Foundations and Methods for GPU based Image Synthesis

vom Fachbereich Informatik
der Technischen Universität Darmstadt

zur Erlangung des akademischen Grades

Doktor-Ingenieur
(Dr.-Ing.)

Dissertation
VON SVEN WIDMER

Erstgutachter: Prof. Dr.-Ing. Michael Goesele
Technische Universität Darmstadt

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

Darmstadt, 2017

Widmer, Sven: Foundations and Methods for GPU based Image Synthesis
Darmstadt, Technische Universität Darmstadt
Jahr der Veröffentlichung der Dissertation auf TUprints: 2018
Tag der mündlichen Prüfung: 14. Juli 2017

Abstract

Effects such as global illumination, caustics, defocus and motion blur are an integral part of generating images that are perceived as realistic pictures and cannot be distinguished from photographs. In general, two different approaches exist to render images: ray tracing and rasterization. Ray tracing is a widely used technique for production quality rendering of images. The image quality and physical correctness are more important than the time needed for rendering. Generating these effects is a very compute and memory intensive process and can take minutes to hours for a single camera shot. Rasterization on the other hand is used to render images if real-time constraints have to be met (e.g. computer games). Often specialized algorithms are used to approximate these complex effects to achieve plausible results while sacrificing image quality for performance.

This thesis is split into two parts. In the first part we look at algorithms and load-balancing schemes for general-purpose computing on graphics processing units (*GPUs*). Most of the ray tracing related algorithms (e.g. KD-tree construction or bidirectional path tracing) have unpredictable memory requirements. Dynamic memory allocation on GPUs suffers from global synchronization required to keep the state of current allocations. We present a method to reduce this overhead on massively parallel hardware architectures. In particular, we merge small parallel allocation requests from different threads that can occur while exploiting SIMD style parallelism. We speed-up the dynamic allocation using a set of constraints that can be applied to a large class of parallel algorithms. To achieve the image quality needed for feature films GPU-cluster are often used to cope with the amount of computation needed. We present a framework that employs a dynamic load balancing approach and applies fair scheduling to minimize the average execution time of spawned computational tasks. The load balancing capabilities are shown by handling irregular workloads: a bidirectional path tracer allowing renderings of complex effects at near interactive frame rates.

In the second part of the thesis we try to reduce the image quality gap between production and real-time rendering. Therefore, an adaptive acceleration structure for screen-space ray tracing is presented that represents the scene geometry by planar approximations. The benefit is a fast method to skip empty space and compute exact intersection points based on the planar approximation. This technique allows simulating complex phenomena including depth-of-field rendering and ray traced reflections at real-time frame rates. To handle motion blur in combination with transparent objects we present a unified rendering approach that decouples space and time sampling. Thereby, we can achieve interactive frame rates by reusing fragments during the sampling step. The scene geometry that is potentially visible at any point in time for the duration of a frame is rendered in a rasterization step and stored in temporally varying fragments. We perform spatial sampling to determine all temporally varying fragments that intersect with a specific viewing ray

at any point in time. Viewing rays can be sampled according to the lens uv-sampling to incorporate depth-of-field. In a final temporal sampling step, we evaluate the pre-determined viewing ray/fragment intersections for one or multiple points in time. This allows incorporating standard shading effects including and resulting in a physically plausible motion and defocus blur for transparent and opaque objects.

Zusammenfassung

Computergenerierte Objekte und Szenen in Filmen, Werbung und Computerspielen zeichnen sich durch einen hohen Realisumus aus. Diese können kaum noch durch den Betrachter von klassisch fotografierten Objekten unterschieden werden. Dabei sind Effekte wie globale Beleuchtung, Kaustiken sowie Bewegungs- und Tiefenunschärfe integrale Bestandteile für eine realistische Darstellung. Es existieren zwei unterschiedliche Verfahren zur Erzeugung solcher Bilder. Die Verfolgung von Strahlen (Ray tracing im folgendem genannt) wird im Bereich der fotorealistischen Bildsynthese für Filme eingesetzt. Dabei ist die Bildqualität und physikalische Korrektheit entscheidend. Die Berechnung eines einzelnen Bildes kann mehrere Minuten oder auch Stunden dauern, spielt aber im Gegensatz zur Korrektheit nur eine untergeordnete Rolle. Rasterisierung auf der anderen Seite findet hauptsächlich Verwendung im Bereich der Computerspiele und anderer interaktiver Medien. Oft kommen dabei vereinfachte Modelle und Verfahren zum Einsatz, die schnell ein glaubwürdiges Bild erzeugen und wichtige Effekte aufgrund der limitierten Rechenzeit nur approximieren.

Die Dissertation beleuchetet neue Algorithmen für interaktive Anwendungen mit dem Ziel, die visuelle Qualität und Korrektheit von Ray tracing basierten Verfahren zu erreichen. Dazu ist die Dissertation in zwei Teile gegliedert. Im erste Teil werden Algorithmen und Lastverteilungsverfahren für Grafikkarten untersucht. Für Algorithmen wie Bidirectionales Path Tracing oder die Konstruktion von KD-Bäumen als Beschleunigungsstruktur kann der Speicherverbrauch nicht zum Startzeitpunkt angegeben werden und ist abhängig von Eingabedaten. Dynamische Speicherverwaltung kann das Problem minimieren. Bei der Verwendung massiv paralleler Hardware ist darauf zu achten, dass der Zustand der Speicherverwaltung immer konsistent ist. Dies erfolgt mittels Synchronisationobjekten, deren Verwendung aber Performanceeinbußen hervorrufen. Wir präsentieren ein Verfahren, welches den Synchronisationsaufwand auf massiv paralleler Hardware signifikant reduziert. Im Speziellen werden einzelne Allokationsaufrufe unterschiedlicher nebenläufiger Threads zusammengefasst. Durch weitere restriktive Anforderungen an die Anwendung, etwa das Freigeben von Speicher am Ende des Programms, kann die Laufzeit verbessert werden. Im Bereich der Filmproduktion ist der Rechenaufwand zur Bildgenerierung besonders hoch. Daher werden häufig CPU Rechnerverbünde eingesetzt, um den zeitlichen Aufwand zu reduzieren. Im Zuge dieser Arbeit präsentieren wir ein System zur dynamischen Lastverteilung in Grafikkarten basierten Rechnerverbünden. Das Verfahren minimiert speziell die Antwortzeit interaktiver Anwendungen.

Im zweiten Abschnitt der Dissertation kombinieren wir Ray tracing basierte Verfahren mit Rasterisierung um die Bildqualität für interaktive Anwendungen zu verbessern. Wir stellen dazu eine adaptive Beschleunigungsstruktur für Ray tracing Verfahren im Bildraum vor. Die Struktur approximiert dabei die sichtbare Szenengeometrie anhand von Flächen. Der

Vorteil ist ein schneller Ausschluss von Leerraum innerhalb der Geometrie, die nicht durch einen Strahl getroffen werden. Desweiteren kann der Ansatz einen exakten Schnittpunkt, basierend auf der Approximation der Fläche, berechnen. Das Verfahren erlaubt es uns komplexe Phänomene, wie Tiefenunschärfe und korrekte Reflektionen auf rauen Oberflächen, in Echtzeit zu simulieren. Um Bewegungsunschärfe in Kombination mit transparenten und opaquen Objekten zu simulieren haben wir einen auf Ray tracing basierenden Algorithmus entwickelt. Dieser trennt das Samplen der Raum- von der Zeitkomponente. Es ermöglicht uns durch die Wiederverwendung von sichtbaren Fragmenten interaktive Bildwiederholraten zu erreichen. Die zu jedem Zeitpunkt eines Bildes potenziell sichtbare Szenengeometrie wird rasterisiert. Im Anschluss erfolgt das räumliche Sampling um alle Fragmente zu ermitteln, die einen Sichtstrahl schneiden. Sichtstrahlen werden dabei anhand der Linse der Kamera erzeugt, um Tiefenunschärfe zu simulieren können. Abschließend wird die zeitliche Komponente evaluiert, der Farbwert berechnet und alle Fragmente eines Pixels akkumuliert.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Dr.-Ing. Michael Goesele for the continuous support of my Ph.D study and related research, for his patience, motivation, immense knowledge, and valuable feedback. His guidance helped me through out the time of research and writing of this thesis. I would also like to thank Prof. Dr.-Ing. Carsten Dachsbacher who kindly agreed to review this thesis. Furthermore, I thank David Luebke and his research group for the opportunity of an internship at NVidia. Especially, I'd like to thank Dawid Pajak for the tremendous help, the valuable discussions, and pushing me to achieve the best possible solutions as well as Douglas Lanman for the interesting task and insights into a different research topic. Thanks are due to Prof. Dr. Kay Hamacher and his Computational Biology and Simulation group at the TU Darmstadt. Without his interest in general-purpose computing on graphics processing units I probably would not have done my Ph.D thesis.
I would like to thank the Graduate School of Excellence Computational Engineering at the TU Darmstadt for their support and would also like to thank all external guests at our group retreats for the feedback and suggestions.
Special thanks go to all my colleagues in GCC who have accompanied me for the last years, in particular to Dominik Wodniok, Daniel Thuerck, Nicolas Weber, Andre Schulz, Daniel Thul, and Carsten Haubold. Without their support, programming and debugging skills, the collaboration, and helpful feedback from so many discussions the thesis would not have been possible.
Last but not least, I would like to thank my family: my parents and sister for supporting me throughout writing this thesis and my life in general.

Contents

Chapter 1

Introduction

Figure 1.1: In 2014, around 60 - 75 percent of all IKEA's product images (pictures showing only one single product) were computer generated. 35 percent of all non-product images were fully rendered and not composited afterwards [Parkin, 2014]. The photo shows a rendered kitchen scene [Giovanni_cg, 2015], Creative Commons CC0.

Our daily life is greatly influenced by media such as movies, games, and magazines. They use computer generated images to build new fantastic worlds, tell a new story or show beautiful and appealing images in advertisements to influence our decisions and actions (e.g. images of the latest cars, recent blockbuster movies, or the furniture in a catalog as shown in Figure 1.1). Nowadays, most people cannot distinguish between a computer generated picture and a real photograph anymore. An example can be found in the blog entry "Render Me Speechless: Get a Peek at Future of Design at SIGGRAPH 2015" [Estes, 2015].
Simulating phenomena such as global illumination, caustics, defocus and motion blur are an integral part to generating images. They are vital to perceive a rendered image as

a realistic picture which can not be distinguished from a photograph. Defocus blur for example is often used as an artistic style element to emphasize a certain character or object. Motion blur on the other hand is an integral part for perceiving the velocity and motion of an object. The simulation and rendering of such complex light and material interactions is very compute and memory intensive. It can take from minutes up to hours to render one single frame of a sequence. The image quality and physical correctness are the driving forces [Pantaleoni et al., 2010, Seymour, 2014]. On the other side of the spectrum are computer games and interactive media in general. For those applications performance is the key factor. The image quality and physical correctness of the simulation has to be reduced to achieve interactive or real-time frame rates with a refresh frequency of at least 30Hz. Computationally intensive phenomena such as global illumination as well as defocus and motion blur that make an image perceived as realistic are simplified and approximated.
To summarize, the main difference between offline and real-time rendering is the achieved image quality and the computation time for a single image. As the requirements are the opposite of each other two different approaches exist to generate images. *Ray tracing* is a widely used technique to generate photorealistic and physically correct results. In contrast, the *rasterization* of triangles is used if the performance of the application is critical. Advances in hardware architecture and novel algorithms made it possible to use insights from offline rendering and transfer them to real-time rendering. Therefore, a lot of research effort has been spent to bridge the gap between offline and real-time rendering using ray tracing for real-time applications. Unfortunately, the performance increase is still not sufficient enough to apply ray tracing in its general form. Most applications still rely on approximations of computationally intensive effects. With this thesis we try to close the gap further. In the following section, we briefly describe the two principle rendering techniques and motivate our contributions.

1.1 Image Generation

Ray tracing and rasterization are the two most common techniques to render images. Both are used for distinct purposes. Ray tracing is the preferred approach for photorealistic image in movies where rasterization is more commonly found in real-time applications. In this section, we would like to outline the two different rendering techniques.

1.1.1 Ray tracing

Ray tracing is the foundation of photorealistic image synthesis. Algorithms based on ray tracing (e.g. recursive ray tracing [Whitted, 1980], distributed ray tracing [Cook et al., 1984], path tracing [Kajiya, 1986], metropolis light transport [Veach, 1998], and photon mapping [Jensen, 1996]) can simulate a wide range of complex phenomena such as global illumination, caustics, depth of field, and motion blur. All these variants have in common that they simulate the transport of light to the eye, either based on following the path of rays of light or collecting photons. The underlying ray tracing algorithm is based on geometrical optics. Thereby, the light transport is described in terms of rays.
Ray tracing can be split up into several independent building blocks — the camera model, the ray-triangle intersection test, the light and material interaction, the recursive tracing

to gather all incoming radiance, and the acceleration structure to speed-up the ray traversal. The camera is the starting point for photo-realistic and real-time rendering. In the first step primary rays are generated based on the origin and view direction of the camera. Thereby, different camera models can be used such as the pinhole camera (Section 2.1.1) or a more complex model using the thin lens approximation for depth-of-field effects. The intersection of these rays with the scene geometry provides a point at which light and material interaction occur. The light and material interaction computes the incoming and outgoing radiance for the given intersection point. Therefore, we sample a bidirectional scattering distribution function (*BSDF*). As light may interact with several surfaces and materials before reaching the intersection point it is necessary to trace additional rays using the recursive nature of ray tracing. According to the incoming ray, the surface normal, and the BSDF we create new rays. This may result in several billion rays which have to be traced to converge to the final result. Bounding volume hierarchies (*BVH*) or other acceleration structures can be used to speed-up the ray traversal.
We will discuss the building blocks and their extension to distributed ray tracing in more detail in Section 2.1.

1.1.2 Rasterization

Rasterization [Pineda, 1988] on the other hand is typically used to render images if real-time constraints have to be met (e.g. computer games or other interactive media). Simple rendering primitives (e.g. triangles, lines, or complex polygons) are projected onto a 2D plane and converted into a raster format (pixel image) to display the primitives (see Section 6.2.1 for more details).
Unfortunately, the algorithm cannot handle complex phenomena natively. Specialized algorithms are often used to approximate some of those effects to achieve plausible results. These approximations are always driven by the underlying hardware architecture to achieve real-time frame rates. With the increasing compute capabilities more and more complex phenomena become available to rasterization. In recent years hybrid approaches using specialized variants of ray tracing and the underlying acceleration structures have emerged (e.g. sparse voxel octrees [Laine and Karras, 2010] and screen space ray tracing [McGuire and Mara, 2014]).
We will discuss the process of rasterization, the general rendering pipeline, and hybrid algorithms such as screen space ray tracing in Section 6.2.1.

1.2 Problem Statement

Despite the large body of literature and ongoing research in the area of real-time rendering the integration and usage of offline rendering algorithms and techniques into interactive applications has still not been solved. The "Holy Grail" of rendering - creating photorealistic images which are not distinguishable from a photograph in real-time - is not yet reached.
To achieve photorealistic results (production quality) the propagation of light in a scene using geometrical optics has to be simulated. Geometrical optics describes the light propagation in terms of rays. Kajiya [1986] derived the light transport equation (or rendering equation) using geometrical optics and the underlying ray approximation.

$$L_o(p, \omega_o, t) = L_e(p, \omega_o, t) + \int_\Omega f_r(p, \omega_o, \omega_i, t) L_i(p, \omega_i, t)(-\omega_i \cdot n) d\omega_i, \tag{1.1}$$

where $L_o(p, \omega_o, t)$ denotes the outgoing radiance in the direction ω_o from the surface point p at the time t. $L_e(p, \omega_o, t)$ expresses the emitted radiance at p in the direction ω_o where $L_i(p, \omega_i, t)$ is the incoming radiance at the surface point from the direction ω_i. The given equation can be integrated over time to support motion blur. The radiance that is reflected from ω_i into the direction ω_o is calculated by the phase function $f_r(p, \omega_o, \omega_i, t)$ (BSDF).
Overall, the radiance that arrives at a pixel on the image plane from a visible point is the sum of radiance reflected by other surfaces and the radiance emitted from a light source onto that point (Equation 1.1) (see Sections 2.1.4 and 2.1.5 as well as Pharr and Humphreys [2016] for more details). Monte Carlo integration is used when solving multiple integrals. A different approach is based on finite element methods. The computational power and memory bandwidth needed to solve the light transport equation is tremendous. To reduce the noise introduced by the Monte Carlo sampling methods we have to generate a large amount of samples per pixel. Each sample is related to a set of rays we have to trace through the scene, compute the intersection with the scene geometry, and evaluate the BSDF. With respect to the overall goal, creating photorealistic images in real-time, the following problems and challenges can be derived:

1. Novel rendering algorithms try to combine rasterization and ray tracing to achieve a higher image quality for real-time applications. Screen space ray tracing projects a ray into the image space. The ray traversal is simplified to an algorithm which is similar to the digital differential analyzer (*DDA*). It can be used to simulate a variety of different effects such as reflections and depth-of-field. Due to the perspective division oversampling can increase the number of traversal steps. This leads to unnecessary memory operations and therefore reduces the performance. To overcome the downside a fixed number of traversal steps can be used [McGuire and Mara, 2014]. Unfortunately, this can lead to artifacts due to incorrect intersection points and a reduced image quality.

   How can we speed-up the ray tracing process without reducing the image quality?

2. Nowadays, distribution effects such as depth-of-field and motion blur play an important role for the image quality and story telling in games. To achieve these effects real-time frame rates constraints and approximations have to be applied. The produced results are visually pleasing but not physically correct. To solve the visibility and dis-occlusions at any point in time between two frames requires a tremendous amount of memory as well as computing power. By introducing transparency the problem becomes more complex as the order of objects has to be taken into account. Navarro et al. [2011] gives a comprehensive introduction to motion blur and discusses several motion blur rendering algorithms for ray tracing (e.g. distributed ray tracing [Cook et al., 1984]) and real-time rendering (e.g. the accumulation buffer approach [Haeberli and Akeley, 1990]). Both approaches can be applied to depth-of-field rendering as well.

   How can we achieve a physically correct result which is still applicable to real-time applications?

3. For production quality rendering such as movies compute clusters are often used. They render different self-contained images of an animation sequence in parallel. Recent approaches try to employ a cloud or cluster computing to speed-up real-time rendering and increase the image quality. Crassin et al. [2015] describe the asynchronous computation of indirect lighting using a "cloud" environment. The main drawback of those techniques are the network latency and bandwidth requirements with respect to real-time user input — input delays have to be minimized. A different problem arises by using multiple processing units, cluster, and cloud computing. An optimal partition of work into smaller chunks and scheduling is not feasible. The computation time of a chunk may not be known in advance or differ from other chunks significantly. This will increase the overall computation time.

   How can we schedule and distribute chunks in a cluster environment more efficiently to reduce the overall computation time for interactive applications?

4. An important algorithmic building block to solve the problem is dynamic memory allocation. The usage of a dynamic memory allocator can reduce the performance dramatically. Thousands of allocations from independent threads on a GPU will increase the synchronization overhead.

   How can we reduce the synchronization overhead introduced by dynamic memory allocation on a GPU?

1.3 Contributions

The thesis contains several contributions in the area of specialized ray tracing for real-time rendering as well as algorithms for general-purpose computing on graphics processing units (*GPGPU*). In particular, we extend the state-of-the-art for screen space ray tracing as we used a novel acceleration structure to speed-up ray tracing effects (e.g. multi-view synthesis and depth-of-field rendering) for real-time applications. We propose to decouple space and time sampling to handle transparent and opaque fragments in a unified pipeline for interactive applications. The driving motivation is to bridge the gap between offline rendering and real-time rendering in terms of achieved image quality and performance. In addition we show how to employ different scheduling schemes to realize interactive applications on a GPU-cluster. Furthermore, an algorithm to speed-up dynamic memory allocation on a GPU is proposed. This algorithm can handle unpredictable memory peaks, e.g. during the construction of acceleration structures for ray tracing or the number of rays needed over the lifetime of a path during tracing.
Most main contributions were published as peer-reviewed publications at international conferences and journals.

- We present a method to reduce the required global synchronization for dynamic memory allocations on massively parallel hardware architectures (see Challenge 4 in Section 1.2). In particular we merge small parallel allocation requests from different threads that can occur while exploiting SIMD style parallelism. In addition we speed-up the dynamic allocation using a set of constraints that can be applied to a wide range of parallel algorithms.

  Chapter 4, [Widmer et al., 2013]

- A framework is developed which employs dynamic load balancing and applies fair scheduling to minimize the average execution time of spawned computational tasks on a GPU-cluster and allows concurrent access on GPUs. The load balancing capabilities are evaluated using the irregular workload generated by a bidirectional path tracer. The proposed framework allows us to render complex effects at near interactive frame rates (see Challenge 3 in Section 1.2).

  Chapter 5, in coop. with D. Wodniok, C. Haubold, and M. Goesele, unpublished

- We propose an adaptive acceleration structure to speed-up real-time screen-space ray tracing to tackle Challenge 1 in Section 1.2. The acceleration structure represents the scene geometry as a combination of bounding boxes and planar approximations. The benefit is a fast way to skip empty space and compute exact intersection points based on the planar approximation. The technique can handle several applications, including depth-of-field rendering, stereo warping, and screen-space ray traced reflections at real-time frame rates.

  Chapter 7, [Widmer et al., 2015]

- A unified rendering approach is presented that jointly handles motion blur, transparency and defocus at interactive frame rates (see Challenge 2). The scene geometry that is potentially visible at any point in time for the duration of a frame is rendered in a rasterization step and stored in temporally-varying fragments. A decoupled space and time sampling allows use to reuse fragments during the sampling step. In combination with a specialized intersection test for temporally varying fragments we achieve interactive frame rates.

  Chapter 8, [Widmer et al., 2016]

1.4 Thesis Outline

In Chapter 2 we introduce the main building blocks (e.g. camera model, light and material interaction, and acceleration structures) and outline the needed steps to render images using ray tracing. Based on these foundations we extend the building blocks by introducing distributed ray tracing [Cook et al., 1984] to support soft phenomena such as depth-of-field and motion blur.
Afterwards, the thesis is split up into two parts. Part I introduces massively parallel architectures, describes the mapping of ray tracing to a GPU, and outlines possible drawbacks. We show how to speed-up dynamic memory allocation (Chapter 4) for arbitrary allocation requests. In Chapter 5 we introduce a dynamic load balancing and distribution algorithm to support interactive applications in GPU cluster environments.
Part II of the thesis tries to bridge the gap between real-time and production rendering. Chapter 6 describes rasterization, screen space ray tracing, and illustrates state-of-the-art depth-of-field and motion blur algorithms. Based on these insights we propose an acceleration structure for screen space ray tracing (Chapter 7) and try to increase the quality of distribution effects for real-time applications (Chapter 8).
We finally summarize and discuss our achieved results in Chapter 9 and lay out possible future developments.

Chapter 2

Background

One of the key foundations for rendering photorealistic images is ray tracing. In the previous section we outlined the main building blocks of ray tracing: The pinhole camera model and ray generation, the ray triangle intersection, the light and material interaction, recursive tracing, and the need for acceleration structures to speed-up the rendering process. In this section we will discuss these in further detail and describe how distribution ray tracing [Cook et al., 1984] is integrated to simulate soft phenomena such as motion blur, depth-of-field, and soft shadows.

2.1 Ray tracing

Appel [1968] describes several techniques for shading solid objects. One of those is an algorithm based on marching along a ray to find the closest intersection point with a solid object for hidden surface removal and to compute shadows. In modern days, ray casting is widely used in different standard algorithms in computer graphics such as hidden surface removal or direct volume rendering. Important phenomena such as reflections and shadows can be integrated using the recursive nature of ray tracing as shown by Whitted [1980]. To simulate soft phenomena (e.g. motion blur, depth-of-field, and soft shadows) it is necessary to generate several rays for one pixel on the image plane. Cook et al. [1984] showed how to distribute the directions of a ray to sample the function. Kajiya [1986] introduced the rendering equation (or light transport equation - Equation 1.1) a generalization of several known rendering algorithms such as the Whitted approximation [Whitted, 1980]. To solve the light transport equation Kajiya [1986] outlined an algorithm using Monte Carlo methods - the foundation of path tracing. As the algorithm suffers from noise he proposed several variance reduction techniques. Bidirectional path tracing (*BDPT*) [Lafortune and Willems, 1993, Veach and Guibas, 1994] creates two random walks — one from a randomly sampled point on a light source and the other from the camera. The random walks are traced and the radiance and importance are evaluated and later combined to reduce the variance further. In the next step both walks are connected and the visibility between the intersection points is evaluated. For each possible path from the camera to the light source the contribution is computed. All possible combinations of all possible connections are created by multiple importance sampling using the maximum heuristic. The additional rays traced to solve the visibility between the intersection points

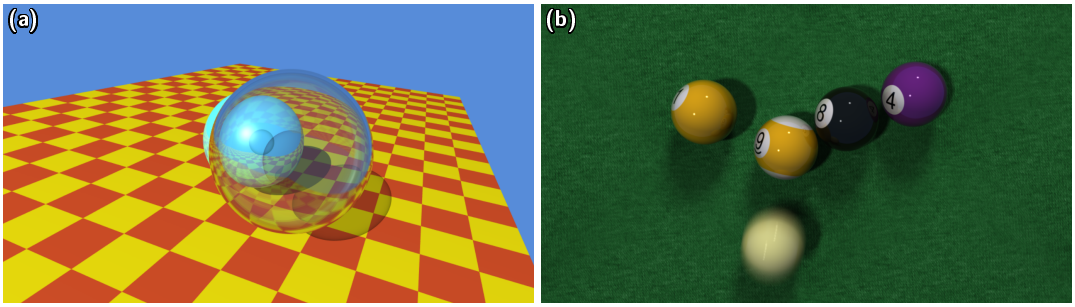

Figure 2.1: **(a)** Rendering using the Whitted style recursive ray tracing [Whitted, 1980] showing phenomena such as reflections, refractions, and shadows. The image **(b)** shows motion blur, depth-of-field and soft shadows rendered using distributed ray tracing [Cook et al., 1984].

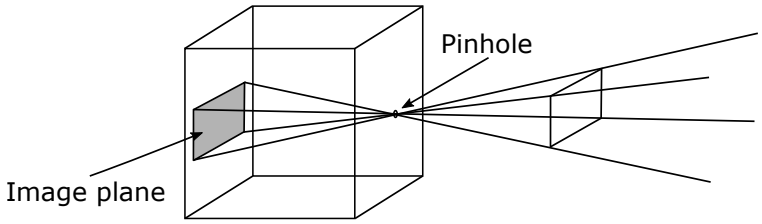

Figure 2.2: **Pinhole camera** — In the simplified camera model all light rays have to go through a pinhole to reach the image plane. As the pinhole is infinitely small the model does not support depth-of-field.

reduces the performance. As the algorithm may converge faster the performance loss is mitigated. In addition, BDPT can render effects such as caustics which are not feasible by a standard path tracer. Metropolis Light Transport (*MLT*) [Veach, 1998, Veach and Guibas, 1997] reuse paths that contribute to the final image. They are created by mutating existing paths. The mutation strategies are designed to sample different phenomena such as caustics more efficiently. Thereby, MLT explores the path space locally around the important path. For a broader introduction on algorithms and techniques used to solve the rendering equation please refer to the following books [Dutre et al., 2006, Pharr and Humphreys, 2016].
At first we take a look at a simple camera model, the first building block of ray tracing. For the next sections we follow the notations as given in "Physically Based Rendering" [Pharr and Humphreys, 2016]. To reduce complexity and simplify the overview we assume that our scene only consists of triangles and only consider the light transport in a vacuum. The interaction of rays with participating media (e.g. smoke) will not be discussed.

2.1.1 Camera

Everybody is familiar with cameras (e.g. digital single-lens reflex camera, smartphone cameras, or video cameras) to simply take snapshots or high quality photographs. The physics of these devices is surprisingly complex. A correct simulation would involve a lot

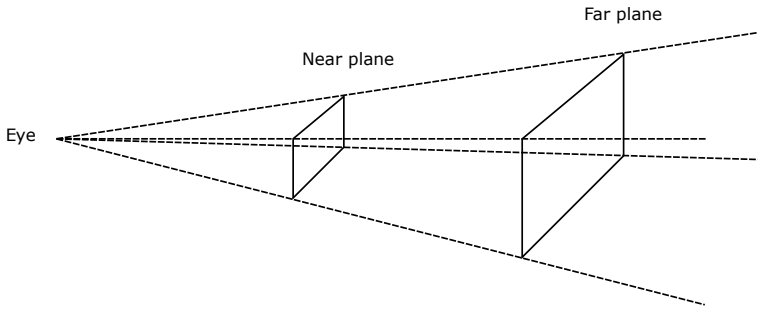

Figure 2.3: A modified pinhole camera model where the extent of the view volume is given by the near and far plane. The image plane can be located between the eye position and the near plane. In case of ray tracing the image plane is often the near plane.

of computation. Effects introduced by complex lens systems, motion blur from the shutter movement itself, lens aberrations, and aperture shape have to be considered. To simplify the model approximations were made. The simplest form, the **pinhole camera model**, consists of a light-tight box with a small hole in the middle of one side. On the opposite side of the box a paper, film or image plane is attached that captures all incoming light. Due to the infinitely small hole a long exposure time is needed to capture the scene. The rays with their origins at the four corners of the image plane going through the pinhole are spanning a view volume (see Figure 2.2). Every object inside or intersecting the volume is visible on the image plane.

In computer graphics to project the content inside the view volume onto the image plane a **projective camera model** is generally used. The projection is described by a 4×4 projection matrix. Thereby we can distinguish between an orthographic and a perspective projection which mimics the behavior of a pinhole camera model. The orthographic projection is a parallel projection where the projection rays are orthogonal to the image plane. Every surface plane (e.g. triangle) is transformed using an affine transformation. In contrast the perspective projection includes foreshortening. Objects in the background are projected smaller onto the image plane than objects with the same size in the foreground. Thereby geometric properties are not preserved such as the distance and angle. Parallel lines are not parallel anymore.

We can use the **perspective camera model** to describe a transformation and projection that mimics the behavior of a pinhole camera model. In addition we place a plane in front of the pinhole — the near plane. The pinhole itself lies at the eye position (also called the view position). The near and far plane describe the view volume. Every object inside the volume will be rendered. Both planes are used to map the depth between $z = 0$ and $z = 1$. Figure 2.3 shows the modified pinhole camera model (perspective camera model) as used in computer graphics. In general, the image plane is positioned between the near plane and the eye position. To render an image (compute the color) we have to sample the image plane and trace a new ray for each sample point. Thereby the ray direction is given by the eye position and the sample point on the image plane.

2.1.2 Ray/Triangle intersection

After creating a new ray we have to find the nearest intersection point of the ray with the given scene primitives. A ray is described by the following equation:

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

where $\vec{o}$ is the origin of the ray, $\vec{d}$ the direction vector, and t the scaling parameter of the direction vector to compute any point along the ray with $t[0, \infty]$ as we only trace forward. A simple triangle contains at least the vertices describing the position and the face normal needed for further operations. Other possible triangle attributes are the shading normal for proper shading, material information (e.g. color or complex material attributes), and a set of texture coordinates.
Möller and Trumbore [1997] introduced an efficient algorithm for ray triangle intersection. The benefit of the method is that the plane equation need not be computed on the fly nor to be stored. This can reduces the memory consumption of a triangle mesh significantly without loss in performance. The algorithm translates the origin of the ray $\vec{o}$ and then changes the base to yield a vector $(t\, b_1\, b_2)^T$, where t is the distance to the plane in which the triangle lies and b_1 and b_2 represents the coordinates inside the triangle. A parameterization of a triangle using barycentric coordinates b_1 and b_2 can be written as the following equation where $b_1 \geq 0, b_2 \geq 0$ and $b_1 + b_2 \leq 1$:

$$p(b_1, b_2) = (1 - b_1 - b_2)\vec{p_0} + b_1\vec{p_1} + b_2\vec{p_2} \tag{2.2}$$

$$r(t) = p(b_1, b_2) \tag{2.3}$$

$$\vec{o} + t\vec{d} = (1 - b_1 - b_2)\vec{p_0} + b_1\vec{p_1} + b_2\vec{p_2} \tag{2.4}$$

$$\begin{pmatrix} -\vec{d} & \vec{p_1} - \vec{p_0} & \vec{p_2} - \vec{p_0} \end{pmatrix} \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \vec{o} - \vec{p_0} \tag{2.5}$$

By solving the linear Equation 2.5 using Cramer's rule we obtain the ray parameter t to describe the intersection point as well as the barycentric coordinates b_1 and b_2 which later can be used to interpolate per vertex triangle attributes.
In the past years several hardware dependent and independent optimizations for the ray triangle intersection algorithms have been proposed (e.g. Schmittler et al. [2004], watertight intersection test by Woop et al. [2013]).
As we need to find the nearest intersection point from the origin of the ray we have to perform the test for every possible triangle in the scene. As the scene can contain millions of triangles a brute-force test, with an algorithmic complexity of $O(n \cdot m)$ with n the number of rays and m the number of triangles, against all triangles is not feasible. To speed-up the rendering process we can use different acceleration structures (see Section 2.1.6).

2.1.3 Monte Carlo integration

The rendering equation presented in Section 1.2 can be approximated using Monte Carlo methods. In this section we will give a short overview of evaluating an integral using

Monte Carlo integration. For an in-depth view into Monte Carlo methods we refer the reader to the following books [Dutre et al., 2006, Pharr and Humphreys, 2016, Rubinstein and Kroese, 2016]. We assume a function $f(x)$ with $x \in [0,1]$ and evaluate the following integral:

$$I = \int_0^1 f(x)dx. \tag{2.6}$$

To compute the integral we choose N samples to estimate the value of the integral. Thereby we randomly select the samples over the domain of I with a probability distribution function $p(x)$ (*pdf*), evaluate the function f using the sample, and average the weighted sampled values.

$$\langle I \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}. \tag{2.7}$$

As the expected value of the estimator is $E[\langle I \rangle] = I$ the estimator is unbiased. If $E[\langle I \rangle] \neq I$ the estimator is biased and the bias is $B[\langle I \rangle] = E[\langle I \rangle] - I$. A biased estimator is consistent if $\lim_{N \to \infty} B[\langle I \rangle] = 0$.

The Monte Carlo integration can be extended to multiple dimensions

$$I = \int\int f(x,y)dxdy, \tag{2.8}$$

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

Monte Carlo integration is a general method that can be used to estimate arbitrary functions f by sampling a pdf, using the samples to evaluate f, and average the weighted sampled values.

2.1.4 Light distribution and surface scattering

We obtained the nearest intersection point and interpolated attributes such as shading normals and colors using the barycentric coordinates b_1 and b_2, as well as some non-interpolated material properties.
In this step we compute the amount of radiance that is reflected from the intersection point into the direction of the camera. Therefore, we have to randomly select a light source from the scene and evaluate the visibility. The light source can either be a point light, spot light, directional light or area light. In case of an area light we have to sample a point on the geometry of the light. After randomly selecting the sampling point we have to evaluate the visibility for the intersection point. Therefore an additional ray with the intersection point as origin and the direction towards the light position is traced. The resulting scaling parameter t of the ray gives us the distance to the nearest intersection point with the scene geometry. If t is smaller than the distance to the light source the intersection point lies in shadow. If the point is visible we can calculate the incoming radiance, and compute the reflected amount using the assigned BSDF. For this section we assume direct lighting only.

This means we take only the amount of radiance into account which is directly reflected from the light source to the camera without the reflection from other surfaces.
As we want to compute the distribution of light energy in a scene we first want to give an overview over the most important quantities for light transport. The basic quantity is radiant power Φ. It represents the total amount of energy flowing from or to a surface per unit time in watts W (joules/sec).
Irradiance E is the incident radiant power on a surface per unit surface area in W/m^2:

$$E = \frac{d\Phi}{dA}. \tag{2.10}$$

Radiant exitance M describes the exitant radiant power per unit surface area in W/m^2:

$$M = \frac{d\Phi}{dA}. \tag{2.11}$$

Radiance L is the radiant power per unit solid angle per unit projected area in $W/(sr \cdot m^2)$:

$$L = \frac{d^2\Phi}{d\omega dA \cos\theta}. \tag{2.12}$$

Thereby the radiance depends on the position p and the direction vector ω_i.
Based on this information we can now compute the reflected radiance in the direction ω_o leaving the surface at point p to the camera. The bidirectional reflectance distribution function (*BRDF*) [Nicodemus, 1965] describes the fraction of the incident irradiance from a direction ω_i as reflected radiance in direction ω_i.
The irradiance function for a solid angle:

$$dE(p, \omega_i) = L_i(p, \omega_i) \cos\theta_i d\omega_i, \tag{2.13}$$

with $L_i(p, \omega_i)$ the incident radiance from the direction ω_i, θ_i the angle between ω_i and the surface normal n, the differential irradiance $dE(p, \omega_i)$ at p from direction ω_i.

$$f_r(p, \omega_o, \omega_i) = \frac{dL_o(p, \omega_o)}{dE(p, \omega_i)} = \frac{dL_o(p, \omega_o)}{L_i(p, \omega_i) \cos\theta_i d\omega_i} \tag{2.14}$$

Important physical properties of a BRDF are:

1. Reciprocity: $f_r(p, \omega_o, \omega_i) = f_r(p, \omega_i, \omega_o)$
2. Energy conservation: The total energy of light reflected is less than or equal to the energy of incident light.
3. Range: $f_r(p, \omega_o, \omega_i) \geq 0$

One of the most common BRDFs is the diffuse BRDF. Such materials reflect the light uniformly over the complete hemisphere. Thereby, the reflected radiance is independent of ω_o and the value of BRDF is constant for all directions ω_o and ω_i.

$$f_r(p, \omega_o, \omega_i) = \frac{\rho_d}{\pi}, \tag{2.15}$$

with $\rho_d \in [0, 1]$ the fraction of incident radiance that is reflected at the surface.

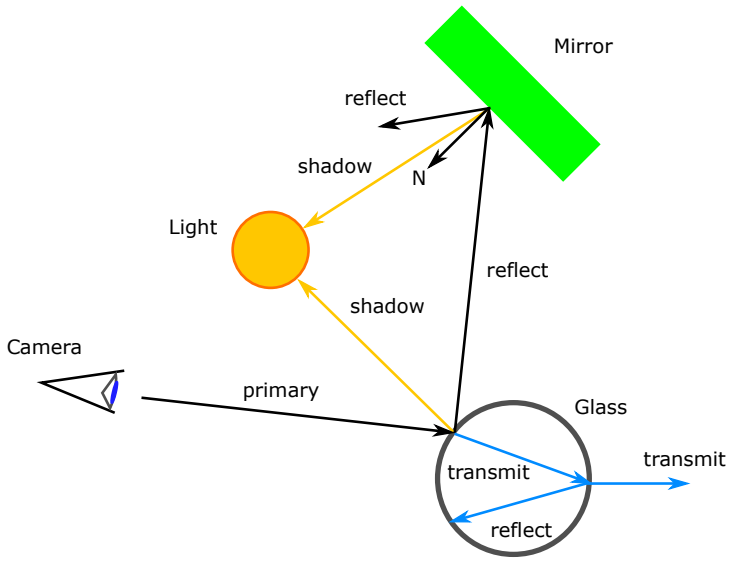

Figure 2.4: The figure illustrates the principle of recursive ray tracing [Whitted, 1980]. The algorithm can incorporates perfect reflections, refraction, and shadows to renderings. A primary ray from the camera is reflected and transmitted at the glass sphere. From the intersection point we recursively trace new rays to evaluate the perfect reflections and refractions as well as the shadow.

2.1.5 Recursive ray tracing and path tracing

In Section 2.1.4 we only take the energy into account which is directly reflected from a light source (direct lighting) to the camera. To achieve realistic results light reflections from one surface to another have to be taken into account. As introduced in the problem statement 1.2 we have to solve the light transport equation [Kajiya, 1986] (Equation 1.1). Earlier, Whitted [1980] highlighted the recursive nature of ray tracing. To support mirrors he proposed to reflect a ray at the intersection point about the face normal and trace a novel ray in the reflection direction. The incoming light energy arriving at the intersection point is then summed up. Thereby, it is necessary to follow the ray recursively over several bounces if another mirror was hit. Transparent materials can be handled in a similar way by computing the refraction vector instead. Whitted-style ray tracing only evaluates the light transport equation for perfect reflections and refractions (Figure 2.4).

2.1.6 Acceleration structure

To significantly speed-up the computation time different acceleration structures can be applied. One can group them into two different categories, object hierarchies and spatial hierarchies. Both organize a soup of primitives (e.g. triangles, rectangles, and complex polygons) in a n-dimensional tree structure. An **object hierarchy** uses a simple volumetric object (e.g. sphere, axis aligned bounding box, or object oriented box) to describe the extents of a subset of primitives from the complete soup. The parent is recursively split into n child nodes beginning from the root. Thereby child nodes can overlap each other.

The corresponding bounding volumes are typically stored at the inner nodes while the leaf nodes contain pointers to the primitives. In contrast, a **spatial hierarchy** splits up the space populated by the primitives into a set of half-spaces based on a near optimal splitting plane. Afterwards, the primitives will be grouped and sorted to the left or the right side of the plane. Each half-space is recursively split until certain criteria are met.
A typical object hierarchy is the bounding volume hierarchy (*BVH*) where an axis aligned bounding box is used to describe the extents of a node. This acceleration structure is widely used in ray tracing as it is easy and fast to build in parallel on multi- and many-core architectures [Wald, 2007, 2012]. In contrast, KD-trees [Zhou et al., 2008] partition the space into a hierarchy of disjoint sets. Wu et al. Wu et al. [2011] presented an efficient construction algorithm for GPUs. Up until now, BVHs construction on many-core architectures easily outperform kD-tree construction algorithms. Vinkler et al. [2014] compared the ray tracing performance for a GPU ray traversal algorithm based on Aila and Laine [2009]. They showed that for simple and moderate complex scene the BVH ray tracing performance outperforms kD-trees. KD-trees, on the other hand, perform better for complex scenes. Stich et al. [2009] apply spatial splits in bounding volume hierarchies (*SBVH*) and gain similar tracing performances as the kD-trees but can be constructed faster. An optimized acceleration structure for many-core architectures are multi-branch BVHs [Dammertz et al., 2008, Ernst and Greiner, 2008, Wald et al., 2008]. It reduces the depth complexity of a BVH tree by merging inner nodes to test multiple child nodes in a single traversal iteration. The structure increases memory access coherence and reduces the number of traversal steps by performing multiple child node intersection tests using vector instructions. In both cases, object and spatial hierarchies, the termination criteria for the construction of the tree can either be the number of primitives in a child node or the surface area heuristic (*SAH*). The best way to reduce the overall traversal costs for both data structures is to use a top-down, greedy SAH based construction algorithm [Wald, 2007]. Thereby the expected costs of the ray traversal have to be minimized. Equation 2.16 estimates the expected traversal cost. We assume a volume V encompasses N triangles which can be split into two child nodes of the tree V_L and V_R with N_L and N_R.

$$Cost() = K_T + K_I\left(\frac{SA(V_L)}{SA(V)}N_L + \frac{SA(V_R)}{SA(V)}N_R\right), \tag{2.16}$$

where $SA(V)$ is the surface area of the volume V, and K_T and K_I are constants which estimate the cost of the traversal and triangle intersection. For BVH trees the volume V can be split into two child nodes by choosing an axis aligned split plane in one of the three dimensions which minimizes the cost function. The axis aligned bounding box (*AABB*) of the volume is uniformly subdivided into K bins. This gives us $K-1$ ways per dimension to partition the set of triangles. In the next step all triangles are sorted into the bins using their centroids. Each bin has knowledge of the number of triangles it contains and their tight AABB. With this information we can evaluate the cost function (Equation 2.16) of each possible bin combination that constructs V_L and V_R. We choose the combination with minimal cost and split the two child nodes further. The algorithm can be terminated if the number of triangles is under a certain threshold, the costs cannot be minimized further or the size of the AABBs is too small. For an in-depth discussion of the algorithm and its parallelization we refer the reader to Wald [2007]. Recent research by Aila et al. [2013] and Wodniok and Goesele [2016] has shown that the SAH cost can not predict the gain in traversal performance well.

