonstrates super-linear behavior for the entire rendering process, observed consistently for a large ranges of nodes in a path-tracing experiment.

The strengths of the framework are based on two main contributions: (1) the focus on one-sided and asynchronous communication strategies and (2) the tile conquest algorithm for inexpensive and data locality-preserving load-balancing. As a consequence of an aggregate cache effect, the efficiency of one node increases with increasing number of nodes in the cluster. With the unprecedented scaling behaviour exhibited by the new distribution framework significantly higher performance and quality levels can be achieved simply by increasing the cluster size.

By refining the approach further, based on the ideas expressed in the Results section, it should be possible to increase algorithmic efficiency and prevent performance degradation in edge cases.

Chapter 6

Application and Integration

In this final chapter, a real-time global illumination application is presented that is built on top of the ray tracing kernels researched and developed throughout this dissertation. The application is intended as an example to demonstrate one of the many possible usage scenarios of the ray tracing kernels, their scalability and their integration.

The utility of the application is the presentation and editing of large CAD scenes based on triangular meshes. According to the intended work flow, a scene is loaded from file and explored interactively in photo-realistic quality. The look of the scene is adjusted as desired by modifying lighting and material and removing erroneous or superfluous geometry. Finally, after saving the scene as configured back to a file, the result can be presented interactively or images from different views can be exported to be used in a presentation, for example. This work flow has been applied to prepare some of the benchmark scenes employed for the experimental results of the dissertation (R8, BOEING).

Figure 6.1 provides a structured view of a general rendering application in the form of three main modules. The *logic module* defines the functionality of the application, i.e., what can be done with it. For example, a game would implement its set of rules and goals, the exemplary application would implement everything needed to support the pre-described work flow. The part of the logic intended for user control is exposed through the *user interface*. The *rendering module* produces images as instructed by the application logic and feeds the result back to the user interface.

The illustration of the rendering module in Figure 6.1 is further structured into sub-modules. The *GI* sub-module contains the rendering or light transport algorithm solving the rendering equation to the desired degree, which is assumed to include global illumination in this context. The *scene* sub-module contains the scene descriptions such as geometry, lights and materials. Light-material interaction is defined by the *shaders*. Resulting from the research in this dissertation is the essential *kernel* [71] sub-module, which provides parallel ray tracing and load balancing to the GI algorithm.

In the following, the different modules are explained in detail along with the exemplary application, followed by a look at how the kernel enables the application to scale over several devices and a final conclusion.

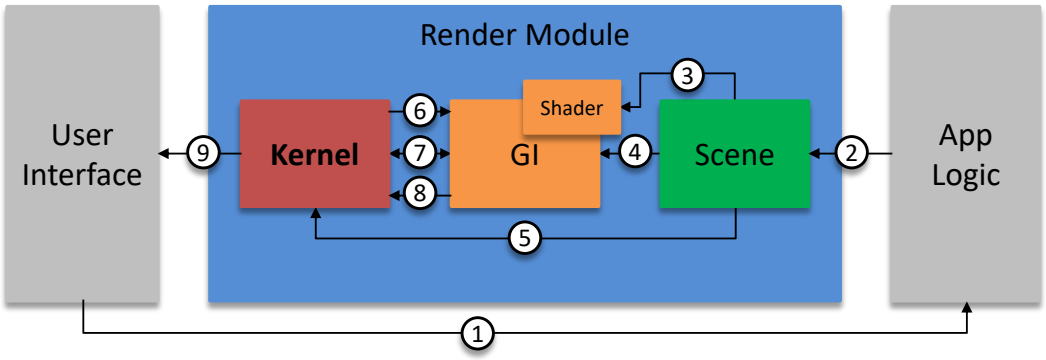

FIGURE 6.1: Structure of the exemplary application represented by three main modules (user interface, render module, application logic) and three sub-modules (Kernel, GI, Scene). The shaders are part of the GI sub-module but specific materials are defined by the parameters provided by the scene module. Interaction with the user interface triggers the application logic (1), which translates the the user interaction, for example, into a change of camera position or geometry. This change is applied to the scene (2), which initiates a new frame to be rendered (4) and optionally communicates changes to geometry and lights or materials to the Kernel (5) and shaders (3), respectively, in order to update the internal data structures accordingly. During rendering, the GI implementation repeatedly requests new tiles (6), traces the corresponding rays (7) and commits the finished tile (8). The load balancing and ray queries are managed transparently by the kernel, which makes intermediate and final image results available to the user interface (9).

6.1 Application Logic and User Interface

This section describes the exemplary application in terms of usability and functionality. The application logic is supposed to enable the work flow as sketched in the introduction, exposing the corresponding controls via the the user interface, captured in Figure 6.2.

The viewport window shows the rendered scene, which can be navigated by rotating, translating and zooming the virtual camera in real time. The panel on the right side shows one of three different tabs. The *Main* tab is sectioned into three blocks, the control block providing loading, importing and saving of scene data as well as exporting of the current rendered frame as an image and choosing the particular GI implementation (*pt*, for example). Also, the settings and HDR block allow to adjust the rendering quality parameters, which are discussed later.

The image presented in the main window in Figure 6.2 appears slightly noisy, because the rendering of the image is ongoing, as indicated by the progress bar at the bottom of the panel. Progressive refinement of the image allows to navigate quickly even on slower systems, while the full image quality materializes within a few moments after the camera stopped moving.

Important for the work flow is the *Material* tab, shown on the right side of Figure 6.2. Here, shaders (*Phong*, for example) and material parameters, such as diffuse color or texture (*Diffuse*), specular color or texture (*Specular*), glossiness (*Glossy*), weighing

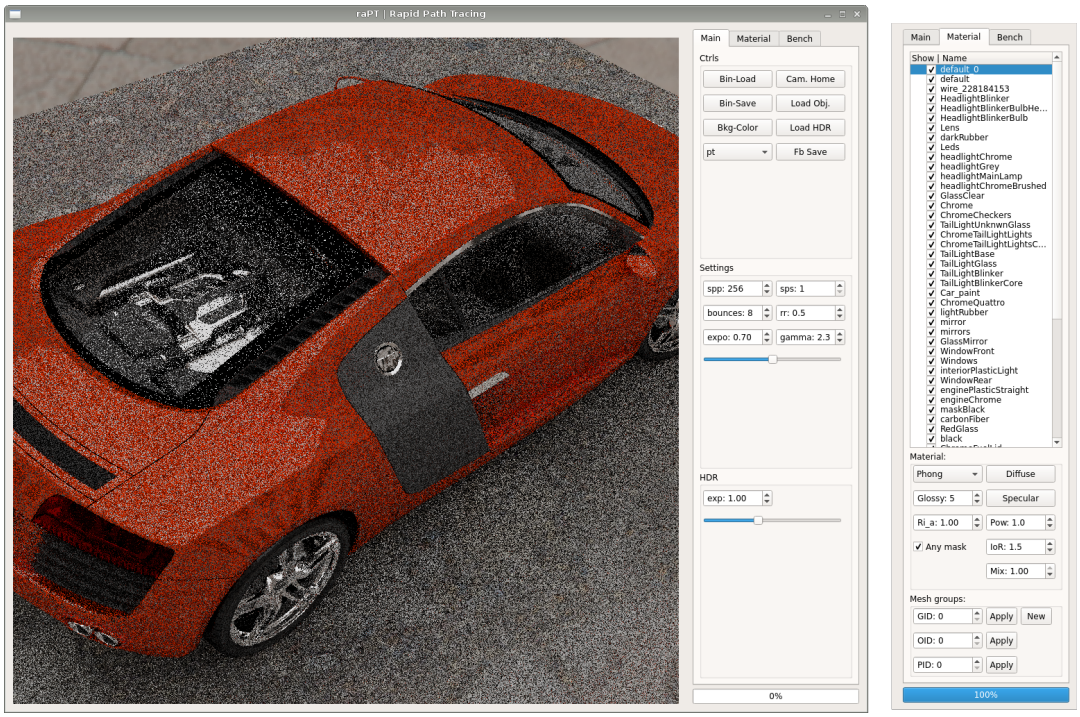

FIGURE 6.2: User Interface of the exemplary application. The camera can be zoomed, rotated and translated using the mouse. The progress of the image refinement is indicated by the progress bar, which is at 0%, meaning that so far only a single sample per pixel has been computed. The *Main* and *Material* tabs allow to change the render settings and material parameters, respectively, as described in the text.

between diffuse and specular contribution (*Mix*) and the index of refraction (*IoR*) can be adjusted. New materials are created by pressing *New* and selected materials are assigned based on mesh IDs (*OID*,*GID*) or primitive ID (*PID*) using the respective *Apply* button. A primitive and its corresponding materials and IDs can be selected by picking it directly from the main window.

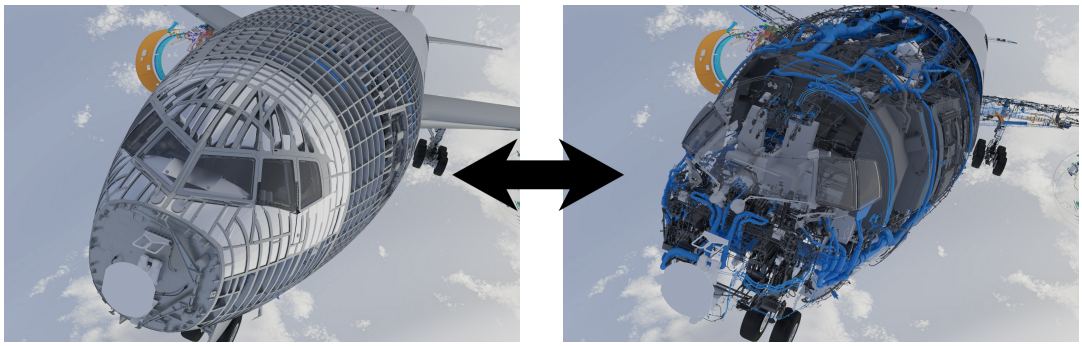

FIGURE 6.3: Altering visibility based on material, mesh or primitive granularity interactively.

An important feature is the visibility option for each individual material in order to support fast exploration of a scene. Removing the tick from the check box next to the material name in the material list instantly fades out all corresponding geometry. This is demonstrated in Figure 6.3 with the BOEING model consisting of 300 Million

FIGURE 6.4: Using dynamic masks for delete. Removed geometry (red) can be restored at any time.

triangles: the outer hull and framing are removed, revealing pipes, wiring and the interior of the cockpit.

Often, CAD data contains erroneous or superfluous geometry introduces during automated generation or processing of the data. Such geometry can be removed from a scene by assigning it a special "delete" material which is invisible by default. Geometry deleted by mistake may be recovered by changing the visibility of the delete material and re-assigning the original material to the geometry parts in question. Figure 6.4 shows in red the material deleted from the original BOEING CAD data in order to obtain a "camera-ready" version.

Usability

The usability of the exemplary application critically depends on the speed of the rendering process. Noticeable lag between input and visual response makes navigation in the scene imprecise and tiresome, thus reducing productivity.

Still, it is desirable to have the application function reasonably well on moderately powerful systems such as a laptop, not only on scarcely available clusters. This is made possible by using iterative refinement of the rendered image: the display is continuously updated as the rendering progresses, showing a quite noisy representation of the scene in the beginning which quickly improves in quality with each additional iteration. The advantage is that the application remains responsive even though the system is only capable of rendering just a fraction of the full image in real-time. If the camera moves, the refinement of the current frame stops and the next frame starts from scratch. The ease of navigation remains mostly intact despite the reduced quality intermediate images.

Table 6.1 lists a qualitative evaluation of working with the R8 scene on three different platforms: a laptop, a workstation and a cluster. There is a significant discrepancy in computational complexity between an inside view and an outside view of the R8 because inside the car ray paths become much longer on average and thus are more expensive.

TABLE 6.1: Qualitative evaluation of working with the exemplary application and the R8 scene on different devices. Depending on the device, a different combination of traversal algorithms are active, as noted. *Navigation* is defined in terms of smoothness, i.e., if there is a noticeable lag between mouse movement and the visual response of the initial rendering iteration. *Final image* indicates the time frame required to complete all of the 256 refinement iterations. A distinction is made between *inside* and *outside* views of the car.

Device	Laptop	Workstation	Cluster
Microarchitecture	Haswell	Knights Landing	Sandy Bridge
Cores/Threads	2/4	68/272	1200/1200
Traversal	CLPT/ORST	WIVEC/WIVE	CLPT/ORST
Navigation			
Outside	Slight lag	Smooth	Very smooth
Inside	Extreme lag	Slight lag	Very smooth
Final image			
Outside	Few minutes	Few seconds	Below second
Inside	Several minutes	Several seconds	Few seconds

Navigation is rated by how fast the application responds with a single iteration image during camera movement (see Figure 6.2). The cluster has no difficulty with either inside or outside views and always delivers a smooth experience. The workstation also allows smooth navigation outside the car but produces small but noticeable lag between input and output from inside. Even with the laptop working with the scene from outside is mostly pleasant, however from inside extreme lag makes navigation very jerky and tiresome.

After the camera stops moving, the refinement of the image continues for 256 iterations after which visual noise has mostly disappeared. The cluster finishes an outside view usually below one second while an inside view take a few additional seconds. The workstation also stays within the "seconds regime" but tends to get close to a minute for an inside view. The laptop requires quite a bit more time which limits its usability. For productive work sessions at least the performance of a workstation is required.

6.2 Rendering Module

The rendering module delivers rendered images to the application for presentation in the viewport window. It also provides the load balancing required for multiple threads and nodes so that different devices are transparently supported.

6.2.1 Scene

The scene sub-module contains all the data of the scene description, such as geometry, materials, light sources and cameras.

The geometry is represented by an indexed triangle mesh so that a single vertex can be shared by multiple triangles. The properties of a vertex include position, surface normal and texture coordinates. Triangles contain three indices to vertices defining the triangular face in a counterclockwise orientation and three IDs: material ID, mesh ID and object ID. Mesh and object IDs allow grouping of triangles into aggregate surfaces and objects for interacting with the scene.

The material ID is a reference to a material definition which is composed of a fixed-function shader type and corresponding parameters and texture maps. The shader types implemented in the exemplary application's global illumination algorithm include *phong* for diffuse, glossy and perfectly reflective materials, *glass* for transparent materials [93] and *light* for emissive materials [95]. An environment light can be defined for the scene in addition to emissive surfaces. This can either be a constant color or an environment map, for example, a HDR photograph [81, 27].

The scene sub-module interacts with both the GI and kernel sub-modules, providing surface properties and light sources for path generation as well as scene geometry for BVH construction, respectively.