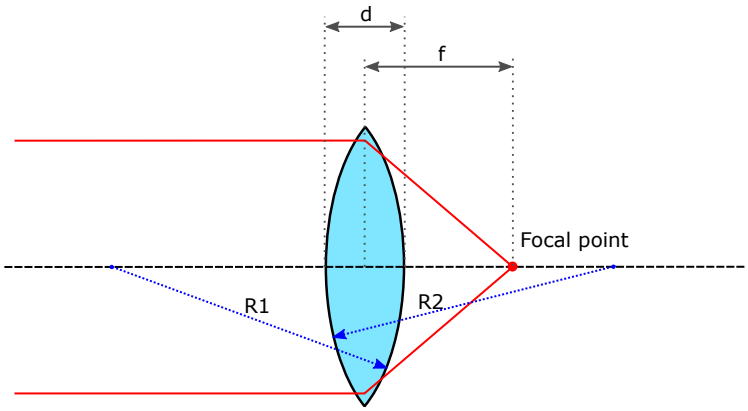

Figure 2.5: The figure illustrates the thin lens approximation. All rays that are parallel to each other will converge in the focal point after they where refracted by the lens. The focal length f describes the distance of the focal point from the lens midpoint. The lens thickness d can be neglected under the thin lens approximation if the radii of the lens surface R_1 and R_2 (blue lines) are significantly larger.

2.2 Depth-of-field

A camera uses a complex system of lenses to focus light through the aperture onto the camera sensor. The larger the aperture diameter the more light can reach the sensor. If the aperture diameter is decreased the depth-of-field is increased. Thereby, depth-of-field is the distance between the closest and furthest points which are still recognized as in-focus. Using a narrow depth-of-field, only a small amount of the image is in focus and vice versa for a wide depth-of-field. The amount of light that reaches the sensor can be limited by a shutter (or diaphragm). The thin lens approximation can be used to model the finite aperture.

2.2.1 Thin lens camera model

The thin lens model approximates an optical system (set of lenses) using a single lens. It assumes that the thickness of the lens is negligible in comparison to the radius of curvature of the lens surfaces (see Figure 2.5). Therefore, ray tracing can be simplified. All optical effects based on the thickness of lenses are ignored. As we assume the surrounding medium of the lens is vacuum the index of refraction is $n = 1$. According to the thin lens approximation all rays that are parallel to each other will converge in the same point after the refraction, the focal point at the distance f from the lens. If we place an image plane at the focal point all objects with an infinite distance from the lens are always in focus (infinity focus).

In the following we use the Cartesian sign convention for the Gaussian form of the lens equation. Thereby, the distance to the object o is negative and the distance to the image positive, the light travels from the left to the right side. All distances are measured from a refracting surface. As d is negligible we use the midpoint of the lens in all figures.

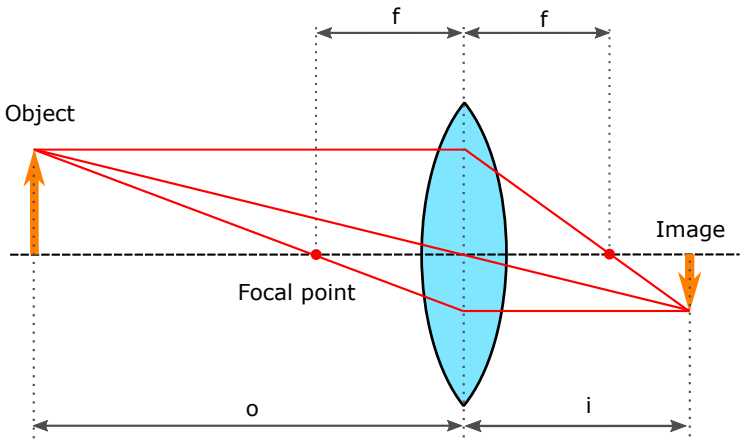

Figure 2.6: Geometric description of the Gaussian lens equation 2.17. The distance o is the scene distance of the object to the midpoint of the lens and i the distance on the image plane size.

$$\frac{1}{i} - \frac{1}{o} = \frac{1}{f}, \tag{2.17}$$

where o is the distance of the object to the lens and i is the signed distance from the lens where the object is in focus. If we set $o = -\infty$ in Equation 2.17, this yields $i = f$ as described above. All parallel rays converge at the focal point (see Figure 2.6).

$$i = \frac{f\,o}{f + o}. \tag{2.18}$$

The Gaussian lens equation 2.18 can be used to compute the distance i at which the point P at distance o would be in focus (point I in Figure 2.7). The projection of the point becomes a disk on the image plane — the circle of confusion (*COC*) with a diameter of d_c. The extent of the circle depends on the diameter of the aperture.
We can compute the circle of confusion d_c (Equation 2.19 and Equation 2.20) by applying the properties of similar triangles based on the diameter of the lens d_l and i. The ratio must be the same as d_c to $i^{'} - i$ (see Figure 2.7).

$$\frac{d_l}{i} = \frac{d_c}{|i' - i|}, \tag{2.19}$$

$$d_c = |\frac{d_l(i^{'} - i)}{i}|. \tag{2.20}$$

Based on the size of the COC d_c and the size of a pixel d_p we can derive a formal definition for the depth of field. A point is in-focus if the corresponding size of the COC d_c is smaller than the size of the pixel d_p on the image plane. We can express the relation between depth-of-field and the aperture size using Equation 2.21, where o is the distance of the object center in-focus, $o^{'}$ the distance of the frontmost point on the object in-focus. The

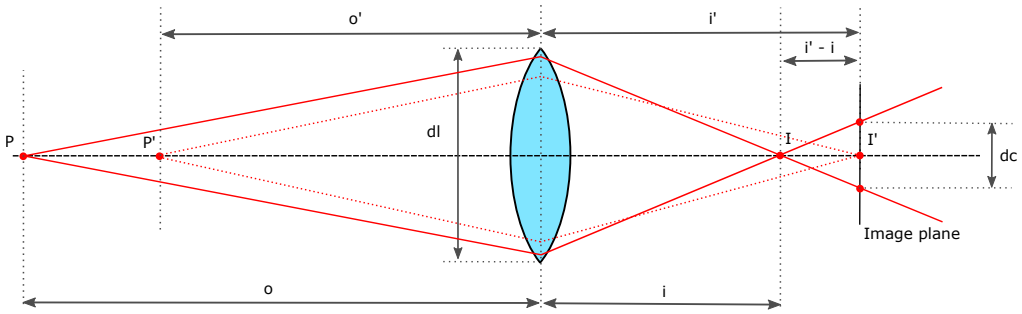

Figure 2.7: The Figure illustrates the geometric meaning of Equation 2.20 to calculate the diameter of the circle of confusion, d_c. Given d_l, the diameter of the lens, a point $P^{'}$ in focus results in a point on the image plane $I^{'}$ with the distance $i^{'}$ from the lens.

Figure 2.8: The photos show object motion blur. Due to the motion of the subject it is blurred on the images (motion of the water, moving bus and cars) and the background is sharp as the camera does not move, Creative Commons CC0.

point is on a sphere with the radius r. Therefore we can say that with an increasing aperture size the depth-of-field becomes smaller.

$$r < |\frac{od_s(f+o)}{d_l f + d_s(f+o)}| \implies r \propto \frac{1}{d_l} \tag{2.21}$$

As our main goal is to achieve a more realistic image at real-time frame rates we stick to the thin lens approximation. For an introduction to realistic camera models using a lens system and thick lenses please refer to "Physically Based Rendering" [Pharr and Humphreys, 2016]. In Section 2.4 we discuss the implementation of the thin lens approximation for ray tracing. Later on, in Section 6.2.3 we will introduce depth-of-field for real-time applications. Therefore, the circle of confusion is an essential concept.

2.3 Motion blur

The exposure time of a camera describes the length of time the camera sensor is exposed to light. During that time-span objects can move and will introduce blur (motion blur) to an image. Figure 2.8 shows motion blur based on object motion relative to the camera. The moving objects get blurred as the background and static objects such as the telephone

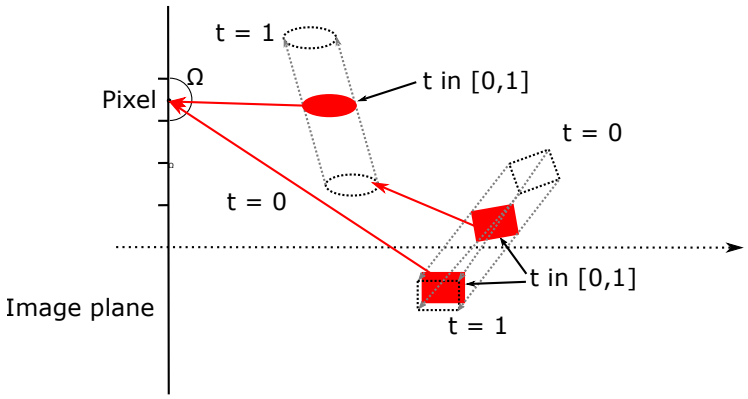

Figure 2.9: The figure illustrates Equation 2.23. The function integrates the incoming light $L_l(\omega, t)$ over the hemisphere Ω at a pixel. The visibility of the rays is solved by $g_l(\omega, t)$.

booth remain sharp.
Sung et al. [2002] formalized the exposure process using Equation 2.22. Thereby, (x, y) represents the pixel position on the image plane I. The captured light is the integration of the radiance $L(\omega, t)$ over the exposition time T at the sensor. Using the reconstruction filter $r(\omega, t)$ we can model physical properties such as the diameter, geometry, and movement of the aperture.

$$I(x, y, t) = I(\omega, t) = \int_{\Omega} \int_{T} r(\omega, t) L(\omega, t) dt d\omega \tag{2.22}$$

We can extend Equation 2.22 by modeling the visibility of objects l of a scene in the direction of ω at time t. Thereby the function $g_l(\omega, t)$ evaluates the visibility and $L_l(\omega, t)$ computes the radiance for the object l. Current algorithms do not take the motion blur introduced by the shutter movement into account [Sung et al., 2002]. This can simplify the reconstruction filter $r(\omega, t)$.

$$I(\omega, t) = \sum_{l} \int_{\Omega} \int_{T} r(\omega, t) g_l(\omega, t) L_l(\omega, t) dt d\omega \tag{2.23}$$

Using Monte Carlo integration we can approximate Equation 2.23. Therefore, we sample in the spatio-temporal domain (Equation 2.24). This approach can be directly mapped to distributed ray tracing [Cook et al., 1984]. We will discuss this in the following Section 2.4.

$$I(\omega, t) \approx \frac{1}{N_j} \frac{1}{N_k} \sum_{j} \sum_{k} \sum_{l} r(\omega_j, t_k) g_l(\omega_j, t_k) L_l(\omega_j, t_k) \tag{2.24}$$

Navarro et al. [2011] has analyzed the physical properties of motion blur and methods that can simulate it in computer generated images. They introduced and categorize the existing algorithms based on their differences, strengths and limitations.

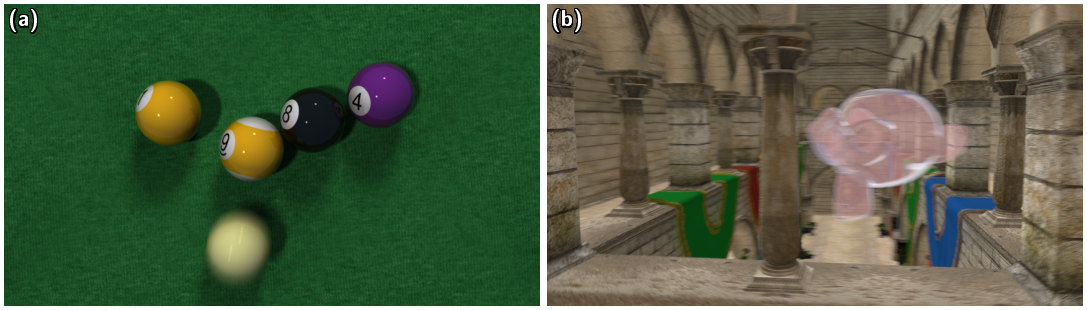

Figure 2.10: The image **(a)** shows motion blur, depth-of-field and soft shadows rendered using distributed ray tracing [Cook et al., 1984]. Figure **(b)** illustrates motion blur introduced by camera and object motion.

2.4 Distributed ray tracing

Several phenomena such as perfect specular reflections and transparent objects can be rendered using recursive ray tracing [Whitted, 1980] (see Section 2.1.5). To simulate motion blur, depth-of-field, and soft shadows it is necessary to generate several rays per pixel. Cook et al. [1984] proposed sampling strategies to incorporate these effects. Thereby, a new ray with an uniformly distributed origin on a camera lens is generated. In addition we have to sample in the time domain to create motion blur. Therefore, the simplified pinhole camera model 2.1.1 has to be replaced by a lens and aperture.

Depth-of-field The basic idea of depth-of-field rendering using distributed ray tracing is to sample the associated size of the lens aperture and create a ray from the lens position with a direction through the focal plane. Therefore we need the know the focal distance to compute the focal point.
At first we sample a point on the unit disk. Therefore we map a randomly selected point on the unit square (ξ_1, ξ_2) to a point on a unit disk so that we achieve a uniform distribution. The most basic way to map the unit square is to use a polar mapping such as $r = \sqrt{\xi_1}$ and $\theta = 2\pi\xi_2$. Shirley and Chiu [1997] presented an algorithm that maps points onto a unit disk $(\xi_1, \xi_2) \in [-1, 1]^2$ to concentric circles. To compute a new ray origin we multiply the sampled point with the lens radius (size of the lens aperture).
To create the new ray direction we assume that the focal plane is perpendicular to the z-axis. This simplifies the computation of the focal point $\vec{fp}$ as we know that a ray which passes through the center of the lens is not refracted — the direction is not changed (see Section 2.2.1). We need to compute the intersection of the original ray r with the focal plane using the focal distance i as follows:

$$\vec{fp} = \vec{o}_r + \frac{i}{d_{z_r}}\vec{d}_r. \tag{2.25}$$

Motion blur In addition to the spatial domain (lens (u, v)) the time domain t is sampled to incorporate motion blur. All that is needed is to calculate the position of an object at the time t between two transformations with $t = 0$ and $t = 1$. The calculation of the position can be arbitrarily complex as in general the motion of an object does not have to be linear. To simplify the problem it is possible to assume that the complex motion is

piecewise linear so we can simply interpolate between $t = 0$ and $t = 1$. Afterwards, the intersection of the transformed object with the ray has to be computed.
To accelerate distributed ray tracing one can use specialized acceleration structures which encompass the motion trajectory. For example, a BVH tree can be constructed for a certain t (e.g. $t = 0.5$). During the ray traversal each bounding box must be interpolated according to t. A tighter bounding volume can reduce the number of intersection tests. The intersection test itself becomes more complex and therefore the overall performance can suffer. Olsson [2007] extended kD-trees by adding a temporal split in the time domain. The increasing number of object references introduce a significant memory overhead, which limits its practical applicability. Grünschloß et al. [2011] proposed a 4D space-time extension to the spatial split BVH algorithm [Stich et al., 2009].

Part I

Foundations for GPU Computing

Chapter 3

Motivation

With the introduction of the programmable rendering pipeline, general purpose computing on GPUs became available not only for computer graphics but also gained a wide acceptance in the high performance and scientific computing community. In this part of the thesis we will introduce the current GPU architecture, discuss its implications on programming models, describe the mapping of ray tracing on a GPU, and outline possible drawbacks. We present a dynamic memory allocation algorithm (Chapter 4) for arbitrary allocation requests optimized for many-core architectures such as GPUs. The allocator can be applied not only to ray tracing but to high performance computing as well. In Chapter 5 we introduce a dynamic load balancing and distribution algorithm to support interactive applications in a GPU cluster environment. We use a classic bidirectional path tracer to evaluate the scalability and performance of our approach.

3.1 Introduction

Parallel architectures can be classified using Flynn's taxonomy [Flynn, 1972]. They are based on single instruction multiple data (*SIMD*) model or multiple instructions multiple data (*MIMD*). Multiple processing units are executing the same single instruction on different data in parallel or multiple instructions in case of MIMD. Based on the viewpoint, current CPUs and GPUs can be grouped into different classes. A single core of a CPU works in the SIMD fashion while multiple cores can execute different instructions in parallel and therefore will be classified into MIMD — going up to clusters of compute units.

3.2 Background

In contrast to CPUs, GPUs use a modified version of SIMD — single instruction multiple threads (*SIMT*). A group of threads contains a program counter to efficiently support branching. The number of threads in such a group is given by the SIMD-width. Current GPUs are using a width of 32 or 64 threads which are grouped to one unit of execution. The execution of instructions inside a group of threads (so-called warp) is implicitly synchronous.

Figure 3.1: The diagram in this figure shows all components of a Streaming Multiprocessor (*SM*) of the GP104 architecture [NVIDIA, 2016](redrawn after [NVIDIA, 2016]).

3.2.1 GPU architecture

To understand the advantages and disadvantages of GPU based algorithms we have to take a look at the current state-of-the-art GPU architecture based on the NVidia Geforce GTX 1080 and the underlying Pascal architecture [NVIDIA, 2016]. Thereby, we will focus on the SIMT execution model, how it is mapped to the GPU hardware, and outline possible performance bottlenecks that may occur. For simplicity we will use the notations introduced by NVidia. But the core principles can be easily adapted to other programming languages and to GPU architectures of other vendors.
The basic unit of execution on a GPU are threads as well as warps. A warp operates in a SIMD fashion. All threads in a warp are executing the same instruction on different data. As branching may happen during the execution, the number of threads that execute an instruction is reduced. Those who are not following the branch are masked out. At the end of the branching path all threads are running synchronously due to the SIMD execution model.
To map threads to the hardware each Streaming Multiprocessor (*SM*) on a GPU contains 128 cores (arithmetic logic units) whereby each core executes one thread (see Figure 3.1 - light green boxes). As threads are grouped in warps, an SM can run four different warps with different program counters in parallel. A multiprocessor has access to a set of special function units (SFU) to execute transcendental instructions such as sin, cosine, reciprocal, and square root operations. The load/store units (LD/ST) handle the bidirectional transfer of data between memory and registers.
Another crucial part for an efficient execution of GPU kernels is the memory model. Current GPU architectures contain several different memory areas which differ in scope and therefore in read/write bandwidth, latency, and size due to different technologies used (Figure 3.1 - blue boxes). The hierarchy contains the register file set, shared memory, and global memory. Shared memory is often used for data distribution as it is shared between all threads in a thread block. To reduce the high latency a GPU uses a hierarchy of caches. The L1 data and instruction caches increase the performance of instruction decoding and

data fetches. Texture caches as well as L2 cache reduce the latency of global memory. A typical non-cached read operation from a kernel can take up to several hundred GPU-cycles.

A typical multiprocessor of a state-of-the-art GPU may has a 256 Kbyte register file, 96 Kbyte shared memory, and a 48 Kbyte working set for L1 cache / texture cache. In addition 10 Kbytes are available to cache the access to constant memory, a part of global memory only available for constant variables. 2048 Kbyte L2 cache are used to cache global memory access for the entire GPU. If a thread uses too many register or the data structures become too large they are spilled into local memory — essentially a memory area residing in the global memory. Each thread can use 512 Kbyte of local memory at max.

In the end, five Streaming Multiprocessors are combined into one Graphics Processing Cluster (*GPC*). A GPU can contain a different number of GPCs depending on the exact model.

3.2.2 Programming model

Based on the described hardware architecture, we will now introduce the programming model and discuss some implications of the underlying hardware.

Parallel programs can decompose a problem — an algorithm or a program — in two different ways. Task parallelism distributes a set of tasks across different processing units. Thereby a task, a C like kernel , is executed concurrently by one processor. Each processor runs the same or a different kernel on the same or a different set of data — classified as MIMD. In contrast, in a decomposition using data parallelism all processors run the same task on different set of data and is classified by Flynn's taxonomy as SIMD. Each parallel task and data chunk can further be decomposed into concurrent sub-tasks and smaller data independent chunks. So we can combine task and data parallelism.

The parallel programming model for GPUs uses task and data parallel decomposition. Independent tasks are executed concurrently by different warps using a task parallel decomposition model. Each thread in a warp runs the same instruction off the task on a different data word. Thereby it is important that tasks and data chunks are independent to minimize possible synchronization.

While developing algorithms for GPUs it is essential to think about SIMD efficiency, the overlay of memory access and computation, and the memory access pattern for good cache usage to achieve the best possible performance.

In a SIMD efficient algorithm all threads are working entirely in parallel over the complete execution time of a kernel. Branching code introduced by conditional statements or loops can reduce the SIMD efficiency. Based on the statement not every thread executes the same instruction at the same time. A second problem is the latency of the memory access as described in the previous section. A typical uncached read operation can take up to several hundred GPU-cycles. In this case a new warp has to be scheduled to overlay memory operations with computation. We can use shared memory to speed-up memory reads by pre-fetching the data we use for computation or rely on the cache hierarchy.

To facilitate the possibilities of general purpose computation on GPUs (*GPGPU*) modern high-level programming languages and APIs were introduced, such as NVidia's CUDA [NVIDIA, 2015] and the cross-platform OpenCL standard [Khronos Group].

3.2.3 Ray tracing on GPUs

After the short description of the current NVidia Pascal architecture the question arises how to map ray tracing onto GPGPUs. A basic and the most common mapping of ray tracing onto the SIMD model is the mapping of one ray onto one thread. Each thread of a warp traverses the bounding volume hierarchy, intersects the tree nodes, computes the intersection with the triangles, and does the final shading computation independently from other threads.
Ray tracing on GPUs is a widely studied field [Aila and Karras, 2010, Aila and Laine, 2009, Laine et al., 2013, van Antwerpen, 2011]. The main performance drawback is caused by incoherent rays [Aila and Karras, 2010]. These are rays which are handled by neighboring threads in a warp but traverse a different path in the BVH tree. Thereby, the memory access while fetching the child nodes is not contiguous. This results in increased memory traffic, cache trashing, and serialization of threads and therefore reduces the achieved SIMD efficiency [Aila and Laine, 2009]. A different source of a reduced SIMD efficiency is branching inside a warp. As rays can traverse different paths in the tree the neighboring rays can have different states. One can still traverse the BVH, another one may do the AABB intersections with some nodes of the tree, and another may already have found a leaf node with a triangle and do the triangle intersection test. All threads in a warp are executing a different instruction and have to wait until they reach the same instruction. To alleviate the problem we use a while-while ray traversal kernel [Aila and Laine, 2009].
Depending on the light transport algorithm used, the memory footprint for book keeping can be very high. Techniques such as bidirectional path tracing have to store the complete path (several ray segments) to generate different combinations of eye and light paths. Normally one can allocate the needed buffers for the temporary data in advance. Due to the different path lengths (based on material properties and random events) memory can either be wasted as too many rays were pre-allocated or the size of the allocation was insufficient.
In large scale production rendering, CPU based path tracers such as Arnold Solid Angle [2016] and Disney's production renderer Hyperion [Eisenacher et al., 2013, Seymour, 2014] are often used. Limitations of GPUs are the small amount of global memory available, the additional data streaming costs for out-of-core algorithms, well understood hardware with existing code base and years of experience on optimizing CPU based path tracing using Streaming SIMD Extensions (*SSE*). GPUs are often applied to pre-computation of certain effects and data structures Pantaleoni et al. [2010] or pre-visualization during production. Often large computing clusters are used with little support of GPUs and FPGAs for special tasks [Seymour, 2014]. Christensen and Jarosz [2016] give an overview of current technologies used in production rendering.

3.2.4 GPU cluster computing

At the end of the previous section we have outlined some reasons why production quality path tracers are CPU based. All use CPU clusters to render the set of final images. They distribute different self-contained images of an animation sequence and render them in parallel. Recent approaches in real-time rendering try to use GPU based cloud or cluster computing to speed-up real-time rendering and increase the image quality [Crassin et al., 2015]. A drawback of these approaches are the network latency, bandwidth requirements, and input lag which have to be minimized. Production quality rendering can

be partitioned into smaller chunks such as tiles of an image or the radiance of indirect light passes. Unfortunately, the computation time is not known in advance and may differ from tile to tile due to different material shader complexity or other effects. Therefore, the overall computation time can suffer from an irregular workload and a less optimal distribution of work (Challenge 3 in Section 1.2).
In Chapter 5 we present a framework which can distribute and reschedule GPU computations in a heterogeneous environment. We show how to reduce the overall rendering time of an interactive application by using work stealing to alleviate the uneven workload distribution.

3.2.5 GPGPU dynamic memory allocation

In many scenarios it is beneficial to know the memory consumption in advance. Therefore, the most efficient implementations of parallel algorithms work on fixed size buffers which are allocated using an estimated upper bound of the memory consumption of the algorithm over the complete runtime of a kernel. One reason to use these pre-allocated buffers is the lack of dynamic memory allocation on the kernel side. Another reason is the performance impact of the allocator itself. If many threads request memory independently at any time the synchronization overhead introduced by the dynamic memory allocation increases the kernel runtime significantly. On the other hand, using fixed buffers with an upper bound size can waste memory as the buffer may be too large.
More importantly, well known algorithms and data structures exist where the amount of temporary data needed for the construction of a data structure cannot be computed in advance (e.g. KD-tree construction [Vinkler and Havran, 2014]). More problematic is an unpredictable memory consumption which is based on the input data in combination with random events. As described in the previous Section 3.2.3, algorithms based on ray tracing and traversal can have an unpredictable memory consumption which is not known ahead of time (Challenge 4 in Section 1.2).
In the following chapter we describe an algorithm for dynamic memory allocation (Chapter 4) which reduces the synchronization overhead significantly and can be applied on a per thread basis for ray tracing. This can reduce the amount of wasted memory as we do not rely on pre-allocated buffers.

Chapter 4

Fast Dynamic Memory Allocator for Massively Parallel Architectures

Abstract

Dynamic memory allocation in massively parallel systems often suffers from drastic performance decreases due to the required global synchronization. This is especially true when many allocation or deallocation requests occur in parallel. We propose a method to alleviate this problem by making use of the SIMD parallelism found in most current massively parallel hardware. More specifically, we propose a hybrid dynamic memory allocator operating at the SIMD parallel warp level. Using additional constraints that can be fulfilled for a large class of practically relevant algorithms and hardware systems, we are able to significantly speed-up the dynamic allocation. We present and evaluate a prototypical implementation for modern CUDA-enabled graphics cards, achieving an overall speedup of up to several orders of magnitude.

4.1 Introduction

Dynamic memory allocation is one of the most basic features programmers use today. It enables memory allocation at runtime and is especially useful, if the amount of memory needed is not known ahead of time. Modern operating systems provide therefore easy to use interfaces to allocate and free memory arbitrarily. Unfortunately, these approaches do not generalize directly with good performance on massively parallel architectures such as current graphics processing units (GPUs) or even many-core systems. The key problem is hereby that bookkeeping during naïve allocation and deallocation requires a form of global synchronization. This is a severe performance bottleneck when systems become more and more parallel.
This effect can, e.g., be observed in practice when using the C functions `malloc()` and `free()` that were recently included into NVIDIA's CUDA framework [NVIDIA, 2015] to allocate memory dynamically at runtime. Several approaches have therefore been developed to build a dynamic memory allocator capable of working in a massively parallel environment, where traditional approaches do not work, including XMalloc [Huang et al., 2010] and ScatterAlloc [Steinberger et al., 2012]. Although these implementations show better results than the built-in CUDA allocator, their application still results in a noticeable slowdown.
Our key observation is that massively parallel architectures typically operate in a SIMD fashion where a single instruction is physically executed in parallel. In CUDA, this corresponds to the concept of a warp. Memory allocation and deallocation should take this into account and will ideally yield a significant speedup when operating in this granularity. We additionally propose multiple constraints and assumptions which are fulfilled in many practically relevant algorithms and hardware systems that yield a different and much faster implementation. In particular, we assume that a systemwide default memory allocator is available. Further, we expect the application to free memory not arbitrarily but free all memory at certain points during the execution.
These assumptions fulfill the SIMD parallel programming scheme and can be applied to various algorithms, in particular algorithms with an unpredictable transient or output data size such as Monte Carlo-based simulation techniques, graph layout algorithms, or adaptive FEM simulations. All these algorithms can be implemented without dynamic memory allocation by interrupting the GPU computations at critical points and allocating additional needed memory. The introduced global synchronization can be at least partially reduced if not eliminated by use of dynamic memory allocation.
Our contributions are as follows:

- an improved memory allocator for massively parallel architectures with a wide SIMD width such as Nvidia's CUDA and Intel's Xeon Phi. It drastically increases performance and works for a wide variety of applications.
- a comparison of the proposed allocator with the standard CUDA malloc and ScatterAlloc.

4.2 Related work

Dynamic memory allocation is nowadays an ubiquitous operation. Therefore most programming languages provide some mechanism to allocate memory at runtime. The com-

plexity of such operations is transparent for most programmers. They are in most cases not aware of the implications the operations have on the overall runtime of an application. This is specially the case for multi-processors and in particular multi-core CPU systems.
Many different memory allocators and algorithms for memory management where proposed over time. An introduction and basic overview is given by Knuth [1997] and Tanenbaum [2007]. Wilson et al. [1995] have evaluated and compared the most common algorithms regarding the overall memory consumption.
Over the past years a lot of dynamic memory allocators for multi-processor and multi-core systems were proposed. Gloger [1998] extended the well known and widely used dlmalloc [Lea, 1996] to support multi-thread environments. Berger et al. [2000] uses per processor heaps in addition to a global heap to increase scalability. Dice and Garthwaite [2002] as well as Michael [2004] introduced memory allocators based on lock-free data structures.
Modern parallel architectures have a wide variety of active hardware threads. They range from multi-core systems with up to 16 cores over many-core systems to massively parallel architectures such as GPUs which can execute thousands of threads concurrently. With the increasing number of threads running concurrently the synchronization overhead increases and becomes a severe bottleneck. As a result the scalability and in particular the SIMD scalability decreases.
In this chapter we will focus on dynamic memory allocation for massively parallel architectures with a wide SIMD width such as GPUs or Intel's Xeon Phi. Our main goal is hereby to increase the SIMD scalability.

4.2.1 Dynamic memory allocation on CPU

With the introduction of multi-processor and multi-core systems the classic memory allocation algorithm became a severe bottleneck. To increase the scalability of an allocator on a multi-processor system Häggander and Lundberg [1998] proposed two optimizations. A parallel heap and memory pools for commonly used object types were implemented to gain a significant speedup. For every processor a heap area is created. A thread can allocate memory using this area of the processor it is executed on. If the heap area is occupied by a different thread one can try to lock a heap area of another processor. A similar approach, the Hoard allocator, was proposed by Berger et al. [2000]. They augment a global heap with a per-processor heap that every thread may access. Hoard caches a limited number of superblocks (a chunk of memory) per thread. To keep fragmentation at a minimum, unused superblocks can be migrated into the global heap and used by other processors.
Most dynamic memory allocators rely on atomic operations or even require mutual exclusion locks to handle their critical section and keep shared data structures consistent. This can have a significant performance impact and reduced scalability with increasing number of cores per processor. To reduce mutual exclusion locks, lock-free data structures which build on the atomic Compare-and-Swap (CAS) or the Load-Linked and Store-Conditional (LL/SC) operation are supported by almost all current CPU architectures. In this context Michael [2004] presents a completely lock-free memory allocator.
Hudson et al. [2006] presented McRT-malloc, a scalable non-blocking transaction-aware memory allocator that is tightly integrated with a software transactional memory system. It avoids expensive CAS operations by accessing only thread-local data and increases scalability even further.

4.2.2 Dynamic memory allocation on massively parallel architectures

Publications dealing with dynamic memory allocation for GPGPU applications are scarce. Huang et al. [2010] introduce the problems that have to be faced when building a memory allocator for massively parallel architectures, identifying the need to synchronize access to header data as the main issue. This synchronization serializes execution and therefore decreases the performance gain from parallel architectures.

XMalloc uses a similar approach as the Hoard allocator [Berger et al., 2000] by introducing superblocks and using the atomic CAS operation from Michael [2004] to reduce synchronization overhead. To parallelize memory allocation, all threads are divided in smaller groups. Each group maintains their own superblocks. This allows groups to work independently from each other and to only access the global memory management when allocating a new superblock. Each superblock can be divided into several smaller blocks and distributed among the threads in the group.

ScatterAlloc [Steinberger et al., 2012] expands XMalloc [Huang et al., 2010] by introducing a new approach to further reduce simultaneous access from different threads to the same memory region. Their implementation does not search linearly for a free memory slot, but instead scatters the memory access. This reduces the concurrent access to the same memory region and speeds up the allocation process.

In contrast to XMalloc and ScatterAlloc we introduce a voting algorithm to determine a single worker thread, reducing the amount of concurrent access to the critical section and increasing SIMD scalability. We propose assumptions that minimize the algorithmic complexity and suggest an allocation principle based on superblocks that increases the overall performance.

4.3 Design

Various modern many-core architectures (e.g., GPUs or accelerator cards) are executing a group of threads in a SIMD style fashion, each thread corresponding to a SIMD lane. All those threads must execute the same instruction to minimize divergence and achieve the best performance. A dynamic memory allocator for those systems must scale when thousands of threads allocate different chunks of memory at the same time. In the following, we call a group of SIMD lanes of one streaming processor core a warp, similar to the CUDA terminology. An important consequence of the SIMD nature is that threads in a warp are implicitly synchronized.

Allocation algorithms such as Hoard [Berger et al., 2000] are optimized for multi-threading environments. They do not scale with increasing numbers of warps. Allocators proposed for many-core architectures (e.g. XMalloc [Huang et al., 2010]) are too general since they are based on the assumption that all threads are independent and not executed in SIMD style.

The main design goal for our allocator was to increase the SIMD scalability for small, frequent memory allocations and therefore endeavor to reduce the branch divergence. To achieve this, we rely on the following three assumptions.

1. A system wide default memory allocator exists and works fast, as long as there are only few simultaneous requests.

2. There is no need to free single chunks of memory during the execution. It is sufficient that the complete allocated memory of a group of threads can be freed at a certain point during the execution.

3. Most memory requests are smaller than some threshold.

ScatterAlloc [Steinberger et al., 2012] and XMalloc [Huang et al., 2010] are using superblocks, a chunk of memory that is allocated for a group of threads. They divide this superblock into several smaller memory chunks that are only accessed by this group. This reduces the number of global memory allocation requests and there is no need for global synchronization when manipulating a superblock. In our approach one superblock is shared by all threads in a warp. To reduce the simultaneous memory requests a voting is performed that determines a so called worker thread. This thread does all the work for his group. Thereby we can reduce the invocations up to SIMD width times.
Our second assumption makes it obsolete to have any header data for the superblock except for one pointer register, which points to the next unoccupied chunk inside the superblock. With this simplification, the time needed to allocate memory inside a superblock is reduced significantly. We further reduce the synchronization and memory overhead introduced by a general `free()` method.
The last assumption ensures that the default allocator is used as little as possible. To guarantee not only good performance in allocating memory but also efficient cache use, we aggregate all memory requests inside a warp.

4.3.1 Data Layout

Similar to the parallel heap used by Hoard [Berger et al., 2000] as well as Häggander and Lundberg [Häggander and Lundberg, 1998] we create a heap per warp accessible for all threads in the corresponding warp. The so called **WarpHeader** organizes all memory requests inside a group of SIMD lanes. Figure 4.1 shows the data layout and how the objects are organized. Each header contains a pointer to the current **SuperBlock** and a pointer to a list (**SuperBlock_List**) that stores all pointers to superblocks that have been allocated using the default memory allocator. The size of the list is fixed. If it is full, it is replaced by a new empty list and the old list is registered in the new list, so that the reference is not lost. **SB_Counter** denotes the number of allocated elements in the list. Besides its memory allocation region, a superblock also stores the amount of allocated memory in **SB_Allocated**.
Two additional variables are stored inside the warpheader for later use. The **TotalCount** describes the number of threads inside a warp which use dynamic memory allocation. The second **ActiveCount**, contains the number of threads that have not finished execution.

4.3.2 Initialization

To initialize the dynamic memory allocator, every thread that needs the ability to request memory dynamically, has to obtain a warpheader. At first all threads inside a warp have to determine how many threads in a warp require the warpheader. This can be realized by a voting function. After that a worker thread has to be declared by using the position of the most significant set bit of the voting mask as the ID of the thread. The worker thread allocates the warpheader using the default memory allocator and distributes the

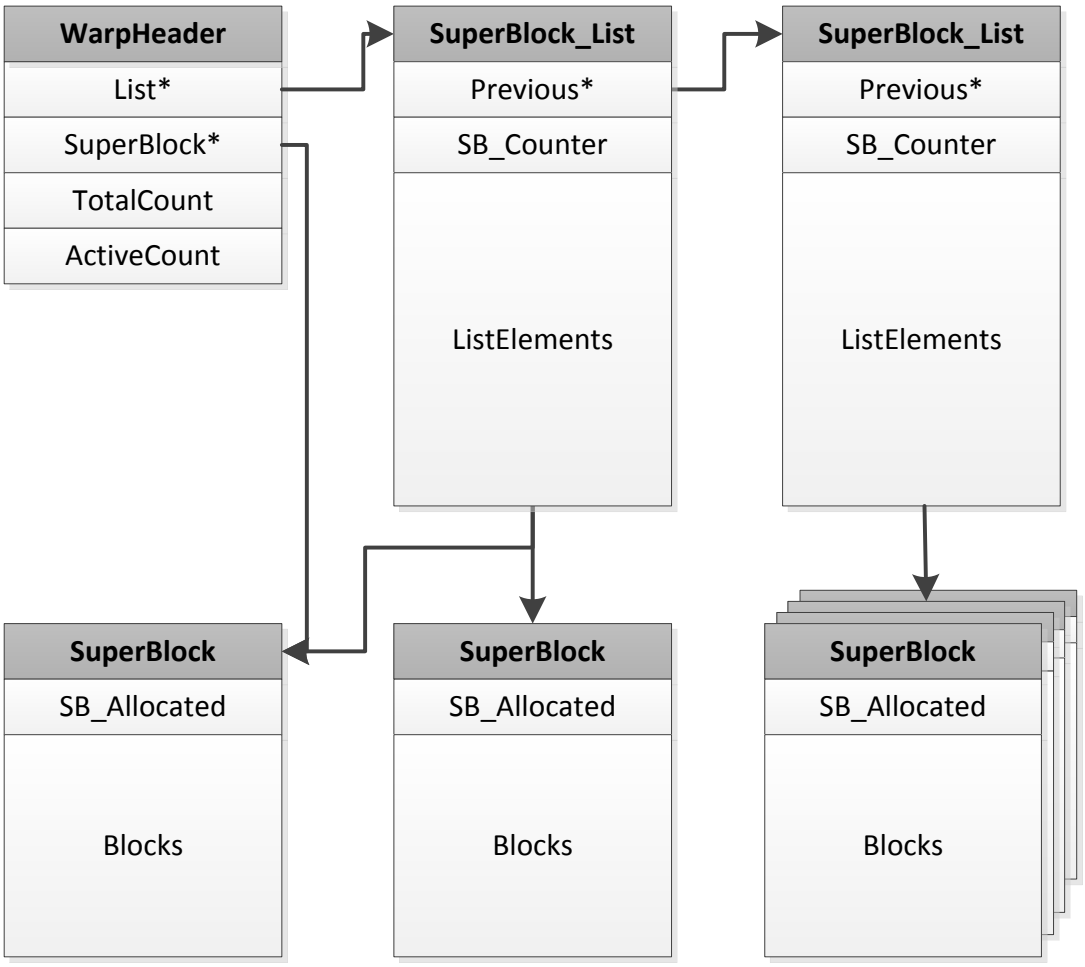

Figure 4.1: Overview of the data layout and organization of the three data structures used by our allocator.

pointer to the other threads. This worker thread also allocates the current superblock and registers it in the list. The algorithm is illustrated in Figure 4.2.

4.3.3 Allocation

The allocation process is divided into two phases (see Figure 4.3). In the first phase, the required memory amount of each requesting thread is mapped to the next multiple of a minimum allocation block size. The default minimum allocation block size is 16 bytes, but it can be adjusted to fit the application needs. Memory can only be allocated in blocks, therefore allocating 17 bytes would result in a 32 byte allocation. This guarantees correct alignment inside memory for better cache reads and writes.
In the second phase, it is checked if the total requested memory size of all requesting threads is smaller than the maximum superblock size. If this is the case, the threads try to allocate memory in the associated superblock. All threads that have not been able to allocate memory in the superblock again perform a voting to decide on a worker thread, which allocates a new superblock and registers it in the superblock list. The remaining

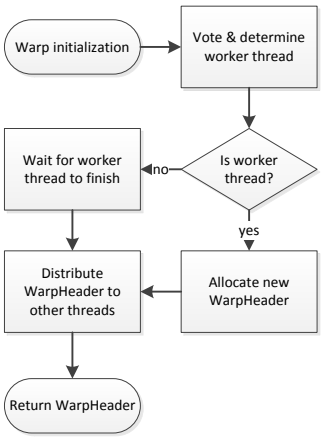

Figure 4.2: Warp execution flow for initialization of the dynamic memory allocator.

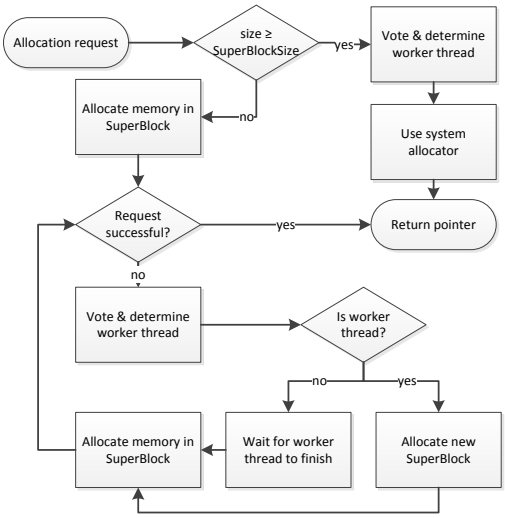

Figure 4.3: Warp execution flow for a memory allocation request.

threads allocate memory in the new superblock.
If the total requested memory size exceeds the maximum superblock size the whole request is served using the default allocator. The returned pointer is registered in the superblock list, so that the developer does not need to know whether the request was handled by a superblock or the default allocator.
As mentioned before, there is no need for any header data in a superblock except for a pointer register, which points to the last used block. To enable coalesced memory access, all requests are mapped to a contiguous memory region. All active threads calculate the sum of requested memory for all other threads up to their own thread ID. This can be done by a prefix sum or a simple iteration. The sum is used as an offset to calculate the thread's own block position in the superblock. The active thread with the highest ID has all information needed to update the allocation status information of the superblock.

4.3.4 Garbage Collection

The proposed memory allocator keeps track of allocated memory using a list. To free these allocations we use three strategies:

1. **Clean all dynamically allocated memory:** A function traverses all lists and frees all stored memory pointers. After this, a new empty superblock will be allocated.

2. **Shutdown dynamic memory allocation:** All allocated memory including the warp header is deallocated. This strategy is meant to be invoked at the end of the kernel, so that all memory is freed correctly and no memory leaks occur. After this function has been executed, it is no longer possible to allocate memory.

3. **Make allocated memory available to other kernels after kernel termination:** In contrast to clean up the complete warpheader at the end of kernel execution a pointer to the warpheader is returned. This allows us to use allocated memory over several kernel calls but it is still required to eventually invoke the previous strategy.

In each strategy the calling threads atomically decrement **ActiveCount**. The thread that reduces the count to zero executes the respective strategy. For the first strategy it additionally resets **ActiveCount** to **TotalCount**. This follows from the assumption that all threads with a warpheader keep the ability to dynamically allocate memory.