6.2.2 Global Illumination

The global illumination algorithm implemented in the exemplary application is unidirectional path tracing. Paths start at the camera origin and are subsequently extended at surface intersection points. Depending on the surface material paths are either reflected or transmitted. If both events are possible either one is selected by a random draw with weights determined by a proportionality constant (Figure 6.2, *Mix*). For glossy and diffuse materials the outgoing path direction is importance-sampled according to the material's BSDF.

In addition to the material sample a light sample is taken into the direction of a randomly chosen area light source [95] or into the direction obtained from importance-sampling the environment map [81, 27]. The light sample cannot be further extended: if a clear line-of-sight between light source and surface point exists the light contribution counts, otherwise the light sample is dismissed. The material sample and the light sample are combined using multiple importance sampling [103, 101].

The path is terminated either if a light source is hit directly or randomly by "Russian roulette". Russian roulette is an unbiased technique to reduce the average length of a path. After each sample, a random number is drawn and compared to the path throughput or a parameter (Figure 6.2, *rr*): if the number is above the threshold the

path is terminated [7]. An absolute limit on the number of extensions (Figure 6.2, *bounces*) ensures that the path length cannot grow indefinitely if Russian roulette fails to end the path.

Ray Management

Taking advantage of ray coherence necessitates a strategy for ray management since multiple rays must be processed in parallel, which, in turn, demands a separation of the rendering pipeline into distinct stages. Each phase consecutively operates on the same ray stream and stores intermediate results in buffers.

In contrast, for single ray tracing, initiating and obtaining the result of an operation is sequential. For example, in order to compute the color contribution of a surface point within the corresponding material shader, a light sample can be generated and immediately evaluated, subsequently followed by the completion of the material shader using the obtained lighting result. For a ray stream approach, the material shader must be split into two stages and the light sample traced in between a third stage in order to process multiple rays together.

The exemplary application is designed for ray stream processing from the ground up to support traversal algorithms that exploit ray coherence. For single ray traversal algorithms the stream approach is maintained by traversing each ray in the stream sequentially. This approach can be compared to the concurrent work of [2]. The main difference is in the data organization, which uses the structure-of-array format instead. While their work focuses on coherent shading where the structure-of-array format is optimal, this dissertation focuses on coherent traversal where the array-of-structure format simplifies loading and storing complete stream elements, i.e., rays. Coherent shading is not supported by the exemplary application, even though shaders are implemented using vector instructions on a per sample granularity. A recent approach for exploiting coherence throughout the entire rendering pipeline in a production path tracer is presented by [77].

In practise, for each tile, the exemplary application performs a coherent packet traversal for the camera rays initially. For every camera ray with a valid surface intersection the shading stage generates a fixed number of extension and light rays and puts them into two separate streams, respectively. This arrangement supports a technique called *splitting* [7] if the fixed number is larger than one (Figure 6.2, *sps*). Splitting in this case reuses camera rays to generate multiple paths more efficiently. The light ray stream is traversed first and the contributions accumulated for the corresponding paths. Then, the extension ray stream is traversed, followed by the shading stage. Depending on the outcome of Russian roulette, the shading stage potentially generates one new extension ray and one new light ray per shaded sample, which are put into their respective streams again, replacing the previous ray data. The stream processing is iterated until either the extension stream becomes empty or the maximum number of iterations, i.e., bounces, are reached.

6.2.3 Kernel

The kernel abstraction encapsulates all the complex algorithms and corresponding data structures developed throughout this dissertation and exposes a minimal interface to support global illumination rendering based on ray tracing.

The main operation of the kernel is processing ray queries via the *trace** functions, providing global access to the scene geometry at any step of the rendering algorithm. Depending on its type a query returns either the closest or any intersection point for the given ray, supplemented by further data such as surface identifier, texture coordinates, etc. Multiple variants of *trace**, i.e., *traceSingle*, *tracePacket* and *traceStream*, support queries of single rays, packets and streams, respectively.

Prior to rendering the kernel is initialized with the scene geometry, which triggers the kernel's BVH construction algorithm. Any changes to the scene geometry must be communicated to the kernel in order to keep the internal BVH synchronized. The kernel integrated into the exemplary application supports two different update mechanisms: either, for changes to the visibility of the geometry, the update proceeds with the *dynamic mask* algorithm described below, or, for general changes to the geometry, a full rebuild by the BVH construction algorithm is initiated. In general, many applications perform changes to the scene geometry which only occur within a small region compared to the full scene, where it is more economical to partially rebuild or refit the existing BVH.

The kernel also manages load balancing for multi-core and cluster setups transparently through the *getTile* and *commitTile* semantics. A thread requests an image tile for rendering which it then processes without further synchronization requirements. Following completion of the tile the result is committed back to the kernel. Once the kernel has allocated all tiles for the current frame it initiates the transition to the next frame so that new tiles become available. Once all allocated tiles for a frame are committed the kernel releases the finished image to the application. If activated, the kernel also updates and maintains the accumulation buffer.

The accumulation buffer allows to progressively refine the rendered image. In the first iteration, only a single primary sample is computed per pixel and the resulting image is presented on the screen. The following iterations produce additional samples and accumulate the corresponding colors per pixel. After each iteration, the accumulated pixel values are normalized, i.e., averaged, in order to produce an updated image for display. Once the desired number of iterations / samples per pixel (Figure 6.2, *spp*) are computed the accumulation stops and the image is complete. It is also possible to let the accumulation continue indefinitely for the Monte Carlo integration to converge as much as possible. This is similar to frame-less rendering [15, 26], except when the scene changes or the camera moves the accumulation buffer is cleared by discarding previous samples and the accumulation starts anew.

Dynamic Masks

The ability to quickly blend in and out large parts of the scene geometry, as demonstrated in Figures 6.3 and 6.4, is implemented in the kernel in terms of *dynamic masks*.

Every inner node in the scene BVH encodes a mask with one bit per child node to indicate if the child node is active or inactive. Inactive child nodes already exist by default since not all slots of a multi-branch node are always occupied. Combining this child mask with the mask resulting from the bounding box test automatically filters out the inactive child nodes in case of a falsely detected intersection. Thus, by disabling the bit of the child mask the entire sub-tree is skipped during traversal

and the geometry disappears in the rendered image.

In order to restore the visibility of the geometry it is necessary to differentiate a disabled sub-tree from an empty child slot. This can be achieved either by keeping a copy of the original mask or by encoding the original mask in the the bounding box planes corresponding to the empty slot. The original mask can be either kept in a separate structure or within the BVH, depending on space constraints. To construct the original mask from the bounding box planes, a degenerate box with all planes set to zero may represent an empty slot and everything else a disabled sub-tree.

The visibility of individual triangles within a leaf node is controlled in the same way using the leaf node mask. If the number of triangles per leaf is unrestricted, i.e., can be larger than the branching factor, than the leaf mask might not provide a sufficient number of bits. A possible solution is to rearrange the list of triangles in the leaf node and put the active ones first, the inactive ones last, and adjust the triangle count to correspond to the active triangle number.

Triangles may be enabled or disabled individually, i.e., based on triangle ID, or based on a mesh ID or a material ID. In general, the algorithm for updating the BVH masks performs a post-order traversal over the entire BVH. At the leaves, the triangles are checked for a matching ID and the corresponding leaf mask / triangle order is updated. If all triangles in the leaf become disabled so does the leaf itself and this change is propagated back up the hierarchy in order to prune a disabled sub-tree as early as possible from the active BVH. In case a bounding box is available for all triangles corresponding to a particular mesh ID or material ID (for a primitive ID the bounding box can be easily calculated) the parts of the BVH which do not overlap the query bounding box can be safely skipped during an update.

The update algorithm is implemented within the parallelization framework developed for BVH construction in Chapter 4 and thus exhibits near linear scalability on multi-core CPUs. For common scenes a full update takes a few milliseconds at most, and for a huge data set like the BOEING the timing is around 700ms on a dual socket workstation.

6.3 Conclusion

This chapter has introduced an exemplary application featuring photo-realistic image quality for working with large CAD models interactively, including fast exploration and editing of materials and geometry visibility. The application is made possible by the performance delivered by the parallel algorithms developed throughout this dissertation, aggregated and organized into a single kernel. A demonstration of the exemplary application running on different devices ranging from laptop to high-performance cluster shows its scalability obtained transparently through the kernel. A progressive refinement of the image ensures that interactivity is maintained even for slower devices, with full image quality reached after a few seconds.

Chapter 7

Conclusions and Future Work

This dissertation has made significant contributions towards the long-standing goal of real-time photo-realistic rendering by research focused on parallel, highly scalable algorithms for ray tracing. The research has been organized into three building blocks which are essential for almost every photo-realistic rendering system, i.e., ray traversal, construction of acceleration structures and workload distribution. The corresponding key contributions and results are summarized below:

- **Vectorized ray traversal**

  Four new traversal algorithms have been introduced, named WIVE, ORST, CLPT and WIVEC, respectively. Each is specialized for, and performs best in, a different use case regarding ray coherence, hardware capabilities and rendering application structure. The new algorithms have in common designs focused on scalable data-parallel processing, which allow to use wider vector operations more effectively compared to previous approaches. As the presented experimental results have demonstrated, the higher vector utilization translates into superior performance ranging from 10% to 300% over industry-leading ray traversal implementations.

- **Parallel BVH construction**

  BVH construction is important to ray tracing, because (1) its quality directly affects ray tracing performance and (2) its construction determines the time-to-image after loading or modifying geometry. The newly introduced construction algorithm combines both maximum quality and high scalability for best performance, through improved scheduling, memory management and vector processing. Its high scalability enables the efficient utilization of large multi-core processors, which has been demonstrated by experiment to outperform previous approaches significantly, up to a factor of eight.

- **Asynchronous distributed computing**

  A framework for real-time, low-latency rendering on distributed computers has been introduced, which for the first time manages network communication with fully asynchronous and one-sided techniques, both on the hardware and software level. Load balancing with the new tile conquest algorithm in addition brings decentralization of communication and data-locality awareness, which spreads out and reduces data transfers and improves caching efficiency. The combined benefit is significantly improved scaling behaviour compared to previous approaches: unprecedented in distributed real-time rendering, super-linear scaling has been observed consistently over a large range of nodes in experiments.

Based on these contributions, real-time photo-realistic rendering has been realized on a 60 node cluster, delivering a throughput of about one Billion rays per second even for very large scenes. A more modern system of similar size today can already be expected to achieve up to one order of magnitude higher performance using the same scalable algorithms.

Graphics applications built on top of the three building blocks inherit the corresponding scalable rendering performance. This has been demonstrated in by the development of an application for interactive analysis and editing of very large triangulated scenes with photo-realistic quality. The application runs on parallel systems ranging from laptop to supercomputer, adjusting the performance according to the computational resources available.

Currently, the computational power of a cluster is required for very smooth photo-realistic rendering, and sometimes even a bit more depending on the complexity of the global illumination. The key point, however, is expressed by the hypothesis formulated earlier in the dissertation's introduction: the supercomputer of today is the handheld device of tomorrow. And indeed, during the time span of this research, the compute resources of a tablet have increased from a homogeneous dual core to a heterogeneous eight core configuration interconnected by a complex on-chip network. Also, the latest high performance processors feature multi-chip module designs with a stark resemblance to a cluster-on-chip. Hence, the scope and relevance of the presented research on scalable algorithms is expanding, from HPC to mainstream computer graphics.

In the future, the major issues for further advancements in real-time photo-realistic rendering are power efficiency and size. A cluster may comprise a sufficient number of nodes for the building blocks to enable the desired rendering quality and speed, however the form factor and power consumption is not practical for most use cases. Integration of a cluster-like system into a single package is required. Further, dedicated hardware units specialized in ray traversal and BVH construction provide the opportunity for further acceleration compared to the software implementations of this dissertation's algorithms, with lower and smaller area footprint as well.

Bibliography

[1] Attila T. Áfra. "Incoherent Ray Tracing without Acceleration Structures". In: *Eurographics 2012 - Short Papers*. Ed. by Carlos Andujar and Enrico Puppo. The Eurographics Association, 2012. DOI: 10.2312/conf/EG2012/short/097-100.

[2] Attila T. Áfra, Carsten Benthin, Ingo Wald, and Jacob Munkberg. "Local Shading Coherence Extraction for SIMD-efficient Path Tracing on CPUs". In: *Proceedings of High Performance Graphics*. HPG '16. Dublin, Ireland: Eurographics Association, 2016, pp. 119–128. ISBN: 978-3-03868-008-6. DOI: 10.2312/hpg.20161198.

[3] Timo Aila, Tero Karras, and Samuli Laine. "On Quality Metrics of Bounding Volume Hierarchies". In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: ACM, 2013, pp. 101–107. ISBN: 978-1-4503-2135-8. DOI: 10.1145/2492045.2492056.

[4] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.

[5] Arthur Appel. "Some Techniques for Shading Machine Renderings of Solids". In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: ACM, 1968, pp. 37–45. DOI: 10.1145/1468075.1468082.

[6] James Arvo. "Backward Ray Tracing". In: *In ACM SIGGRAPH '86 Course Notes - Developments in Ray Tracing*. Vol. 12. 1986, pp. 259–263.

[7] James Arvo and David Kirk. "Particle transport and image synthesis". In: *ACM SIGGRAPH Computer Graphics* 24.4 (1990), pp. 63–66. DOI: 10.1145/97880.97886.

[8] Ulf Assarsson and Tomas Möller. "Optimized View Frustum Culling Algorithms for Bounding Boxes". In: *Journal of Graphics Tools* 5.1 (2000), pp. 9–22. ISSN: 1086-7651. DOI: 10.1080/10867651.2000.10487517.

[9] Didier Badouel. *Graphics Gems*. Ed. by Andrew S. Glassner. San Diego, CA, USA: Academic Press Professional, Inc., 1990. Chap. An Efficient Ray-polygon Intersection, pp. 390–393. ISBN: 0-12-286169-5.

[10] Steve Bako, Thijs Vogels, Brian Mcwilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony Derose, and Fabrice Rousselle. "Kernel-predicting convolutional networks for denoising Monte Carlo renderings". In: *ACM Transactions on Graphics* 36.4 (2017), pp. 1–14. ISSN: 07300301. DOI: 10.1145/3072959.3073708.

[11] Rasmus Barringer and Tomas Akenine-Möller. "Dynamic Ray Stream Traversal". In: *ACM Transactions on Graphics* 33.4 (July 2014), pp. 1–9. ISSN: 0730-0301. DOI: 10.1145/2601097.2601222.