4.3.5 Constraints and Limitations

Our proposed allocator design implies a few constraints which we summarize here. These constraints must be fulfilled to guarantee fast and correct operation.

- All constraints of the default allocator are inherited and are still applicable.
- Threads that have requested a WarpHeader eventually must call one of the clean up functions. Otherwise the memory will not be freed as described in Section 4.3.4.
- The used hardware must provide a voting function for an efficient implementation.

4.4 Implementation

We used Nvidia's CUDA Version 5.0 NVIDIA [2015] to implement our proposed allocator (Fast Dynamic GPU Memory Allocator – **FDGMalloc**). The design is kept very generic. This allows the system to be used with most current many-core architectures (e.g., Intel Xeon Phi INTEL [2012]) if the architecture supports a warp voting function (see Section 4.3).

4.4.1 Allocator on GPU using CUDA

As Figures 4.4 and 4.5 illustrate, the CUDA toolkit built-in memory allocation function (**CUDAMalloc**) is fast for large and few simultaneous allocation requests. Therefore, it is used for the allocation of new superblocks. The bottleneck for CUDAMalloc (as well as for most other allocators) is the SIMD scalability. We reduce the amount of concurrent requests and therefore increase the scalability.
To determine a worker thread we use the CUDA voting function *__ballot*, which essentially returns the lane mask for all participating threads when called with a predicate not equal to zero. The corresponding bit of non-participating threads is automatically set to zero. Afterwards we us *bfind* to find the most significant set bit and declare the corresponding thread as the worker thread.
To create a warpheader all threads in a warp use the CUDA voting function *__ballot* to determine how many threads in a warp require the warpheader. The worker thread allocates the warpheader using CUDAMalloc and distributes the pointer to the other threads.
The distribution of the pointer to the warpheader as well as later the pointer to a chunk of memory can be realized in two different ways. For compute capability 2.0 shared memory is used to distribute the pointer to the other threads. If compute capability 3.0 or higher is available, the pointer exchange between threads within a warp is realized using the function *__shfl*, removing the need for shared memory. Performance analysis has shown, that there is no difference in execution time by using shared memory or the *__shfl* function.
As described in Section 4.3.3, all allocation requests that are smaller then a certain threshold are served by a superblock without any atomic operations involved. In this case we

reduce the number of global memory allocation requests and there is no need for global synchronization or thread serialization.

4.4.2 Usage

Listing A.1 in Appendix A shows a simple example of how to use the allocator. All threads request a warpheader, allocate some memory, and free all memory at the end of the kernel. The example demonstrates usage of the *tidyUp()* function, which implements the first strategy for garbage collection. Each thread allocates some memory every time the loop is executed. memory is freed at the end of every loop cycle. After the execution of the warp the warpheader and all other header data is freed.
To prevent memory leaks, the function *end()* has to be executed at the end of the kernel. Otherwise at least one list, one superblock, and the warpheader per warp will not be freed. The example in Listing A.2 shows a misusage of our allocator. Not all threads that allocate memory have also requested the warpheader. Further not all threads that requested the warpheader also exectue the *end()* function. As a consequence, the warpheader will not be freed.

4.5 Evaluation

All experiments were performed on a PC running Windows 7 64-bit version and Nvidia driver (version 306.97). The system is equipped with an Intel Core i7 920, 12 gigabytes of RAM, an Nvidia Geforce GTX 480 (primary device), and a GTX 680 with 2 gigabytes of RAM (headless device). All tests were performed on the Geforce GTX 680.

4.5.1 Different allocation sizes

First, we compare the proposed allocator with the default CUDA allocator (CUDAMalloc) and ScatterAlloc [Steinberger et al., 2012].
In the following test scenario X threads are created which allocate N times S bytes of memory. At the end all memory is freed. For FDGMalloc this is done by invoking the *end()* method. In the case of CUDAMalloc and ScatterAlloc it has to be done by hand. We measure the runtime performance by allocating memory chunks of different sizes. Therefore the parameters for the test scenario have been set to $X = \{64\}$, $N = \{16\}$ and $S = \{16, 32, 64, \ldots, 8160, 8176, 8192\}$. The size of a superblock is 32 kilobytes and the minimum allocation block size is 16 bytes. The list of a warpheader contains 126 entries to manage superblocks before allocating a larger list. To analyze the impact of FDGMalloc's allocation strategy and the usage of superblocks we implemented two different versions. One uses superblocks, the other one gathers all requests and allocates memory directly using the CUDA memory allocator.
Figure 4.4 shows the time needed to allocate S bytes. While CUDAMalloc and ScatterAlloc have to synchronize to serve the alloc request for each thread in a warp, we use the implicit synchronization between the threads. The chosen thread performs the request just by a simple atomic operation and propagates the pointer to the other participating threads. The graph shows that our allocator requires always the same amount of time to allocate a chunk of memory. Allocation time increases linearly because the operation needs more superblocks with growing memory consumption.

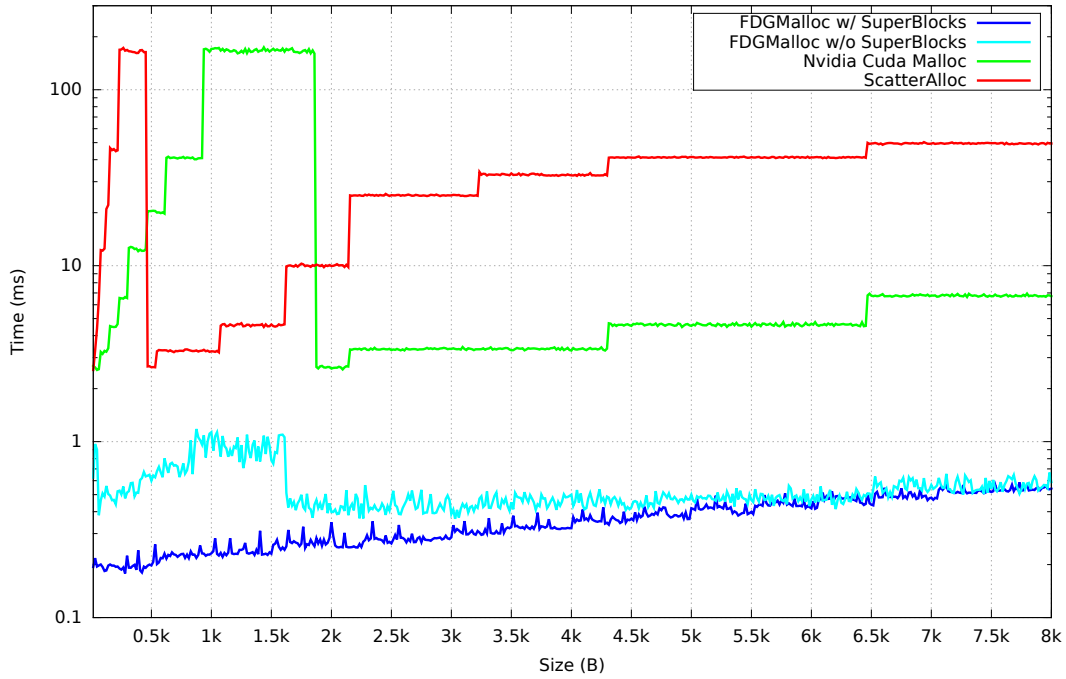

Figure 4.4: Performance Comparison of our proposed FDGMalloc, CUDAMalloc[NVIDIA, 2015] and ScatterAlloc[Steinberger et al., 2012].

4.5.2 Varying thread count

In the second evaluation test case we vary the number of threads allocating a memory chunk. Because of the different amount of time needed to allocate different sizes of memory chunks the count and size of the allocations has been varied. In the end the mean of all results for one thread count has been calculated. The limited GPU Memory does not allow us to perform all tests with all combinations.

The values used for the tests have been X = {1, 2, 4, ..., 16384, 32768, 65536}, N = {16, 32, 64, 128, 256, 512} and S = {16, 32, 64, 128, 256, 512}. Figure 4.5 shows the results of this comparison as absolute (left figure) and relative (right figure, relative to FDGMalloc) values.

Since we allocate the first superblock during the initial phase our proposed FDGMalloc with superblocks is nearly ten times faster at the beginning than any other measured allocator. The speedup between both FDGMalloc (blue and light blue lines in Figure 4.5) is in the range of 10 to 300 depending on the number of threads requesting memory simultaneously. Here, one can clearly see the benefit of superblocks.

Comparing the default CUDA allocator (green line) with FDGMalloc without superblocks the gained speedup is related to the used voting function and reduction of concurrent memory requests. With a small number of threads (one to three) simultaneously requesting memory CUDAMalloc is faster. With more concurrent allocations the synchronization becomes a severe bottleneck.

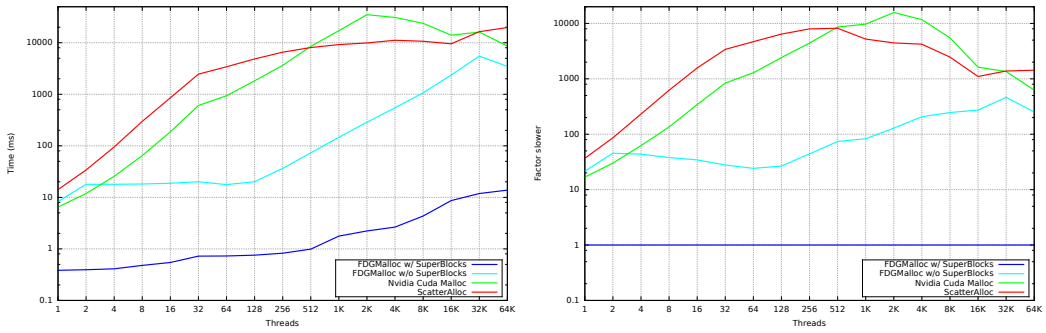

Figure 4.5: The figures show the performance comparison under varying thread count. On the left side with absolute values in milliseconds and the slowdown relative to FDG-Malloc on the right.

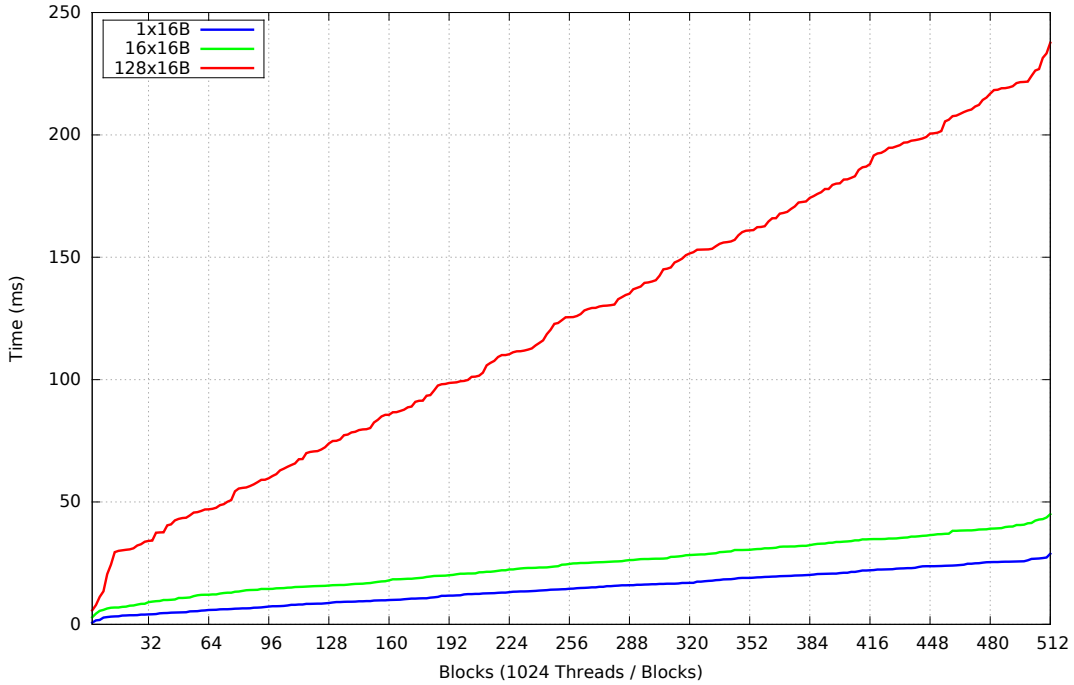

Figure 4.6: The chart shows a comparison of different allocation counts over the number of blocks using FDGMalloc with superblocks.

4.5.3 Scalability

For the scalability evaluation of FDGMalloc with superblocks we used 1 to 512 blocks with 1024 threads each, which allocate 1, 16 and 128 times 16 bytes of memory ($X = 1024*[1,512], N = \{1,16,128\}, S = 16$). Our results shown in Figure 4.6 clearly indicate the linear scaling of our allocator depending on the number of active threads. The bigger gradient in the range from 1 to 16 blocks is probably caused by hardware constrains. The GTX 680 has 8 multiprocessors which can handle up to 2 blocks with 1024 threads at once. With more blocks scheduled, the multiprocessors are able to hide the waiting time of a block by executing another block in the meantime.

4.6 Conclusion

We presented a dynamic memory allocator for many-core architectures with a wide SIMD width. Frequent and concurrent requests are reduced and handled efficiently by a voting function in combination with a fast allocation inside a superblock. The performance evaluations have shown that our proposed allocator is able to speed up dynamic memory allocation by several orders of magnitude although it relies on the CUDA allocator. All in all we increase the SIMD scalability significantly for frequent dynamic memory allocations. The implementation shows that concurrent dynamic memory allocations in massively parallel architectures do not need to be slow. However, the assumption of the allocator do not allow it to be an all-round solution. It is still necessary to improve dynamic memory allocation schemes that allow memory to be arbitrarily freed during the execution. Our implementation is based on CUDA but could be extended to any of the other hardware architectures supporting a voting function. In the future we would like to evaluate the performance using Intel's SPMD Program Compiler (ISPC) using AVX vector units and Xeon Phi.

Chapter 5

Dynamic Load Balancing and Fair Scheduling for GPU Clusters

Abstract

Despite the tremendous compute power of modern GPUs, many compute-intensive problems cannot be solved on a single GPU in ample time. GPU clusters can handle the increasing demand of compute power but require efficient ways to execute the concurrent tasks submitted by the users. We present a dynamic load balancing approach and apply a fair scheduling scheme to minimize the average execution time of spawned computational tasks. This is especially important if multiple tasks, applications, and/or users access the cluster concurrently. We demonstrate our approach using a CUDA-based framework supporting fine grained task parallelism and transparent distribution to cluster nodes and analyze the performance on a GPU cluster with eight multi-GPU nodes. The proposed system is evaluated using two example applications running concurrently including a bidirectional path tracer running entirely on the GPUs. The load balancing capabilities are shown by the irregular workload of the bidirectional path tracer using different scenes.

5.1 Introduction

Even with current hardware, many compute-intensive applications cannot be executed with sufficient performance on a single graphics processing unit. Examples cover a wide range of applications from graphics and simulation to domain specific applications. The reasons for this fact are not only the high demand for computational power but also the irregular work distribution and in many cases the requirement to support interactive manipulations of a system (e.g., when navigating through a scene rendered by a bidirectional path tracer requiring interactive frame rates).
One way to address this problem are GPU clusters. A GPU cluster in the classical sense is a set of workstations with identical hardware (including identical GPUs) connected via low latency and high bandwidth links. Such clusters can either be programmed by adapting traditional frameworks for distributed computing such as the message passing interface (MPI) or by developing specialized solutions for GPUs [Fan et al., 2008, Frey and Ertl, 2010, Müller et al., 2009, Strengert et al., 2008]. The latter have the advantage that they can be easily tailored to the peculiarities of GPU computing. However, to our knowledge none of the existing systems support dynamic load balancing and task stealing, two concepts required to handle irregular workload, and to take the possibly varying performance of the GPUs in the cluster into account. Likewise, none of the existing systems contains a fair scheduling scheme which is essential to equally share the available resources when allowing concurrent access by multiple applications or users to the cluster while minimizing the average execution time for each application.
We present an approach that integrates a set of workstations equipped with potentially different GPUs (hardware generations, compute capabilities), ranging from single to multi-GPU systems, into a loosely coupled GPU cluster. The critical part in such a heterogeneous environment is efficiently using the available resources and balancing irregular workload under the aspect of concurrent and interactive usage from different programs. Our approach addresses these issues. It uses the resources in a possibly heterogeneous GPU compute cluster in an efficient way by applying the task parallel and data parallel programming model to decompose a problem into a number of smaller independent chunks. Interactivity is provided on the cluster level by the scheduling and load balancing schemes. These are distributed and run in parallel on different GPUs. Our contributions are

- a CUDA-based cluster approach for efficient and transparent distribution of data and compute jobs onto a heterogeneous GPU cluster,
- fair scheduling of jobs and dynamic load balancing including task stealing using a two level hierarchical scheduling system for concurrent and interactive access to the cluster,
- an extensive evaluation using two applications with different characteristics: mutual information computation and a bidirectional path tracer implemented completely on the GPU with highly irregular workload.

5.2 Related work

The increasing computational resources of GPUs have been used for quite some time for general purpose computations. To facilitate programming, specialized programming envi-

ronments, such as Brook [Buck et al., 2004] which uses the GPU as a streaming processor, were developed. Modern high-level languages for general purpose GPU programming include ATI's Stream [ATI, 2007], Nvidia's CUDA [NVIDIA, 2015], and the cross-platform OpenCL standard [Khronos Group]. In the following, we first discuss related work on GPU clusters and then focus on dynamic load balancing of GPU workload, a key component of our system.

5.2.1 GPU Cluster Computing

Zippy [Fan et al., 2008] was the first framework using GPUs in a cluster environment. It abstracts the complexity of GPU cluster programming via a two-level parallelism hierarchy using global arrays. Computation is performed in a classical GPGPU way using shaders and (possibly multiple) rendering passes.

In contrast, CUDASA [Strengert et al., 2008] extends the CUDA C programming language by additional keywords to support multi-GPU systems and GPU-cluster environments. The extensions are handled by an independent compiler. The underlying data sharing between different GPU nodes is expressed by the application programmer using the language extensions. To minimize the communication overhead, a data locality aware static scheduling mechanism was later added by Müller et al. [2009]. The PaTraCo (Parallel Transparent Computation) [Frey and Ertl, 2010] framework distributes parallel applications to different kinds of compute resources (e.g., CPU, GPU, and Cell). They use a critical path analysis as scheduling method to minimize the overall execution time. This takes device speed, availability, and transfer cost statically into account, i.e., *before* the execution of the device specific kernel implementation.

The middleware rCUDA [Duato et al., 2010] virtualizes a CUDA compatible GPU and therefore enables remote computation when no capable device is installed locally. The client-side wraps the CUDA API calls and uses remote procedure calls to configure and launch kernel functions on the compute server.

Based on the MOSIX cluster management system the Virtual OpenCL (VCL) cluster platform [Barak and Shiloh, 2011] is an implementation of the OpenCL standard that allows unmodified OpenCL applications to transparently use many devices in a cluster. Similar to rCUDA the VCL framework virtualizes OpenCL capable devices for the application. The communication between application and the back-end daemon is handled by the VCL broker, a daemon running on the host-side. The broker allocates and monitors devices for a running application.

Sequoia++ [Fatahalian et al., 2006] is a programming language for writing portable parallel programs. The language exposes the underlying structure of the memory hierarchy to programmers in an abstract manner to ensure portability across a wide variety of contemporary machines. The compiler maps the memory hierarchy levels to host memory, GPU device memory, threadblocks and threads using the Sequoia GPU backend. The Sequoia runtime environment and programming model can exploit clusters by using MPI.

Note that traditional frameworks for distributed computing such as MPI can use GPUs as well. This is, e.g., used in CUDASA [Strengert et al., 2008], rCUDA [Duato et al., 2010] and Sequoia++ [Fatahalian et al., 2006]. One of the key differences between our system and MPI driven applications is, however, that in our case jobs are transparently distributed over the network whereas in the case of MPI the user has to model the data sharing between different cluster nodes. Likewise, grid batch computing frameworks (e.g.,

BOINC [Anderson, 2004]) can make use of underlying GPUs but typically focus on large scale computations with completely independent tasks (e.g., SETI@home [Anderson et al., 2002]).
In contrast to rCUDA and VCL we employ a two level scheduling scheme and work stealing over all compute servers to handle irregular workload. Furthermore we allow concurrent access to the compute resources and therefore use a fair scheduling scheme to avoid starvations of running applications.

5.2.2 Dynamic Load Balancing

Typical cluster environments use a batch scheduling system allocating user defined resources if and when available. To increase resource utilization, load balancing algorithms try to distribute workload according to the available heterogeneous resources. These algorithms can be classified into two main categories: static and dynamic load balancing. Static load balancing algorithms allocate tasks based on information known at compile time (e.g., resource consumption). In contrast, scheduling decisions for dynamic load balancing are made at runtime using online information about each cluster node [Casavant and Kuhl, 1988, Zaki et al., 1996]. Allowing concurrent access to cluster resources additionally requires a fair scheduling scheme which assigns the available GPUs to the different applications, ensuring that all applications get on average an equal share of computation time.
The out-of-core data management layer for path tracing on heterogeneous architectures presented by Budge et al. [2009] applied an on-demand scheme for load balancing where each consumer prioritizes the available tasks in the kernel queues and selects the highest task allowed. To prioritize the workload, a cost-driven model is employed where the task is evaluated according to its workload and memory transfer. They claim without further details that the system can run on multiple cluster nodes.
In our work we use a centralized scheduler which distributes the work equally in a heterogeneous cluster, optionally weighting the workload based on a performance metric of the resource. On each cluster node, we adopt the fair-share and lottery scheduling scheme to allow for concurrent access to the GPUs from different applications. To handle the irregular workload in an appropriate way we incorporate work stealing as presented by Blumofe and Leiserson [1994] when resources are idle. We focus on dynamic load balancing without any prior knowledge of the work distribution and irregularity of an application. But we are aware of the many application specific load balancing algorithms and therefore add support for user specified scheduling to exploit the knowledge.

5.3 Overview

In our approach, we use three layers as illustrated in Figure 5.2: the application, the service, and the computation layer. The application layer creates and distributes new workload. The service layer performs resource allocation and load balancing. The computation layer executes the scheduled jobs independently. In this section, we give an overview over the task and job decomposition used to enable parallel execution of applications on the cluster. Details about the layer structure, the job distribution, and the algorithms used for load balancing, scheduling, and work stealing will be discussed in the next section.

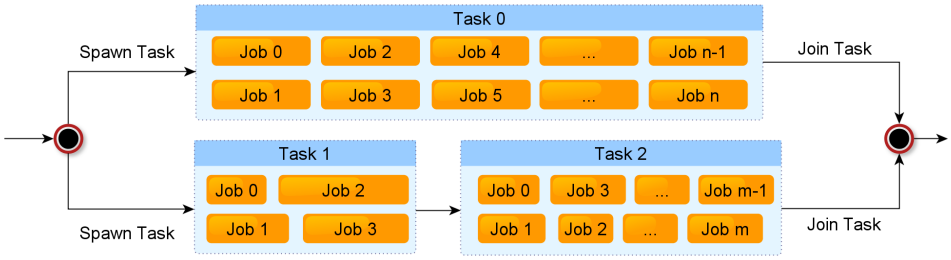

Figure 5.1: Possible decomposition of an application into a set of dependent tasks (Task 1 and Task 2) and jobs. The jobs spawned by each task have to be independent from each other. Dependencies can be modeled as tasks as shown in the diagram.

5.3.1 Task and Job Decomposition

To enable a flexible decomposition of applications, we support two levels of parallelism (besides the obvious SIMT parallelism inherent in the execution of a CUDA kernel). At the coarse grained level, the application creates a set of concurrent, mutually dependent or independent *tasks*, each running in a separate CPU thread of the application. As illustrated in Figure 5.1, the dependency graph between tasks forms a general directed acyclic graph (DAG). Each task consists of several independent *jobs*, forming the second level of parallelism. We typically use a global array approach assigning each job in a task a unique identifier yielding a data decomposition. Other decompositions such as a second level of task decomposition are, however, possible as well.

5.3.2 Programming Model

Our goal is to minimize the changes (resp. the programming overhead) when porting an existing CUDA application or implementing a new application on the cluster. The Listings B.1 and B.2 in Appendix B, excerpts from our bidirectional path tracer (see Section 5.5.2), illustrate the programming interface. The interface to enqueue allocations, kernel configurations, and the launch environment is therefor identical to the Nvidia CUDA Driver API. The exceptions are that some calls receive a kernel ID as an additional argument, in our examples **m_rndFunc**, and the changed prefix **cl** of all keywords. Porting an existing CUDA application to the cluster environment is therefore straightforward.
A new *task* has to implement a virtual method from its base class that initializes a batch of *jobs*. In addition a second method can be provided that will spawn new jobs after the initial set has been finished and the result was fetched, working on the same dataset. This method is essential for our mutual information computation example (see Section 5.5.1) and can be used to further improve the result of a task, e.g., tracing additional paths in the refinement step of a path tracer (see Section 5.5.2). In short the task schedules, dispatches, and retrieves the results of all jobs as described in Section 5.4.2.
On the *job* scope of our decomposition scheme the user has to reimplement two virtual methods, **dispatch()** and **retrieve()**. These methods, shown in Listings B.1 and B.2, are called automatically when a job is dispatched and when the result of the computation is passed on, respectively. The dispatch method generates a proxy object which is implemented as a command queue. It is identified by a unique job identification number, generated by the parent task, given as a parameter and represents the CUDA allocations,

kernel configurations, and an arbitrary number of dependent kernel calls. This proxy object contains all information to execute a given CUDA kernel. Once the dispatch method has flushed the command queue, it is automatically serialized, scheduled, and sent to a compute node with matching compute capabilities.
Allocations can either have task or job scope. A *task allocation* will be fetched by the compute node and cached afterwards to minimize network traffic and overhead. These allocations can be used for constant datasets such as triangle meshes and will be shared by all jobs over the lifetime of the task. CUDA kernel source files are handled like task allocations and are also cached. A *job allocation* is used for varying, job dependent data and is not cached. Incoming results are fetched by the second method reimplemented by the application programmer.
Each job instance is executed by a separate CPU thread in every stage of execution (schedule, distribute, and receive result). After issuing a stage change the job releases the thread. The underlying CPU thread is reissued for a new job. To avoid starvation due to too many active threads we use a thread pool assigning free hardware threads. To maximize the scalability of the approach we reduce the need for synchronization by employing scalable memory pools and lock- or wait-free data structures if applicable.

5.4 Architecture

In this section, we give an overview over the basic architecture and take a look at the layer structure, the job distribution and the algorithms used for load balancing, scheduling, and work stealing. For an in-depth overview over distributed systems and a general discussion of the relevant issues, we refer the reader to basic text books such as "Distributed Systems: Principles and Paradigms" [Tanenbaum and van Steen, 2006].

5.4.1 Structure

With our approach, we allow for interactive applications using our loosely coupled heterogeneous GPU resources concurrently as a compute cluster. Essential for the proposed system is the scheduling of irregular, unpredictable workload and concurrent access of different applications with different and potentially varying resource requirements. The system is designed with a three-tier client server architecture in mind. The application layer requests a scheduling plan from the service layer. It then sends compute jobs to computation nodes and receives the result directly as shown in Figure 5.2.

Application Layer

Compute jobs are created and managed by the application as described in Section 5.3.1 in a way that is completely transparent to the user. We enforce that the compute jobs of a particular task must be independent of each other and thus do not need to model data sharing between compute nodes as in MPI systems. This allows us to completely hide the complexity of the network communication from the programmer. In practice, this leads to a potentially larger communication overhead which can be reduced as outlined in Section 5.4.2. To distribute programs to the compute nodes, the system uses the Parallel Thread Execution Assembly Language Files (PTX files) as an intermediate representation of the CUDA kernels. Since PTX is platform independent and only later compiled for particular

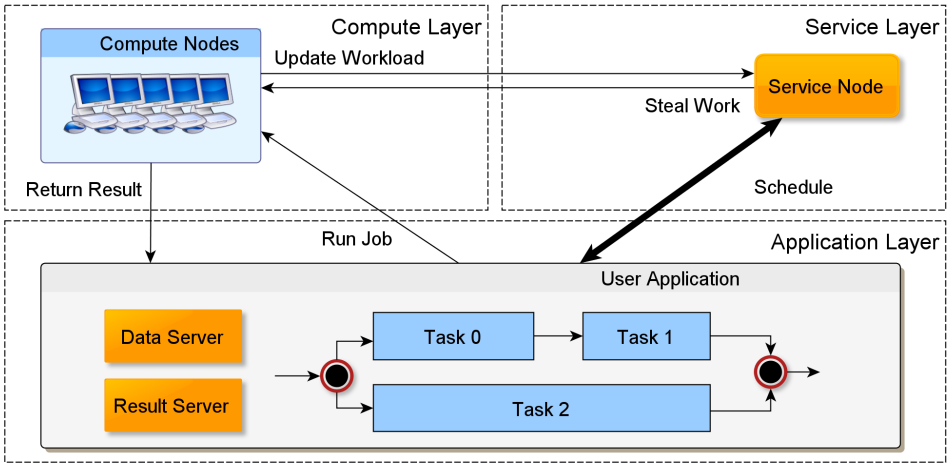

Figure 5.2: System overview. The three-tier client server structure is composed of the user applications, a service node handling scheduling and work stealing, and a set of compute nodes which perform the actual computation on their GPUs.

hardware, this allows us to use different compute capabilities and types of cluster nodes for the computation and permits the nodes to run on either Linux, Windows, or MacOS. It is therefore possible to not only use a dedicated cluster but also collect suitable machines from an office environment for computation regardless of its operating system.
The application automatically creates a data server thread which is used to answer requests for missing task allocations and CUDA kernel files. After a computation is completed the result is sent back to the application's result server as shown in Figure 5.2. Although data and result server could theoretically be merged into a single server, we separated the two to increase scalability (see Section 5.6.1). In addition, all jobs are executed in parallel to decouple the requests and allow for asynchronous modification of the result. This approach can lead to a high memory usage during job distribution, which is circumvented by splitting the distribution process into batches.

Service Layer

We use a centralistic scheduling approach to allocate workload to the compute nodes. The service node governs all compute nodes and schedules incoming computation requests. Handling too many requests at the same time can unfortunately lead to delays and decrease the scalability of the cluster system. We therefore batch the requests from the application and employ a second scheduling level directly at the compute nodes to minimize network traffic and overhead. Each application is registered at the service node and receives a cluster wide unique ID used for caching task allocations and kernel sources. During the registration process a specific service thread is generated that contains a subset of compute nodes suitable to fulfill all application demands and runs the desired scheduler. The set of available compute nodes is restrained by the available CUDA compute capability and the required GPU memory. As we want to incorporate office workstations into our cluster, we need to be able to add and remove GPU compute nodes at runtime. Therefore new nodes can be registered dynamically at the service node. If an application is running while a node is added, work stealing as shown in Section 5.4.3 is applied to make use of

the newly available compute capacities. If a compute node is removed, its remaining jobs are rescheduled.

Computation Layer

Nodes at the computation layer execute job requests. The first stage during the process is to fetch all missing allocations and kernel PTX-files from the application and to add them to the on-GPU caching facility. Task allocations and compiled CUDA modules are cached using a least recently used scheme and shared across the jobs of the same application within a CUDA context. The CUDA kernel is compiled via the CUDA JIT compiler using a target profile specifying the minimal compilation target. After a compute job is created from the serialized proxy object (Section 5.3.1) it will be enqueued in a work-queue assigned to the scheduled GPU device. At the end of the kernel execution the result is sent back to the application. During the initialization of each device we measures the floating point operations per second using the CUDA BLAS library and parse the capabilities of the current GPU. The amount of floating point operations per seconds is used by the scheduler as a performance heuristic for scoring the current device.

5.4.2 Distribution

In our approach the distribution of jobs is application centric (as stated in the previous section). This can lead to a large communication overhead and a high number of messages between the nodes of each layer. To tackle this drawback we propose batching the requests between the application and the service node as well as the application and the computation layer. A task generates a batch of compute jobs and requests one scheduling plan for the complete set of compute jobs. The scheduling plan is batched by compute nodes and GPU devices per node. This reduces the communication between the service node and the application to a minimum.
During the execution of the scheduling request the application generates proxy objects for the compute nodes in parallel. To limit the memory consumption we only handle a batch of proxy objects at a time. After the scheduling result was received, the application aggregates all proxy objects per compute node and GPU to minimize the network communication overhead at this point (more specifically the number of messages) when spawning hundreds of jobs at the same time. These sets are then sent to each node. At the compute node the packet of jobs is split up and added to the work-queues. The result of a compute job will be sent to the application immediately and the compute node notifies the service about the completion of the job. The scalability, communication overhead and performance impact of the batching will be discussed in Sections 5.6.1 and 5.6.2.

5.4.3 Dynamic Load Balancing

As our goal is to use the system in an interactive, concurrent manner for different applications with varying resource utilization, we require an efficient allocation of the available resources. This is particularly important when assigning GPU devices because the execution of CUDA kernel is generally not preemptive. To avoid starvation of fast compute nodes, to minimize response time, and to maximize throughput we propose an efficient multi-level load balancing system. We distinguish between long term scheduling decisions

established by a global scheduler and short term scheduling decisions initiated by a local scheduler.

- **Global scheduling** decisions are based on the application requirements, e.g. the minimal compute capabilities needed, the overall system load and resource usage. The scheduling layer maximizes the overall utilization of the given cluster resources. Therefore we utilize work stealing to handle irregular workload and differences in compute performance between cluster nodes.
- **Local scheduling** decisions are based on a fair scheduling scheme employed by each compute node to avoid starvation of an application spawning less compute jobs than others and ensures concurrent access of jobs from different applications.

Global Scheduler

The global scheduler, which is part of a service node, distributes the incoming computation requests based on the workload of each node in the computation layer and the application's requirements. The default behavior is a First-Come First-Served strategy. But since there exist more suitable load balancing algorithms for various problems, we offer the possibility to change the algorithm by injecting a new implementation via a shared object file for each application.
A possible implementation of such a system in the case of a tile based bidirectional path tracer (see Section 5.5.2 for more details) is a scheduler which takes the temporal coherence during camera movement into account. The part of a scene rendered by a tile typically does not change significantly. The execution time of the tile is therefore similar to the execution time in the last frame. Using this information in the scheduling process can lead to a better initial distribution of the tiles. Another approach would be a data distribution-sensitive scheduler for a finite element method.

Local Scheduler

For the local scheduling at the computation layer we use a two-level scheduling strategy. A priority queue is created for every application assigning a job to the compute node. If no priority is assigned by the global scheduler the job will be executed in a First-Come First-Served manner.
Since CUDA does not allow to interrupt running jobs, we implemented a lottery scheduler to randomly select a queue from which the next job is taken based on a weighted fair-share scheduling scheme [Kay and Lauder, 1988]. We modify the probability of a queue being chosen depending on the last selection, i.e., for a selected queue the probability will be decreased and the probability for every other queue increased. If a new application is assigned to the node the probabilities are re-weighted. The goal of the local scheduler is to minimize the average execution time of an application when sharing concurrent resources with other applications.

Work Stealing

Unpredictable workload, e.g., the irregular work generated by a bidirectional path tracer as described in Section 5.5.2, can reduce the overall utilization of the cluster resources and increase the execution time for all programs using the cluster environment. In particular, it

can lengthen the response time for interactive applications. To handle such unpredictable workload we integrated two different work stealing mechanisms in the multi-level scheduler hierarchy.
At the computation layer, a CUDA device that has finished a compute job and has an empty device work queue just steals a job from another compute device of the same node (assuming that there are multiple GPUs on the node and/or multiple compute devices per GPU). If all allocations and CUDA kernels are already cached on the node, we start the execution immediately. This approach at the lowest hierarchy level allows us to efficiently utilize a multi-GPU system and to significantly reduce the network traffic. If a computation node is running out of jobs, the global scheduler will reschedule a compute job from another node to maximize the overall utilization of the cluster.
In Sections 5.6.3 and 5.6.4, we will evaluate the proposed load balancing schemes using our two example applications.

5.5 Applications

We evaluate our cluster framework using two applications with different characteristics: distributed mutual information computation as a standard benchmark and ray tracing of a complex scene with global illumination effects using bidirectional path tracing.

5.5.1 Mutual Information

In the recent past, genome sequencing hardware became widely affordable, which led to the availability of large genome sequence as well as protein sequence datasets. These datasets can be analyzed for structures that are co-evolving among individuals, which can in turn be used for structural and functional protein analysis [Martin et al., 2005]. One way of analysis is the computation of Mutual Information (MI) among protein sequence positions.
The datasets being used should be normalized due to finite-size effects. This can be done in a stochastic manner by iteratively (usually for around 10,000 iterations) shuffling the sequence set and comparing the "true" MI to the MI computed from the shuffled versions [Hamacher, 2008, Weil et al., 2009]. This so called shuffling null-model is highly compute intensive. Computation can take on the order of days or even weeks for typical dataset sizes on a CPU.
Since MI computation as well as shuffling scale well even for a large number of processing elements, the MI computation was recently ported to the GPU, achieving speed-ups between 10 and 20 compared to state-of-the-art multi-core CPUs [Waechter et al., 2012].
We ported this system to our cluster framework. The whole shuffling null-model computation is a single task. Each iteration of the shuffling null-model (which includes shuffling the sequence set and computing the MI from the shuffled version) is a single job. The jobs are then distributed among the cluster nodes by our framework. The size of the result of a single job is quadratic in the length of a sequence, which results in sizes ranging from 5MB to 100MB for typical datasets. This forces us to dispatch iterations in batches with size depending on sequence length to reduce the application side memory footprint by using the task mechanism to spawn additional jobs. The runtime of a single kernel increases linearly with the number of sequences and quadratically with sequence length.

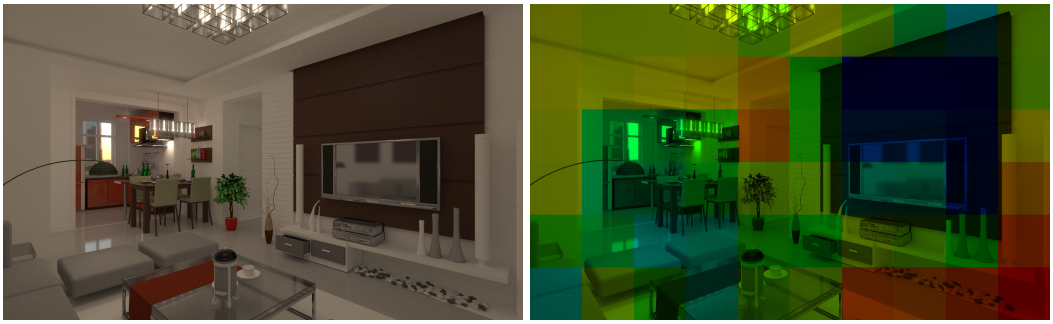

Figure 5.3: Image of a 478k triangle living room scene rendered with our bidirectional path tracer in about 250 seconds on a cluster with 32 GPUs. Resolution is 1920 by 1152 with 4096 samples per pixel which results in 36 million samples per second for the cluster (1.12 million samples per second per GPU). The distribution of the computation time per tile is shown in the heat map, ranging from 36 seconds (blue) to 76 seconds (red).

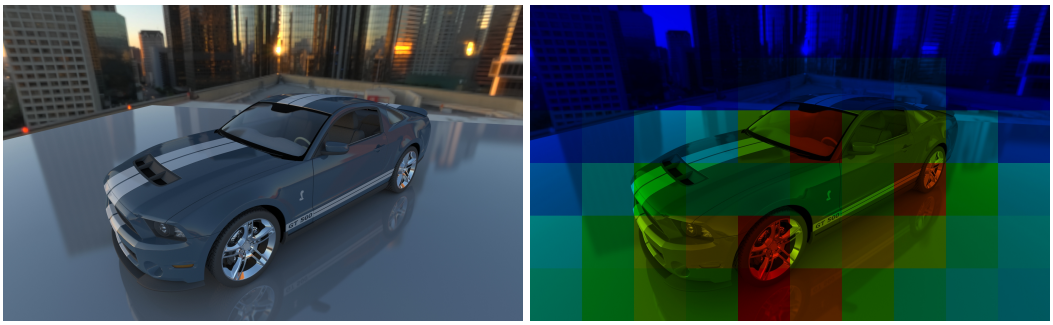

Figure 5.4: Image of a 855k triangle mustang scene rendered with our bidirectional path tracer in about 171 seconds on a cluster with 32 GPUs. Resolution is 1920 by 1152 with 16,384 samples per pixel which results in about 212 million samples per second for the cluster (6.6 million samples per second per GPU) including all possible overhead. The heat map shows the distribution of the tile computation times for the given view. From blue to red computation times range from 38 seconds to 88 seconds.

5.5.2 Bidirectional Path Tracing

Bidirectional path tracing is a global illumination technique concurrently introduced by Lafortune and Willems [1993] and Veach [Veach, 1998, Veach and Guibas, 1994]. The key idea of the approach is to first trace partial paths both from the camera and the light sources into the scene. In a second step, the resulting sub-paths are combined, requiring an intersection test and the evaluation of the appropriate reflectance functions before a contribution is added to the corresponding image pixel.

We implemented a Veach style bidirectional path tracer for triangulated scenes that completely runs on GPUs. Acceleration structure building and rendering is done in two separate tasks. We use a bounding volume hierarchy (BVH) with branching factor two and spatial splits [Stich et al., 2009] for the ray/scene intersections. The structure is build on the CPU.

The rendering task partitions the image to be rendered into tiles which are processed as rendering jobs by the cluster. A rendering job launches a single CUDA uberkernel

which performs all bidirectional path tracing calculations from construction of eye and light paths via importance sampling and ray traversal, to path combination and evaluation with multiple importance sampling (*MIS*). We employ the recursive MIS computation scheme from van Antwerpen [2011] to efficiently parallelize calculations of path combinations and MIS-weights. To bound the possibly exploding number of combinations, eye- and light-subpath length are limited to a maximum of 16 each. For shading we use the bidirectional scattering distribution function and importance sampling techniques from Walter et al. [2007], which allowed us to extend the Ashikhmin-Shirley bidirectional reflectance distribution function by a bidirectional transmittance distribution function for rough surfaces. To increase SIMD efficiency for incoherent rays we employ the persistent-while-while kernel design from Aila and Laine [2009] for ray traversal. As BVH nodes are accessed much more frequently than geometry during traversal, we fetch them directly from cached global memory and read geometry from textures in order to fit as much BVH nodes into L1 cache as possible.
After an initial image has been rendered we can improve the result by adaptively adding more samples to the image. Therefore we determine the quality of every pixel in terms of the Renyi entropy [Xu et al., 2005] on application side. Then we extract a list with pixel positions of the first P percent of all pixels with the worst quality by fast sorting via the Thrust library [Hoberock and Bell, 2010]. The cluster processes batches of this list by spawning additional refinement jobs.

Irregular Work

The two main sources of irregular workload are non-deterministic path lengths and ray traversal. Path lengths are non-deterministic due to termination of path extension via russian roulette. The more energy is absorbed per path bounce the more likely it will be terminated sooner. Partitioning of the image into tiles leads to grouping of primary rays that hit surfaces with the same material. Tiles covering more absorbing areas of the scene can produce shorter eye-subpaths than tiles covering highly reflective areas. The length of light-subpaths is independent of the tile they belong to. This way the average number of combinations per tile varies between tiles. Ray traversal performance depends heavily on the coherence of a batch of processed rays. Tiles covering specular, or at least highly glossy areas can keep rays coherent for a longer time during path construction, while coherence is lost much faster for tiles covering diffuse areas.
Thus association of cluster jobs with image tiles leads to highly varying computation times, which motivates the need for load balancing. Figures 5.3 and 5.4 show the varying computation times of each tile for two example scenes. For both scenes the distribution of tile computation times is depicted in Figure 5.5.
To save network bandwidth, compute nodes store the scene geometry and acceleration structure as static allocations. This way a compute node can process an undeterministic amount of render and refinement jobs without the need to refetch the scene, which is necessary for efficient dynamic load balancing.