[12] Carsten Benthin, Ingo Wald, Sven Woop, Manfred Ernst, and William R. Mark. "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 (2012), pp. 1438–1448. ISSN: 1077-2626. DOI: 10.1109/TVCG.2011.277.

[13] E. Wes Bethel, Christopher Sewell, Torsten Kuhlen, Kenneth Moreland, Jeremy Meredith, Hank Childs, Berk Geveci, and Will Schroeder. "Research Challenges for Visualization Software". In: *Computer* 46 (May 2013), pp. 34–42. ISSN: 0018-9162. DOI: 10.1109/MC.2013.179.

[14] Nikolaus Binder and Alexander Keller. "Efficient Stackless Hierarchy Traversal on GPUs with Backtracking in Constant Time". In: *Proceedings of High Performance Graphics*. HPG '16. Dublin, Ireland: Eurographics Association, 2016, pp. 41–50. ISBN: 978-3-03868-008-6.

[15] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen J Scher Zagier. "Frameless Rendering: Double Buffering Considered Harmful". In: *SIGGRAPH '94 Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), pp. 175–176. ISSN: 0097-8930. DOI: 10.1145/192161.192195.

[16] Jiří Bittner, Michal Hapala, and Vlastimil Havran. "Fast insertion-based optimization of bounding volume hierarchies". In: *Computer Graphics Forum* 32 (2013), pp. 85–100. ISSN: 01677055. DOI: 10.1111/cgf.12000.

[17] Jiří Bittner, Michal Hapala, and Vlastimil Havran. "Incremental BVH construction for ray tracing". In: *Computers and Graphics (Pergamon)* 47 (2015), pp. 135–144. ISSN: 00978493. DOI: 10.1016/j.cag.2014.12.001.

[18] Brent Bohnenstiehl, Aaron Stillmaker, Jon J. Pimentel, Timothy Andreas, Bin Liu, Anh T. Tran, Emmanuel Adeagbo, and Bevan M. Baas. "KiloCore: A 32-nm 1000-Processor Computational Array". In: *IEEE Journal of Solid-State Circuits* 52.4 (2017), pp. 891–902. ISSN: 0018-9200. DOI: 10.1109/JSSC.2016.2638459.

[19] Solomon Boulos, Ingo Wald, and Peter Shirley. *Geometric and Arithmetic Culling Methods for Entire Ray Packets*. Tech. rep. SCI Institute, University of Utah, 2006, 2006.

[20] Solomon Boulos, Dave Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. "Packet-based Whitted and Distribution Ray Tracing". In: *Proceedings of Graphics Interface 2007*. GI '07. Montreal, Canada: ACM, 2007, pp. 177–184. ISBN: 978-1-56881-337-0. DOI: 10.1145/1268517.1268547.

[21] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. "Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder". In: *ACM Transactions on Graphics* 36.4 (2017), pp. 1–12. ISSN: 07300301. DOI: 10.1145/3072959.3073601.

[22] Robert L. Cook, Thomas Porter, and Loren Carpenter. "Distributed Ray Tracing". In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 137–145. ISSN: 0097-8930. DOI: 10.1145/964965.808590.

[23] Biagio Cosenza, Carsten Dachsbacher, and Ugo Erra. "GPU Cost Estimation for Load Balancing in Parallel Ray Tracing". In: *International Conference on Computer Graphics Theory and Applications* (2013).

[24] Holger Dammertz, Johannes Hanika, and Alexander Keller. "Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays". In: *Computer Graphics Forum* 27.4 (2008), pp. 1225–1233. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2008.01261.x.

[25] Holger Dammertz and Alexander Keller. "The edge volume heuristic - robust triangle subdivision for improved BVH performance". In: *2008 IEEE Symposium on Interactive Ray Tracing*. 2008, pp. 155–158. DOI: 10.1109/RT.2008.4634636.

[26] Abhinav Dayal, Cliff Woolley, Benjamin Watson, and David Luebke. "Adaptive frameless rendering". In: *ACM SIGGRAPH 2005 Courses*. New York, NY, USA: ACM Press, 2005, p. 24. DOI: 10.1145/1198555.1198763.

[27] Paul Debevec. "Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-based Graphics with Global Illumination and High Dynamic Range Photography". In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pp. 189–198. ISBN: 0-89791-999-8. DOI: 10.1145/280814.280864.

[28] David E. DeMarle, Christiaan P. Gribble, and Steven G. Parker. "Memory-savvy Distributed Interactive Ray Tracing". In: *Proceedings of the 5th Eurographics Conference on Parallel Graphics and Visualization* (2004), pp. 93–100. DOI: 10.2312/EGPGV/EGPGV04/093-100.

[29] David E. Demarle, Christiaan P. Gribble, Solomon Boulos, and Steven G. Parker. "Memory sharing for interactive ray tracing on clusters". In: *Parallel Computing* 31.2 (2005), pp. 221–242. ISSN: 01678191. DOI: 10.1016/j.parco.2005.02.007.

[30] *Embree Protoray*. Intel Corporation. 2017. URL: https://github.com/embree/embree-benchmark-protoray.

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

[32] Manfred Ernst and Günther Greiner. "Multi Bounding Volume Hierarchies". In: *2008 IEEE Symposium on Interactive Ray Tracing*. 2008, pp. 35–40. DOI: 10.1109/RT.2008.4634618.

[33] Rob Farber. *Redefining HPC Visualization Using CPUs*. 2017. URL: http://www.hpctoday.com/state-of-the-art/redefining-hpc-visualization-using-cpus.

[34] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071.

[35] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Sept. 2018. URL: https://www.agner.org/optimize/instruction_tables.pdf.

[36] MPI Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA, 1994.

[37] Valentin Fuetterling. "Methods, Computer Program and Apparatus for an Ordered Traversal of a Subset of Nodes of a Tree Structure and for Determining an Occlusion of a Point along a Ray in a Raytracing Scene". Pat. 20190035138. 2019.

[38] Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, Bernd Hamann, and Achim Ebert. "Accelerated Single Ray Tracing for Wide Vector Units". In: *Proceedings of High Performance Graphics*. HPG '17. Los Angeles, California: ACM, 2017, 6:1–6:9. ISBN: 978-1-4503-5101-0. DOI: 10.1145/3105762.3105785.

[39] Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, and Achim Ebert. "Efficient Ray Tracing Kernels for Modern CPU Architectures". In: *Journal of Computer Graphics Techniques (JCGT)* 4.4 (2015), pp. 89–109.

[40] Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, and Achim Ebert. "Parallel Spatial Splits in Bounding Volume Hierarchies". In: *Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by Enrico Gobbetti and E. Wes Bethel. The Eurographics Association, 2016. ISBN: 978-3-03868-006-2. DOI: 10.2312/pgv.20161179.

[41] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. "ARTS: Accelerated Ray-Tracing System". In: *IEEE Computer Graphics and Applications* 6.4 (1986), pp. 16–26. ISSN: 0272-1716. DOI: 10.1109/MCG.1986.276715.

[42] Per Ganestam and Michael Doggett. "SAH guided spatial split partitioning for fast BVH construction". In: *Computer Graphics Forum* 35.2 (2016), pp. 285–293. DOI: 10.1111/cgf.12831.

[43] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. "Simpler and Faster HLBVH with Work Queues". In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. HPG '11. Vancouver, British Columbia, Canada: ACM, 2011, pp. 59–64. ISBN: 978-1-4503-0896-0. DOI: 10.1145/2018323.2018333.

[44] *Gaspi : Global Address Space Programming Interface*. Version 17.1. GASPI-Forum. Feb. 2017.

[45] Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič, and Philipp Slusallek. "Light transport simulation with vertex connection and merging". In: *ACM Transactions on Graphics* 31.6 (2012), p. 1. ISSN: 07300301. DOI: 10.1145/2366145.2366211.

[46] Andrew S. Glassner. "Space subdivision for fast ray tracing". In: *IEEE Computer Graphics and Applications* 4.10 (1984), pp. 15–24. ISSN: 0272-1716. DOI: 10.1109/MCG.1984.6429331.

[47] Jeffrey Goldsmith and John Salmon. "Automatic Creation of Object Hierarchies for Ray Tracing". In: *IEEE Computer Graphics and Applications* 7.5 (1987), pp. 14–20. ISSN: 0272-1716. DOI: 10.1109/MCG.1987.276983.

[48] Christiaan Gribble. "Node Culling Multi-hit BVH Traversal". In: *Proceedings of the Eurographics Symposium on Rendering: Experimental Ideas & Implementations*. EGSR '16. Goslar Germany, Germany: Eurographics Association, 2016, pp. 85–90. ISBN: 978-3-03868-019-2. DOI: 10.2312/sre.20161213.

[49] Christiaan Gribble, Alexis Naveros, and Ethan Kerzner. "Multi-Hit Ray Traversal". In: *Journal of Computer Graphics Techniques (JCGT)* 3.1 (2014), pp. 1–17. ISSN: 2331-7418.

[50] Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. "Efficient BVH Construction via Approximate Agglomerative Clustering". In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: ACM, 2013, pp. 81–88. ISBN: 978-1-4503-2135-8. DOI: 10.1145/2492045.2492054.

[51] Vlastimil Havran. "Heuristic Ray Shooting Algorithms". Ph.D. Thesis. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2000.

[52] Brian Hayes. "A Lucid Interval". In: *American Scientist* 91.6 (2003), p. 484. ISSN: 0003-0996. DOI: 10.1511/2003.6.484.

[53] Alan Heirich and James Arvo. "A competitive analysis of load balancing strategies for parallel ray tracing". In: *The Journal of Supercomputing* 68.1998 (1998), pp. 57–68.

[54] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS". In: *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*. 2010, pp. 108–109. DOI: 10.1109/ISSCC.2010.5434077.

[55] Mark Howison, E. Wes Bethel, and Hank Childs. "MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems". In: *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)* (2010), pp. 1–10. ISSN: 1941-0506. DOI: 10.1109/TVCG.2011.24.

[56] *InfiniBand Architecture Specification Volume 1*. Release 1.3. InfiniBand Trade Association. Mar. 2015.

[57] *InfiniBand Architecture Specification Volume 2*. Release 1.3.1. InfiniBand Trade Association. Nov. 2016.

[58] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-040. Intel Corporation. Apr. 2018.

[59] *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A, 2B, 2C, and 2D: Instruction Set Reference, A-Z*. 325383-069US. Intel Corporation. Jan. 2019.

[60] Fraunhofer ITWM. *GPI-2*. 2018. URL: http://www.gpi-site.com/.

[61] Thiago Ize, Carson Brownlee, and Charles D. Hansen. "Real-Time Ray Tracer for Visualizing Massive Models on a Cluster". In: *EuroGraphics Symposium on Parallel Graphics and Visualization* (2011), pp. 61–69. DOI: 10.2312/EGPGV/EGPGV11/061-069.

[62] Thiago Ize, Ingo Wald, and Steven G. Parker. "Ray Tracing with the BSP Tree". In: *2008 IEEE Symposium on Interactive Ray Tracing* (2008), pp. 159–166. DOI: 10.1109/RT.2008.4634637.

[63] Charles. R. Johns and Daniel A. Brokenshire. "Introduction to the Cell Broadband Engine Architecture". In: *IBM Journal of Research and Development* 51.5 (2007), pp. 503–519. ISSN: 0018-8646. DOI: 10.1147/rd.515.0503.

[64] James T. Kajiya. "The Rendering Equation". In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 143–150. ISBN: 0-89791-196-2. DOI: 10.1145/15922.15902.

[65] James T. Kajiya and Timothy L. Kay. "Ray Tracing Complex Scenes". In: *ACM SIGGRAPH Computer Graphics* 20.4 (1986), pp. 269–278. ISSN: 00978930. DOI: 10.1145/15886.15916.

[66] James T. Kajiya and Brian P. Von Herzen. "Ray Tracing Volume Densities". In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 165–174. ISSN: 0097-8930. DOI: 10.1145/964965.808594.

[67] Tero Karras and Timo Aila. "Fast parallel construction of high-quality bounding volume hierarchies". In: *Proceedings of the 5th High-Performance Graphics Conference on - HPG '13* (2013), p. 89. DOI: 10.1145/2492045.2492055.

[68] Alexander Keller. *Quasi-Monte Carlo Image Synthesis in a Nutshell*. 2012.

[69] Andrew Kensler. "Tree rotations for improving bounding volume hierarchies". In: *2008 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2008, pp. 73–76. ISBN: 978-1-4244-2741-3. DOI: 10.1109/RT.2008.4634624.

[70] Andrew Kensler and Peter Shirley. "Optimizing Ray-Triangle Intersection via Automated Search". In: *2006 IEEE Symposium on Interactive Ray Tracing*. 2006, pp. 33–38. DOI: 10.1109/RT.2006.280212.

[71] David Kirk and James Arvo. "The Ray Tracing Kernel". In: *In Proceedings of Ausgraph*. 1988, pp. 75–82.

[72] Krzysztof S. Klimaszewski and Thomas W. Sederberg. "Faster Ray Tracing Using Adaptive Grids". In: *IEEE Computer Graphics and Applications* 17.1 (Jan. 1997), pp. 42–51. ISSN: 0272-1716. DOI: 10.1109/38.576857.

[73] Jaroslav Křivánek, Iliyan Georgiev, Anton S. Kaplanyan, and Juan Cañada. "Recent Advances in Light Transport Simulation: Theory and Practice". In: *ACM SIGGRAPH 2013 Courses*. SIGGRAPH '13. Anaheim, California: ACM, 2013, pp. 1–5. ISBN: 978-1-4503-2339-0. DOI: 10.1145/2504435.2504439.

[74] Eric P. Lafortune and Yves D. Willems. "Bi-Directional Path Tracing". In: *Proceedings of the third International Conference on Computational Graphics and Visualization Techniques (COMPUGRAPHICS '93)* (1993), pp. 145–153.

[75] Samuli Laine. "Restart Trail for Stackless BVH Traversal". In: *Proceedings of the Conference on High Performance Graphics*. HPG '10. Saarbrucken, Germany: Eurographics Association, 2010, pp. 107–111.

[76] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. "Fast BVH Construction on GPUs". In: *Computer Graphics Forum* 28.2 (2009), pp. 375–384. ISSN: 01677055. DOI: 10.1111/j.1467-8659.2009.01377.x.

[77] Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. "Vectorized Production Path Tracing". In: *Proceedings of High Performance Graphics*. HPG '17. Los Angeles, California: ACM, 2017, 10:1–10:11. ISBN: 978-1-4503-5101-0. DOI: 10.1145/3105762.3105768.