5.6 Evaluation

We evaluated the proposed architecture under two aspects, scaling behavior and load balancing. The main experiments were performed in a homogenous hardware environment,

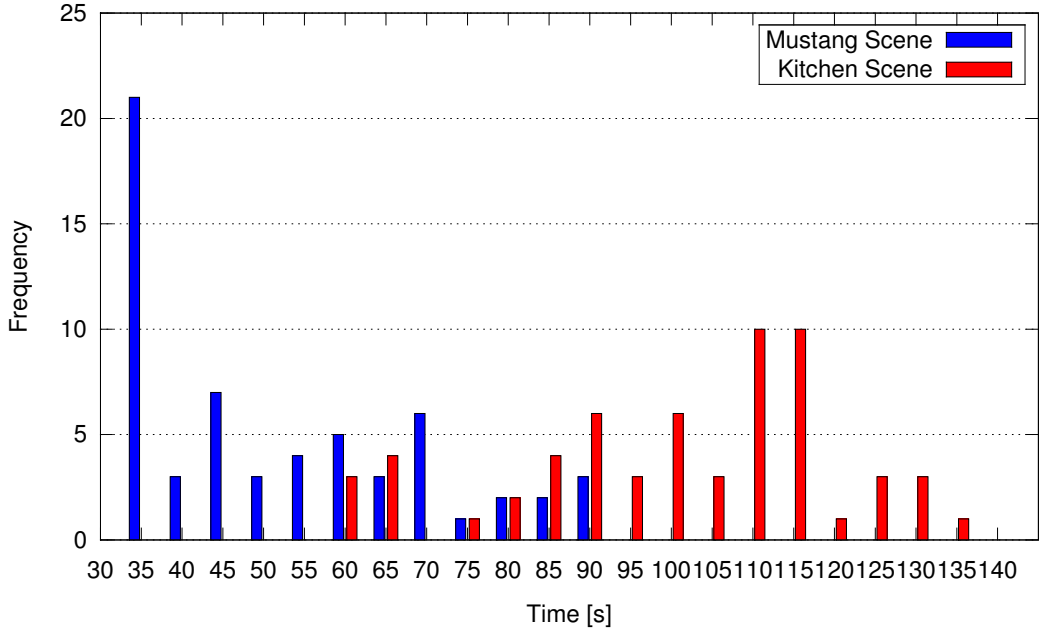

Figure 5.5: Histogram over all tile computation times for both scenes, illustrating the irregularity of bidirectional path tracing.

a cluster of PCs running Linux with NVidia drivers version 290.10. Each node is equipped with two Intel Xeon DP E5649 hexacore CPUs, 48 GB of RAM, and two NVidia Geforce GTX 590 (each consisting of two GPUs). All nodes are connected over Gigabit Ethernet and 4x QDR Infiniband. For Ethernet connections we use the UDP-based Data Transport (UDT) protocol [Gu et al., 2004], a reliable UDP based application level data transport protocol. The Infiniband layer relies on the Sockets Direct Protocol (SDP). The cluster achieves 20.8 TFlops for single-precision floats, measured by the CUDA BLAS library using a matrix multiplication.
For the heterogeneous setup we attached a wide range of different compute nodes to the homogenous cluster nodes. These nodes are using Gigabit Ethernet only and are located in a typical office environment. The application runs for all test settings on the head node of the cluster. The node is equipped with a Intel Core i7 970 CPU, 24 GB of RAM, and two GTX 480 (one headless). The head is connected via 4x QDR Infiniband and Gigabit Ethernet to the cluster.

5.6.1 Scalability

For the evaluation of scalability we use the computation of mutual information for single-precision floating point values as described in Section 5.5.1. The test runs on a fixed set of eight cluster nodes using four GPUs per node. We use three sets of sequences. The first set consists of over 45,000 sequences of length 1.400. Computation time of one iteration on a single GPU without any overhead was about 19.72 seconds on average. The second set with 4176 sequences and a length of 5025 is computed in 8.95 seconds per iteration. The third one with 20,000 sequences and a length of 2189 took 6.707 seconds per iteration without any overhead. During the test the number of iterations is gradually increased

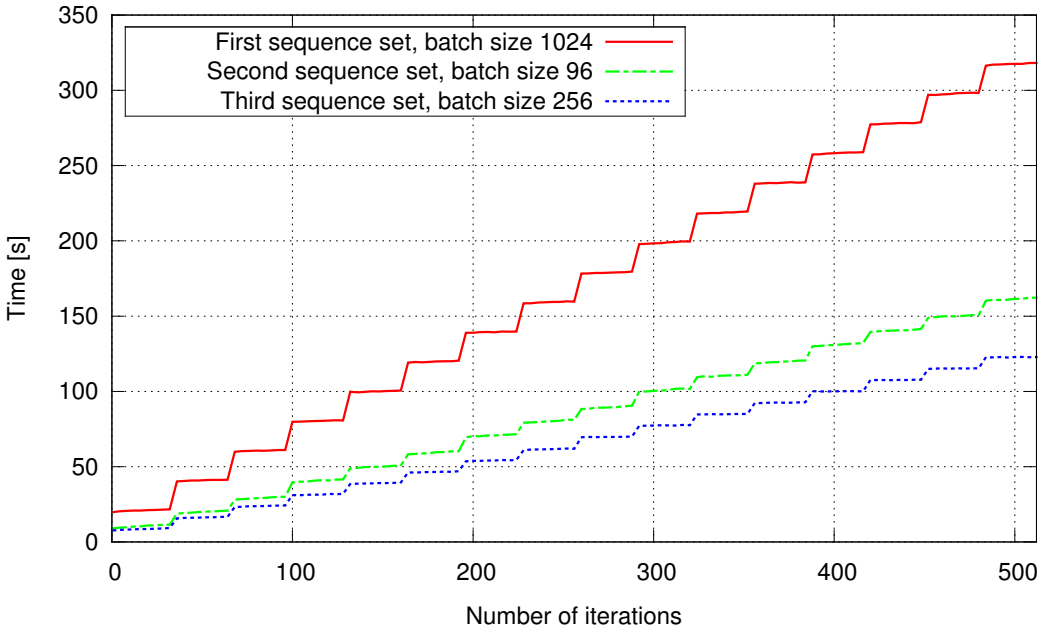

Figure 5.6: Average execution time for the computation of mutual information under varying number of iterations. All experiments were run on 8 nodes equipped with two NVidia Geforce GTX 590 (overall 4 GPUs per node.).

from 1 to 512 to measure the scalability of our system. Each iteration is a single compute job.

Figure 5.6 shows the resulting runtime of an application with an increasing number of iterations. The time is an aggregation of the communication overhead, I/O traffic plus the kernel compute time over all iterations. Our approach results in a linear speed-up for all three sequence sets. The speed-up for the first set converges to 31.5 for 32 GPUs as stated in Figure 5.7, showing that the computation time prevails over the communication overhead. In the second case the communication overhead is much higher. The memory consumption for each compute job is more than twelve times higher than in the first case. As a result we have to reduce the number of jobs running in parallel to 96 and spawn additional jobs after the computation of the previous batch has been finished until we reach the desired number of iterations. This introduces more communication between the different layers. The third sequence set has similar computational demands having half the sequence length of the second dataset but four times more sequences. The speedup difference between both datasets, shown in Figure 5.7, is minimal. From this it follows, that the computation of mutual information in our scenario is compute bound and not limited by network latency and bandwidth. In this case our proposed system scales linearly with the number of available GPUs when using a homogenous environment like a cluster.

The local maxima seen in Figure 5.7 occur when the number of iterations is a multiple of the GPU count. At this point our approach maximizes the utilization of the test system and reaches the lower bound of the communication overhead.

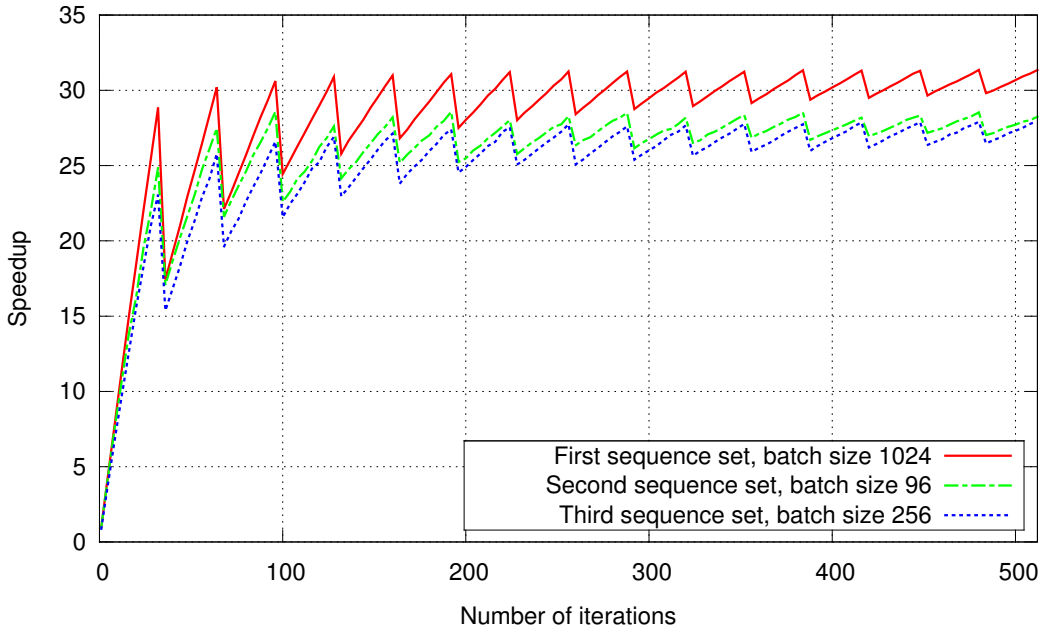

Figure 5.7: Reached speedup for a single GPU vs. 32 GPUs under varying number of iterations and for all sequences.

5.6.2 Network Overhead

For the evaluation of the communication overhead we use the MI computation with the same sequence set as in the previous section, with 45,000 sequences of a length of 1400. We fix the number of iterations to 64 but vary the number of available cluster nodes from one to eight and switch between the usage of one and four GPUs per node. As can be seen in Figure 5.8, the difference in communication overhead between single and multi--GPU nodes is negligible, only the results are sent back separately. It would be possible and beneficial to cluster the results for multi-GPU usage when homogeneous workload is spawned, but interactive applications would suffer from a slower response time. In our test cases the accumulated overhead ranges from about 3.1 to 20.8 seconds. Similar as in Section 5.6.1 the overhead increases when the number of jobs is not a multiple of the number of available GPUs.
Using five compute nodes with 4 GPUs each we can observe an increased overhead, because 64 jobs can not be distributed evenly to 20 GPUs resulting in one node having to compute the remaining four jobs. As can be seen in Figure 5.8 the network overhead can be reduced by batching requests when we achieve a uniform distribution of jobs for multiple GPUs per node. For our test scenario this is the case for two, four and eight compute nodes. On the other side we do not batch incoming results from the same compute node, which can lead to network overhead and less scalability of the application which is noticeable for 8 compute nodes.

5.6.3 Load Balancing

We examine the load balancing capabilities of our proposed system using the bidirectional path tracer described in Section 5.5.2. Therefore we run two different test setups measur-

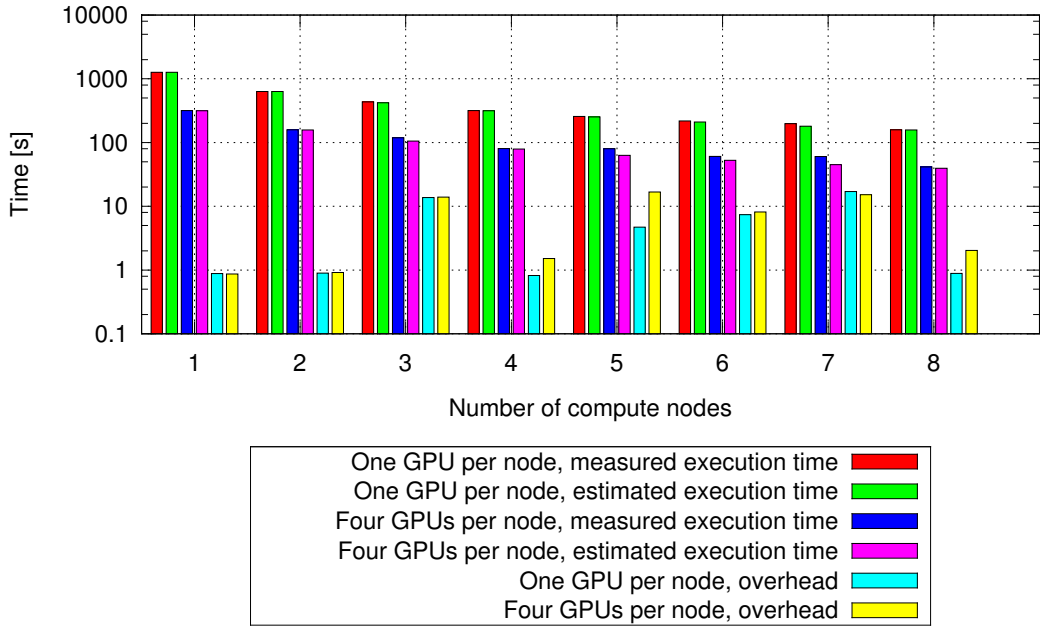

Figure 5.8: Average execution time for the computation of MI for the first sequence with a fixed number of iterations. The experiments vary the number of utilized compute nodes and GPUs (one or four.).

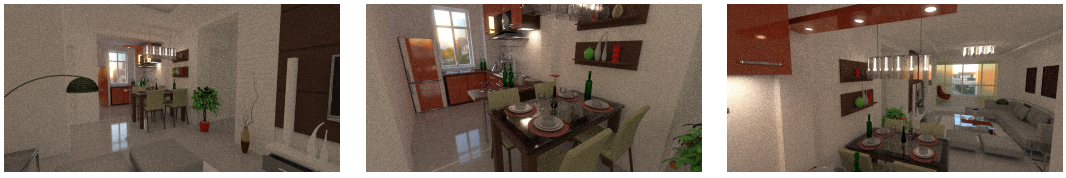

Figure 5.9: Excerpt from the fly through of our kitchen scene. The sequence was generated using a resolution of 1024 by 512 with 256 samples per pixel and contains 100 frames. The average execution time is 3.14762 seconds per frame (min 2.78142 s, max 3.43880 s) with load balancing disabled.

ing the effectivity of the approach. The first setup simulates high computational demand and as a result less network traffic. We use two different scenes, the kitchen scene (Figure 5.3) and the mustang scene (Figure 5.4) for the test. The path tracer configurations for both scenes is similar, using a resolution of 1920 by 1152 and a fixed tile size of 64 by 64 pixels, resulting in 540 compute jobs to be spawned. The number of samples per pixel is set to 4096 for the kitchen and 32,768 for the mustang scene respectively. As shown in Figure 5.5 both scenes generate a different irregular workload distribution. The benefit of load balancing for high computational demand can be seen clearly in Table 5.1. The compute time varies a lot across jobs due to incoherent rays. By moving jobs from busy devices to GPUs which are close to finishing, or have already finished their work, we can assure the best possible resource utilization. We can enable both layers of load balancing separately, task stealing within a single compute node to circumvent additional network traffic, and overall load balancing using global task stealing as well.

Scene	none	global TS only	local TS only	full
Kitchen	227.105	203.453	212.26	201.597
Mustang	236.324	193.776	210.258	190.51

Table 5.1: Runtime in seconds of the bidirectional path tracer without load balancing, global task stealing only, local task stealing on compute layer only, and full load balancing. The Full HD images of both scenes are rendered on 32 GPUs.

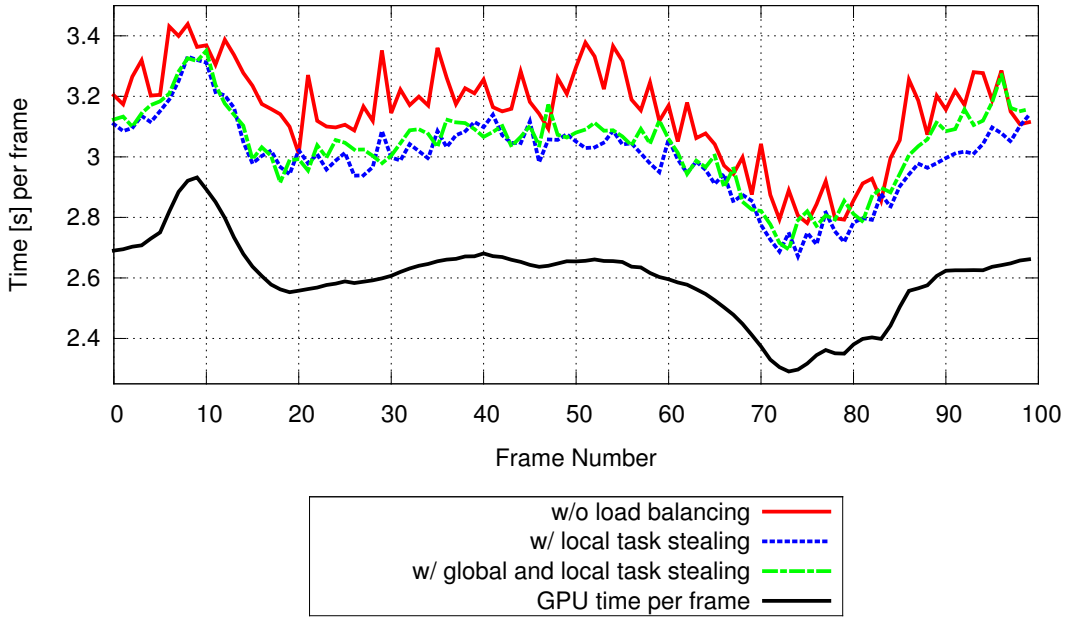

Figure 5.10: The average render times per frame, with and without load balancing and local task stealing only. The black line shows the wall time, the accumulated CUDA kernel runtime.

The second setup mimics a setup for near interactive usage of our system. Therefore we employ a fly through of our kitchen scene (Figure 5.9) and have chosen a scene configuration to achieve a frame duration of about three seconds in the case of disabled load balancing. Even such a setup with high network traffic can benefit from task stealing as shown in Figure 5.10. The average execution time is reduced in all cases. Without load balancing we need 314.76 seconds for the complete camera path, for global only we obtain 303.7 seconds and 300 seconds for full load balancing enabled. Because of the higher network overhead and traffic the performance can decrease for full load balancing in comparison to local task stealing only. The additional communication between compute nodes and the service node is reduced to a minimum. Therefore local task stealing is enabled by default and global stealing can be activated by each application separately.

Figure 5.11 visualizes a run of the path tracer under high computational demand without and Figure 5.12 with load balancing. For the experiment we enabled global and local task stealing and used eight compute nodes with four GPUs each. A single line segment in the graphs represents a compute job on a specific GPU. A compute node is encoded by the alternating black-white pattern. The length of a segment expresses the computation time

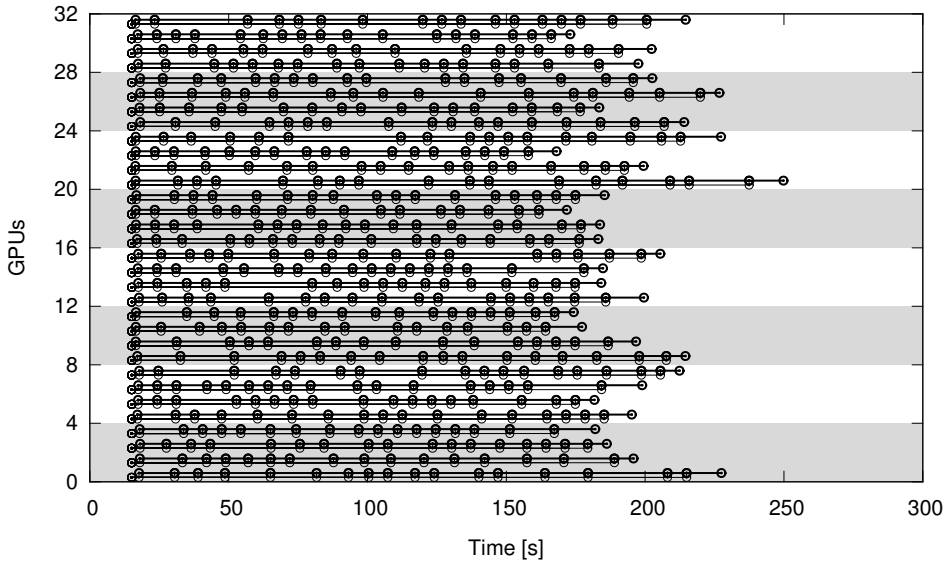

Figure 5.11: Running the path tracer without load balancing scatters the termination time due to the irregular job computation duration. The execution times of the application for both scenes are stated in Tabel 5.1.

for a particular job. As stated in Section 5.5.2 and visualized by the histogram in Figure 5.5 the bidirectional path tracer generates irregular and unpredictable workload. We use the static mustang scene for the load balancing test. The jobs where distributed using the default first-come, first-served scheduling policy without any prior knowledge about the previous computation times.
Figure 5.12 illustrates the job migration from one GPU to another. Local migration is visualized using red lines and migration over compute node boundaries with blue. Runtime can be reduced significantly as shown in Table 5.1 which leads us to the conclusion that global and local task stealing are crucial parts of a load balanced cluster to handle irregular workload.

5.6.4 Concurrent Access

Figure 5.13 shows 16 rows of compute devices. Always four GPUs are grouped in one node. For every device the execution of jobs is represented as bold lines with dots marking begin and end. The colors display whether a job belongs to the bidirectional path tracer (black) or the MI computation (red). Below the line of execution, we display points indicating the time when jobs were scheduled on a device.
One can see that jobs of both applications are executed alternatingly, following the fair lottery scheduling from Section 5.4.3. This ensures that both applications receive a fair share of compute time and are both finished within ample time. As long as no load balancing is enabled jobs execute on the same device they were scheduled on.

5.6.5 Heterogeneous System

For our heterogeneous test case we use a set of loosely coupled workstations as a desktop grid. The environment incorporates GPUs with compute capability 2.0 and 3.0 that are

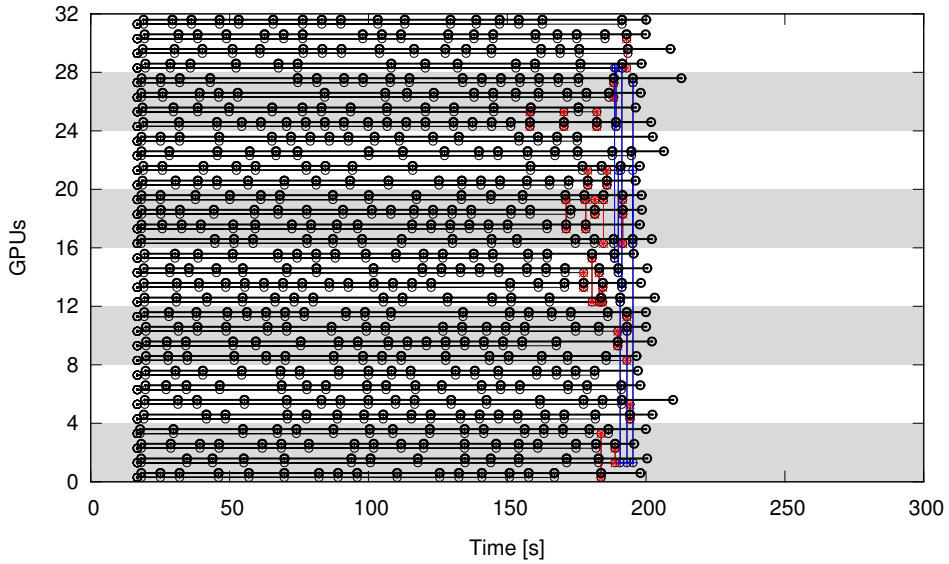

Figure 5.12: Running the path tracer with dynamic load balancing enabled, jobs are passed to nodes reaching the end of their work queue, leveling the overall computation time of all nodes. The execution times of the application for both scenes are stated in Tabel 5.1.

Scene	none	global TS only	local TS only	full
Mustang	823.19	708.16	715.52	681.53

Table 5.2: Runtime in seconds of the bidirectional path tracer in our heterogeneous setting without load balancing, global task stealing only, local task stealing on compute layer only, and full load balancing.

connected using Gigabit Ethernet. If two workstations are linked together via Infiniband in addition the cluster system will automatically adopt to the faster connection. In our case we will mix both network architectures.
The desktop grid is composed of three Windows workstations, two equipped with a GeForce GTX 680 each and one machine with a GTX 570. We added one additional Linux system, using a Tesla C2070. All systems are connected via Gigabit Ethernet only. As components equipped with Infinband network adapters we attached the three cluster nodes (two GTX 590 each) and the head node using a GTX 480. The accumulated FLOPS for single-precision floats, measured by the CUDA BLAS library, amounts to 11.2 TFlops varying from 583 GFlops for the Tesla C2070 to 1027 GFlops for a GTX 680.
The load balancing capabilities of our proposed system are exposed by the bidirectional path tracer. We render one image of the mustang scene with a resolution of 1920 by 1152 pixels using 65,536 samples per pixel and a tile size of 64 by 64. This setting results in 540 compute jobs handled by the desktop grid. Table 5.2 shows the overall runtimes with load balancing enabled and disabled, respectively. The computation time is reduced from 823.19 seconds to 681.53 by applying full load balancing. Local task stealing will be applied first to reduce network communication. As a result the GTX 590 cards from the cluster nodes will steal more tasks then the systems equipped with the GTX 680 despite the fact that the GTX 680 is about 1.5 times faster.

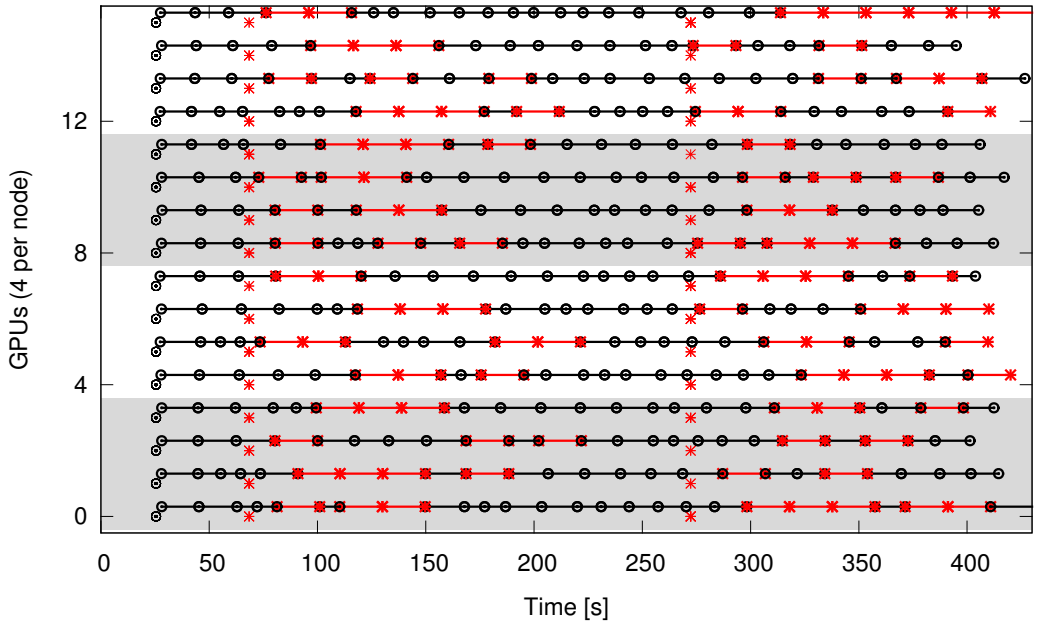

Figure 5.13: Timeline of two applications being run in parallel on 16 GPUs. The path tracing kernel (black) execution times are varying and were all scheduled at the beginning, whereas the homogeneous work of the MI computation (red) is scheduled in batches. Notice the alternating execution of jobs from each application. Scheduling times are marked by symbols shifted slightly below the lines.

The Figures 5.14 to 5.17 are showing the job distribution for all four possible load balancing configurations. The first fact shown in all diagrams is the distribution of compute jobs according to the heuristic. Both GTX 680 (IDs 14 and 15 in Figure 5.14) are getting more jobs assigned then the Tesla C2070 (ID 12). Because we use a different CUDA launch configuration when running the path tracer on cards with compute capability 3.0, to optimize utilization for the Kepler architecture, the scheduled jobs on those cards are executed in a shorter amount of time. In our setup these cards are well suited for load balancing. The Figures 5.15, 5.16, and 5.17 show that most jobs are stolen by both GTX 680. A performance impact of mixing network technologies can be seen as the difference between the scheduling time (thin circle) and the execution time (bold circle) at the beginning of the run. The Infiniband enabled cluster nodes (GPU ID 0 to 11) and the head node (GPU ID 13) fetch the static allocations faster than all other nodes using Gigabit Etherent only.
Running our sample applications on this system with varying GPUs using different network technologies shows that our framework is capable of handling heterogeneous environments reliably.

5.7 Conclusion and Future Work

We presented a cluster system for interactive and concurrent access to a heterogeneous GPU cluster that includes fair scheduling, dynamic load balancing, and work stealing. Data is efficiently distributed and persistently stored in the compute nodes. Likewise, ex-

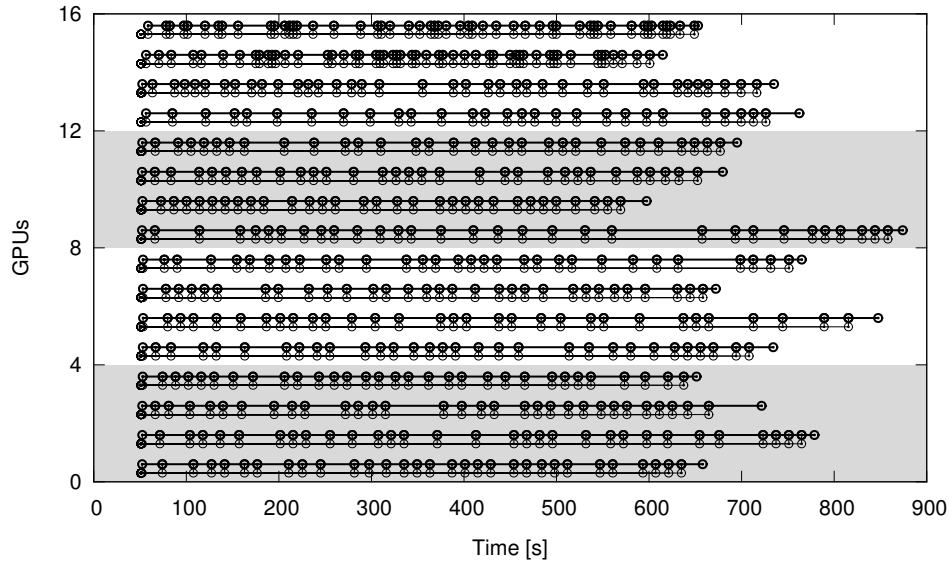

Figure 5.14: The diagram shows the job distribution based on our heuristic without load balancing capabilities of our proposed system.

ecutables are only transmitted once and cached in the compute nodes. We demonstrated the feasibility of such a highly flexible and dynamic approach. The system scales nearly linearly with the number of compute devices except for a minimal communication overhead. As applications, we demonstrate an implementation of a bidirectional path tracer, implemented completely on the GPU and a bio informatics application computing the mutual information of sequences.
Our system is based on CUDA but could be extended to any of the other current languages like OpenCL. In the future, we would like to employ the LLVM [Lattner and Adve, 2004] to reduce the complex development of parallel applications for our system by supporting the CUDA Runtime API instead of the Driver API. As an addition we would like to distribute the LLVM intermediate code and incorporate the Just-In-Time compiler to support a broader range of different compute devices. Another direction we would like to invest further is the preemption of GPU programs using a snapshot system. This could allow us to schedule high priority tasks more efficiently.

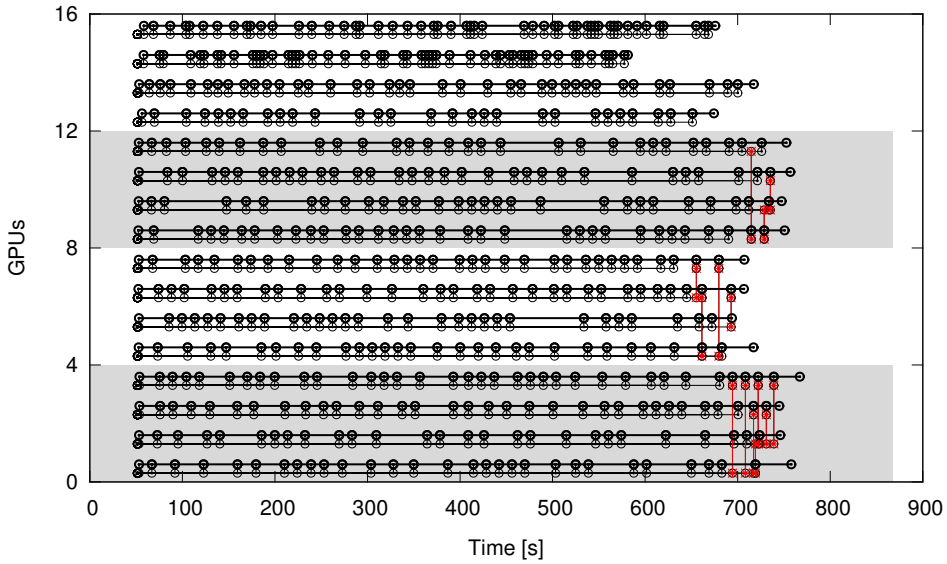

Figure 5.15: The graph visualize the local job migration from one GPU to another in a multi-GPU systems.

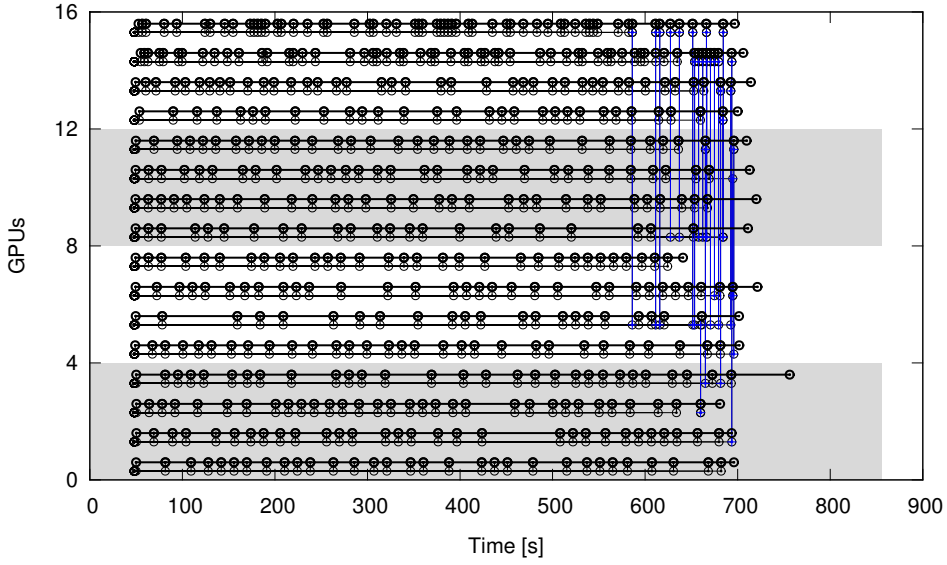

Figure 5.16: The figure shows the work distribution and load balancing capabilities with global task stealing enabled only.

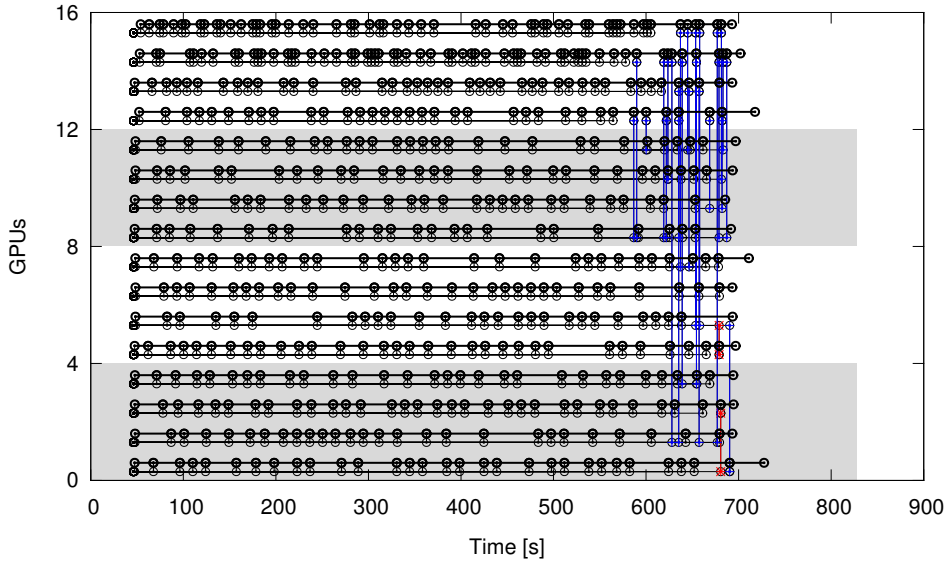

Figure 5.17: The diagram shows the full potential of the work distribution and load balancing capabilities of our proposed system.

Part II

Hybrid Ray Tracing

Chapter 6

Motivation

Despite the huge amount of research on alternative rendering algorithms in recent years such as stochastic rasterization [Akenine-Möller et al., 2007, McGuire et al., 2010], standard rasterization [Pineda, 1988] is still the state-of-the-art rendering algorithm for real-time applications. Current GPU architectures (see Section 3.2.1) are optimized for vector, matrix, and per fragment operations and can therefore deliver an astonishing image quality at real-time frame rates. As outlined rasterization cannot handle complex phenomena (e.g. soft shadows, motion blur and global illumination) natively. In this part, we describe two techniques that use ray tracing in combination with rasterization to render distribution effects.

At first we will outline the basic programmable rendering pipeline (Section 6.2.1) and introduce screen space ray tracing (Section 6.2.2). In the following Chapter 7 we present an adaptive acceleration structure based on a plane-based multi-layer approximation of the depth buffer to increase the screen space ray tracing performance significantly. We combine rasterization and distributed ray tracing in Chapter 8. A decoupled space and time sampling approach allows us to reuse ray tracing results for motion sampling so we can evaluate additional motion samples without the tracing overhead.

6.1 Introduction

As GPUs become more powerful they are capable of rendering high quality images (see Chapter 3) within reasonable time frames using path tracing. Unfortunately, hardware limitations (as outlined in Section 3.2.3) still render real-time path tracing impractical. The image quality achieved by specialized algorithms for rasterization is better than any general real-time ray tracing solution currently available. The open question we would like to answer is how we can extend current rasterization algorithms by applying ray tracing to reduce the algorithmic complexity and achieve a higher image quality without sacrificing performance? Which effects can be implemented and how can we speed-up ray traversal?

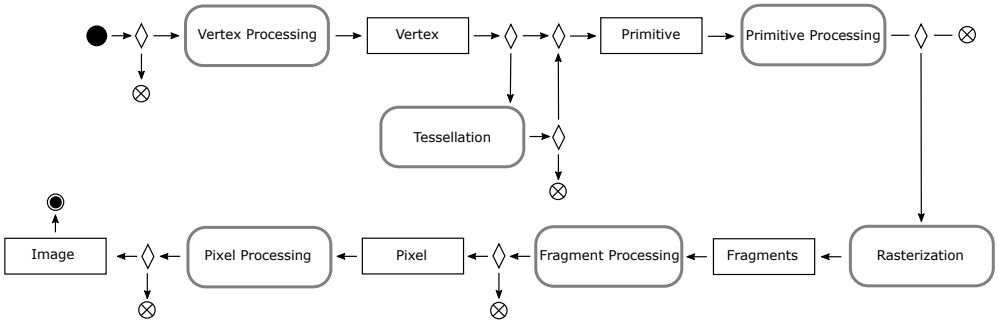

Figure 6.1: The figure shows a simplified diagram of a programmable rendering pipeline based on OpenGL 4.4 - (Complete OpenGL 4.4 pipeline map [Khronos Group, 2014]).

6.2 Background

6.2.1 Rasterization

Rasterization in general describes the projection of a rendering primitive (e.g. triangles, lines, or complex polygons) onto a 2D plane and converting them into a raster format (pixel image) to display them. In this section we will outline the most basic steps from a draw call to the final pixel on the display.

The pipeline (Figure 6.1) can be divided into different phases and stages — vertex specification stage, vertex processing phase, primitive processing, rasterization, fragment processing, and finally the per-sample operations in the pixel processing phase. In the following paragraph we will go through each phase and stage step by step. A complete description can be found in the OpenGL specification [Khronos Group, 2014].

At first the application prepares all vertex data streams (e.g. triangles meshes), sets up the pipeline, and issues the necessary calls to draw the vertices (**vertex specification stage**). During the **vertex processing phase** each individual vertex from the stream is transformed and projected into clipping space using a vertex shader. Application defined vertex attributes are modified, and the result stored in the output stream of vertices. In addition the vertices (describing patches) can be tessellated and attributes interpolated using the tessellation phase.

The next step is vertex post-processing. This includes the clipping stage which removes all primitives that are outside of the view volume described by the camera (see Section 2.2). Intersecting primitives are split up so that the complete primitive is inside the volume. During this phase new primitives can be generated using a geometry shader.

The **primitive processing phase** collects and compacts all primitives into a single stream. The output of the phase is a sequence that describes a set of simple rendering primitives (e.g. triangles, lines, or points). At the end all triangles that are not facing towards the camera will be removed by the face culling and the primitive positions are transformed into window space by applying the perspective division and the viewport transform.

During the **rasterization phase** the sequence is rasterized based on the triangle coverage onto a block of pixel (e.g. 8 x 4 pixel of the image plane). A new fragment is created for each raster element of a primitive that is covered. Thereby, a fragment contains a position, interpolated triangle attributes (e.g. a normal, a color, and texture coordinates — similar

as described in Section 2.1.2 using the barycentric coordinates) as well as a depth value.
In the **fragment processing phase** the fragments can simply be shaded based on the given input data (e.g. color, normal, and texture coordinates) using a fragment shader. It is possible to modify the color and depth values of a fragment. The output stream contains the color, depth, and a stencil value per fragment.
The values are used for the per sample operations in the **pixel processing phase** afterwards. The most important part of the phase is the depth test to solve the visibility of individual fragments for a rasterization algorithm. A fragment is simply discarded and not stored in the final framebuffer if the depth comparison function (e.g. larger than, less, or equal) with depth of other fragments at the same position fail. After all fragment test operations where applied, the color values of each fragment can be blended. The resulting color and depth of the final fragment are written to the framebuffer and later on displayed on the screen.
The GPU executes the phases and stages of the described pipeline in parallel. The GPU architecture described in Section 3.2.1 can be directly mapped to the pipeline stages. The rasterization phase basically works on the previously described warps, each thread in a warp can simply operate independently on a vertex, and fragments are executed in the same fashion. As the workload of the phases can be different each multiprocessor can execute a different phase of the rendering pipeline.

6.2.2 Screen space ray tracing

Several specialized algorithms for real-time rendering exist that extend rasterization with ray tracing to generate soft shadows, reflections [Mara et al., 2014], and global illumination [Crassin et al., 2011]. Within these we can identify two main tracing approaches. Voxel based tracing approaches [Crassin et al., 2011, Laine and Karras, 2010] approximate the scene using a sparse voxel grid and operate as classic ray tracing in either world space or camera space. In contrast, screen space approaches such as relief mapping [Policarpo et al., 2005] or the tracing of height fields [Tevs et al., 2008] rely on ray marching along a texture to solve the visibility.
For screen space ray tracing we render the scene at first into the depth buffer to obtain the depth as a texture. In the second pass, for each pixel of the final image a ray is generated and transformed into screen space (also known as texture space). In the next step we have to travel along the ray using a fixed step size. With each marching step we have to sample the depth from the depth buffer and compare it with the current depth of the ray. If the depth of the ray is smaller we traverse further and create a new sample point. We have found an intersection point if the sampled depth is in an ϵ range or we are outside of the depth texture (x and y position of the sample point are not between 0 and 1). At this point we would like to refer the reader to a comprehensive description of the algorithm by McGuire and Mara [2014].
The main drawback of the algorithm is the oversampling and undersampling of the depth buffer. Based on the number of marching steps and the step size we can issue too many texture reads. This either reduces the overall performance or due to undersampling leaves holes in the reconstruction. A hierarchical depth buffer [Uludag, 2014] can be used for fast rejection of samples and reducing unnecessary texture reads. Furthermore, the algorithm cannot cope with dis-occlusion if only a single depth layer is used. By introducing multiple depth layers holes can simply be filled by tracing the additional layers Mara et al. [2013].

We describe a acceleration structure (Chapter 7) and show how to apply it to screen space ray tracing to speed-up the rendering of different effects, for example multi-view synthesis and ray traced reflections. Another possible application for the acceleration structure is the rendering of depth-of-field effects as well as ray tracing of rough glossy reflections.

6.2.3 Depth-of-field

Based on the foundations we described in Section 2.4 we would like to outline possible approximations and rendering algorithms for depth-of-field in the context of real-time rendering. Those techniques can be classified into object or image based approaches. Object based approaches such as distributed ray tracing [Cook et al., 1984] use the underlying geometry to solve visibility. In contrast, image based algorithms employ the color and depth buffer. Notable artifacts of incorrect depth-of-field approaches are sharp edges and leaking of sharp foreground objects onto blurred backgrounds.
Distributed ray tracing [Cook et al., 1984] (see Section 2.4) is the most common way to solve depth-of-field using an object based ray tracing approach. As of today the technique described is not feasible for real-time applications. The second object based approach which is applicable to real-time rendering to a certain degree is the usage of the accumulation buffer [Haeberli and Akeley, 1990]. The scene is rendered multiple times from different positions on the lens. The resulting images are blended using the accumulation buffer. With an infinite number of rendering passes the result converges to the quality achieved by distributed ray tracing with an infinite number of samples. Less passes result in artifacts such as ghosting. These techniques can be applied to real-time applications, but with higher quality of the depth-of-field effect and size of the circle of confusion the number of passes increases significantly. This is comparable with increasing the number of samples when using distributed ray tracing.
The group of layered depth-of-field algorithms decompose the rendered image into several layers based on the depth from the depth buffer. Each layer is blurred independently using different approaches such as a Fourier Transform [Barsky et al., 2002], a pyramidal image [Kraus and Strengert, 2007] or splatting [Lee et al., 2008]. Dis-occlusions can be handled by interpolation of the reconstructed pixel neighborhood or adding additional depth layers using depth-peeling [Lee et al., 2008]. A similar approach uses light fields which are created by warping the rendered image to nearby views [Yu et al., 2010]. However, Selgrad et al. [2015] uses a multi-layer filtering which is unable to handle motion blur at the same time.
Reverse-mapped z-buffer approaches are most common for real-time applications such as games. The rendered color buffer is filtered with different filter kernels and sizes based on the used approach. For each pixel the amount of blur is computed according to its depth which directly corresponds to the diameter of the circle of confusion [Potmesil and Chakravarty, 1981]. In contrast the forward mapped z-buffer depth-of-field approaches are not widely used in real-time applications as they mostly render circles into the framebuffer and therefore perform a scatter operation.

6.2.4 Motion blur

Navarro et al. [2011] gives an excellent overview of current offline and real-time rendering algorithms for motion blur. He classified the problem in seven categories: analytic methods, geometric substitution, texture clamping, Monte Carlo, post-processing, hybrid,

and mechanical and optical methods. In this section we will focus on recent and real-time capable approaches.

The accumulation buffer [Haeberli and Akeley, 1990] can be used to render motion blur (see Section 6.2.3). Multiple images rendered at different points in time are averaged. Similar to distributed ray tracing [Cook et al., 1984] the algorithm converges to the correct result. Unfortunately, the rendering of several frames can be slow, and undersampling may result in ghosting artifacts.

A lot of games and interactive media rely on post processing and other filter based approaches such as blurring the texture on moving objects with multiple or anisotropic samples [Loviscach, 2005]. As the technique is simple and fast, artifacts such as sharp silhouettes and texture seams of moving objects can be noticed. Geometric approaches such as Tatarchuk et al. [2003] extrude or augment moving object geometry using convex geometry. To reduce artifacts the approach relies on a pixel-perfect depth sorting.

McGuire et al. [2012] describe a motion blur filter for real-time applications. The 2D post-process filter works on the standard framebuffer augmented with a velocity buffer. The velocity buffer encodes the pixel offset and depth value to its corresponding position of the previous frame. Guertin et al. [2014] performs motion blur as a post-processing filter. They address issues of single-velocity approaches using a robust sampling and filtering scheme. However, combining a motion with defocus blur filter requires per fragment information and assumes that each pixel corresponds to a single fragment. This assumption is no longer valid once the first filter has been applied. Multiple layers per fragment for the depth and velocity buffer are needed.

Stochastic rasterization Akenine-Möller et al. [2007], Fatahalian et al. [2009], and McGuire et al. [2010] have extended the standard rasterization algorithm to stochastic sampling for defocus and motion blur.

Akenine-Möller et al. [2007] presented an algorithm that rasterizes time-continuous triangles (*TCT*). Thereby, attributes of a TCT can vary in time t over the time span of a single frame. As the rasterization of the time-continuous triangles is very expensive they used a screen space acceleration structure that at first renders a tight object oriented box (*OBB*) for each TCT. For each fragment inside the OBB a per-pixel evaluation of the time-dependent edge functions is performed. In addition, they use Zmin/Zmax-culling to conservatively reject triangles and fragments before performing actual sample evaluation. Furthermore, time-dependent textures were introduced to support motion blurred shadows. As the attributes of a TCT vary and are sampled for an exact t the approach supports motion blurred reflections as well as other time dependent shading effects.

McGuire et al. [2010] show a real-time capable and completely dynamic stochastic rasterizer. The algorithm uses a stochastic visibility evaluation to approximate motion and defocus. For each triangle the encompassing geometry is created which conservatively covers all pixels that the triangle affects over the time span of a frame. Thereby motion as well as defocus is considered. Then the convex hull is rasterized. For each rasterized fragment, a ray-triangle intersection test is performed using stochastically sampled rays to determine the visibility. Thereby, the rays are sampled over the spatio-temporal domain. Non-rejected samples are shaded and the depth of the intersection test is used to later on determine the visibility between different triangles. For the depth test the conventional depth-buffer is used.

Sampling analysis and reconstruction filer Stochastic sampling methods such as distributed ray tracing [Cook et al., 1984] need a large number of samples to reduce the image noise significantly. This results in an increase of ray traversal for visibility tests and shading operations, especially for complex light situations and phenomena such as defocus and motion blur. One possible improvement to reduce the number of shading operations is to cache shaded samples [Ragan-Kelley et al., 2011].
Egan et al. [2009] analyzed the frequency of motion blurred scenes which include moving objects, reflections, and shadows. Based on the analysis they compute adaptive space-time sampling rates. In addition, they propose a sheared reconstruction algorithm that extends filter support along the motion. This reduces the number of samples needed significantly.
Layered reconstruction filters [Lehtinen et al., 2011, Munkberg et al., 2014, Vaidyanathan et al., 2015] generate a higher quality image from a sparsely sampled light field for motion and defocus blur.
Hasselgren et al. [2015] presented performance improvements to the state-of-the-art layered reconstruction algorithms. In particular, they use hardware texture filters, merging layers and sparse statistics to reduce computational complexity. The algorithm was optimized for current GPUs which results in a performance improvement of 2× up to 5× and still achieves a similar image quality compared to previous algorithms. The work makes reconstruction filter applicable to real-time applications.
In Chapter 8 we show how to apply distributed ray tracing to solve visibility in the rasterization process and apply depth-of-field and motion blur.

Chapter 7

Adaptive Acceleration Structure for Screen-space Ray Tracing

Abstract

In this chapter we propose an efficient acceleration structure for real-time screen-space ray tracing. The hybrid data structure represents the scene geometry by combining a bounding volume hierarchy with local planar approximations. This enables fast empty space skipping while tracing and yields exact intersection points for the planar approximation. In combination with an occlusion-aware ray traversal our algorithm is capable to quickly trace even multiple depth layers. Compared to prior work, our technique improves the accuracy of the results, is more general, and allows for advanced image transformations, as all pixels can cast rays to arbitrary directions. We demonstrate real-time performance for several applications, including depth-of-field rendering, stereo warping, and screen-space ray traced reflections.

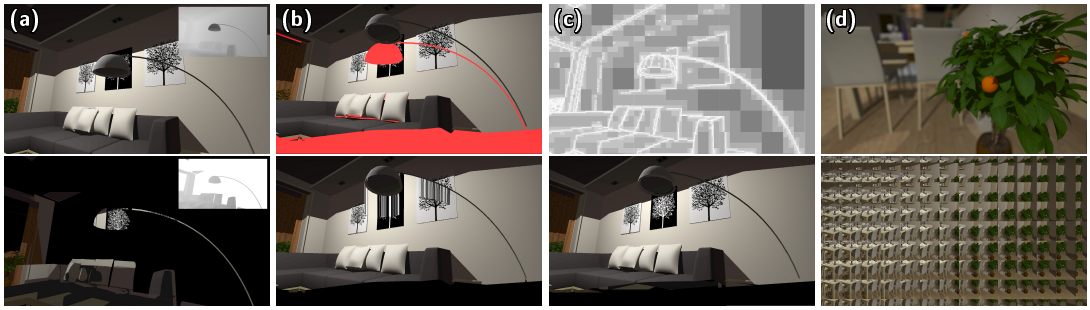

Figure 7.1: We perform fast screen-space ray tracing through single- and multi-layered depth representations. Because we efficiently obtain valid hits from occluded geometry, our approach can address many problems of traditional screen-space methods. Examples: **(a)** Reference view color+depth buffers for the first (top) and second layer (bottom). **(b)** Large camera motion produces extensive disocclusions (top, red), which are difficult to inpaint by forward warping (bottom). **(c)** Our compressed depth representation lowers the required ray-AABB intersection count (top, black corresponds to 0 and white to 9 intersections) and allows for an efficient reprojection with only two depth+color layers (bottom). **(d)** Our ray-tracing approach generalizes well to other real-time applications, including depth-of-field rendering (top) and light-field rendering (bottom).

7.1 Introduction

Many real-time rendering techniques operate in screen-space in order to be computationally efficient. This includes techniques for approximating realistic lighting, such as screen-space ambient occlusion [Mittring, 2007], soft shadows [Guennebaud et al., 2006], global illumination effects [Mara et al., 2014, Ritschel et al., 2009], and camera effects such as depth-of-field (DoF) [Lee et al., 2009]. These screen-space techniques trade precision and quality for performance.

In fact, some of those algorithms use screen-space ray tracing, or rather ray marching [Sousa et al., 2011]. Ray marching is attractive as no additional data structure needs to be built, but tracing rays for long distances becomes prohibitively expensive quickly. So precision is often sacrificed for performance by restricting the number of samples along the ray to reduce texture lookups, which can miss geometry – even when combined with a final binary-search refinement step. Classic ray marching methods, like DDA, are prone to over and under-sampling, unless perspective is accounted for [McGuire and Mara, 2014]. Most screen-space ray tracing methods use only a single depth layer, as a naïve extension to multiple layers is costly [Mara et al., 2013].

We address many of these shortcomings of screen-space ray tracing. Namely, we implement an efficient and scalable screen-space ray tracing algorithm that employs a dynamically created acceleration data structure, which enables efficient empty space skipping. If desired, our algorithm can trade accuracy for speed and can efficiently handle multi-layered screen-space representations to yield higher quality. The construction of the acceleration structure is extremely fast and can easily be done on a per-frame basis allowing us to also handle dynamic content.

Our method's efficiency makes it not only useful for screen-space effects such as depth-of-field, but also for reprojection tasks even into many views, as required for light field dis-

plays, that previously mapped very poorly to GPUs. We demonstrate our method in several classic applications (stereo warping, temporal upsampling, depth-of-field rendering, multi-view synthesis, and glossy as well as specular reflections).
Our contributions are:

- A novel data structure that stores the depth buffer in a compressed format—a mixture of AABB and planar approxmations—that enables early ray traversal termination and leads to an improved performance. The approximation can be further tuned for specific applications (e.g., depth-of-field) providing a significant performance boost.
- A new algorithm for screen-space ray tracing that is particularly well suited for GPUs. In contrast to state-of-the-art methods (e.g., ray marching [McGuire and Mara, 2014]) the ray traversal does not rely on a predefined or maximal number of steps and results in an accurate intersection point (assuming each pixel is planar). Our efficient occlusion handling allows tracing multiple depth layers of our acceleration structure without individually tracing each ray against each layer.
- A real-time implementation of thin-lens-based depth-of-field rendering. We extend previous work by introducing a secondary sampling stage (possible thanks to our compressed scene representation), which significantly reduces noise without sacrificing the defocus blur quality.
- A reprojection application that enables arbitrary existing content to be rendered on light-field displays with many views.
- Efficient multi-bounce specular and glossy reflections, utilizing our adaptive data structure on a (multi-layer) cube map representation of the scene.

7.2 Related work

7.2.1 Screen-space ray tracing and applications

In 1986 Fujimoto et al. [1986] proposed the 3D-DDA line traversal algorithm for quickly tracing rays through a regular grid or octree. It inspired the improved 3D-DDA line traversal algorithm by Amanatides and Woo [1987], which serves as the basis for many screen-space ray tracing methods [Ganestam and Doggett, 2014, McGuire and Mara, 2014, Sousa et al., 2011].
The work by Sousa et al. [2011] was the state-of-the-art for many years. It linearly ray marches a (reflection) ray in 3D, based on 3D-DDA, for a bounded distance. Each 3D point is reprojected into the frame buffer and classified as a hit if it lies behind the depth at the projected pixel. It does not do any space skipping, which was addressed by Ganestam and Doggett [2014], where an additional BVH is created. Employing a linear 3D-DDA traversal might lead to missed samples in screen-space. McGuire and Mara [2014] address this with a perspective 3D-DDA, ensuring no screen-space samples are skipped. In contrast, we build a screen-space acceleration structure on the fly that allows us to efficiently trace rays without the need for stepping along a line in small increments with 3D-DDA.
A wide range of applications and techniques use screen space ray tracing to simulate effects like ambient occlusion, view interpolation, or reflections. Most of those methods reuse shading information across frames to speed up computation, since this information (e.g., complex material evaluation) is expensive to recompute [Nehab et al., 2007,

Sitthi-amorn et al., 2008]. Herzog et al. [2010] combined shading reuse and spatio-temporal upsampling with the focus on reduction of shading cost. In contrast, our method focuses on a general acceleration data structure that allows for fast reprojection.
Another typical screen-space application is depth-of-field. These methods often employ approximations and application-specific algorithms, such as approximate cone tracing [Lee et al., 2009], or bounding the ray footprint [Lee et al., 2010]. While our technique is more general, we show that we also achieve better performance than these methods. Yu et al. [2010] suggested to warp a frame buffer to create a full light field, which is then combined to create depth-of-field effects. They use forward warping and simply splat larger pixels into the target views to prevent holes. Our technique can also be used to create a light field, but is much more efficient, as the run-time is independent on the number of views generated.

7.2.2 Ray tracing data structures for GPUs

Many different data structures have been proposed for GPU-based ray tracing applications. Bounding volume hierarchies (BVH) have been used extensively on GPUs. Lauterbach et al. [2009] create LBVHs by linearizing primitives along a space filling curve, yielding near optimal hierarchies and good overall performance for both construction and traversal. This was improved upon with a hierarchical version [Pantaleoni and Luebke, 2010] and work queues [Garanzha et al., 2011]. Rasterized bounding volume hierarchies (RBVH) [Novák and Dachsbacher, 2012], where leafs contain height fields that are ray marched, allow for efficient but approximate ray casting. Similar to our method, it also allows one to trade level of detail for computational efficiency. However, RBVH are not geared towards screen-space ray tracing, as the construction of the data structure is too slow.
Very recently, fast parallel construction of high-quality BVHs have been demonstrated on GPUs [Karras and Aila, 2013], yielding about 90% of the ray tracing performance of offline methods. K-d trees can also be constructed on GPUs [Zhou et al., 2008], even including the surface area heuristic (SAH) [Wu et al., 2011]. However, construction is generally more costly than for a BVH.
Voxelized scene representations have been used for various applications in real-time rendering. Efficient sparse voxel octrees [Laine and Karras, 2010] offer excellent ray casting performance, but can require non-negligible construction time and memory. Voxelized scene representations have been used extensively when high resolution and accuracy is less critical, for instance, in indirect illumination [Crassin et al., 2011].
Unlike these methods, our technique is geared specifically towards ray tracing through layered 2.5D height fields, exploiting their structure for considerable performance gains over standard ray tracing acceleration structures.

7.2.3 Ray tracing of relief and height fields on GPUs

Many techniques have been proposed since the early 1980's to render height fields, usually using ray tracing [Cohen and Shaked, 1993, Cohen-Or et al., 1996, Musgrave, 1988]. These methods generally differ in their choice of acceleration data structure. For instance, the early work by Cohen and Shaked [1993] uses a quad-tree and is the original inspiration for our method. Using ray tracing to render height fields and reliefs has also become popular in GPU-based real-time rendering. Tracing rays by uniformly stepping through the

height field in conjunction with a binary search is a common approach [Policarpo et al., 2005]; Newton iterations is another [Wyman, 2005]. While these methods are simple to implement, they can lead to missed intersection points. This can be fixed through the use of safety zones [Baboud and Decoret, 2006, Donnelly, 2005], but at a higher computational cost and using precomputed data structures.
These methods only support a single layer, the work by Policarpo and Oliveira [2006] adds the ability to render layered height fields by packing four layers into a single texture in order to trace through them simultaneously, speeding up rendering. For our use cases, we argue that most of the time only the first layer is hit by a ray, and the other layers are rarely needed. We therefore trace into deeper layers only if the first layer received no hit (for fewer than 1% of pixels).
Tevs et al. [2008] accurately render height fields by creating a min-max mipmap hierarchy over the depth map on the fly, which allows ray tracing with empty space skipping. This improves the pyramidal displacement mapping technique [Oh et al., 2006], sharing the min-max mipmap acceleration data structure with previous work [Carr et al., 2006, Guennebaud et al., 2006, Kolb and Rezk-Salama, 2005]. While this data structure—it corresponds to a fully sub-divided quad-tree—is attractive for height field rendering, it does not directly support discontinuities and multiple layers. Traversal requires looping to step into the correct hierarchy level, whereas our method uses simple bit patterns to yield the correct level.
The min-max mipmaps [Tevs et al., 2008] have also been used to render volumetric shadows [Chen et al., 2011], where epipolar rectification of the shadow map ensures that a ray traverses along a row, reducing ray traversal to a 1D-problem. Unfortunately, we cannot use this insight, as the construction is slow and the structure is only valid between a single reference view and one novel view, reducing applicability.
Like many of the methods cited here, our method is related to layered depth images (LDIs) [Shade et al., 1998]. Just like LDIs, our scene representation consists of possibly multiple layers of depth plus color and we also support reprojection of the scene. However, LDIs were geared exclusively toward scene reprojection, whereas our method is more general enabling several different applications. Furthermore, the original splatting-based LDI rendering technique was not very GPU-friendly, and has been superseded by much of the work cited in this section.

7.3 Data structure

In this section, we first introduce our data structure for representing multiple layers of scene geometry, and how it is constructed in real time. We then describe our approach for ray traversal in Sec. 7.4.

7.3.1 Compressed depth representation

We start by rendering the reference view into a set of color+depth buffers [Shade et al., 1998], which serves as an over-complete representation of the scene. This kind of rendering workload can be implemented efficiently via *depth peeling* [Lee et al., 2010, Mara et al., 2013, Policarpo and Oliveira, 2006]. We decided to use a simple k-buffer algorithm, which turned out to be sufficient and fast enough. Our method, however, is not bound to any particular approach and will work with any depth peeling method.

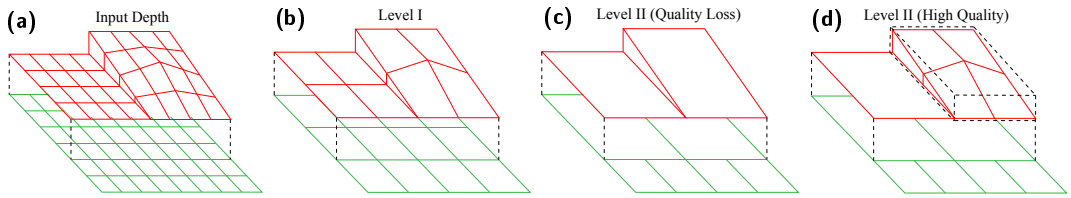

Figure 7.2: A toy example of quad-tree construction. **(a)** We start the construction with two input depth layers (red/green) with per-pixel normal vectors (visualized via plane rotation here). **(b)** Each 2×2 set of adjacent and non-overlapping cells is analyzed to generate a parent that best describes them. Depending on the compression setting, slightly misaligned children are replaced by a **(c)** single parent plane which then becomes a new leaf node (tree branch has been pruned) or a **(d)** AABB parent node that encompasses the children.

Next, for every layer from this 2.5D stack, we compute a *quad-tree* ray traversal acceleration structure (see Figure 7.2). Each node in the quad-tree stores either a 3D *axis-aligned bounding-box* (AABB) or, at leaf nodes, a 3D plane that represents the geometry. Note that since the quad-tree is built from frame buffer image data, the screen-space 3D AABBs actually correspond to frusta in world-space. The proposed data structure is related to the min-max pyramids [Guennebaud et al., 2006], in a sense that we use non-overlapping (in screen-space) bounding-volume hierarchies (BVH) to accelerate ray tracing through efficient empty space skipping (AABB misses). However, in contrast to this previous work, our quad-tree can be adaptively pruned, when the leaves represent the geometry by a single plane.
This has two important consequences for ray traversal. First, since the plane nodes represent the underlying layer geometry, a ray-plane intersection test simultaneously determines if the ray hits *both* the node *and* the geometry. The bounds defined by such a plane are tighter than by the enclosing AABB, which makes skipping empty space during the traversal more efficient. This translates into significant performance gains due to reduction in both GPU memory bandwidth pressure and thread divergence. Furthermore, as the ray-plane intersection test results in an exact intersection point, we do not need a "refinement" stage, such as binary search [Policarpo and Oliveira, 2006, Tevs et al., 2008], to remove artifacts. The second feature of our data structure is that we can use it directly to approximate and *compress* the screen-space geometry by controlling when and how a node's geometry is replaced with a *proxy*-plane. This allows us to trade off ray tracing precision for performance.

7.3.2 Bottom level generation

The quad-tree is built in a bottom-up fashion. The depth of the decomposition could go to $\lceil \log_2 \max(width, height) \rceil$ levels. Both construction and ray tracing are done in NDC (*Normalized Device Coordinates*) space mapped between $[0,1]$, where all potential scene points (x,y,z) are inside of a unit cube. The bottom level of the quad-tree is initialized directly from the depth buffer. Each pixel is represented as a plane with a normal vector $\vec{N}$ and plane origin P_{origin}. In practice, we only store the Z coordinate of P_{origin}, as its 2D coordinates are known from the node position in the quad-tree. In addition to $\vec{N}$ and P_{origin}, we store a binary flag O, which indicates whether the plane is close to a depth dis-

Algorithm 1: GLSL pseudo-code for generating the bottom level of our traversal acceleration structure. λ_h and λ_d are set to 10^{-3} in our implementation.

```
   input  : depth ;                                          /* depth buffer data */
   output: out ;                                             /* node texture at level 0 */
1  D_{0,0...2,2} ← depth ;                                   /* read 3x3 depth neighborhood */
2  /* discontinuity hint computed via Laplacian thresholding                          */
3  O ← step (λ_d, getLaplacian (D))
4  /* compute forward and backward differentials                                      */
5  df_xy ← vec2 (D_{2,1} − D_{1,1}, D_{1,2} − D_{1,1})
6  db_xy ← vec2 (D_{1,1} − D_{0,1}, D_{1,1} − D_{1,0})
7  /* enforce smoothness by picking the smallest derivative                           */
8  d_xy ← mix (df_xy, db_xy, abs (df_xy) < abs (db_xy))
9  /* zero large derivatives that connect different surfaces                          */
10 d_xy ← step (λ_h, abs (d_xy)) · d_xy
11 /* compute normal                                                                  */
12 N⃗ ← normalize (cross (vec3 (P_size.x, 0, d_x), vec3 (0, P_size.y, d_y)))
13 /* compute plane's top-left corner z-coordinate                                    */
14 P ← D_{1,1} − dot(N⃗_xy / N⃗_z, −0.5 · P_size))
15 out ← outputPlane (N⃗, P, O);                             /* output a plane node */
```

continuity. We use this information during ray tracing to adaptively dilate the screen-space bounding-box to mask tiny cracks between neighbors with different orientations.
Normal vectors can be obtained through deferred rendering (via a *g-buffer*) or computed from depth data directly. All our examples use the latter approach, although g-buffer normals should produce slightly better quality. For depth-derived normals special care needs to be taken when generating P_{origin} for the bottom level of the quad-tree. By default, the GPU rasterizer produces pixel samples that are located at the center of each pixel, corresponding to a $(0.5, 0.5)$ shift. However, we represent the plane with the Z-coordinate of its *top-left* corner, which we need to take into account. Therefore, to produce a valid Z for the plane origin, we use regular plane-ray intersection, which in this case reduces to an equation from line 14 in Algorithm 1.

7.3.3 Quad-tree generation and planar approximation

The remaining quad-tree levels are generated with our adaptive pruning algorithm demonstrated in Figure 7.2 (see Algorithm 2). The idea behind the algorithm is simple. For each output node we consider its 2×2 children nodes, and if they can be approximated *well enough* with a plane, we store the plane, otherwise we define the node as a regular AABB and store its corresponding min/max Z values.
Successfully approximating a subtree by a plane node requires fulfilling three conditions: **(1)** all the children have to be plane nodes, **(2)** the maximum angular difference between proxy-plane and child plane normals has to be less than γ_{norm}, and **(3)** the maximum distance from child plane corners to the proxy-plane has to be less than γ_{dist}. Thresholds γ_{norm} and γ_{dist} can be fixed, or modulated adaptively to implement LOD-like functionality. If the children do not meet these conditions, we output a parent node as AABB that encompasses all of them.

Algorithm 2: GLSL pseudo-code for construction and compression of a single level of the quad-tree.

```
   input  : in ;                                   /* node texture at level i (i > 0) */
   output: out ;                                   /* node texture at level i-1 */
 1 Q_{0...3} ← in ;                                /* read 2x2 neighboring nodes */
 2 if containOnlyPlanes (Q_{0...3}) then
 3     /* get plane normal vector, origin and "discontinuity" flag                  */
 4     (N, P, O)_{0...3} ← getPlaneData (Q_{0...3})
 5     /* set proxy plane normal to mean of children normals                        */
 6     N_proxy ← normalize (mean (N_{0...3}))
 7     /* compute angle differences via dot-product                                 */
 8     float d_{0...3} ← 1− dot (N_proxy, N_{0...3})
 9     /* P_proxy is least-square fit to child planes with fit errors stored in p_{0...3} */
10     (P_proxy, p_{0...3}) ← getPlaneOrigin (N_proxy, N_{0...3}, P_{0...3})
11     /* output plane if the proxy is close enough in terms of orientation and position */
12     if max (d_{0...3}) < γ_norm and max (p_{0...3}) < γ_dist then
13         out ← outputPlane (N_proxy, P_proxy, any (O_{0...3}))
14         return
15     end
16 end
17 /* output AABB node that encompasses all children                               */
18 vec2 Z_{0...3} ← getMinMaxZ (Q_{0...3})
19 float min_z ← min (Z_{0...3}.x)
20 float max_z ← max (Z_{0...3}.y)
21 out ← outputAABB (min_z, max_z)
```

An accurate planar approximation of children requires solving an optimization problem with 4 unknowns $(\vec{N}_{proxy}, P_{proxy})$, which is too slow for real-time applications. We simplify the problem by first estimating the normal $\vec{N}_{proxy}$ as the average of child normals, and then finding the plane's Z coordinate P_{proxy} that minimizes the distance of child plane corners to the proxy plane with

$$\arg\min_{P_{proxy}} \sum_{i=0}^{3}\sum_{j=0}^{3}\left(\frac{P_{proxy} - p_{ij}\cdot\vec{N}_{proxy}}{\vec{v}\cdot\vec{N}_{proxy}}\right)^2,$$

where $\vec{v}$ is the view direction we optimize for and p_{ij} is the j^{th} corner of i^{th} child plane represented in the coordinate space of the proxy-plane. By default we set $\vec{v} = (0,0,1)$ to

Scene	Triangle count	Average construction time [ms]
SPONZA	227k	0.646 ± 0.042
LIVINGROOM	456k	0.636 ± 0.006
SANMIGUEL	6550k	0.657 ± 0.042

Table 7.1: Average quad-tree construction and compression time changes with the output quad-tree node count and is almost invariant to the scene complexity. See Sec. 7.5 for configuration details.

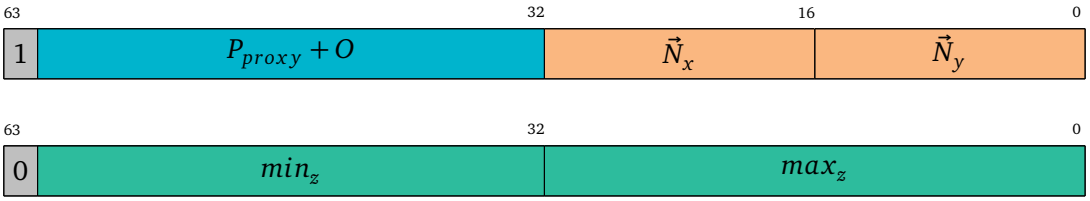

Figure 7.3: Quad-tree 64-bit node layout. Top corresponds to the plane node (1x32-bit float + 2x16-bit float) and bottom (2x32-bit float) to the AABB node.

maximize the reconstruction quality of the quad-tree for the reference view. In Table 7.1 we demonstrate the performance of the solver on the GPU. The entire construction algorithm is very fast and does not depend on the geometric complexity of the underlying scene.

7.3.4 Quad-tree node format

Each quad-tree node can store either an AABB or a 3D-plane. We managed to reduce the node's memory footprint by fitting both structures in just 64 bits (see Figure 7.3).
Interpretation of the node data is based on the value of the most significant bit of the first word, which corresponds to the *sign* bit in single-precision floating-point number representation. As both min_z and $P_{proxy}+O$ are always known to be positive, we choose to flip the sign of $P_{proxy}+O$ so we can disambiguate whether the node stores a plane or AABB by simply inspecting the sign bit. When encoding the plane, we store two components of a normal vector in half-float precision and recover the third one with $\vec{N}_z \leftarrow \sqrt{1-\vec{N}_x^2-\vec{N}_y^2}$.

7.4 Ray traversal

The core of our ray tracing method is depicted in Figure 7.4 (see Algorithm 4 in Appendix C for pseudo-code). At a high level, the algorithm performs a classic quad-tree ray traversal [Cohen and Shaked, 1993] with several application-specific customizations. For now we describe our method for a single layer and ignore disocclusions, which we detail in the next section.
First, the node type determines the intersection procedure, and we test either for an intersection with the plane or the AABB. Second, to reduce branching and thread divergence, we developed a method for efficient child selection in case of a node hit, and efficient successor selection in case of a node miss. Both functions are numerically stable and resistant to singularities. In case of a node hit, we compute the child position based on the parent node quadrant in which the intersection point landed. In case of a node miss, we generate the successor by evaluating which edge the ray hit when leaving the parent node (see Ray #1 case in Figure 7.4).
Finally, similar to Frisken and Perry [2002], we observe that quad-tree node x- and y-coordinates Q_x and Q_y encode the traversal stack up to the root node. By simply right bit-shifting Q_x and Q_y by one, we can generate the parent coordinates of the current node. This property allowed us to reduce the number of intersection tests significantly. After finding the coordinates (Q_x^* and Q_y^*) for the successor at the same quad-tree level (e.g., P2 is the successor of P1 in Figure 7.4), we could directly proceed to it, but this would provide inefficient fixed-step traversal similar to 2D DDA line-drawing algorithms. After all, the direct

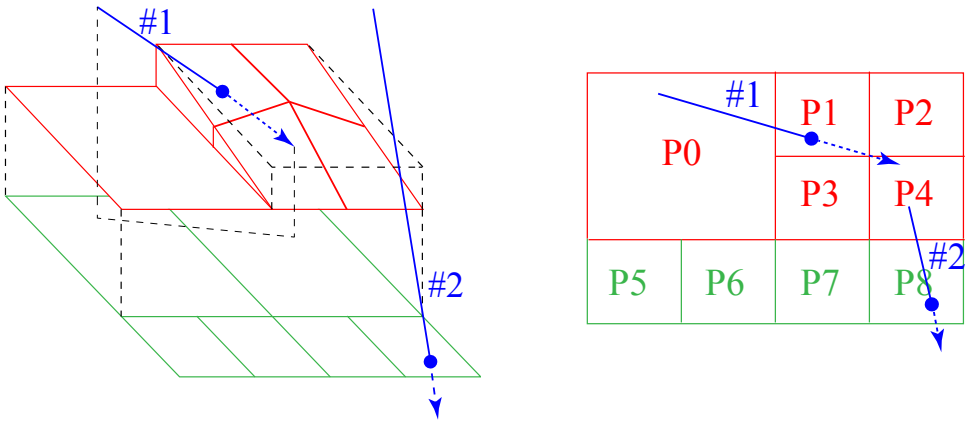

Figure 7.4: A toy ray-tracing example. Using the data structure from Figure 7.2d (right side shows XY-plane view) we cast two rays into the scene. Ray #1 misses leaf P0 but then hits the AABB node encompassing leaves P1–P4. Based on the AABB intersection point we derive the next intersection candidate (P1 leaf), which produces the final hit point. Ray #2 initially misses top layer completely (no intersection with P1–P4 AABB), but hits AABB encompassing P7–P8. The final hit point is computed via intersection with P8.

successor or one of its ancestors might be missed, so to maximize the benefit of empty space skipping, we need to select the largest possible parent. This means that an optimal strategy would involve (re-)starting the traversal from the root, which is inefficient (see Figure 7.5 for a 1D example); ideally we would like to start from one level below the last common ancestor. Because the node position encodes the full traversal path, we can compute this point via simple bit manipulation. Specifically, the number of levels we need to move up in the hierarchy is defined by the index of the most significant bit at which the coordinates of the current and successor nodes differ. This maps to $findMSB(Q_x \oplus Q_x^*)$ (where $\oplus$ is bit-wise XOR) and can be extended to $findMSB((Q_x \oplus Q_x^*)|(Q_y \oplus Q_y^*))$ for 2D, see Algorithm 3 in the appendix. The entire procedure maps very well to current GPUs as all the instructions are hardware-accelerated. In fact, this stack-less traversal is 25% faster than stack-based.

7.4.1 Disocclusion handling

Efficient handling of disocclusions in our 2.5D representation is a non-trivial task. A naïve solution would trace the ray against each quad-tree and pick the nearest hit among all hits. This, however, is slow and does not scale well with increasing number of layers. Some methods [Policarpo and Oliveira, 2006] save on the ray traversal time by bundling multiple layers and casting rays through all of them simultaneously. This works much better, but still seems sub-optimal. Most screen-space applications, such as time-warping, DoF rendering, and stereo-warping, have small reprojection requirements, i.e., relatively few pixels end up being fetched from background layers. Our approach efficiently deals with these scenarios by tracing rays individually and indicating not only a valid hit event, but also if the ray has passed through the occlusion volume in the scene.
An occlusion volume describes the 3D space occluded by the data in a depth layer (similar to a shadow volume). Since we do not know what might be hidden in the occlusion

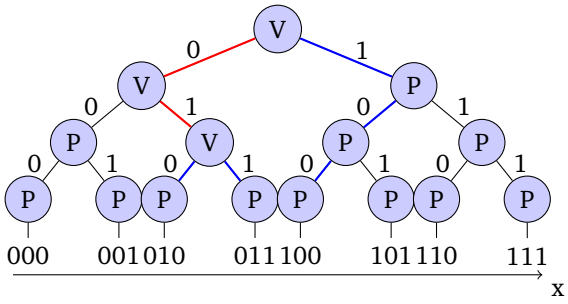

Figure 7.5: A 1D-view of ray traversal. Nodes representing planes (P) and AABBs (V) are stored in a hierarchical data structure. Moving from node at position 011 to 010 is fast as both nodes share most of the path from the root (red edges). In contrast, moving from 011 to 100 requires going all the way up to the root. Our algorithm computes the position of the last common ancestor between two arbitrary nodes at a given quad-tree level without the use of stack or loops, making it particularly friendly to GPU implementations.

volume of the foreground layer, rays that hit it, i.e., rays that would pass between occlusion boundaries, need intersection information from the background layer. This is because after reprojection, the background hit might end up being in the front of the foreground hit. To handle this, we test for intersections with the primitive (plane and AABB) *and* its occlusion volume. We never explicitly create this occlusion volume, but rather extrude a given node during intersection testing.

Knowing if the result of tracing the foreground is final, or whether we still need to trace the background, allows us to tremendously speed up the multi-layer tracing version of our algorithm. We have measured the ray distribution across different layers in the DoF rendering application, and even for large defocus blur, less than 1% of rays end up in the background layer. Note that the decision-making process is cascaded by nature. For example, adding a third layer will only impact rays that have missed both previous layers or hit an occlusion volume in the second layer. Hence the performance of our method scales well with the number of layers. In fact, for typical scenes, adding more than 3 layers has a negligible impact on ray tracing speed, and the overall system performance is limited by initial depth peeling and quad-tree construction stages.

The occlusion volume logic is not useful for the single-layer depth+color case, where no information about occluded geometry is available. Usually, the best thing one can do is to inpaint the resulting hole with background data (see Figure 7.6). For this particular case, we modify the tracing algorithm to provide a fast inpainting of disocclusions. Specifically, instead of traversing through the occluded part of space, we stop the traversal and return a valid hit at the intersection point with the occlusion volume. To make sure the point belongs to the background image data, we perform the intersection test with a slightly dilated occlusion volume. This effectively turns our algorithm into a fast height-field rendering method, and despite the disocclusion information being hallucinated, as we show in the next section, the method is still quite useful in the context of screen-space ray tracing.

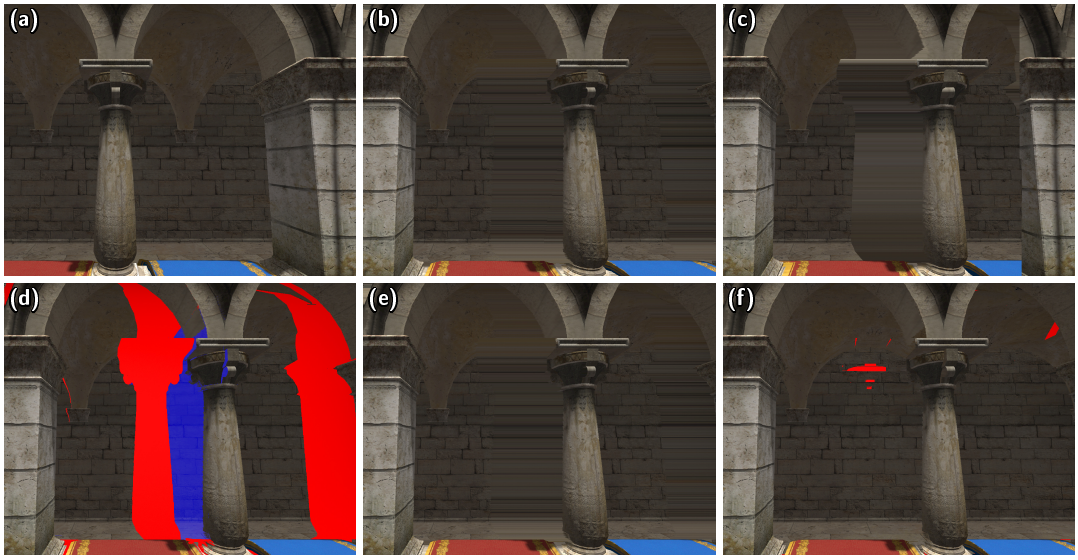

Figure 7.6: Comparing view synthesis approaches for large camera motion. **(a)** Reference view. **(b)** Mesh-based reprojection fills disocclusions by stretching background triangles. **(c)** Height-field rendering methods Tevs et al. [2008] produce less appealing results due to foreground preference during inpainting. **(d)** Our method tracing through a single depth layer. Apart from the regular hits we get misses (red) and hits of occlusion volume (blue). **(e)** In the fastest variant we fill both with the nearest background pixels. **(f)** Another variant performs tracing through the second depth layer and recovers the majority of misses. This can be repeated on subsequent layers to yield a full reconstruction.

7.5 Results

We now evaluate the performance and quality of our method for several screen-space applications. All experiments were conducted on a PC running Windows 7 64-bit version and the NVIDIA driver version 347.52. The system is equipped with an Intel Core i7-3930K, 64GB of RAM, and an NVIDIA Geforce GTX 980 with 4GB of RAM. All images were rendered at 1600×900 resolution, unless specified otherwise. The test sequences vary in the amount of camera motion and geometry complexity (see Table 7.1)—from the relatively simple SPONZA to the detailed LIVINGROOM (see Figure 7.1) and the SAN-MIGUEL scenes. The timings in this section exclude quad-tree construction, which is about 0.6ms per frame, unless otherwise noted; see Table 7.1 for detailed quad-tree construction timings.

7.5.1 View synthesis

We compare our approach to efficient implementations of three classes of view synthesis methods: mesh-based forward warping, height-field rendering and screen-space ray tracing.

The mesh-based warping methods represent the reference view as a regular grid mesh and rely on fixed-functionality GPU hardware to warp and rasterize it into a new view [Bowles et al., 2012, Didyk et al., 2010]. These approaches generally trade mesh resolution for performance. To have a fair comparison with our method, which resolves details

Method	Scene	Average time [ms]			$\frac{Mrays}{s}$
		T_{ref}	T_{eye}	T_{total}	
Mesh-based warping	SPONZA	0.59	1.47	3.56	N/A
	LIVINGROOM	0.76	1.37	3.53	N/A
	SANMIGUEL	2.73	1.37	5.46	N/A
Height-field tracing	SPONZA	0.59	3.5	7.7	405.75
	LIVINGROOM	0.79	3.85	8.5	373.78
	SANMIGUEL	2.75	4.33	11.4	332.41
DDA single layer	SPONZA	0.523	6.06	12.71	237.56
	LIVINGROOM	0.78	5.77	12.33	249.28
	SANMIGUEL	2.75	6.01	14.78	239.48
DDA two layers	SPONZA	1.13	8.26	17.65	174.06
	LIVINGROOM	1.41	7.80	17.01	184.60
	SANMIGUEL	5.51	8.22	21.93	175.05
Our single layer	SPONZA	0.59	1.1	3.43	1309.69
	LIVINGROOM	0.78	1.49	4.35	980.9
	SANMIGUEL	2.74	1.7	6.78	846.56
Our two layers	SPONZA	1.2	1.34	4.52	1076.22
	LIVINGROOM	1.45	1.7	5.45	850.5
	SANMIGUEL	5.52	2.02	10.24	709.8

Table 7.2: Performance comparison of our approach in stereo-warping application. We render a central reference view (T_{ref}) and warp it to the left and right eye. T_{eye} is the averaged time for a warp to a single eye and T_{total} corresponds to the total stereo frame render time.

at sub-pixel resolution, we have set up a mesh with one vertex per input image pixel. This simplifies the implementation as no mesh refining (snapping vertices to the nearest depth-discontinuities) or reprojection point optimization is required.