[78] Kwan Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. "A Data Distributed, Parallel Algorithm for Ray-traced Volume Rendering". In: *Proceedings of the 1993 Symposium on Parallel Rendering*. PRS '93. San Jose, California, USA: ACM, 1993, pp. 15–22. ISBN: 0-89791-618-2. DOI: 10.1145/166181.166183.

[79] Jeffrey Mahovsky. "Ray Tracing with Reduced-Precision Bounding Volume Hierarchies". PhD thesis. University of Calgary, 2005, p. 206. ISBN: 0-494-06958-9.

[80] Michael Mara, Morgan McGuire, Benedikt Bitterli, and Wojciech Jarosz. "An efficient denoising algorithm for global illumination". In: *Proceedings of High Performance Graphics on - HPG '17*. New York, New York, USA: ACM Press, 2017, pp. 1–7. ISBN: 9781450351010. DOI: 10.1145/3105762.3105774.

[81] Scott Miller and Charles R. Hoffman. "Illumination and Reflection Maps : Simulated Objects in Simulated and Real Environments". In: *Course Notes for Advanced Computer Graphics Animation, SIGGRAPH '84*. 1984.

[82] Tomas Möller and Ben Trumbore. "Fast, Minimum Storage Ray-Triangle Intersection". In: *Journal of Graphics Tools* 2.1 (1997), pp. 21–28. DOI: 10.1080/10867651.1997.10487468.

[83] Shinji Ogaki and Alexandre Derouet-Jourdan. "An N-ary BVH Child Node Sorting Technique for Occlusion Tests". In: *Journal of Computer Graphics Techniques (JCGT)* 5.2 (2016), pp. 22–37.

[84] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. "Large ray packets for real-time whitted ray tracing". In: *RT'08 - IEEE/EG Symposium on Interactive Ray Tracing 2008, Proceedings* (2008), pp. 41–48. DOI: 10.1109/RT.2008.4634619.

[85] Jacopo Pantaleoni and David Luebke. "HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry". In: *Proceedings of the Conference on High Performance Graphics*. HPG '10. Saarbrucken, Germany: Eurographics Association, 2010, pp. 87–95.

[86] Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, and Peter Shirley. "Interactive Ray Tracing for Volume Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 5.3 (July 1999), pp. 238–250. ISSN: 1077-2626. DOI: 10.1109/2945.795215.

[87] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David Mcallister, and Martin Stich. "OptiX: A general purpose ray tracing engine". In: *ACM Transactions on Graphics* 29.4 (2010), pp. 1–13. ISSN: 07300301. DOI: 10.1145/1833349.1778803.

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

[89] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. "Rendering complex scenes with memory-coherent ray tracing". In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97* (1997), pp. 101–108. ISSN: 00978930. DOI: 10.1145/258734.258791.

[90] Daniel Pohl, Gregory S. Johnson, and Timo Bolkart. "Improved Pre-warping for Wide Angle, Head Mounted Displays". In: *Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology*. VRST '13. Singapore: ACM, 2013, pp. 259–262. ISBN: 978-1-4503-2379-6. DOI: 10.1145/2503713.2503752.

[91] Stefan Popov, Iliyan Georgiev, Rossen Dimov, and Philipp Slusallek. "Object Partitioning Considered Harmful: Space Subdivision for BVHs". In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: ACM, 2009, pp. 15–22. ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572772.

[92] Christoph Schied, Marco Salvi, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, and Aaron Lefohn. "Spatiotemporal variance-guided filtering". In: *Proceedings of High Performance Graphics on - HPG '17*. New York, New York, USA: ACM Press, 2017, pp. 1–12. ISBN: 9781450351010. DOI: 10.1145/3105762.3105770.

[93] Christophe Schlick. "An Inexpensive BRDF Model for Physically-based Rendering". In: *Computer Graphics Forum* 13.3 (1994), pp. 233–246. ISSN: 14678659. DOI: 10.1111/1467-8659.1330233. arXiv: 9809069v1 [arXiv:gr-qc].

[94] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. "Ray-Triangle Intersection Algorithm for Modern CPU Architectures". In: *in Proceedings of GraphiCon 2007*. 2007, pp. 33–39.

[95] Peter Shirley, Changyaw Wang, and Kurt Zimmerman. "Monte Carlo techniques for direct lighting calculations". In: *ACM Transactions on Graphics* 15.1 (1996), pp. 1–36. ISSN: 07300301. DOI: 10.1145/226150.226151.

[96] Abe Stephens, Solomon Boulos, James Bigler, Ingo Wald, and Steven Parker. "An Application of Scalable Massive Model Interaction Using Shared-memory Systems". In: *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*. EGPGV '06. Braga, Portugal: Eurographics Association, 2006, pp. 19–27. ISBN: 3-905673-40-1. DOI: 10.2312/EGPGV/EGPGV06/019-026.

[97] Martin Stich, Heiko Friedrich, and Andreas Dietrich. "Spatial Splits in Bounding Volume Hierarchies". In: *Proceedings of the Conference on High Performance Graphics 2009 (HPG'09)* (2009), pp. 7–14. DOI: 10.1145/1572769.1572771.

[98] Georg Tamm and Philipp Slusallek. "Web-enabled server-based and distributed real-time Ray-Tracing". In: *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization (EGPGV-16)* (2016). DOI: 10.2312/pgv.20161182.

[99] Thiago Ize. "Robust BVH Ray Traversal". In: *Journal of Computer Graphics Techniques (JCGT)* 2.2 (2013), pp. 12–27. ISSN: 2331-7418.

[100] John A. Tsakok. "Faster Incoherent Rays: Multi-BVH Ray Stream Tracing". In: *Proceedings of the Conference on High Performance Graphics 2009 (HPG'09)* (2009), pp. 151–158. DOI: 10.1145/1572769.1572793.

[101] Eric Veach. "Robust Monte Carlo Methods for Light Transport Simulation". AAI9837162. PhD thesis. Stanford, CA, USA, 1998. ISBN: 0-591-90780-1.

[102] Eric Veach and Leonidas J. Guibas. "Metropolis light transport". In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97*. New York, NY, USA: ACM Press, 1997, pp. 65–76. ISBN: 0897918967. DOI: 10.1145/258734.258775.

[103] Eric Veach and Leonidas J. Guibas. "Optimally combining sampling techniques for Monte Carlo rendering". In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques - SIGGRAPH '95*. New York, NY, USA: ACM Press, 1995, pp. 419–428. ISBN: 0897917014. DOI: 10.1145/218380.218498.

[104] Carsten Wächter and Alexander Keller. "Instant ray tracing: the bounding interval hierarchy". In: *Proceedings of the 17th Eurographics conference on Rendering Techniques* (2006), pp. 139–149. DOI: 10.2312/egwr/egsr06/139-149.

[105] Ingo Wald. "Fast construction of SAH BVHs on the Intel Many Integrated Core (MIC) architecture". In: *IEEE Transactions on Visualization and Computer Graphics* 18.Mic (2012), pp. 47–57. ISSN: 10772626. DOI: 10.1109/TVCG.2010.251.

[106] Ingo Wald. "On Fast Construction of SAH-based Bounding Volume Hierarchies". In: *RT'07 - IEEE/EG Symposium on Interactive Ray Tracing 2007 Proceedings* 1 (2007), pp. 33–40. DOI: 10.1109/RT.2007.4342588.