In Tables 7.2 and 7.3 we evaluate the performance for spatial and temporal reprojections. The performance of our approach is strongly correlated with the complexity of per-layer quad-trees. For the SPONZA scene, which has relatively simple geometry and produces few disocclusions, we are faster than mesh-based warping, regardless of the number of layers used. However, with detailed geometry that has large background/foreground depth differences, the speed advantage of our solution decreases. This is demonstrated in the SANMIGUEL sequence, where large camera motion produces lots of disocclusions and occlusion volume hits, which forces our method to shoot rays through the second layer for a significant portion of the image. However, unlike mesh-based forward warping, our method produces images with correctly resolved disocclusions (Figure 7.7) and allows for arbitrary per-pixel reprojections, which enables single-pass light-field (Sec. 7.5.3) and DoF rendering (Sec. 7.5.2).

Another class of view-synthesis methods is height-field rendering [Musgrave, 1988, Policarpo and Oliveira, 2006], which aims to efficiently visualize an elevation map from an arbitrary point of view. We have evaluated the performance of our approach with respect to a recent GPU height-field rendering method [Tevs et al., 2008] that uses ray marching in a min-max pyramid followed by a binary search intersection refining step. Their ray traversal routine allows for fast arbitrary per-pixel reprojections, but does not support tracing through multiple depth layers and therefore fails to resolve disocclusions correctly. This makes it equivalent to a single-layer version of our approach. We have used the authors' GLSL implementation and selected their fastest iterative version of the algorithm. To improve the speed further we have disabled bilinear patch interpolation. This normally produces pixel level staircase artifacts common to voxelization algorithms, but due to rel-

Method	Scene	Average time $[ms]$			$\frac{Mrays}{s}$
		T_{ref}	T_{syn}	T_{amt}	
Mesh-based warping	SPONZA	0.54	1.43 ± 0.07	1.2	N/A
	LIVINGROOM	0.78	1.4 ± 0.04	1.25	N/A
	SANMIGUEL	2.75	2.23 ± 0.13	2.36	N/A
Height-field tracing	SPONZA	0.54	2.33 ± 0.1	1.88	617.76
	LIVINGROOM	0.79	2.2 ± 0.15	1.84	655.43
	SANMIGUEL	2.76	3.07 ± 0.73	2.99	468.6
DDA single layer	SPONZA	0.58	3.87 ± 3.67	3.05	371.70
	LIVINGROOM	0.77	1.05 ± 0.4	0.98	1368.36
	SANMIGUEL	2.71	3.11 ± 0.82	3.02	462.13
DDA two layers	SPONZA	1.25	4.08 ± 3.80	3.37	352.5
	LIVINGROOM	1.42	1.53 ± 0.99	1.5	940.56
	SANMIGUEL	5.5	3.55 ± 0.92	4.03	404.95
Our single layer	SPONZA	0.55	0.92 ± 0.24	0.83	1558.44
	LIVINGROOM	0.78	0.89 ± 0.14	0.86	1608.93
	SANMIGUEL	2.71	1.2 ± 0.29	1.57	1201.0
Our two layers	SPONZA	1.15	1.13 ± 0.45	1.13	1269.84
	LIVINGROOM	1.44	1.03 ± 0.73	1.13	1395.34
	SANMIGUEL	5.55	2.18 ± 1.35	3.02	660.85

Table 7.3: Performance comparison of our approach for 15Hz to 60Hz conversion application. The mean reference frame rendering time T_{ref} together with new view synthesis time T_{syn} is used to compute amortized frame time T_{amt} for 60Hz rendering.

atively small zoom-in factors of our reprojection applications, we have not found this to be an issue. Despite these optimizations, our approach is still 2.5× faster on average (see Table 7.2 and 7.3). The performance improvement comes from our compressed depth-representation and more efficient traversal algorithm. Both reduce the overall intersection count and number of nodes visited, which directly maps to reduced texture fetch count and shorter run-times.

Finally, we compare against a recent screen-space GPU ray tracing method that supports tracing through multiple depth layers McGuire and Mara [2014]. The traversal algorithm is based on the idea of perspective-correct DDA line rasterization, which minimizes the number of duplicated intersection tests and texture fetches. The method does not require pre-computation or any ancillary data structure. However, due to ray marching nature and fixed traversal step size, its performance tends to degrade proportionally to the ray hit distance. To reduce the variance of the performance the authors introduce an upper bound on per ray marching step count. In our experiments we used a minimum value at which the DDA method produced results that are artifact-free and equivalent to ours. Specifically, we set the maximum step count to 200 and 500 for stereo-warping and temporal upsampling applications respectively. For stereo-warping, where the reprojection is relatively small and constant, our method is from 4× to 6× faster (Table 7.2). The DDA method becomes more competitive for temporal-upsampling (Table 7.3), which has different (rotation + translation) and varying reprojection requirements. Moving away from the reference frame increases the reprojection magnitude, which maps to longer epipolar lines (rays) and results in high timings variance for SPONZA and SANMIGUEL scenes. The variance for the LIVINGROOM case remains relatively small because of the slow camera motion that produces small differences between adjacent reference frames.

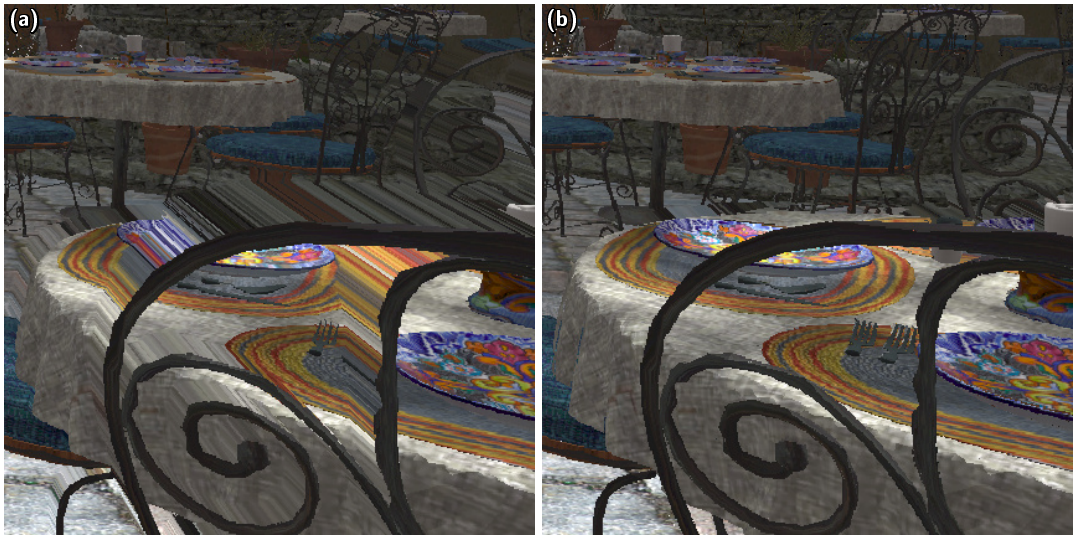

Figure 7.7: Time-warping for large camera translation. **(a)** Mesh-based forward warping produces visible background stretching artifacts. **(b)** We use information from the background layers, avoiding these artifacts.

7.5.2 Depth-of-field rendering

Screen-space ray tracing is often used to simulate complex lens effects. Cook et al. [1984] showed that rendering phenomena like motion blur, depth-of-field (DoF), and shadow penumbras is feasible via lens sampling and proper ray distribution. We followed this methodology, and implemented a naïve DoF algorithm that samples the lens aperture [Shirley and Chiu, 1997] and traces a fixed number of rays per pixel according to the thin-lens model. The per-pixel ray batches are then accumulated to form the final image. As we show in Figure 7.9, the knowledge of occluded regions in the scene is critical for high-quality DoF simulation. This is especially the case for defocus blur at large depth discontinuities, where rays travel into occluded parts of the scene geometry. Our method addresses this by tracing through multiple layers. Performance-wise, the DoF rendering workload represents the opposite scenario to Sec. 7.5.1. Here, most of the rays are short, incoherent and few of them end up in background layers. As shown in Table 7.4, our scheme breaks the 2.16 *billion* Rays-Per-Second (RPS) barrier for a single-layer rendering and 1.96 billion RPS for two layers. Interestingly, the DoF workloads are also handled relatively well by the DDA method [McGuire and Mara, 2014]. Setting maximum ray marching step count to 25 produces optimum performance with rendering quality equivalent to ours. While our approach has a significant advantage in terms of RPS, the absolute FPS statistics suggest that this improvement is to some extent consumed by the acceleration structure build overhead.

Unfortunately, random lens sampling with just a few rays produces visible noise in the final image. To reduce the noise, we have implemented a secondary sampling stage that exploits the characteristics of our quad-tree data structure. Specifically, after hitting the plane leaf-node, we generate additional rays in close proximity to the primary ray (using the same random distribution), but instead of tracing them through the scene, we assume their visibility and simply intersect them with the primary ray hit plane. Some intersection points might land “in the air”, therefore we lower their color sample contribution to the

Method	N_{pri}/N_{sec}	T_{ref} $[ms]$	T_{dof} $[ms]$	$\frac{Mrays}{s}$	FPS
DDA single layer	4/1	0.56	4.74	1215.18	188.4
	8/1	0.54	8.15	1412.11	114.85
DDA two layers	4/1	1.25	6.56	876.97	127.87
	8/1	1.23	9.34	1233.4	94.60
Our single layer	4/1	0.54	2.8	2055.67	251.13
	8/1	0.54	5.32	2167.05	152.67
	4/4	0.56	3.47	1657.07	214.5
	8/4	0.54	6.63	1737.55	127.73
Our two layers	4/1	1.22	3.21	1792.72	197.08
	8/1	1.21	5.85	1967.21	129.71
	4/4	1.24	3.84	1496.88	174.58
	8/4	1.22	7.22	1594.46	109.51

Table 7.4: Impact of the aperture sampling configuration on performance of depth-of-field rendering in SPONZA sequence. Each test case is configured to cast N_{pri} rays and accumulate them with N_{sec} color samples per ray. T_{ref} denotes the rendering time for the reference layer(s) and T_{dof} is the DoF image rendering time. Additionally, we provide the ray tracing speed and the overall application FPS (including quad-tree construction time).

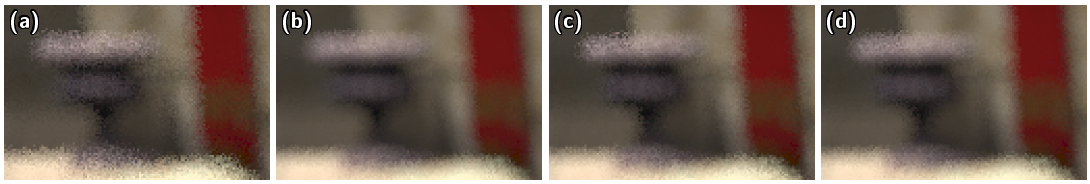

Figure 7.8: Depth-of-field rendering noise reduction through over-sampling. **(a)** Integration of only 8rpp (rays per pixel) produces noisy result. **(b)** 32rpp improves the quality, but is 3.9x more costly to compute. **(c)** 4rpp combined with 7 additional samples per ray produces in-between quality while being 25% faster than 8rpp alone. **(d)** Finally, 8rpp combined with 3 extra samples per ray produces results as in (b) while being only 20% slower than (a).

final pixel estimate by weighting them by $e^{-(z_i - z_b)^2/\sigma^2}$, where z_i is the intersection z value and z_b is the depth buffer value at the intersection point. Figure 7.8 shows the impact this has on DoF rendering noise levels. The proposed sampling strategy only approximates the physically-correct solution, but as we show in Table 7.4, it allows us to significantly reduce the ray tracing overhead without sacrificing the defocus blur quality.

7.5.3 Image retargeting for multi-view displays

Glasses-free 3DTVs, particularly those using parallax barriers [Ives, 1903] or lenticular arrays [Lippmann, 1908], require multiple views of the same scene. Unfortunately, rendering and transmission of dozens of views is expensive both in terms of computation and bandwidth/storage requirements. One way to address this problem is to send/compute only a small subset of all views, so called *reference views*, and use these to synthesize missing in-between views. Unlike existing 3DTVs, near-eye light-field displays [Lanman and Luebke, 2013] require rendering from hundreds to thousands of individual scene views. In the original paper, the authors describe two rendering approaches for their display. The first one relies on GPU ray tracing (with NVIDIA OptiX) to produce accurate elemental im-

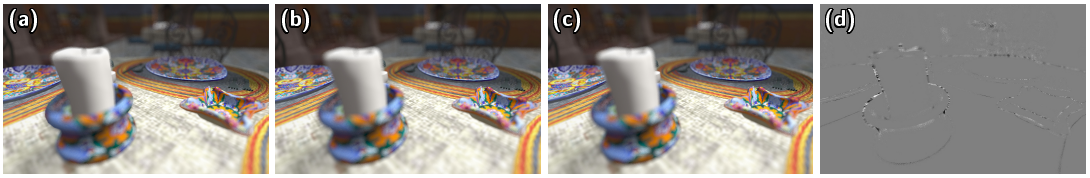

Figure 7.9: Depth-of-field rendering using the thin-lens model. **(a)** With a single depth layer we cannot resolve defocus blur at large depth discontinuities (e.g., the edge of the candle). **(b)** Our method produces correct image by tracing rays through two layers, **(c)** or by tracing through only a single depth layer but hallucinating background through inpainting (see Sec. 7.4.1), which approximates (b) and is faster than (a). **(d)** 5× difference between (b) and (c).

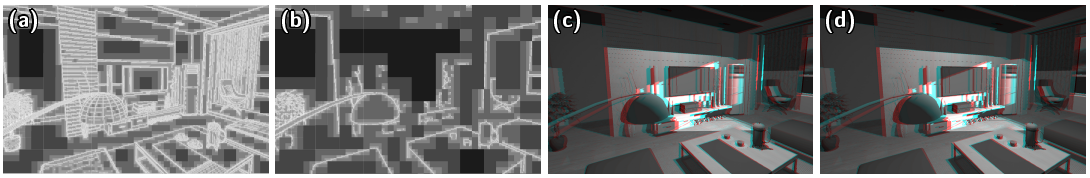

Figure 7.10: Impact of quad-tree compression on stereo-warping. **(a)** AABB intersection count visualization for a close to lossless compression of corresponding depth layers of the reference cyclopean view. **(b)** Aggressive compression of the quad-tree speeds up ray tracing by 12%. **(c,d)** Despite the difference in approximation quality between (a) and (b), the resulting stereo image pairs are visually indistinguishable.

age array for the display. The other one is a much simpler, where frames from the left and right eye are placed on a virtual plane and sampled to produce an image for micro-lens array. While efficient, this approach significantly under-utilizes the capabilities of a near-eye light-field display, as it cannot reproduce large disparities or accommodation effects this way. Here we implement the first approach: specifically, given color+depth buffers for the left and right eye, we generate a complete elemental image set with our ray tracing approach (see Figure 7.1d). This preserves the disparity range of the original stereo content and also enables the eyes to accommodate. Our results are visually indistinguishable from the OptiX solution, and depending on the scene and ray tracer configuration, it takes from 1.1ms to 3ms to reproject the image into an image array for a single eye, about three times faster than OptiX.

7.5.4 Ray-traced reflections

Many ray tracing algorithms exist to create plausible glossy reflections. As mentioned earlier, screen-space ray tracing is commonly used for this, e.g., the algorithm of Sousa et al. [2011] or its more accurate recent variants [Mara et al., 2013, McGuire and Mara, 2014] that can also handle multiple layers to solve disocclusions. As reflections are likely to come from outside the current viewport/screen-space, methods exist to enable this. Umenhoffer et al. [2007] create a cube map at the center of a reflective object, including the respective depth maps and possibly multiple layers. Reflections are rendered by screen-space ray tracing in one or more of the cube map faces. We take a similar approach, but speed up ray tracing using our adaptive data structure for each cube map face.
To show the applicability of our proposed data structure and ray traversal algorithm to

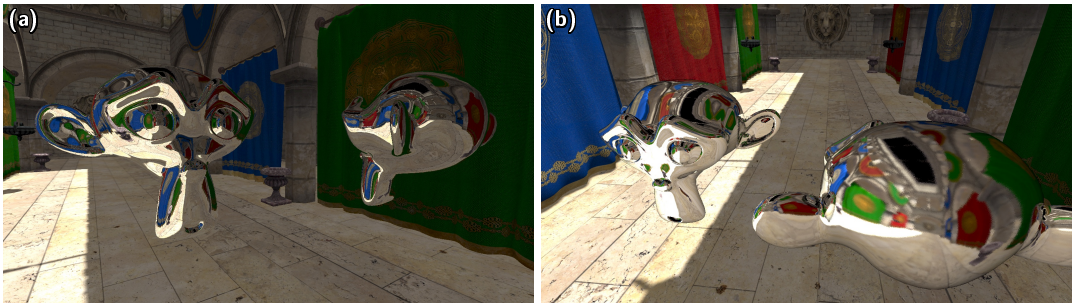

Figure 7.11: **(a)** Single depth+color layer ray-traced specular reflections (without self--reflections). Our per cube map face acceleration data structure is static and build in the first frame in 5.277ms. It takes 1.518ms to render the scene view and 1.062ms to trace the reflections, resulting in 387 average FPS. The two layer version takes 2.55ms to trace, yielding 234 average FPS. Note that since our algorithm effectively **reprojects** the cube map during tracing, we can use the same cubemap to render **correct** reflections for both objects. **(b)** Ray-traced rough reflection according to the Blinn-Phong BRDF with up to three bounces. Here, the front object has a specular coefficient of $N = 1000$, the object in the back is essentially specular ($N = 100000$).

non-screen-space effects, we use ray tracing to generate specular (Figure 7.11a) and glossy reflections (Figure 7.11b) including multiple bounces and self-reflections on non-planar reflective objects as described by Cook et al. [1984]. This can lead to more incoherent ray tracing workloads, especially when applying small reflection exponents to simulate rough surfaces. We generate a cube map ($6 \times 1024 \times 1024$ pixels big) at the camera location, and create our adaptive data structure for each face of the cube map. Note that in theory we could support other environment map representations, but only if they do not lead to curved lines. In our implementation, we simply render the scene six times, once to each cube map face. More efficient solutions, such as viewport multi-casting exist, but this is not the focus of our work. The primary intersection point is computed using rasterization, rendering the reflective objects only. For each intersection point we sample a new direction for the reflection vector by sampling the normalized Blinn-Phong BRDF at this position. If we hit a reflective surface, we generate a new ray until the maximum number of bounces is reached. At a diffuse surface we discontinue tracing.
The core ray traversal algorithm requires very few modifications for supporting tracing in the cube map. We begin by determining the frustum (cube face) the ray originates from. Then we trace the ray in the corresponding acceleration structure. If no valid hit was found, we transition to a new face based on which of the frustum planes of the current view the ray hits. If the ray hits the far plane of any view (ray is leaving the scene) we terminate the trace. We show the performance of our ray tracing approach for various scenarios in Figure 7.11, 7.12 and Table 7.5.

7.6 Discussion

7.6.1 Depth compression impact on performance and quality

By increasing γ_{norm} and γ_{dist} the quad-tree construction algorithm coarsens—with gradually increasing tolerance—smoothly-varying geometry in the scene, which in the limit

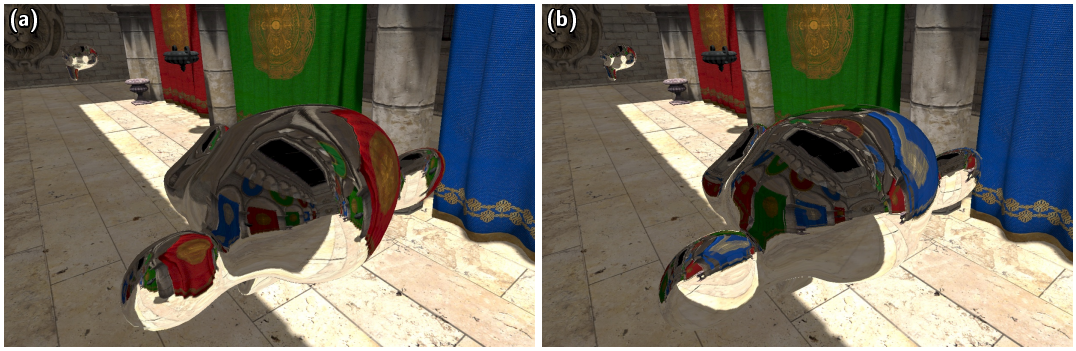

Figure 7.12: A comparison with a standard cube-map-based reflections. **(a)** Cube map rendered at position of the object in the far back produces incorrect reflections for the object in the front that is far from its center of projection. **(b)** Our method can address this scenario and produce correct reflection by reprojecting the data stored in the acceleration structure.

Method	$N_p/N_s/N_b$	T_{cm} $[ms]$	T_{ref} $[ms]$	T_{rr} $[ms]$	FPS
Our single layer	1/8/1	2.38	0.34	1.974	197.31
	3/8/1	2.39	0.41	6.96	98.68
	3/4/3	2.33	0.366	10.25	75.03
	3/8/3	2.35	0.388	12.38	64.64
Our two layers	1/8/1	5.15	0.415	4.91	92.21
	3/8/1	5.20	0.37	13.5	51.40
	3/4/3	5.19	0.34	22.96	34.64
	3/8/3	5.27	0.34	24.75	32.52

Table 7.5: Rendering performance of complex multi-bounce glossy reflections. Both single- and two-layer based algorithm is configured to cast N_p primary rays, followed by N_s extra samples per ray. Each primary ray is allowed to bounce N_b times. T_{cm}, T_{ref}, and T_{rr} denote the cube map, input frame, and reflection rendering times respectively. Last column shows the average application FPS.

produces a scene filled up with billboard-like objects. However, since we optimize the quad-tree for the reference view, and all the screen-space applications we demonstrate require relatively small parallax shift and ray direction changes, even such a coarse geometry approximation can produce sound results (see Figure 7.10).

In our experiments we set $\gamma_{norm} = \cos(3°)$ and $\gamma_{dist} = 10^{-5}$, which provided a 5-10% performance gain (with respect to lossless settings) without introducing any visible reprojection artifacts. The compression thresholds can be further tuned to exploit the nature of perspective projection—the hit precision requirements fall with the distance to the object. We have implemented a distance-adaptive compression by linearly increasing γ_{norm} and γ_{dist} thresholds with the node's mean depth value, which resulted in, on average, a 10% performance gain for view-synthesis applications and 15% for DoF rendering. Figure 7.13 and 7.14 include more detailed evaluation, where we show the impact of γ_{norm}, γ_{dist} and depth-adaptive quantization on performance and quality of the stereo warping application.

In Figure C.1 we compare the depth reconstruction quality of our method against a gold-standard GPU ray tracer—NVIDIA OptiX [Parker et al., 2010]. Even though we only use

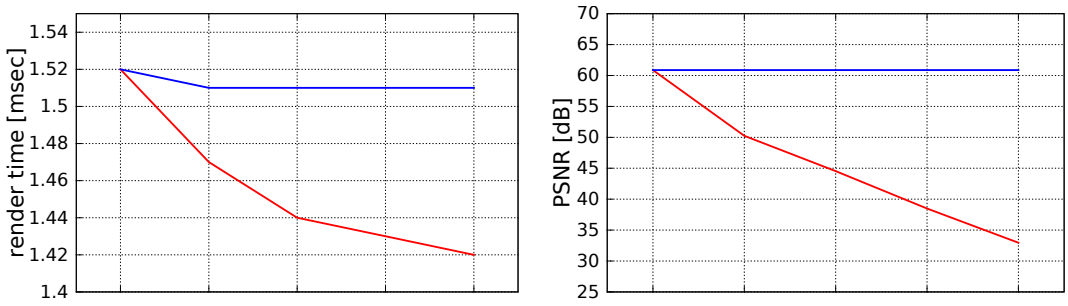

Figure 7.13: Impact of γ_{norm} and γ_{dist} on the quality and performance. **(left)** Frame render time in msec. **(right)** Frame PSNR in dB. **(red)** Fixed $\gamma_{norm} = \cos(3°)$ while logarithmically decreasing $\gamma_{dist} = 10^{-5}$ to $\gamma_{dist} = 10^{-1}$. **(blue)** Fixed $\gamma_{dist} = 10^{-5}$ while γ_{norm} varies between $[\cos(3°), \cos(25°)]$.

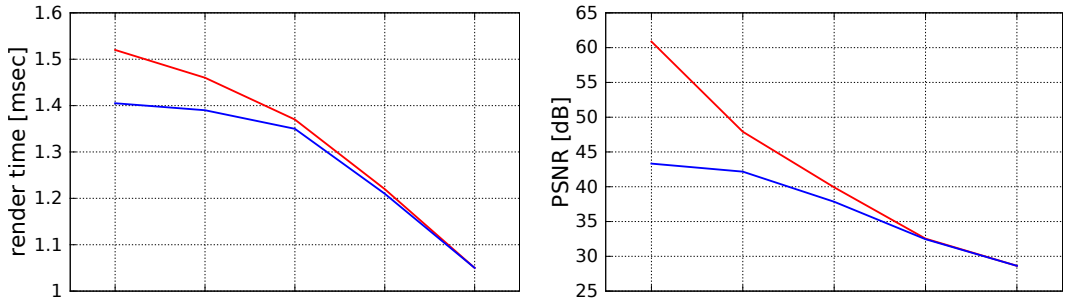

Figure 7.14: Impact of depth-adaptive quantization on quality and performance. The compression values γ_{norm} and γ_{dist} vary between $[\cos(3°), \cos(25°)]$ and $[10^{-5}, 10^{-1}]$ respectively. **(left)** Frame render time in msec. **(right)** Frame PSNR in dB. Depth-adaptive quantization disabled **(red)** and enabled **(blue)**.

two depth layers in this example, our approach correctly evaluates the depth at all pixels (except where three layers would be required), while being 3× faster than general-purpose ray tracer. Note that our approach allows us to inpaint the remaining holes, so that no artifacts appear. Alternatively, one can simply use more layers.

7.6.2 Limitations

While the single-layer (with background inpainting) variant of our method has good all-around characteristics, the performance of the multi-layer version decreases with the number of ray misses and occlusion volume hits. This is because large disocclusions generate long ray traversal paths, which slows down the tracing process, making our method not as efficient for large-displacement view synthesis. In our current multi-layer traversal implementation, when moving up close to geometry, some rays can pass through tiny cracks between unaligned plane nodes that should otherwise form a continuous surface. This could be fixed by generating more precise input normal vectors (use g-buffers instead of depth-based normal reconstruction) or by combining several neighboring planes to implement a more complex bi-linear patch intersection test. Finally, we consider only opaque

geometry and diffuse lighting models. However, support of multi-sample rendering and deferred shading is feasible and can be added as a straightforward extension.

7.7 Conclusion

We have presented a novel screen-space ray tracing method tailored for single- and multi-layered depth representations. We have demonstrated its performance and benefits in several screen-space rendering applications. We achieve real-time performance by combining a compact and steerable traversal acceleration structure with an efficient ray tracing algorithm, reaching the level of specialized state-of-the-art approaches for many applications. While our method is not a replacement for general purpose ray tracing frameworks, such as NVIDIA OptiX, it can be thought of as an efficient alternative for problems requiring 2.5D ray tracing capabilities. In the future, we would like to extend this work to support acceleration of volumetric rendering effects and investigate the application to approximate global illumination. We also plan to work on improving the quad-tree compression algorithm, which currently only considers geometric distortions. Accounting for underlying material properties, such as texture contrast or specular highlights, could further improve the performance and quality of our approach.

Chapter 8

Decoupled Space and Time Sampling of Motion and Defocus Blur for Unified Rendering of Transparent and Opaque Objects

Abstract

We propose a unified rendering approach that jointly handles motion and defocus blur for transparent and opaque objects at interactive frame rates. Our key idea is to create a sampled representation of all parts of the scene geometry that are potentially visible at any point in time for the duration of a frame in an initial rasterization step. We store the resulting temporally-varying fragments (*t*-fragments) in a bounding volume hierarchy which is rebuild every frame using a fast spatial median construction algorithm. This makes our approach suitable for interactive applications with dynamic scenes and animations. Next, we perform spatial sampling to determine all *t*-fragments that intersect with a specific viewing ray at any point in time. Viewing rays are sampled according to the lens uv-sampling for depth-of-field effects. In a final temporal sampling step, we evaluate the pre-determined viewing ray/*t*-fragment intersections for one or multiple points in time. This allows us to incorporate all standard shading effects including transparency. We describe the overall framework, present our GPU implementation, and evaluate our rendering approach with respect to scalability, quality, and performance.

Figure 8.1: Decoupling spatial and temporal sampling allows us to produce physically-based, plausible defocus and motion blur at interactive frame rates (rendered at 1024×576 pixel resolution, 4 dof-samples, 8 motion blur sample per dof-sample in 222ms on a GTX TITAN X).

8.1 Introduction

Defocus and motion blur are integral components to render photorealistic images. They are often used in movies or games to highlight an important situation or object for storytelling purposes. Distribution ray tracing [Cook et al., 1984] is a widely employed unified technique to render these phenomena in production quality applications. In contrast, real-time systems cannot until now use a unified algorithm to solve both effects and instead provide separate solutions for defocus and motion blur: Depth-of-field can be solved via approximate ray tracing [Lee et al., 2009, Widmer et al., 2015], warping light fields [Yu et al., 2010], or applying a multi-layer filter [Selgrad et al., 2015]. Most modern algorithms use multiple layers to handle disocclusions. Motion blur is mostly implemented using filter kernels as post-processing effects [Guertin et al., 2014]. This can reach a visual quality similar to distribution ray tracing without sacrificing performance. However, these approaches cannot handle defocus and motion blur at the same time, especially if a pixel consists of fragments with different depth, i.e. if it is a combination of opaque and transparent fragments.

Stochastic rasterization [Akenine-Möller et al., 2007, Fatahalian et al., 2009] extends the rasterization algorithm and hardware with stochastic sampling to enable defocus and motion blur from the camera and object perspective as well as motion blurred shadow maps [Nilsson et al., 2012]. McGuire et al. [2010] presented a hybrid algorithm for rendering approximate defocus and motion blur with stochastic visibility evaluation within a modern GPU architecture.

We present a unified rendering pipeline for interactive defocus and motion blur rendering for transparent and opaque fragments (see Figure 8.1). We decouple the visibility test,

used to generate depth-of-field in ray tracing, from motion sampling and employ shading after the visibility was determined. This reduces ray traversal and shading cost, as only visible fragments will be shaded, without quality loss compared to other real-time or interactive algorithms. In addition, late shading allows to use a deferred shading approach in combination with correct alpha blending of blurred and transparent fragments to produce physically plausible results. Our contributions are:

- Decoupling time sampling and visibility test, decomposing the traditional 5D sampling into a 4D spatial (xy, uv) and 1D temporal (t) sampling step.
- An intermediate scene representation using temporally-varying fragments (t-fragments) that represent the spatially sampled scene for temporal sampling.
- A disocclusion map that approximates the motion differences between two depth layers in order to identify potentially visible t-fragments for a correct trace result, including a simple edge filter to reduce depth-of-field artifacts at disocclusions. Each pixel in the disocclusion map marks whether it is a source of disocclusion or not. Using this information we modify the depth layers.
- A intersection test for temporally-varying axis-aligned bounding boxes (t-AABBs).
- A unified sorting and late shading pass using t-fragments as shading primitives that enables physically plausible transparency with defocus and motion blur at interactive frame rates.

8.2 Related work

Fundamentally, our proposed algorithm shares similarities with defocus and motion blur rendering for micropolygons by Hou et al. [2010]. They construct an acceleration structure using object aligned bounding boxes in 3D space for both the start time t_0 and end time t_1 of a frame. Assuming linear motion, each pair of bounding boxes forms a 4D hyper-trapezoid in space-time that tightly bounds the object for the entire time interval. A bounding volume hierarchy (BVH) is constructed with the SAH-based BVH construction algorithm of Wald [2007] for bounds at $t = 0.5$. During traversal, rays are associated with a time stamp and intersected with the corresponding interpolated bounds.
In contrast to Hou et al. [2010] we operate on fragments rather than micropolygons. As rasterization can generate several million fragments, a key to better ray traversal performance is fast and aggressive culling without introducing artifacts. At the same time, we aim at interactive rebuilds of the acceleration structure. We also directly intersect 4D hyper-trapezoids to collect fragments for the whole frame.

8.2.1 Defocus and motion blur

Algorithms computing defocus and motion blur fall roughly into two categories: Ray tracing based approaches approximate the effect in a physically based manner while real-time approaches try to create a perceptually plausible effect.

Ray Tracing Cook et al. [1984] present a unified framework for simulating defocus and motion blur with distribution ray tracing (*DRT*). They use stochastic sampling to create phenomena such as depth-of-field, motion blur, and shadow penumbras. All effects are simulated by simultaneously sampling in space and time. Images created by DRT exhibit

a certain level of noise due to the stochastic sampling. Thus, they are usually post-processed using filtering or reconstruction techniques. While early approaches filtered the final rendered images, the state-of-the-art is reconstructing surface lightfields for sample points along each ray [Hasselgren et al., 2015, Lehtinen et al., 2011, Munkberg et al., 2014]. While initially designed for DRT, these reconstruction filtering approaches can be used for all rendering algorithms that provide samples using a stochastic process. In order to reduce the noise prior to filtering without increasing the computational costs, samples should be generated where they improve the rendered image the most. Vaidyanathan et al. [2012] propose an adaptive sampling approach that is based on frequency information obtained through shaders together with the amount of defocus and motion blur in a certain area of the rendered output. They spend more samples in areas of high frequency while smooth areas are filtered more aggressively.
Belcour et al. [2013] improve on this by estimating the covariance along rays instead of combining frequency information from different sources in screen space. Tracing covariance has, however, problems with partial occlusion and transparency. Also, this method becomes less efficient in regions that contain both motion blur and defocus.
A slightly different approach was presented by Gribel et al. [2011]. In order to calculate motion blur, they sample line segments in 4D space-time rather than points in 3D space. The visibility along each line segment is solved analytically. However, adding defocus blur would require multiple line segments or tracing finite patches.

Real-Time Rendering A broad overview of existing techniques to solve motion blur is given by Navarro et al. [2011]. In general, real-time approaches usually fall into two categories. Guertin et al. [2014], for example, perform motion blur as a post-processing filter. There are, however, issues with combining this approach with real-time defocus as both filters require per fragment information and therefore assume that each pixel corresponds to a single fragment. This assumption is no longer true once the first filter has been applied. Selgrad et al. [2015] on the other hand use multi-layer filtering which is unable to handle motion blur at the same time.
In contrast, to these techniques, we only sample in 4D (xy, uv) space to solve visibility for depth-of-field. The time domain is later sampled on the resulting t-fragments, reducing the ray tracing overhead.

8.2.2 Stochastic rasterization

As we rely on rasterization for line-segment generation we share some similarities with stochastic rasterization from Akenine-Möller et al. [2007]. Since the edges of time-continuous triangles (*TCT*) are bi-linear patches rather than planes, their final rendering calculation is very expensive. Therefore, Akenine-Möller et al. [2007] use a screen space acceleration structure that is created on-the-fly by rendering an OBB around each TCT. In addition, they use Zmax-culling to conservatively reject triangles and fragments before performing actual sample evaluation.
Based on this, Fatahalian et al. [2009] partition time into intervals to keep track of moving micropolygons using different approaches for cases with no motion, slow motion and fast motion. Hou et al. [2010] further improve on this by presenting a unified approach for all cases that usually produces higher quality images. Finally, Laine et al. [2011] improve over Fatahalian et al. [2009] with a better sampling pattern defined in dual space.

For better depth test performance, Boulos et al. [2010] define the tz-pyramid. It stores not only Zmax for a certain instance in time but also builds a hierarchy over time, based on the maximum z-values in each node of the z-pyramid.
McGuire et al. [2010] propose a hybrid algorithm based on stochastic rasterization that can do motion blur and defocus at the same time, running on conventional GPUs. While the authors use motion blur and defocus as examples, their focus is on implementing stochastic rasterization on GPUs in general.
To improve performance, Clarberg and Munkberg [2014] propose a deferred shading approach. They first create a per-pixel list of primitives that contribute to each pixel. In a separate shading pass, the primitives are shaded and the colors are averaged. In this setting, the per-pixel list already accounts for motion blur and defocus. In order to increase the stochastic rendering performance in general, Wu et al. [2015] implement efficient sample culling for motion blur and defocus.

8.2.3 Acceleration structure

Acceleration structures are a key component for fast and efficient ray tracing. Most widely used structures are bounding volume hierarchies (*BVHs*) and *k*d-trees, which can be extended to support motion blur. Cook et al. [1984] introduce stochastic sampling to solve distribution effects but leave the question of an efficient acceleration structure open.
Glassner [1988] proposes an acceleration structure that is based on an octree over objects. Each node is split until it contains only a single object or a maximum split level is reached. Each partial object contained entirely in a single octree node is then bound by 4D k-DOPs.

***k*d-Tree** Olsson [2007] extended *k*d-trees by adding a temporal split in the time domain. The increasing number of object references introduce a significant memory overhead, which limits its practical applicability.

BVH The fastest BVH construction algorithm on the GPU is currently the LBVH [Lauterbach et al., 2009] with optimizations from Karras [2012]. It uses Morton codes for the center of each primitive and applies Radix sort to create a linear list of primitives. This linear list is then used to construct the final bounding volume hierarchy. While being extremely fast to create, its efficiency during rendering is up to 85% lower when compared to SAH based construction. Grünschloß et al. [2011] propose a 4D space-time extension to the spatial split BVH algorithm [Stich et al., 2009] called MSBVH. It is mainly suited for irregularly tesselated polygonal scenes and the high construction time renders it mainly relevant for production rendering settings.

8.3 Architecture

Our unified rendering pipeline is focused on combining transparent and opaque fragments with defocus and motion blur. We thus decouple spatial and temporal sampling to reduce the rasterization effort. We also separate visibility and shading by employing late shading after the motion sampling. This allows us to perform correct alpha blending of blurred and transparent fragments. Accurate reflection of the environment for all fragments is accomplished as the reflection vector is computed for the position of the fragment in time. Also,

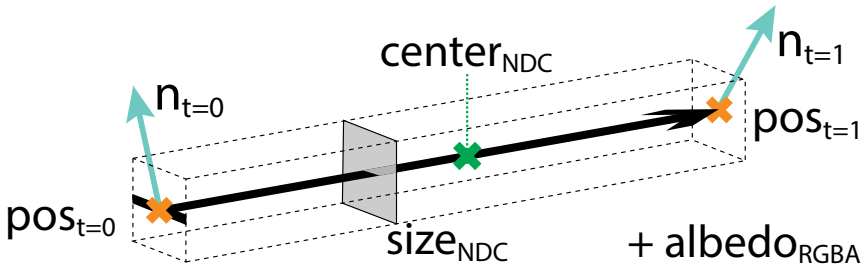

Figure 8.2: All components of a *t*-fragment in NDC space.

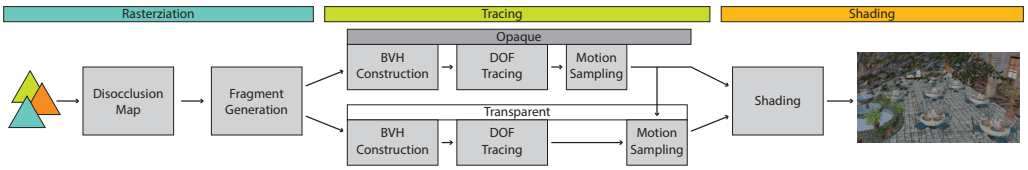

Figure 8.3: Overview of our rendering pipeline consisting of rasterization (generation of *t*-fragments), tracing (spatial and temporal sampling) and shading (lighting and transparency handling). Note that all of these steps are running on the GPU.

our unified rendering pipeline can easily be combined with tile-based deferred rendering to reduce shading overhead.
A central concept is the temporally-varying fragment or *t*-fragment (see Figure 8.2). Without loss of generality, we define the duration of a frame as $[0, 1]$ and we assume that the motion of fragments is linear within frames [McGuire et al., 2010]. Given a regular fragment created at $t \in [0, 1]$, a *t*-fragment represents the oriented line connecting the fragment's positions at $t = 0$ and $t = 1$ in normalized device coordinate (*NDC*) space. The *t*-fragment contains all shading information (time dependent normals, albedo and transparency) and implicitly the size of its footprint. Using the linear motion assumption, we can interpolate these attributes linearly and we derive a capsule with an ϵ radius as ideal object oriented bounding volume for each *t*-fragment.

8.3.1 Overview

Our algorithm consists of three main steps (see Figure 8.3): The first step is the rasterization that creates the disocclusion map and the *t*-fragments. The second step includes the depth-of-field raytracing to resolve general visibility and the motion blur time sampling to determine time-dependent visibility. The last step is the final shading and tone-mapping. The initial pass in the rasterization renders all opaque objects to create the disocclusion map, as well as the first and second layer depth buffer. The disocclusion map (see Figure 8.4 (c)) marks possible disocclusions resulting from camera and object motion or depth-of-field. We create a new depth buffer by combining the first and second layer using the disocclusion map. This buffer is used for early fragment culling during the rasterization of the *t*-fragments in the next pass. Since a *t*-fragment expands over several pixels with high overlap depending on the amount of motion, a fragment linked-list is not a suitable data structure. Thus, we separate opaque and transparent fragments into two different unordered arrays.

In the ray tracing step we construct two bounding volume hierarchies using the unordered arrays. The hierarchies have two different bounding volumes, namely capsules at leaf nodes and temporally-varying axis-aligned bounding boxes (t-AABBs) for the inner nodes to resolve visibility by ray tracing. We implemented a naïve depth-of-field algorithm that samples the lens aperture [Shirley and Chiu, 1997] and trace a fixed number of rays per pixel according to the thin-lens model. This yields a set of t-fragments per viewing ray containing all t-fragments intersected at some point in time $t \in [0,1]$. In the motion sampling pass, we instantiate each capsule for one or multiple $t \in [0,1]$ as a small sphere with radius ϵ at the interpolated location. Fragments are created for all instances that intersect with the viewing ray, sorted according to their depth, and shaded with correct alpha blending for transparency.

8.3.2 Generation of t-fragments

To generate t-fragments, we rasterize the scene at $t = 1$. This will remove artifacts caused by objects appearing during the rendered frame while potentially loosing disappearing objects. Applying a guard band can reduce the artifacts further. In general, any time between 0 and 1 can be chosen in this step. In order to save resources in later stages of the pipeline, we propose a number of additional steps that allow us to reduce the number of t-fragments while at the same time ensuring approximate correctness of the output images.