[107] Ingo Wald, Carsten Benthin, and Solomon Boulos. "Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multi-branching BVHs". In: *RT'08 - IEEE/EG Symposium on Interactive Ray Tracing 2008, Proceedings* (2008), pp. 49–57. DOI: 10.1109/RT.2008.4634620.

[108] Ingo Wald, Solomon Boulos, and Peter Shirley. "Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies". In: *ACM Transactions on Graphics* 26.1 (2007), pp. 1–10. ISSN: 07300301. DOI: 10.1145/1186644.1186650.

[109] Ingo Wald and Vlastimil Havran. "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*. IEEE, 2006, pp. 61–69. ISBN: 1-4244-0693-5. DOI: 10.1109/RT.2006.280216.

[110] Ingo Wald, Philipp Slusallek, and Carsten Benthin. "Interactive Distributed Ray Tracing of Highly Complex Models". In: *Rendering Techniques 2001* (2001), pp. 277–288. DOI: 10.1007/978-3-7091-6242-2_26.

[111] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. "Embree: A Kernel Framework for Efficient CPU Ray Tracing". In: *ACM Transactions on Graphics* 33.4 (2014). ISSN: 15577333. DOI: 10.1145/2601097.2601199.

[112] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. "Interactive Rendering with Coherent Ray Tracing". In: *Computer Graphics Forum* 20.3 (2001), pp. 153–165. ISSN: 0167-7055. DOI: 10.1111/1467-8659.00508.

[113] Ingo Wald, Christiaan P. Gribble, Solomon Boulos, and Andrew Kensler. *SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering*. Tech. rep. 2007.

[114] Bruce Walter, Kavita Bala, Milind Kulkarni, and Keshav Pingali. "Fast agglomerative clustering for rendering". In: *RT'08 - IEEE/EG Symposium on Interactive Ray Tracing 2008, Proceedings* (2008), pp. 81–86. DOI: 10.1109/RT.2008.4634626.

[115] Turner Whitted. "An Improved Illumination Model for Shaded Display". In: *Commun. ACM* 23.6 (June 1980), pp. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882.

[116] Sven Woop and Ingo Wald. "Watertight Ray / Triangle Intersection". In: *Journal of Computer Graphics Techniques (JCGT)* 2.1 (2013), pp. 65–82.

[117] Matthias Zwicker, Wojciech Jarosz, Jaakko Lehtinen, Bochang Moon, Ravi Ramamoorthi, Fabrice Rousselle, Pradeep Sen, Cyril Soler, and Sung-Eui Yoon. "Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering". In: *Computer Graphics Forum* 34.2 (2015), pp. 667–681. ISSN: 0167-7055. DOI: 10.1111/cgf.12592.

List of Figures

List of Tables

List of Abbreviations

AOSOA	**A**rray **O**f **S**tructures **O**f **A**rrays
AOS	**A**rray **O**f **S**tructures
AVX	**A**dvanced **V**ector E**x**tensions
BSDF	**B**idirectional **S**cattering **D**istribution **F**unction
BSP	**B**inary **S**pace **P**artition
BVH	**B**ounding **V**olume **H**ierarchy
CAD	**C**omputer **A**ided **D**esign
CLPT	**C**oherent **L**arge **P**acket **T**raversal
CPU	**C**entral **P**rocessing **U**nit
DMA	**D**irect **M**emory **A**ccess
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DRST	**D**ynamic **R**ay **S**tream **T**raversal
GASPI	**G**lobal **A**ddress **S**pace **P**rogramming **I**nterface
GI	**G**lobal **I**llumination
GPU	**G**raphics **P**rocessing **U**nit
HD	**H**igh **D**efinition
HDR	**H**igh **D**ynamic **R**ange
HPC	**H**igh **P**erformance **C**omputing
HT	**H**yper-**T**hreading
ISA	**I**nstruction **S**et **A**rchitecture
LBVH	**L**inear **B**ounding **V**olume **H**ierarchy
MIMD	**M**ultiple **I**nstruction **M**ultiple **D**ata
NUMA	**N**on-**U**niform **M**emory **A**ccess
ORST	**O**rdered **R**ay **S**tream **T**raversal
PDF	**P**robability **D**ensity **F**unction
PGAS	**P**artitioned **G**lobal **A**ddress **S**pace
RDMA	**R**emote **D**irect **M**emory **A**ccess
SAH	**S**urface **A**rea **H**euristic
SBVH	**S**plit **B**ounding **V**olume **H**ierarchy
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SISD	**S**ingle **I**nstruction **S**ingle **D**ata
SOA	**S**tructure **O**f **A**rrays
SSE	**S**treaming **S**IMD **E**xtensions
TC	**T**ile **C**onquest
WIVEC	**W**ide **Ve**ctor **C**oherent Traversal
WIVE	**W**ide **Ve**ctor Traversal

Publications by Valentin Fütterling

Peer-reviewed Publications

[1] Efficient Ray Tracing Kernels for Modern CPU Architectures, Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, and Achim Ebert, Journal of Computer Graphics Techniques (JCGT), vol. 4, no. 4, 89-109, 2015

[2] Parallel Spatial Splits in Bounding Volume Hierarchies, Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, and Achim Ebert, Eurographics Symposium on Parallel Graphics and Visualization (EGPGV), 2016

[3] Accelerated Single Ray Tracing for Wide Vector Units, Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, Bernd Hamann and Achim Ebert, Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics (HPG), 2017

To be Submitted

[4] Scalable Real-time Path Tracing on Distributed PGAS Systems, Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, Bernd Hamann and Achim Ebert

Invited Talks

[5] Efficient Ray Tracing Kernels for Modern CPU Architectures, Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, and Achim Ebert, ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, San Francisco, California: February 25-27, 2017

[6] Core Algorithms for High-Performance, Interactive Rendering of Large-Scale Scientific Data, Valentin Fuetterling, Platform for Advanced Computing Conference (PASC15), Zürich, Switzerland: June 1-3, 2015

[7] Interaktive Server Basierte Visualisierung großer Daten, Valentin Fuetterling, Herbsttreffen des ZKI-Arbeitskreises "Supercomputing", München, Germany: October 19-20, 2015

Patent

[8] Methods, Computer Program and Apparatus for an Ordered Traversal of a Subset of Nodes of a Tree Structure and for Determining an Occlusion of a Point along a Ray in a Raytracing Scene, Valentin Fütterling, Patent Application filed with the United States Patent and Trademark Office

Curriculum Vitae

Valentin Fütterling

Born in Kirchheimbolanden, Germany

Education

2007	**Abitur** Leiniger Gymnasium, Grünstadt
2012	**Bachelor of Science, Physics** Karlsruhe Institute of Technology Thesis: *Fabrication of Heterometallic Quantum Point Contacts*
2014	**Master of Science, Physics** Karlsruhe Institute of Technology Thesis: *Cavity-Enhanced Emission from Carbon Nanotubes: Nanophotonic Light Sources*

Experience

2005–2014	**Student Research Assistant** Fraunhofer ITWM, Kaiserslautern
2007–2008	**Zivildienst** Pfalzklinikum für Psychiatrie und Neurologie, Kaiserslautern
2015–2018	**Doctoral Researcher** Fraunhofer ITWM, IRTG 2057, Technische Universität, Kaiserslautern
2017	**Graphics Research Intern** Intel Corporation, Santa Clara
2018–present	**Research Associate** Fraunhofer ITWM, Kaiserslautern
2019–present	**Graphics Technology Developer** Haas Schleifmaschinen GmbH, Trossingen