Disocclusion Map The purpose of the disocclusion map is to efficiently cull t-fragments which will not contribute to the final image (see Figure 8.4). We first perform depth only rendering of all front-facing opaque primitives at $t = 1$ to extract the first and second depth layers. Without disocclusion we would only store opaque t-fragments which correspond to the first depth layer and transparent t-fragments in front of the first layer. However, we have to store all t-fragments which are not visible at $t = 1$ but might become visible due to motion in the frame or depth-of-field rays that "look behind" edges. We detect these disocclusions on a per pixel basis and set the values of the first layer depth map to the respective values of the second layer depth map at these locations.
To detect disocclusions caused by motion we compute a velocity map (motion field) for the first opaque layer. Next we compute forward differences of the velocity map in x and y direction. Motion disocclusions can only occur at pixels that have a positive velocity difference. By looking at the depth of the neighboring (right and top) pixels in relation to the current pixel's depth we can decide in which direction the disocclusion needs to be resolved. The velocity difference value corresponds to the severity of the disocclusion and is converted to pixel units. Each pixel now carries information on whether and how big of a disocclusion occurs, thus specifying a rectangular area of disocclusion around the pixel. Such a region is specified with parameters l, r, t, and b for the extent of the disocclusion in left, right, top, and bottom direction respectively. Pseudocode for motion disocclusion map initialization is shown in Algorithm 6 in the appendix. To obtain the binary occlusion map the disocclusion area information from each pixel needs to be collected. This is done by pixels iteratively spreading their disocclusion information over the entire buffer. In the first iteration each pixel will spread its disocclusion information to its direct neighbors and with each iteration the distance that the information is spread will be doubled until the disocclusion information has spread over the entire occlusion map (see pseudocode for

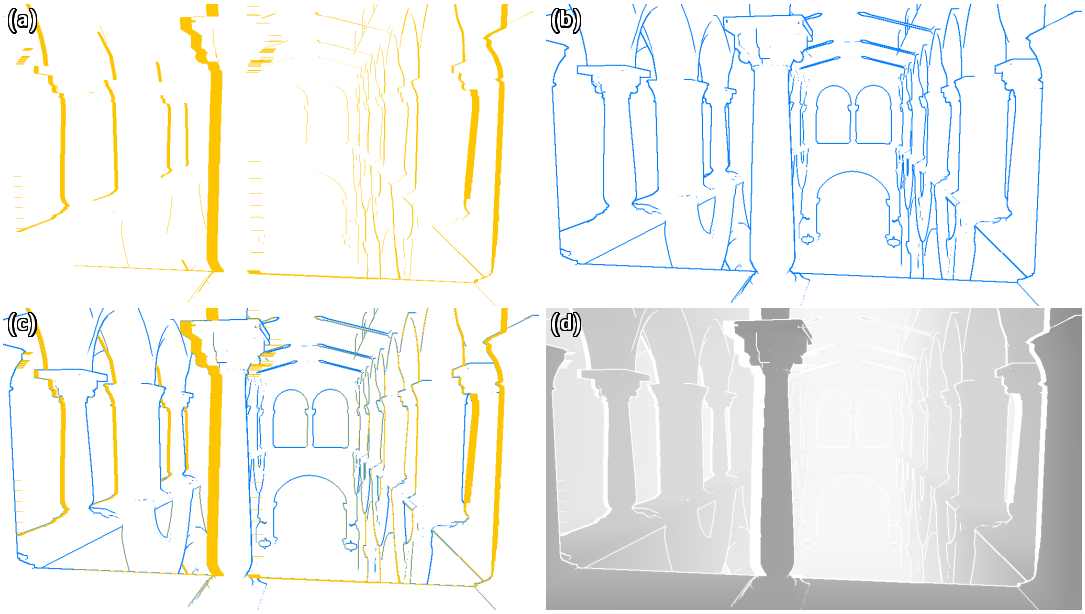

Figure 8.4: Disocclusion map construction: Given the velocity of fragments, we generate an **(a)** occlusion buffer by propagating the velocity differences to neighboring pixels. Next, depth edges are detected and stored in the **(b)** Laplacian buffer. We combine both buffers to the **(c)** disocclusion map, which is used to generate the **(d)** modified depth buffer. This depth buffer is used for early fragment culling. Initially the depth of the first depth layer is used. At pixel positions marked by the disocclusion map the depth value of the second depth layer is used.

disocclusion spreading Algorithm 5).

Correct depth-of-field rendering requires access to geometry that is occluded in the traditional pinhole camera setting. In an effort to balance efficiency and accuracy, we allow for a limited amount of disocclusion in the depth-of-field case. We detect depth discontinuities by thresholding the response of a Laplacian filter on the depth map and store the resulting value in a binary Laplacian map.

In the final step we simply combine the binary occlusion map and the Laplacian map with a simple **OR** operation to yield our novel disocclusion map. Each pixel in the disocclusion map marks whether it is a source of disocclusion or not. Using this information we modify the first layer depth map. For every pixel marked in the disocclusion map we set the depth value back to the second layer. Otherwise we use the first depth layer's value. Figure D.2 shows the difference between using one and two layers.

Generation In a final rasterization pass we create the actual t-fragments. We again start by rendering the scene at $t = 1$ but use the now read-only *modified* depth buffer to enable early fragment culling (at $t = 1$) in hardware. For all fragments that pass the depth test we create a corresponding t-fragment and store it in an unordered array.

8.3.3 Ray tracing acceleration structure

To accelerate collecting *t*-fragments for depth-of-field samples we construct a temporally-varying bounding volume hierarchy (*BVH*). Since *t*-fragments are stored in NDC space, the BVH is built in NDC space as well. Viewports with an aspect ratio $\neq 1$ cause an anisotropic scaling of *t*-fragment coordinates in NDC space. To avoid this distortion we store positions in an *anisotropic* NDC space, preserving the viewport's aspect ratio in the x- and y-dimension. In this undistorted space, we can bound a *t*-fragment with a moving sphere for cheap intersection testing. We define the radius ϵ of the sphere as the radius of the circumsphere of a cube with the side length of a pixel in anisotropic NDC space. Projecting *t*-fragment motion into 3D-space results in a capsule as the bounding volume, which is defined by the two *t*-fragment positions points and a (constant) radius. Thus, no additional memory is needed for *t*-fragment bounds.

We use a fast bottom-up approach for hierarchy construction inspired by LBVH [Lauterbach et al., 2009]. First, all capsules are sorted according to the Morton code of their mid-point in NDC space. We then generate the topology of the hierarchy from this sorted list with the fast construction algorithm of Karras [2012]. *t*-Fragments with identical Morton code belong to the same leaf node. The resulting topology corresponds to a BVH constructed with a spatial-median split strategy. As *t*-fragments are roughly uniformly distributed in x and y direction, this yields a good quality BVH.

Next, we compute bounds for the leaves and inner nodes. We first tried to use tight irregular capsules with different radii at the end points as node bounds. This requires computing tight bounding spheres for a leaf's *t*-fragment bounding spheres at $t \in \{0,1\}$, which is non-trivial for more than two spheres. For inner nodes computation of irregular capsules from children bounds is simple but bounding efficiency proved to be suboptimal and decreased with each level up the hierarchy. Furthermore, intersecting an irregular capsule is non-trivial and expensive. Thus we decided to use temporally-varying axis aligned bounding boxes (*t*-AABBs) for node bounds. *t*-AABBs are defined by a pair $aabb_{t=0}$ and $aabb_{t=1}$ of AABBs for $t \in \{0,1\}$. For leaf nodes computing tight AABBs for *t*-fragment bounding spheres at $t \in \{0,1\}$ is fast and simple, as is propagating *t*-AABBs up the hierarchy. At the same time bounding efficiency is higher than for irregular capsules.

Intersection Intersecting a *t*-AABB with a ray at $t \in [0,1]$ only requires to linearly interpolate $aabb_{t=0}$ and $aabb_{t=1}$ and intersect the interpolated AABB (see Figure 8.5, left). We are instead interested in intersecting a ray against the temporal projection of a *t*-AABB into 3D-space. One possible approach for this is to construct a shaft for $aabb_{t=0}$ and $aabb_{t=1}$ Haines and Wallace [1994], and intersect the ray with the resulting polyhedron. Since an efficient implementation requires to store a significant amount of precomputed data per node, we propose a slightly conservative but simpler *t*-AABB intersection test, which needs no additional memory.

The key idea is to transform the *t*-AABB into a local ray space where the intersection test can be reduced to a two dimensional problem. For this, we first construct an arbitrary 3D orthonormal basis *B* from the ray direction (e.g. [Frisvad, 2012, Hughes and Möller, 1999]) once before traversal of the BVH. Using *B* and the ray origin, we perform an affine transformation of the *t*-AABB into local ray space, where the ray origin is at $(0,0,0)$ and the ray direction corresponds to the z-axis. At this point we can ignore the local z-dimension and reduce the intersection test to a 2-dimensional problem in the local x-y-plane. We

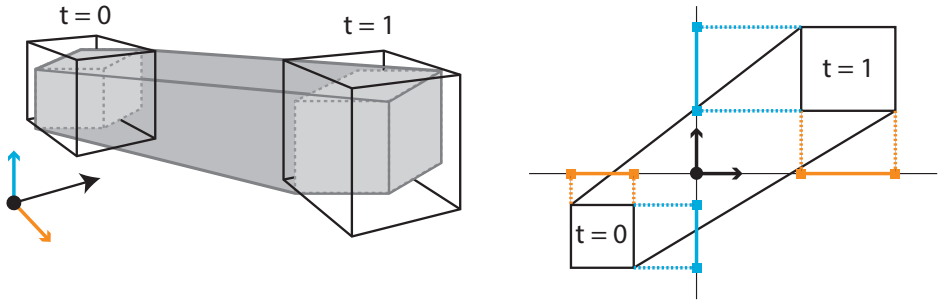

Figure 8.5: Intersection of a ray with a t-AABB. The t-AABB is transformed into a local ray coordinate system, which reduces the intersection test to a 2D problem. The AABBs at $t \in \{0, 1\}$ are conservatively replaced with axis aligned rectangles. Now intersection time intervals are computed separately for projections on both local coordinate axes. A non-empty intersection of the time intervals indicates an intersection of the t-AABB.

could perform a point-in-convex-hull test but extraction of the convex hull is too costly to be performed frequently during traversal. Instead, we test for inclusion of the local origin in the temporally varying projection of the t-AABB on the local x-y-plane. To simplify this test we conservatively replace the projections of $aabb_{t=0}$ and $aabb_{t=1}$ with tight temporally varying axis aligned bounding rectangles. This allows us to further reduce the problem to two one dimensional intersection problems, where we simply have to compute the time intervals for which the local origin is separately contained in the projection of the bounding rectangle onto the local x- and y-axis. If the intersection of both time intervals is non-empty, we intersect the temporally varying axis aligned bounding rectangle and conservatively assume that the t-AABB has been intersected (see Figure 8.5, right). The intersection test is exact if the basis vectors of B are parallel to the NDC coordinate axes. Important for the efficiency of our approach is the fact, that we do not have to actually transform all eight corners of $aabb_{t=0}$ and $aabb_{t=1}$ to compute the projections on the local x- and y-axis. Analyzing the signs of the components of the basis vectors of B we can derive two extreme points for each projection axis, and $aabb_{t=0}$ and $aabb_{t=1}$ separately which suffice to compute the projection bounds, and thus greatly reduce computational cost.
The ray tracer uses a simple stack based ray traversal algorithm. Rays are less coherent for depth-of-field rays. This can cause different rays to find leafs at different points in time. To improve SIMD efficiency in such situations we employ the two-phase while-while traversal from Aila and Laine [2009].

8.3.4 Motion sampling and shading

After collecting all t-fragments we sample in time to generate a final fragment. Given a time sample $\tau \in [0, 1]$ we first find the closest intersection with an opaque t-fragment by intersecting the given ray with the time sampled bounding circumspheres of all t-fragments at $t = \tau$ (see Figure 8.6 and Figure 8.7 in the supplemental material for details). The sphere test is used for fast early rejection. In case of a hit, a tight 3D cube with side length of a pixel in anisotropic NDC is used for the final intersection. The distance of the closest opaque hit point yields the maximum distance for sampling transparent motion: We collect a fixed number (up to 16 in practice) of intersected transparent t-fragments in a

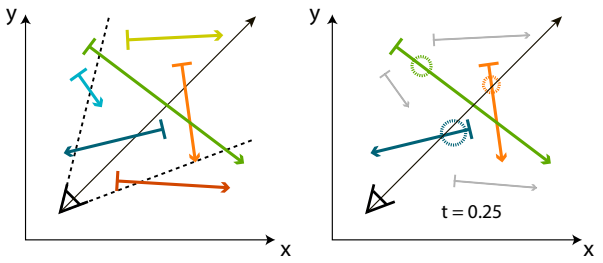

Figure 8.6: **(Left)** A set of t-fragments and their linear movement between $t = 0$ and $t = 1$. Three of the t-fragments are potentially visible for the given viewing ray. **(Right)** To sample the t-fragments at a specific time $t = 0.25$, we create spherical candidate fragments and check for intersection with the viewing ray.

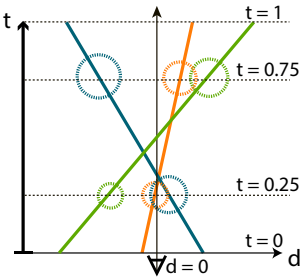

Figure 8.7: The t-fragments for $t = 0.25$ (see Figure 8.6) visualized according to their world-space distance from the viewing ray over time.

similar way to opaque t-fragments. These fragments are sorted back-to-front for shading using an odd-even mergesort sorting network. To reduce memory overhead, we only store the fragment IDs of the opaque and transparent t-fragments plus the sampled time needed to reconstruct the intersection point.
In the shading pass we interpolate the time varying fragment attributes, in our case the position and normal, and shade the fragment according to the material parameters. We first shade the opaque fragment. Afterwards we successively shade the back-to-front sorted transparent t-fragments and blend them together.

8.4 Evaluation

We evaluate our approach using a system equipped with an Intel Core i7-3930K, 64GB of RAM, and an NVIDIA Geforce GTX TITAN X with 12GB of RAM. Unless otherwise noted, timings are measured at a resolution of 1024×576 pixels (1120×672 pixels including the 96 pixels guard band). We use different test scenes and animations with varying amount of camera and object motion, number of transparent layers, and geometric complexity (see Figure 8.8). Besides the SPONZA scene we use the CHALET and SAN MIGUEL scenes, which have higher geometric complexity. The CHALET scene contains many transparent objects (small detailed leafs of the trees, windows, and the balconies), thus creating many transparent fragments and disocclusions to be handled by our algorithm.

8.4.1 Memory consumption

In addition to the memory used by the scene itself, we need buffers for the temporary data which are allocated in advance and must handle the peak usage. The early fragment culling using our disocclusion map significantly reduces the number of opaque fragments without losing quality. The size of the unordered array for storing the t-fragments depends on the resolution and amount of disocclusion. A t-fragment consists of 64 bytes: the positions and normals for $t \in \{0, 1\}$ as well as the unique albedo. A fixed amount of memory is consumed by the temporary result buffers of the depth-of-field tracing. A ray

SPONZA
262K: $2ms + 0.75ms$

Chalet
4,750k: $2ms + 5.5ms$

San Miguel
6,650k: $2ms + 4.7ms$

Figure 8.8: Scenes used for the evaluation, including triangle count, average disocclusion map and t-fragment generation time.

	Fragments	Scenes		
		Sponza	Chalet	San Miguel
Unordered array	OPAQUE	46.0	275.0	
	TRANSPARENT	46.0	183.0	
FLBVH	OPAQUE - TREE	252.6	758.0	
	OPAQUE - TMP CONSTRUCTION	63.0	190.6	
	TRANSPARENT - TREE	252.6	505.3	
	TRANSPARENT - TMP CONSTRUCTION	63.0	127.1	
Tracing result	JOINT STRUCTURE	991.6		
Motion sampling result	OPAQUE	31.5		
	TRANSPARENT	51.9	260.5	

Table 8.1: Maximum memory consumption for each scene in Mbyte at a resolution of 1024×576 (plus guard band) pixels. The amount needed depends primarily on the resolution, the amount of disocclusion, depth complexity, and the number of transparent objects.

consists of 16 bytes (the lens sample (u, v) and the two head pointers for the two linked lists) plus 8 bytes for each t-fragment it collects and stored in a linked list (the t-fragment id and the pointer to the next t-fragment). We use a batchsize of one million rays and limit the number of t-fragments per ray to 128. As seen in Figure D.3, this has no impact on the maximum motion vector. A motion sample uses at least 16 bytes (sampled timer, distance ray t, t-fragment id, head pointer to the linked list for transparent t-fragments). If the sample contains transparent fragments an additional 68 bytes (t-fragment ids) for at most 16 transparent layers are needed. To limit the memory consumption, we use a batch size of two million motion samples. In addition we assume that only 10% of the samples in a batch contain transparent fragments in case of the Sponza and 50% for the Chalet and San Miguel scene.

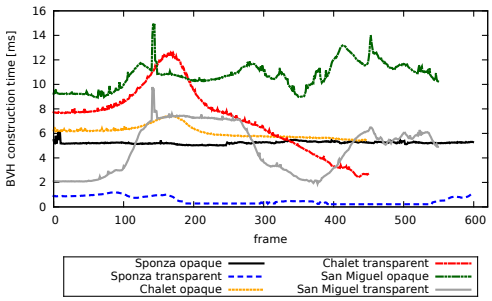

Figure 8.9: BVH construction times in milliseconds for each scene.

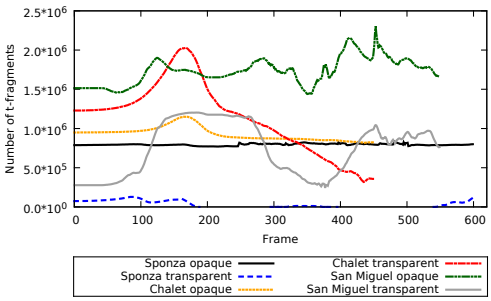

Figure 8.10: Number of opaque and transparent fragments generated using the disocclusion map for early fragment culling.

8.4.2 Construction time

For performance reasons and since the *t*-fragments are not consistent between frames, we need to rebuild the BVH for every frame from scratch. Figure 8.9 shows the BVH construction time for our test scenes. The construction time for the opaque and transparent BVH for the Sponza scene is mostly constant over the camera path. The scene has only a small number of transparent fragments and the disocclusions between the depth layers are not large enough to generate a significantly higher amount of additional opaque fragments. In contrast, the highly detailed leafs of the trees in the Chalet and San Miguel scenes with many transparent fragments create more disocclusions, resulting in more *t*-fragments and therefore in an increased construction time (see Figure 8.8). As the camera motion increases the amount of disocclusion increases and thereby the number of fragments (see Figure 8.10).

8.4.3 Motion sampling

First, we compare our algorithm using motion sampling only against a time sampled ray tracing approach. The ray is associated with a time sample t_i during ray traversal and intersected against our proposed BVH at time t_i. Instead we trace between $t = 0$ and $t = 1$ and collect all possible *t*-fragments that can be intersected during that time and sample in time afterwards. The main benefit is to reduce ray traversal costs while creating additional motion samples. Both approaches work on *t*-fragments.

Performance Figure 8.11 shows timings for construction of the BVH, ray tracing (one ray per pixel), and motion sampling using 4 and 32 samples (top for the Sponza scene and bottom for Chalet). As stated in Section 8.4.2, BVH construction time is nearly constant over the animation. Performance peaks occur around frame 250 and 500 in Sponza as the camera motion between two frames becomes larger. As a consequence our approach has to collect and process more *t*-fragments.
The same behavior is also visible in the comparison between the time sampled ray tracing and our decoupled motion sampling. Our approach is slower with 4 samples but scales better with higher number of motion samples. Most GPU threads are reading the same data during the motion sample stage and can offset the higher ray traversal costs.

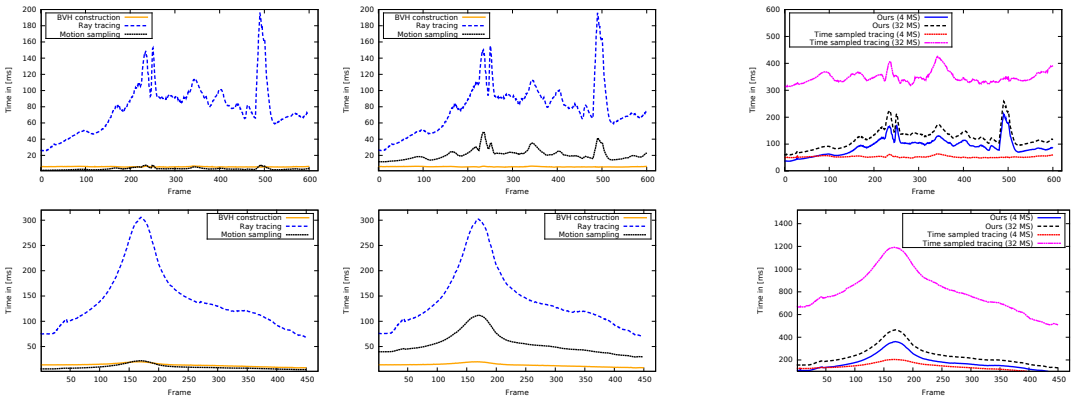

Figure 8.11: The figure shows timings for the BVH construction, ray tracing, and motion sampling passes using **(left)** 4 motion samples and **(middle)** 32 motion samples, respectively. Top row plots give timings for the Sponza and in the bottom the Chalet scene. The **(right)** figure compares overall run-time of our approach with time sampled ray tracing.

The Chalet scene shows similar performance behavior. Overall, performance is slower than in the Sponza scene as more t-fragments are generated (see Figure 8.10) due to the camera motion and more complex geometry, especially the leafs and transparent objects.

Quality Figure 8.12 illustrates a qualitative comparison between our approach and the time sampled ray tracing. The close-up views (Figure 8.12 **(a)** and **(b)**) have no noticeable differences. The difference image (Figure 8.13) shows small discrepancy due to the conservative intersection test during our tracing and motion sampling steps. Small errors (in Figure 8.13 one pixel in size) are visible at edges due to inaccuracies introduced by the ϵ radius approximation used in the intersection test.

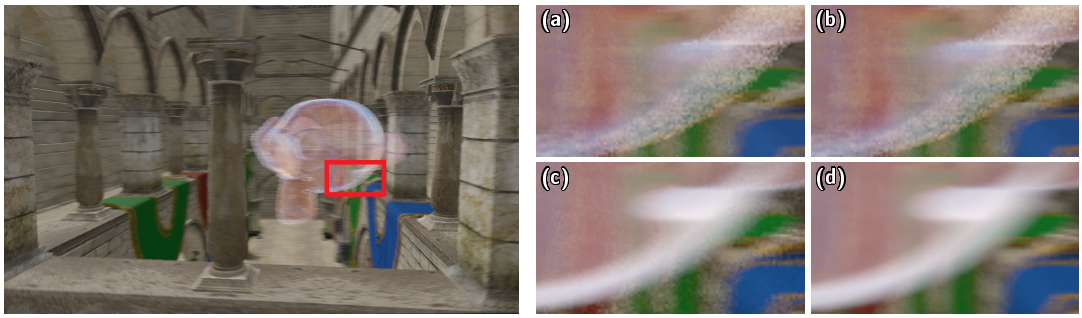

Figure 8.12: Quality comparison of motion blur sampling with **(a)** 32spp for our approach compared to time sampled motion blur tracing **(b)**, respectively. The discrepancy compared to the references (**(c)** with 32spp and **(d)** 256spp) rendered with Blender and Cycles resulting from different shading models in particular for the transparent object. The noise on the wall in the detail images (**(a)** and **(c)**) looks similar. The reference images were rendered in 1.5 and 11.5 minutes respectively, using an Intel Core i7-4870HQ.

Scalability Figure 8.14 shows the scalability of our motion sampling approach with respect to resolution. While increasing resolution, more coherent primary rays are generated

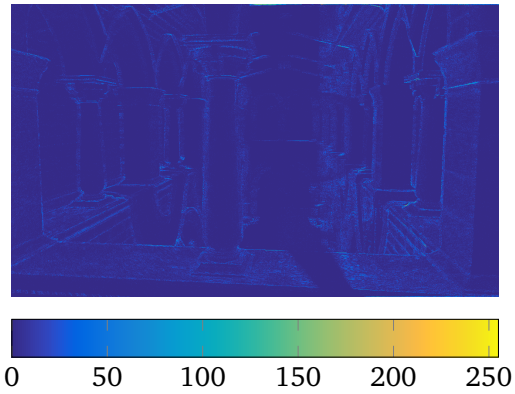

Figure 8.13: Absolute differences of the grayscale images in Figure 8.12 with 32 samples between our approach and time sampled tracing.

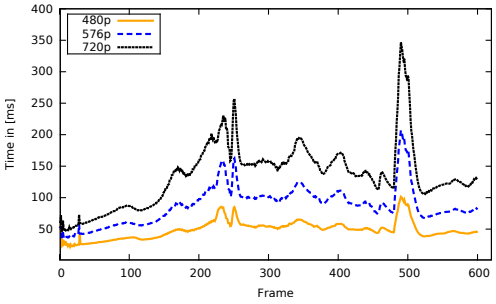

Figure 8.14: Scalability w.r.t. resolution for the Sponza scene for solving motion blur only using 1 depth-of-field ray with 32 motion samples.

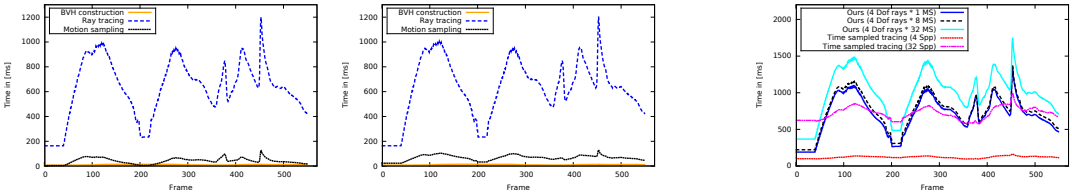

Figure 8.15: The figure shows the timings for the main passes BVH construction, ray tracing, and motion sampling using **(left)** 4 depth-of-field times 1 motion samples and **(middle)** 4 depth-of-field times 8 motion samples for the San Miguel scene. The **(right)** figure compares the overall run-time of our approach with time sampled ray tracing.

per ray batch. These follow similar paths in the acceleration structure during ray traversal which results in better cache utilization and execution flow. A similar cache effect can be observed for motion sampling. Therefore, the performance efficiently scales with increasing number of rays and increasing resolution.

8.4.4 Depth-of-Field

We implemented a simple depth-of-field algorithm that samples the lens aperture [Shirley and Chiu, 1997] and traces a fixed number of rays per pixel according to the thin-lens model.

Performance The detailed timings for our approach using 4 depth-of-field samples show (see Figure 8.15 **(left)** and **(middle)**) that the time for the ray tracing pass is the dominant part in this scenario. The number of incoherent depth-of-field rays (e.g. in case of a large lens radius) reduces cache efficiency and execution flow coherence during ray traversal. The motion sampling pass cannot compensate the tracing overhead as neighboring rays collect less similar t-fragments. Small performance peaks during the motion sampling pass arise when more transparent fragments are generated and shaded. The cost of the BVH construction is negligible. Compared to time sampled ray tracing (Figure 8.15 **(right)**)

with similar sample set-up, our approach is slower during fast motion as the number of collected *t*-fragments increases which explains the variation in rendering performance. On the other hand it still shows better scalability with increasing number of motion samples.

Quality Figure 8.16 shows that our approach does not achieve the same quality compared to time sampled ray tracing in the absence of motion. Full 5D sampling (combined spatial and temporal) reduces noise significantly as for each sample new lens positions are generated (see Figure 8.17).

8.5 Discussion and limitations

Since our technique relies on hardware rasterization, co-planar triangles at $t = 1$ are culled by the hardware. This will lead to artifacts or missing samples in the motion blur and depth-of-field reconstruction. Conservative rasterization can reduce the number of missing triangles but not fully solve the issue. A general problem for real-time motion blur algorithms are fast moving objects that are not visible for $t = 1$ as they pop into the next frame. A guard band can reduce this artifact but cannot handle objects coming from behind the camera.

While our approach cannot compete with specialized defocus or motion blur algorithms for real-time applications (e.g. [Guertin et al., 2014]) in terms of performance, it uses a physically plausible approach with opaque and transparent fragments in a unified pipeline. This results in a higher rendering quality. Note that these specialized approaches do not fit into our pipeline and can thus not be easily implemented.

Shadows are an integral component when rendering photorealistic images. In the case of renderings with motion blur, hard shadow edges can disturb the visual appearance of the image. While we have not implemented motion blurred shadows, our approach can be combined with time-dependent shadow maps (*TSM*) as proposed by Akenine-Möller et al. [2007]. Multiple shadow maps have to be rendered where each layer represents a time slice. In the shading pass a texture lookup is performed using the time sample of the fragment to evaluate the visibility.

Since generating depth-of-field rays is expensive in our approach, it does not perform optimal with large depth-of-field, i.e. extensive blur. However, the other parts of our pipeline, in particular the disocclusion map, the *t*-fragments and the BVH construction can be combined with a non decoupled space-time sampling to jointly handle motion blur, transparency and defocus.

If late shading becomes the bottleneck, a shading cache [Ragan-Kelley et al., 2011] could be used but it will lead to artifacts for fast moving objects with reflective materials.

The noise can be reduced by reusing the set of collected *t*-fragments. A new ray with a small jittered lens position has to be created which is very similar to the original ray. For the new ray we only have to do the motion sampling pass. The results may differ as we cannot resolve all changes in the depth-of-field case and missing disocclusions may generate small artifacts. Geometric aliasing introduced by the rasterization of the *t*-fragments can be reduced by using MSAA and a conservative rasterizer.

8.6 Conclusion

We presented a rendering approach that unifies the handling of order independent transparency, motion blur and defocus in the context of rasterization. Our key idea is to split the sampling phase into independent parts for spatial and temporal sampling. Though gathering moving fragments, which are potentially relevant for a pixel, before time-sampling causes some overhead, this overhead can be quickly amortized with cheap motion samples. Rendering many of those cheap motion samples reduces the noise introduced by motion blur significantly, strongly improving visual quality. In addition our technique is capable of tracing depth-of-field rays with multiple cheap motion samples for improved visual quality.

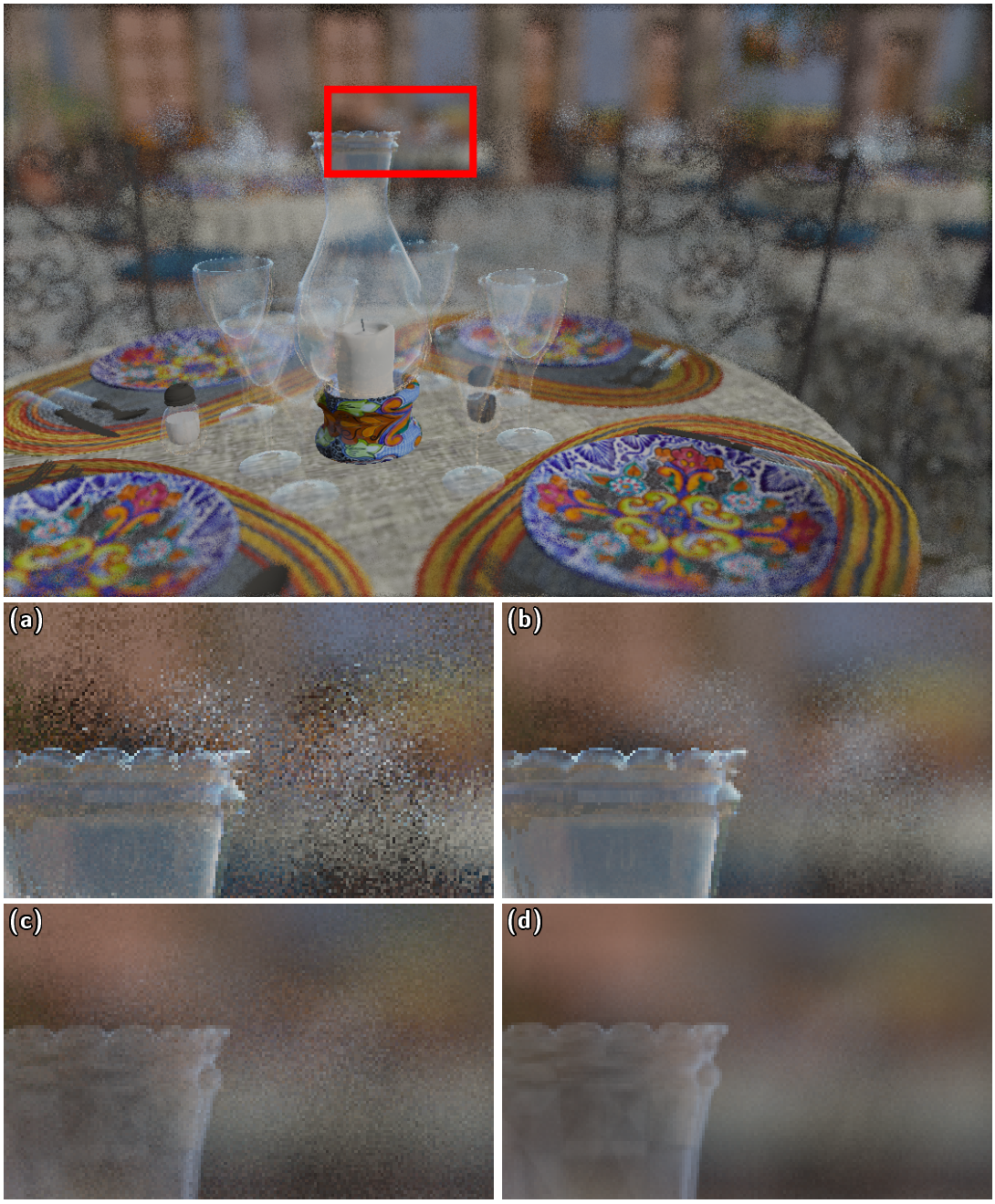

Figure 8.16: Quality comparison of our approach **(a)** using 4 Dof samples times 8 motion samples with time sampled ray tracing **(b)** using 32 samples. Differences of the Blender references (**(c)** 32spp and **(d)** 256spp) to our approach are due to proper 5D sampling. Cycles jitters lens as well as time samples. The reference images were rendered in 2.0 and 14.5 minutes respectively.

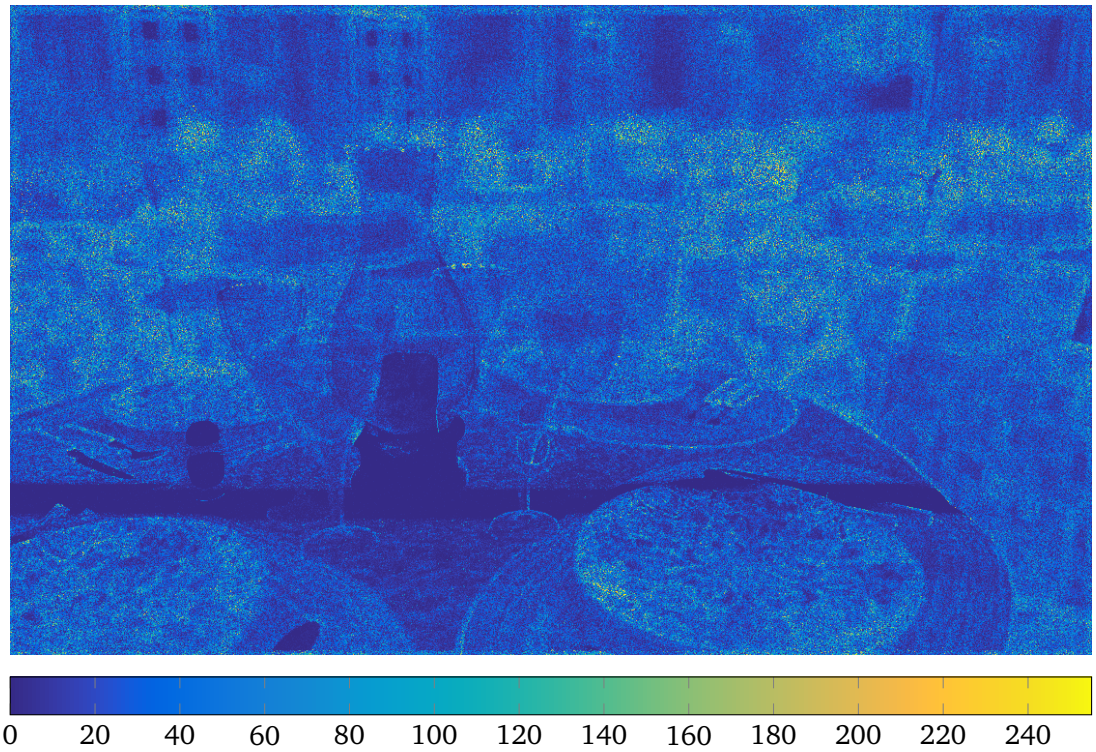

Figure 8.17: Absolute differences of the grayscaled images in Figure 8.16 with 4 depth-of-field sample times 8 motions samples between our approach and time sampled tracing.

Chapter 9

Conclusion

General-purpose computing on graphics processing units becomes more important every year. Besides the classic tasks such as rendering, deep learning, scientific and financial computing are the driving forces for innovations. As the problem size increases, the need for large scale visualization of these complex systems arises. This thesis presented a small step towards interactive visualization in a GPU cluster environment.
The thesis extended the state-of-the-art for efficient screen space ray tracing and hybrid rendering for real-time applications. We have shown an increase in image quality by combining ray tracing and rasterization over different applications such as ray traced reflections, depth-of-field, and motion blur rendering. The acceleration structure in particular could be important for virtual reality systems. These systems need to render frames at 90Hz to 120Hz to alleviate motion sickness. If the frame rate drops one can extrapolate the novel view using multi-view synthesis — which was the driving application of our acceleration structure. One insight of the thesis is the importance of an efficient acceleration structure even for hybrid rendering algorithms. In addition, an aggressive culling of non visible objects as presented in Chapter 8 is very beneficial. One can generalize those insights and apply them to computer science. The biggest performance increase comes from data you do not process.

9.1 Summary

The allocator we presented in Chaper 4 reduces the synchronization overhead for dynamic memory allocations on many-core architectures significantly (Challenge 4 in Section 1.2). Small and frequent parallel allocations from different threads are efficiently merged. In addition we derive a set of constraints to speed-up the dynamic allocation. These can be applied to a wide range of algorithms working within the constraints. A combination of voting function with a fast lock-free allocation inside an own chunk of memory can handle the allocation request efficiently. Thereby our memory allocator outperforms any other GPU based allocator available and we were able to speed up dynamic memory allocation by several orders of magnitude.
Interactive applications and visualization play an important role not only for the graphics community but also for bioinformatics, scientific computing, and large scale data visualization. These areas often use compute clusters with either a CPU only configuration or

a combination of CPU and GPU to solve computation-intensive tasks. Unfortunately, interactive applications suffer from high network latency, limited network bandwidth, and imbalanced workload (Challenge 3 in Section 1.2). This can result in a slow response time of the overall system, unacceptable for any interactive applications. We developed a framework that employs dynamic load balancing and fair scheduling to decrease response time. The proposed scheduling scheme minimizes the average execution time of computational tasks such as the tile-based ray tracing (Section 5.5.2). We show that the applied load balancing capabilities can handle irregular workload such as work generated by production quality rendering system. This allows us to render complex effects almost at interactive frame rates. The proposed system scales nearly linearly with the number of compute devices except for a minimal communication overhead.
The second part of the thesis shed some light on efficient ray tracing for real-time rendering. Screen space ray tracing is an integral part of modern rendering systems [Rasheva, 2015]. The approach is used to render effects such as refraction, glossy reflection, and ambient occlusion [Mara et al., 2013]. We propose an adaptive acceleration structure that is capable of speeding up screen-space ray tracing (Challenge 1 in Section 1.2). The acceleration structure represents the visible scene geometry as a combination of bounding boxes and planar approximations. The planar approximation of the depth buffer allows us to skip empty space and compute exact intersection points. This reduces the oversampling and undersampling of the depth buffer and leads to a better performance and an increased image quality. Dis-occlusions can be handled by a second color and depth layer. A special flag in the acceleration structure indicates a possible dis-occlusion. To solve the dis-occlusion we only trace an additional ray against the second layer if we encounter the flag. We applied the acceleration structure to several different applications, including depth-of-field rendering using an approximation of distributed rendering, multi-view synthesis for light field displays, stereo warping, and screen-space ray traced reflections at real-time frame rates.
After optimizing screen space ray tracing, we examined state-of-the-art rendering systems. A major drawback is the complexity of defocus and motion blur rendering systems when incorporating opaque and transparent geometry. As distributed ray tracing is a general solution to this problem we try to incorporate the algorithm in a unified rendering approach. Thereby, we achieved physically correct results at interactive frame rates for motion blur (Challenge 2 in Section 1.2). Rasterization which solves the visibility at a coarse level and distributed ray tracing are combined. The scene geometry that is potentially visible at any point in time for the duration of a frame is rendered in a rasterization step and stored in temporally varying fragments. A decoupled space and time sampling method allows us to reuse fragments during the sampling step. We gather all moving fragments, which are relevant for a pixel. Afterwards we evaluate the time sample on a subset to avoid the expansive tracing operations. Rendering many of those cheap motion samples reduces the noise introduced by motion blur significantly and improves the visual quality. In addition the approach is capable of tracing depth-of-field rays with multiple cheap motion samples for improved visual quality.

9.2 Discussion and limitations

The dynamic memory allocator can be used in various algorithms used in computer graphics such as ray generation (see Section 1.2) and the construction of an acceleration struc-

ture for ray tracing (e.g. Vinkler and Havran [2014]). The proposed solution outperforms any other approach only if all constraints are fulfilled by the algorithm. Therefore, the allocator cannot be used as an all-round solution. For example, it is often necessary to free memory during the execution at any point in time. Over recent years dynamic memory allocation approaches for GPUs were improved and new specialized algorithms have been proposed. Many new allocation strategies [Vinkler and Havran, 2014] incorporate the merging step to increase SIMD efficiency but do not use the warp local superblock to speed-up the allocator. Unfortunately, the benefits of general purpose allocators on GPUs in other research areas outside computer graphics such as applications in high performance and scientific computing is questionable as most algorithms work on pre-allocated data blocks using fixed-size block allocations.

A natural step to perform interactive path tracing is to distribute the computation on multiple GPUs or large GPU compute clusters. We generalized the question to any interactive application in an heterogeneous compute environment. The main challenges such an application poses to the system is the efficient distribution of tasks. Thereby, most interactive applications suffer from irregular workload. We have shown how to schedule the workload efficiently and use task/work stealing to reduce the overall computation time. Unfortunately, work stealing increases the network traffic. Migrated tasks have to fetch the needed resources (e.g. scene geometry) again if the data is not available on the new compute node. To mitigate the effect we cached data and reduce the network traffic significantly. Another drawback of interactive applications in a cluster environment is the network latency. For camera animations we only send small data packages containing the updated view transformation and get back a small image tile from the compute nodes. This resulted in hundreds and thousands of small data packages which have to be handled in less than 66 milliseconds. We have shown the feasibility of such a complex system. Still there is a lot of engineering on the network layer to do to increase the performance.

To achieve a similar image quality for interactive and real-time application as provided by production quality renderers important effects such as reflections, refraction, and detailed shadows can be rendered using screen space ray tracing. We proposed an adaptive accelerations structure based on the planar approximation of the depth buffer which increased the performance of screen space ray tracing significantly and allowed us to implement distributed ray tracing for depth-of-field and ray tracing of rough reflections for real-time applications. The approximation of the scene depth using planes could lead to tiny holes and cracks. Using a patch based approximation of the depth may reduce the small cracks and increase the image quality [Tevs et al., 2008]. As the acceleration structure only captures the visible geometry it is only suited for screen space effects. The open research question that still remains is how to construct a full 3D acceleration structure for dynamic geometry at real-time frame rates. A possible research direction would be the extension of sparse voxel octrees [Laine and Karras, 2010] in combination with an efficient approximation of the geometry and lazy construction.

To increase the image quality further and incorporate motion blur we traced against capsules instead of planar approximation. We used the capsules to describe the movement of a rasterized fragment between two frames. Based on the soup of capsules we constructed an LBVH and used full ray tracing to collect all fragments that are visible at any time. Two problems arise here. With fast camera motion we render t-fragments with a sizeable pixel footprint and therefore a large bounding volume. The number of collected t-fragments per ray increases significantly, so does the number of complex intersection tests. As a con-

sequence the performance of the algorithm depends on the velocity of the camera — this currently renders the algorithm as impractical for applications such as games compared to post-processing. Unfortunately, this problem is still unsolved. The second problem is the number of *t*-fragments after the rasterization pass. We proposed the dis-occlusion map to cull non-visible *t*-fragments using the early-z test. This resulted in a performance increase as the number of fragments were reduced significantly. The dis-occlusion map combines the occlusion with a buffer propagating the velocity differences and a Laplacian buffer. Instead of the Laplacian buffer a better solution might be to compute the COC for each pixel and propagate the value to the neighboring pixels — similar to the occlusion map. This could result in a better fragment culling in case of depth-of-field. Since our technique relies on hardware rasterization, co-planar triangles are culled by the hardware. This will lead to artifacts or missing samples in the motion blur and depth-of-field reconstruction. The general problem for real-time motion blur algorithms are fast moving objects that are not visible as they pop into the next frame. A guard band can reduce this artifact but cannot handle objects coming from behind the camera.

9.3 Future work

We discussed some future research directions directly related to the presented algorithms and contributions in the previous section. To increase the image quality and performance further, the application of depth-of-field and motion blur reconstruction filter [Hasselgren et al., 2015, Lehtinen et al., 2011] for real-time applications may be a possible extension to the decoupled space and time sampling for motion blur and depth-of-field presented in Chapter 8. A potential issue is the memory consumption of the algorithms as a large amount of rays have to be stored temporally. A possible extension to screen space ray tracing in combination with our proposed acceleration structure would be the integration of packet ray tracing [Gunther et al., 2007]. In certain applications the ray traversal is quite coherent until one reaches a higher mipmap level. One could bundle rays with a similar direction into packages and only trace those packages. Depth-of-field effects with a small aperture and multi-view synthesis algorithms could benefit from this as the number of reduced texture reads and intersection tests may outperform the additional complexity. Further, one could extend the planar approximation of the acceleration structure and take the underlying material properties, such as texture contrast or specular highlights into account. This could further improve the performance and quality of our approach.

Appendices

Appendix A

Fast Dynamic Memory Allocator for Massively Parallel Architectures

```
void __global__ kernel(void) {
  Warp* warp = Warp::start();

  while(condition) {
    void* ptr = warp->alloc(size);

    /* ... some code ... */

    warp->tidyUp();
  }

  warp->end();
}
```

Listing A.1: Simple usage example with first strategy for garbage collection (Section 4.3.4).

```
void __global__ kernel(void) {
  Warp* warp = 0;

  if(threadIdx.x % 5 == 0)
    warp = Warp::start();

  /* ... some code ... */

  void* ptr = warp->alloc(size);

  /* ... some code ... */

  if(threadIdx.x == 5)
    warp->end();
}
```

Listing A.2: Example of a misusage: not all threads request a warpheader but try to allocate memory.

Appendix B

Dynamic Load Balancing and Fair Scheduling for GPU Clusters

```
bool CRenderSceneJob::retrieve()
{
  [...]

  // temp buffer for an image tile
  float* ptr = new float[numPixels * 4];
  clMemcpyDtoH(m_rndFunc, (uchar*)ptr, m_imagePtr, numPixels * sizeof(float) *
       4);

  // transfer rows of the result to the output image
  [...]

  delete [] ptr;

  // free job allocation
  clFree(m_imagePtr);

  return true;
}
```

Listing B.1: The **retrieve()** method copies the result of the job into the image buffer of the application.

```
bool CRenderSceneJob::dispatch(unsigned int id)
{
  // Tile buffer, job allocation
  m_imagePtr = clMalloc(numPixels * sizeof(float4));

  // Bind module and create function handle
  CUKernel* module = task.getModule();
  m_rndFunc = clFunction(module, "render");

  // Create globals and bind a memory block
  constVariable = clModuleGetGlobal(module,
    m_rndFunc, "c_camConf", sizeof(CCameraConfig));

  clMemcpyHtoD(m_rndFunc, constVariable,
    (uchar*)&numSamples, sizeof(CCameraConfig));

  // Set kernel parameter
  clParamIn(m_rndFunc, id);
  clParamIn(m_rndFunc, task.getGeometry());
  clParamIn(m_rndFunc, task.getMaterial());
  clParamIn(m_rndFunc, task.getBVHPtr());

  clParamInOut(m_rndFunc, m_imagePtr);

  //
  clFuncSetCacheConfig(m_rndFunc,
    CU_FUNC_CACHE_PREFER_SHARED);

  // Launch configuration
  clFuncSetBlockShape(m_rndFunc,
    m_blockDimX, m_blockDimY, 1);

  clLaunch(m_rndFunc, m_gridDimX, m_gridDimY);

  return true;
}
```

Listing B.2: The **dispatch()** method enqueues a CUDA module to the job specific command buffer, binds the CUDA kernel, allocates a buffer for the image tile and creates the launch configuration.

Appendix C

Adaptive Acceleration Structure for Screen-space Ray Tracing

C.1 Additional figures

Figure C.1: A comparison of depth reconstruction quality. **(a)** A new view synthesized with our method (using two depth-layers). The full-image took 6.5ms to render. The synthesized depth for **(b)** two-layer and a **(c)** single-layer configuration. **(d)** The reference was generated with NVIDIA OptiX in about 20ms. In both cases we report combined construction and tracing times.

Algorithm 3: GLSL pseudo-code for finding coordinates of the next node that intersects with the ray in screen-space.

```
Function getNextNode(Ray R, ivec2 Q_xy, int Q_level)
  /* get node exit corner coordinate                                   */
1 vec2 B ← (Q_xy + step(0, R.dir.xy)) / 2^Q_level
2 /* get distances to node edges that intersect at B_xy               */
3 vec2 D ← (B − R.origin.xy) / R.dir.xy
4 /* compute position shift                                           */
5 ivec2 S_xy ← sign(R.dir.xy) ∗ step(D.xy, D.yx)
6 return Q_xy + S_xy;                          /* return new position */
```

C.2 Pseudo code

Algorithm 4: GLSL pseudo-code for ray traversal through a single depth layer. For brevity, we assume the quad-tree MIPMAP has power-of-two size.

```
input  : T_{0...n-1} ;                          /* texture MIPMAP storing depth quad-tree */
input  : R ;                                    /* ray structure storing direction and origin */
output: bool rayHit ;                           /* trace result */
output: bool occlusionHit ;                     /* did we hit an occlusion volume? */
output: float d ;                               /* hit-point distance along the ray */
output: vec4 plane ;                            /* hit-plane data */
int Q_level ← n−1 ;                             /* current quad-tree level, start at the root */
ivec2 Q_xy ← ⌊R.origin.xy * sizeof(T_0)/2^Q_level⌋
while insideBounds (pos, T_Q_level) do
    vec2 Q_data ← T_Q_level(Q_xy) ;             /* read the node data */
    if nodeStoresPlane (Q_data) then
        plane ← getPlaneData (Q_data)
        FandN ← getFarNearOfNode (R, Q_xy, Q_level)
        N⃗ ← plane.xyz ;                          /* plane normal */
        P_0 ← vec3 (Q_xy/2^Q_level, plane.w) ;   /* and origin */
        /* compute ray-plane intersection                                  */
        d ← dot(P_0 − R.origin, N⃗)/dot(R.dir, N⃗)
        if dot(R.dir, N⃗) > 0 then /* plane is front-facing the ray        */
            if d < FandN.near then occlusionHit ← true ;
            if d ≥ FandN.near and d < FandN.far then
                rayHit ← true
                plane ← vec4 (N⃗, dot(P_0, N⃗))
                return
            end
        else /* plane is back-facing the ray                               */
            if d ≥ FandN.near then occlusionHit ← true ;
        end
    else
        (hitAABB, hitOV) ← rayIntersectAABB (R, Q_data)
        if hitAABB then
            if hitAABB.near = hitOV.far then
                occlusionHit ← true
            end
            /* progress down to the next child                             */
            ip ← R.dir.xy * hitAABB.near + R.origin.xy
            Q_xy ← Q_xy * 2 + step(0, ip − (Q_xy + 0.5)/2^Q_level)
            Q_level ← Q_level − 1
            continue
        else
            if hitOV then occlusionHit ← true ;
        end
    end
    /* plane or AABB miss, progress to the next node                       */
    /* current node successor position at Q_level                          */
    Q*_xy ← getNextNode (R, Q_xy, Q_level)
    /* compute how many levels up we need to go                            */
    int levelShift ← findMSB ((Q_x ⊕ Q*_x) | (Q_y ⊕ Q*_y))
    /* prevent the traversal from going above the quad-tree root           */
    Q*_level ← min (Q_level + levelShift, n − 1)
    /* update the node location and level to new values                    */
    Q_xy ← ⌊Q*_xy/2^(Q*_level − Q_level)⌋
    Q_level ← Q*_level
end
rayHit ← false
```

Appendix D

Decoupled Space and Time Sampling of Motion and Defocus Blur for Unified Rendering of Transparent and Opaque Objects

D.1 Additional graphs and figures

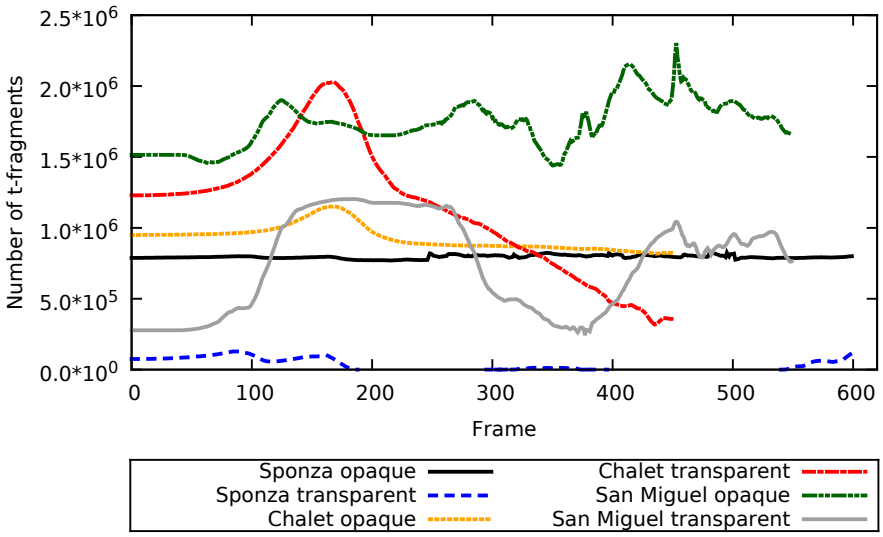

Figure D.1: Number of opaque and transparent fragments generated using the disocclusion map for early fragment culling.

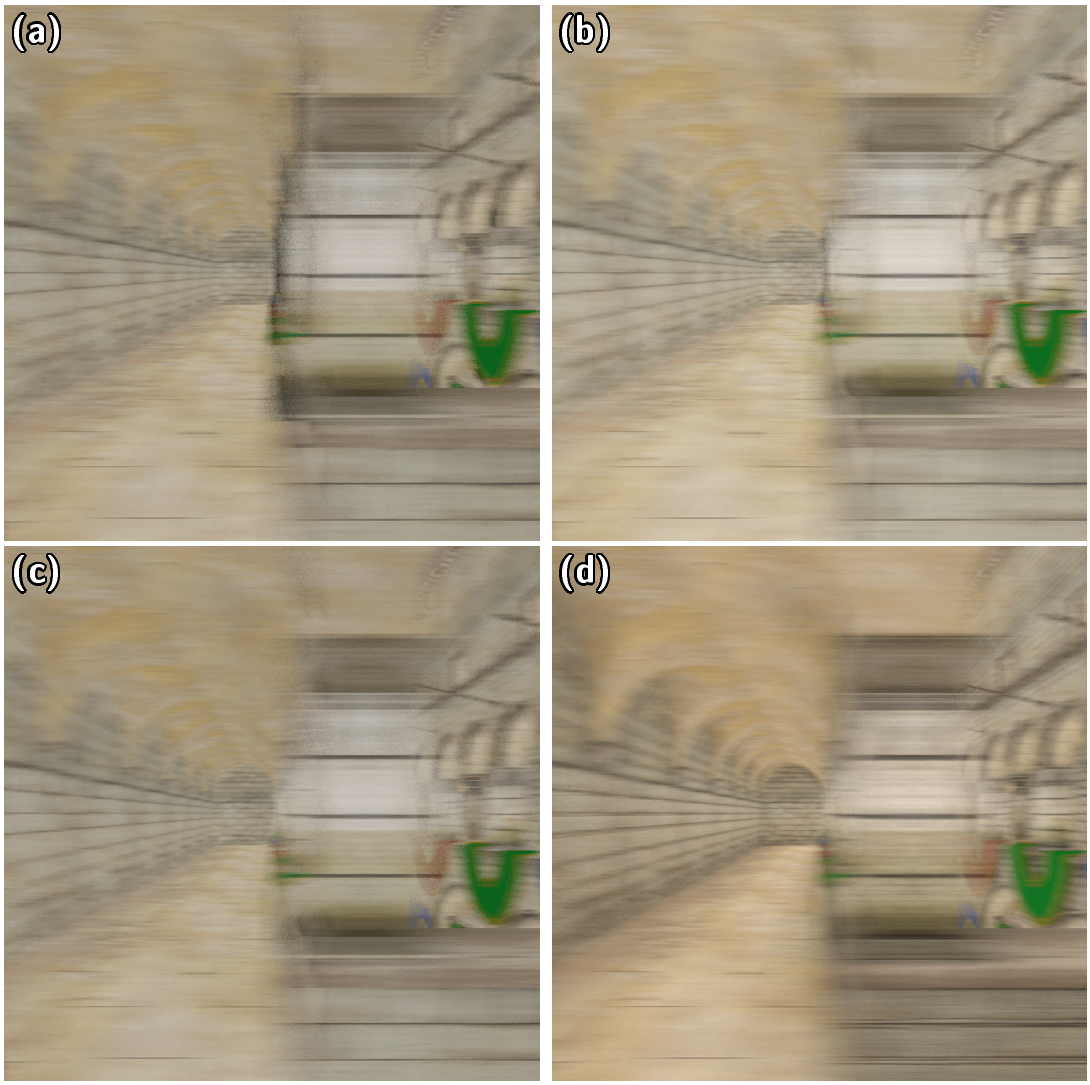

Figure D.2: Comparison of using a two depth layer (**a**) without minimum z-separation, two depth layers with correct z-separation (**b**) and infinite depth (**c**) against Blender reference (**d**) for multiple disocclusions. While there are still some artifact remaining when using the second layer, they are hardly noticeable during animation.

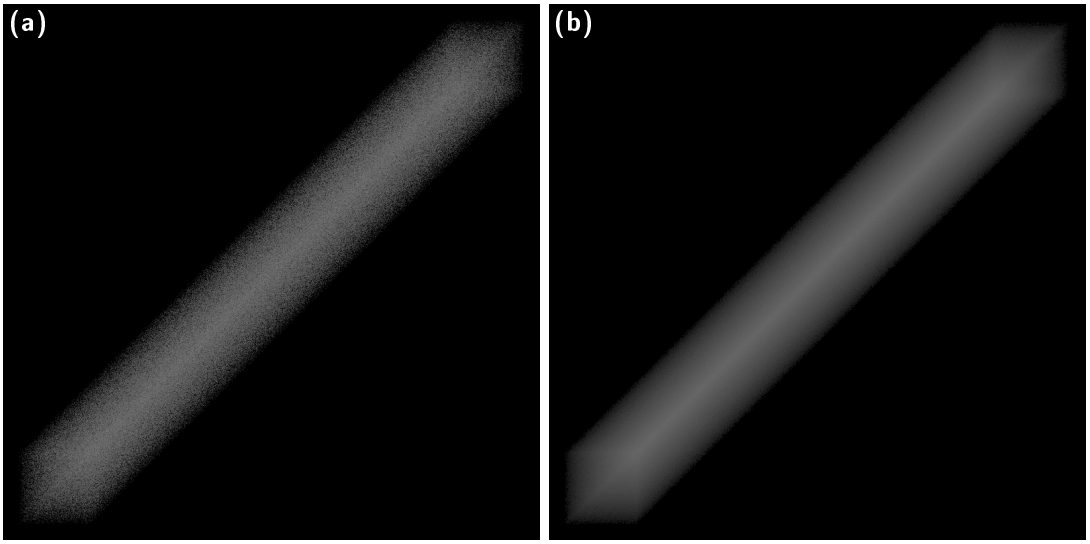

Figure D.3: Comparison of our approach (**a**) against Blender reference (**b**) for very large motion vectors.

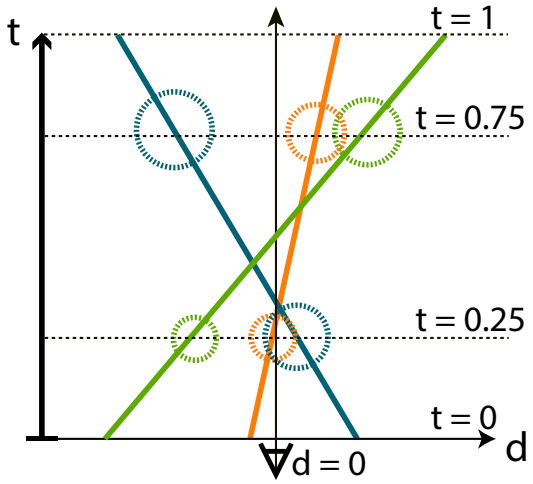

Figure D.4: The t-fragments for $t = 0.25$ (see Figure 8.6) visualized according to their world-space distance from the viewing ray over time.

D.2 Algorithms

Algorithm 5: Pseudocode for spreading of disocclusions caused by motion. Input is a disocclusion map which is initialized with Algorithm 6.

```
in    : dim ;                                   // image dimensions
inout: disMap ;                                 // disocclusion map
// Spread disocclusions
numLevels ← ⌈log2(max(dim.x, dim.y))⌉
level ← 1
m ← 1
srcMap ← disMap
dstMap ← empty
while level ≤ numLevels do
    // Relevant extents and offsets for horizontal and vertical neighbors
    hvNeighbors ← {(r,(−m,0)),(l,(m,0)),(t,(0,−m)),(b,(0,m))}
    // Relevant extents and offsets for diagonal neighbors
    diagNeighbors ← {(rt,(−m,−m)),(lt,(m,−m)),(rb,(−m,m)),(lb,(m,m))}
    foreach p = (x,y) ∈ {0,...,dim.x} × {0,...,dim.y} do
        spread ← srcMap(p)
        // Spread disocclusion from horizontal and vertical neighbors
        foreach (e,o) ∈ hvNeighbors do
            spread.e ← max(spread.e, srcMap(p+o).e − m)
        end
        // Spread disocclusion from diagonal neighbors
        foreach (e,o) ∈ diagNeighbors do
            if all(srcMap(p+o).e − (m,m) > (0,0)) then
                spread.e ← max(spread.e, srcMap(p+o).e − (m,m))
        end
        dstMap(p) ← spread
    end
    level ← level + 1
    m ← m · 2
    swap(srcMap, dstMap)
end
disMap ← srcMap
```

Algorithm 6: Pseudocode for initialization of the disocclusion map for disocclusions caused by motion.

```
in  : dim ;                                          // image dimensions
in  : dMap ;                                         // first layer depth map
in  : vMap ;                                         // first layer velocity map
out: disMap ;                                        // disocclusion map
(vDxMap, vDyMap) ← forward_differences(vMap)
// Initialize the disocclusion map
foreach (x, y) ∈ {0, ..., dim.x} × {0, ..., dim.y} do
    currentD ← dMap(x, y)
    rightD ← dMap(x + 1, y)
    topD ← dMap(x, y + 1)
    vDx ← vDxMap(x, y)
    vDy ← vDyMap(x, y)
    // Compute disocclusion extents at vertical edges
    disocclusionX.(l, r, t, b) ← (0, 0, 0, 0)
    if vDx.x > 0 then
        if currentD > rightD then
            disocclusionX.r ← vDx.x
            if vDx.y > 0 then
                disocclusionX.t ← vDx.y
            else
                disocclusionX.b ← −vDx.y
        else
            disocclusionX.l ← vDx.x
            if vDx.y < 0 then
                disocclusionX.t ← −vDx.y
            else
                disocclusionX.b ← vDx.y
        end
    end
    // Compute disocclusion extents at horizontal edges
    disocclusionY.(l, r, t, b) ← (0, 0, 0, 0)
    if vDy.y > 0 then
        if currentD > topD then
            disocclusionY.t ← vDy.y
            if vDy.x > 0 then
                disocclusionY.r ← vDy.x
            else
                disocclusionY.l ← −vDy.x
        else
            disocclusionY.b ← vDy.y
            if vDy.x < 0 then
                disocclusionY.r ← −vDy.x
            else
                disocclusionY.l ← vDy.x
        end
    end
    disMap(x, y) ← max(disocclusionX, disocclusionY)
end
```

(Co-)Authored Publications

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

Daniel Thuerck, Michael Waechter, **Sven Widmer**, Max von Buelow, Patrick Seemann, Marc E. Pfetsch, and Michael Goesele. A Fast, Massively Parallel Solver for Large, Irregular Pairwise Markov Random Fields, *Proceedings of the 8th Conference on High-Performance Graphics*, Dublin, Ireland, 2016.

Sven Widmer, Dawid Pająk, André Schulz, Kari Pulli, Jan Kautz, Michael Goesele, and David Luebke. An adaptive acceleration structure for screen-space ray tracing. *Proceedings of the 7th Conference on High-Performance Graphics*, Los Angeles, CA, USA, 2015.

Jürgen Bernard, Martin Steiger, **Sven Widmer**, Hendrik Lücke-Tieke, Thorsten May, and Jörn Kohlhammer. Visual-interactive Exploration of Interesting Multivariate Relations in Mixed Research Data Sets. *Computer Graphics Forum*, 2014.

Michael Waechter, Kathrin Jaeger, Daniel Thuerck, Stephanie Weissgraeber, **Sven Widmer**, Michael Goesele, and Kay Hamacher. Using graphics processing units to investigate molecular coevolution. *Concurrency and Computation: Practice and Experience*, 2014.

Daniel Thuerck, **Sven Widmer**, Arjan Kuijper, and Michael Goesele. Efficient heuristic adaptive quadrature on gpus: Design and evaluation. *Parallel Processing and Applied Mathematics*, Springer Berlin Heidelberg, 2014.

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

Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele. Fast dynamic memory allocator for massively parallel architectures. *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, Houston, TX, USA, 2013.

Michael Waechter, Kathrin Jaeger, Stephanie Weissgraeber, **Sven Widmer**, Michael Goesele, and Kay Hamacher. Information-theoretic analysis of molecular (co) evolution using graphics processing units. *Proceedings of the 3rd international workshop on Emerging computational methods for the life sciences*, Delft, Netherlands, 2012.

Michael Waechter, Kay Hamacher, Franziska Hoffgaard, **Sven Widmer**, and Michael Goesele. Is your permutation algorithm unbiased for $n \neq m$? *Parallel Processing and Applied Mathematics*, Springer Berlin Heidelberg, 2012.

Bibliography

Timo Aila and Tero Karras. Architecture considerations for tracing incoherent rays. In *Proc. HPG*, 2010.

Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proc. HPG*, 2009.

Timo Aila, Tero Karras, and Samuli Laine. On quality metrics of bounding volume hierarchies. In *Proc. HPG*, 2013.

Tomas Akenine-Möller, Jacob Munkberg, and Jon Hasselgren. Stochastic rasterization using time-continuous triangles. In *Proc. Graphics Hardware*, 2007.

John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Proc. EG*, 1987.

David P. Anderson. BOINC: A system for public-resource computing and storage. In *Proc. GRID*, 2004.

David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 2002.

Arthur Appel. Some techniques for shading machine renderings of solids. In *Proc. AFIPS*, 1968.

ATI. Stream technology. `http://ati.amd.com/technology/streamcomputing/`, 2007.

Lionel Baboud and Xavier Decoret. Rendering geometry with relief textures. In *Proc. Graphics Interface*, 2006.

A. Barak and A. Shiloh. The virtual opencl (vcl) cluster platform, 2011.

Brian A Barsky, Adam W Bargteil, Daniel D Garcia, and Stanley A Klein. Introducing vision-realistic rendering. In *Proc. EGWR*, 2002.

Laurent Belcour, Cyril Soler, Kartic Subr, Nicolas Holzschuch, and Fredo Durand. 5D covariance tracing for efficient defocus and motion blur. *ACM Trans. Graph.*, 2013.

Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 2000.

R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. Foundations of Computer Science*, 1994.

Solomon Boulos, Edward Luong, Kayvon Fatahalian, Henry Moreton, and Pat Hanrahan. Space-time hierarchical occlusion culling for micropolygon rendering with motion blur. In *Proc. HPG*, 2010.

Huw Bowles, Kenny Mitchell, Robert W. Sumner, Jeremy Moore, and Markus Gross. Iterative image warping. *CGF*, 2012.

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 2004.

B.C. Budge, Tony Bernardin, J.A. Stuart, Shubhabrata Sengupta, K.I. Joy, and J.D. Owens. Out-of-core data management for path tracing on hybrid resources. In *Proc. EG*, 2009.

N. A. Carr, J. Hoberock, K. Crane, and J. C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *Proc. Graphics Interface*, 2006.

T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 1988.

Jiawen Chen, Ilya Baran, Frédo Durand, and Wojciech Jarosz. Real-time volumetric shadows using 1D min-max mipmaps. In *Proc. I3D*, 2011.

Per H. Christensen and Wojciech Jarosz. The path to path-traced movies. *Foundations and Trends in Computer Graphics and Vision*, 2016.

Petrik Clarberg and Jacob Munkberg. Deep shading buffers on commodity GPUs. *ACM Trans. Graph.*, 2014.

Daniel Cohen and Amit Shaked. Photo-realistic imaging of digital terrains. *CGF*, 1993.

D. Cohen-Or, E. Rich, U. Lerner, and V. Shenkar. A real-time photo-realistic visual flythrough. *IEEE TVCG*, 1996.

Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proc. SIGGRAPH*, 1984.

Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. *CGF*, 2011.

Cyril Crassin, David Luebke, Michael Mara, Morgan McGuire, Brent Oster, Peter Shirley, Peter-Pike Sloan, and Chris Wyman. CloudLight: A system for amortizing indirect lighting in real-time rendering. *Journal of Computer Graphics Techniques*, 2015.

Holger Dammertz, Johannes Hanika, and Alexander Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. Proc. EGSR, 2008.

Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *Proc. ISMM*, 2002.

Piotr Didyk, Elmar Eisemann, Tobias Ritschel, Karol Myszkowski, and Hans-Peter Seidel. Perceptually-motivated real-time temporal upsampling of 3D content for high-refresh-rate displays. *CGF*, 2010.

W. Donnelly. Per-pixel displacement mapping with distance functions. In *GPU Gems 2*. Addison-Wesley, 2005.

José Duato, Francisco Igual, Rafael Mayo, Antonio Peña, Enrique Quintana-Ortí, and Federico Silla. An efficient implementation of GPU virtualization in high performance clusters. In *Proc. Euro-Par*. 2010.

Philip Dutre, Kavita Bala, Philippe Bekaert, and Peter Shirley. *Advanced Global Illumination*. AK Peters Ltd, 2006.

Kevin Egan, Yu-Ting Tseng, Nicolas Holzschuch, Frédo Durand, and Ravi Ramamoorthi. Frequency analysis and sheared reconstruction for rendering motion blur. In *Proc. SIGGRAPH*, 2009.

Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. Sorted deferred shading for production path tracing. In *Proceedings of the Eurographics Symposium on Rendering*, Proc. EGSR, 2013.

Manfred Ernst and Gunther Greiner. Multi bounding volume hierarchies. Proc. IEEE/EG IRT, 2008.

Greg Estes. Render me speechless: Get a peek at future of design at siggraph 2015. `https://blogs.nvidia.com/blog/2015/08/05/siggraph/`, 2015.

Zhe Fan, Feng Qiu, and Arie E. Kaufman. Zippy: A framework for computation and visualization on a GPU cluster. *CGF*, 2008.

Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proc. SC*, 2006.

Kayvon Fatahalian, Edward Luong, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proc. HPG*, 2009.

M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. on Computers*, 1972.

Steffen Frey and Thomas Ertl. PaTraCo: A framework enabling the transparent and efficient programming of heterogeneous compute networks. 2010.

Sarah F. Frisken and Ronald N. Perry. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, 2002.

Jeppe Revall Frisvad. Building an orthonormal basis from a 3d unit vector without normalization. *Journal of Graphics Tools*, 2012.

A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated ray-tracing system. *IEEE CG & A*, 1986.

Per Ganestam and Michael Doggett. Real-time multiply recursive reflections and refractions using hybrid rendering. *The Visual Computer*, 2014.

K. Garanzha, J. Pantaleoni, and D. McAllister. Simpler and faster HLBVH with work queues. In *Proc. HPG*, 2011.

Giovanni_cg. Kitchen scene. `https://pixabay.com/de/architektur-design-minimal-1087819/`, 2015.

Andrew S. Glassner. Spacetime ray tracing for animation. *IEEE CG & A*, 1988.

Wolfram Gloger. Dynamic memory allocator implementations in linux system libraries. `http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html`, 1998.

Carl Johan Gribel, Rasmus Barringer, and Tomas Akenine-Möller. High-quality spatio-temporal rendering using semi-analytical visibility. In *Proc. SIGGRAPH*, 2011.

Leonhard Grünschloß, Martin Stich, Sehera Nawaz, and Alexander Keller. MSBVH: an efficient acceleration data structure for ray traced motion blur. In *Proc. HPG*, 2011.

Yunhong Gu, Xinwei Hong, and Robert L. Grossman. Experiences in design and implementation of a high performance transport protocol. In *Proc. SC*, 2004.

Gaël Guennebaud, Loïc Barthe, and Mathias Paulin. Real- time soft shadow mapping by backprojection. In *Proc. EGSR*, 2006.

Jean-Philippe Guertin, Morgan McGuire, and Derek Nowrouzezahrai. A fast and stable feature-aware motion blur filter. In *Proc. HPG*, 2014.

J. Gunther, S. Popov, H. P. Seidel, and P. Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proc. IEEE/EG IRT*, 2007.

Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Proc. SIGGRAPH*, 1990.

D. Häggander and L. Lundberg. Optimizing dynamic memory management in a multithreaded application executing on a multiprocessor. In *Proc. ICPP*, 1998.

Eric A Haines and John R Wallace. Shaft culling for efficient ray-cast radiosity. In *Photorealistic rendering in computer graphics*. 1994.

K. Hamacher. Relating sequence evolution of HIV1-protease to its underlying molecular mechanics. *Gene*, 2008.

Jon Hasselgren, Jacob Munkberg, and Karthik Vaidyanathan. Practical layered reconstruction for defocus and motion blur. *Journal of Computer Graphics Techniques*, 2015.

Robert Herzog, Elmar Eisemann, Karol Myszkowski, and H.-P. Seidel. Spatio-temporal upsampling on the GPU. In *Proc. I3D*, 2010.

Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010.

Qiming Hou, Hao Qin, Wenyao Li, Baining Guo, and Kun Zhou. Micropolygon ray tracing with defocus and motion blur. In *Proc. SIGGRAPH*, 2010.

Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen-mei Hwu. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *Proc. IEEE ICCIT*, 2010.

Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *Proc. ISMM*, 2006.

John F. Hughes and Tomas Möller. Building an orthonormal basis from a unit vector. *Journal of Graphics Tools*, 1999.

INTEL. Intel SPMD Program Compiler. `http://ispc.github.com/`, 2012.

F.E. Ives. Parallax stereogram and process of making same, 1903. US Patent 725,567.

Henrik Wann Jensen. Global illumination using photon maps. In *Proc. EGWR*, 1996.

James T. Kajiya. The rendering equation. In *Proc. SIGGRAPH*, 1986.

Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proc. HPG*, 2012.

Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proc. HPG*, 2013.

J. Kay and P. Lauder. A fair share scheduler. *Commun. ACM*, 1988.

Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

Khronos Group. Opengl 4.4 pipeline map. `https://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGL44PipelineMap.pdf`, 2014.

Donald E. Knuth. *The art of computer programming: fundamental algorithms*. 3rd edition, 1997.

Andreas Kolb and Christof Rezk-Salama. Efficient empty space skipping for per-pixel displacement mapping. In *Proc. VMV*, 2005.

M. Kraus and M. Strengert. Depth-of-field rendering by pyramidal image processing. *CGF*, 2007.

Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proc. Computational Graphics and Visualization Techniques*, 1993.

Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proc. I3D*, 2010.

Samuli Laine, Timo Aila, Tero Karras, and Jaakko Lehtinen. Clipless dual-space bounds for faster stochastic rasterization. In *Proc. SIGGRAPH*, 2011.

Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proc. HPG*, 2013.

Douglas Lanman and David Luebke. Near-eye light field displays. *ACM Trans. Graph.*, 2013.

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO*, 2004.

C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. In *Proc. EG*, 2009.

Doug Lea. A Memory Allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`, 1996.

Sungkil Lee, Gerard Jounghyun Kim, and Seungmoon Choi. Real-Time Depth-of-Field Rendering Using Point Splatting on Per-Pixel Layers. *CGF*, 2008.

Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. Depth-of-field rendering with multiview synthesis. *ACM Trans. Graph.*, 2009.

Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. Real-time lens blur effects and focus control. *ACM Trans. Graph.*, 2010.

Jaakko Lehtinen, Timo Aila, Jiawen Chen, Samuli Laine, and Frédo Durand. Temporal light field reconstruction for rendering distribution effects. In *Proc. SIGGRAPH*, 2011.

G. Lippmann. Épreuves réversibles donnant la sensation du relief. *J. Phys. Theor. Appl.*, 1908.

Joern Loviscach. Motion blur for textures by means of anisotropic filtering. In *Proc. EGSR*, 2005.

Michael Mara, Morgan McGuire, and David Luebke. Lighting deep g-buffers: Single-pass, layered depth images with minimum separation applied to indirect illumination. Technical report, NVIDIA, 2013.

Michael Mara, Derek Nowrouzezahrai, Morgan McGuire, and David Luebke. Fast global illumination approximations on deep g-buffers. Technical report, NVIDIA, 2014.

L C Martin, G B Gloor, S D Dunn, and L M Wahl. Using information theory to search for co-evolving residues in proteins. *Bioinformatics*, 2005.

M. McGuire, E. Enderton, P. Shirley, and D. Luebke. Real-time stochastic rasterization on conventional GPU architectures. In *Proc. HPG*, 2010.

Morgan McGuire and Michael Mara. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques*, 2014.

Morgan McGuire, Padraic Hennessy, Michael Bukowski, and Brian Osman. A reconstruction filter for plausible motion blur. In *Proc. I3D*, 2012.

Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proc. SIGPLAN PLDI*, 2004.

Martin Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH Courses*, 2007.

Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 1997.

Christoph Müller, Steffen Frey, Magnus Strengert, Carsten Dachsbacher, and Thomas Ertl. A compute unified system architecture for graphics clusters incorporating data locality. *IEEE TVCG*, 2009.

Jacob Munkberg, Karthik Vaidyanathan, Jon Hasselgren, Petrik Clarberg, and Tomas Akenine-Möller. Layered reconstruction for defocus and motion blur. *CGF*, 2014.

F. K. Musgrave. Grid tracing: Fast ray tracing for height fields. Technical Report YALEU/DCS/RR-639, Yale University, 1988.

Fernando Navarro, Francisco J. Serón, and Diego Gutierrez. Motion Blur Rendering: State of the Art. *CGF*, 2011.

Diego Nehab, Pedro V. Sander, Jason Lawrence, Natalya Tatarchuk, and John R. Isidoro. Accelerating real-time shading with reverse reprojection caching. In *Proc. Graphics Hardware*, 2007.

Fred E. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Appl. Opt.*, 1965.

J. Nilsson, P. Clarberg, B. Johnsson, J. Munkberg, J. Hasselgren, R. Toth, M. Salvi, and T. Akenine-Möller. Design and novel uses of higher-dimensional rasterization. In *Proc. HPG*, 2012.

J. Novák and C. Dachsbacher. Rasterized bounding volume hierarchies. *CGF*, 2012.

NVIDIA. CUDA compute unified device architecture. `http://www.nvidia.com/object/cuda_home_new.html`, 2015.

NVIDIA. NVIDIA GeForce GTX 1080 — Whitepaper. `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf`, 2016.

K. Oh, H. Ki, and C.-H. Lee. Pyramidal displacement mapping: a GPU based artifacts-free ray tracing through an image pyramid. In *Proc. VRST*, 2006.

J. Olsson. Ray-tracing time-continuous animations using 4d kd-trees. Master's thesis, Lund University, 2007.

Jacopo Pantaleoni and David Luebke. HLBVH: hierarchical LBVH construction for real-time raytracing of dynamic geometry. In *Proc. HPG*, 2010.

Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. Pantaray: Fast ray-traced occlusion caching of massive scenes. *ACM Trans. Graph.*, 2010.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. In *Proc. SIGGRAPH*, 2010.

Kirsty Parkin. Building 3d with ikea. `http://www.cgsociety.org/index.php/cgsfeatures/cgsfeaturespecial/building_3d_with_ikea`, 2014.

Matt Pharr and Greg Humphreys. *Physically Based Rendering, Third Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., 3nd edition, 2016.

Juan Pineda. A parallel algorithm for polygon rasterization. In *Proc. SIGGRAPH*, 1988.

Fabio Policarpo and Manuel M. Oliveira. Relief mapping of non-height-field surface details. In *Proc. I3D*, 2006.

Fábio Policarpo, Manuel M. Oliveira, and J. L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *Proc. I3D*, 2005.

Michael Potmesil and Indranil Chakravarty. A lens and aperture camera model for synthetic image generation. In *Proc. SIGGRAPH*, 1981.

Jonathan Ragan-Kelley, Jaakko Lehtinen, Jiawen Chen, Michael Doggett, and Frédo Durand. Decoupled sampling for graphics pipelines. *ACM Trans. Graph.*, 2011.

Silvia Rasheva. Unity - bedroom demo: archviz with ssrr. `https://blogs.unity3d.com/2015/11/10/bedroom-demo-archviz-with-ssrr/`, 2015.

Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. Approximating dynamic global illumination in image space. In *Proc. I3D*, 2009.

Reuven Y Rubinstein and Dirk P Kroese. *Simulation and the Monte Carlo method*. John Wiley & Sons, 2016.

Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proc. Graphics Hardware*, 2004.

Kai Selgrad, Christian Reintges, Dominik Penk, Pascal Wagner, and Marc Stamminger. Real-time depth of field using multi-layer filtering. In *Proc. I3D*, 2015.

Mike Seymour. Disney's new Production Renderer 'Hyperion' – Yes, Disney! `https://www.fxguide.com/featured/disneys-new-production-renderer-hyperion-yes-disney/`, 2014.

Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. Layered depth images. In *Proc. SIGGRAPH*, 1998.

Peter Shirley and Kenneth Chiu. A low distortion map between disk and square. *Journal of Graphics Tools*, 1997.

Pitchaya Sitthi-amorn, Jason Lawrence, Lei Yang, Pedro V. Sander, and Diego Nehab. An improved shading cache for modern GPUs. In *Proc. Graphics Hardware*, 2008.

Solid Angle. Arnold — Global illumination renderer. `https://www.solidangle.com/arnold/`, 2016.

Tiago Sousa, Nickolay Kasyan, and Nicolas Schulz. Secrets of cryengine 3 graphics technology. In *ACM SIGGRAPH Courses*, 2011.

Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. Scatteralloc: Massively parallel dynamic memory allocation for the GPU. In *Proc. InPar*, 2012.

Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proc. HPG*, 2009.

Magnus Strengert, Christoph Müller, Carsten Dachsbacher, and Thomas Ertl. CUDASA: compute unified device and systems architecture. 2008.

K. Sung, A. Pearce, and Changyaw Wang. Spatial-temporal antialiasing. *IEEE TVCG*, 2002.

Andrew S. Tanenbaum. *Modern Operating Systems*. 3rd edition, 2007.

Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Pearson, 2006.

Natalya Tatarchuk, Chris Brennan, and John Isidoro. Motion blur using geometry and shading distortion. In *ShaderX2 – Shader Programming Tips and Tricks with DirectX 9*. 2003.

Art Tevs, Ivo Ihrke, and Hans-Peter Seidel. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *Proc. I3D*, 2008.

Yasin Uludag. Hi-z screen-space cone-traced reflections. In *GPU Pro 5*. 2014.

Tamás Umenhoffer, Gustavo Patow, and László Szirmay-Kalos. Robust multiple specular reflections and refractions. In H. Nguyen, editor, *GPU Gems 3*. 2007.

Karthik Vaidyanathan, Robert Toth, Marco Salvi, Solomon Boulos, and Aaron Lefohn. Adaptive image space shading for motion and defocus blur. In *Proc. HPG*, 2012.

Karthik Vaidyanathan, Jacob Munkberg, Petrik Clarberg, and Marco Salvi. Layered light field reconstruction for defocus blur. *ACM Trans. Graph.*, 2015.

Dietger van Antwerpen. Improving simd efficiency for parallel monte carlo light transport on the GPU. In *Proc. HPG*, 2011.

Eric Veach. *Robust monte carlo methods for light transport simulation*. PhD thesis, 1998.

Eric Veach and Leonidas Guibas. Bidirectional estimators for light transport. In *Proc. EGWR*, 1994.

Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proc. SIGGRAPH*, 1997.

Marek Vinkler and Vlastimil Havran. Register efficient memory allocator for GPUs. In *Proc. HPG*, 2014.

Marek Vinkler, Vlastimil Havran, and Jiří Bittner. Bounding volume hierarchies versus kd-trees on contemporary many-core architectures. Proc. SCCG, 2014.

Michael Waechter, Kathrin Jaeger, Stephanie Weissgraeber, Sven Widmer, Michael Goesele, and Kay. Hamacher. Information-theoretic analysis of molecular (co)evolution using graphics processing units. In *Proc. ECMLS*, 2012.

Ingo Wald. On fast construction of SAH-based bounding volume hierarchies. In *Proc. IEEE IRT*, 2007.

Ingo Wald. Fast construction of SAH BVHs on the intel many integrated core (MIC) architecture. *IEEE TVCG*, 2012.

Ingo Wald, Carsten Benthin, and Solomon Boulos. Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs -. In *Proc. IEEE IRT*, 2008.

Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. Microfacet models for refraction through rough surfaces. In *Proc. EGSR*, 2007.

P. Weil, F. Hoffgaard, and K. Hamacher. Estimating sufficient statistics in co-evolutionary analysis by mutual information. *Computational Biology and Chemistry*, 2009.

Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 1980.

Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele. Fast dynamic memory allocator for massively parallel architectures. Proc. GPGPU, 2013.

Sven Widmer, Dawid Pająk, André Schulz, Kari Pulli, Jan Kautz, Michael Goesele, and David Luebke. An adaptive acceleration structure for screen-space ray tracing. Proc. HPG, 2015.

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

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. IWMM*, 1995.

Dominik Wodniok and Michael Goesele. Construction of bounding volume hierarchies with SAH cost approximation on temporary subtrees. *Computers & Graphics*, 2016.

Sven Woop, Carsten Benthin, and Ingo Wald. Watertight ray/triangle intersection. *Journal of Computer Graphics Techniques*, 2013.

Yi-Jeng Wu, Der-Lor Way, Yu-Ting Tsai, and Zen-Chung Shih. Clip space sample culling for motion blur and defocus blur. *Journal of Information Science and Engineering*, 2015.

Zhefeng Wu, Fukai Zhao, and Xinguo Liu. SAH KD-tree construction on GPU. In *Proc. HPG*, 2011.

Chris Wyman. Interactive image-space refraction of nearby geometry. In *Proc. GRAPHITE*, 2005.

Qing Xu, Ruijuan Hu, Lianping Xing, and Yuan Xu. Adaptive sampling with Renyi entropy in Monte Carlo path tracing. In *Proc. Int. Symp. on Signal Processing and Information Technology*, 2005.

Xuan Yu, Rui Wang, and Jingyi Yu. Real-time depth of field rendering via dynamic light field generation and filtering. In *CGF*, 2010.

M. J. Zaki, Wei Li, and S. Parthasarathy. Customized dynamic load balancing for a network of workstations. In *Proc. HPDC*, 1996.

Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time KD-tree construction on graphics hardware. *ACM Trans. Graph.*, 2008.

Wissenschaftlicher Werdegang des Autors

2001 – 2008	Studium der Informatik Universität Potsdam
2008	Abschluss: Diplom-Informatiker Abschlussarbeit: „Entwurf und Implementierung eines ECMA-CLI-basierten Betriebssystems”
2009 - 2016	Wissenschaftlicher Mitarbeiter Graphics, Capture and Massively Parallel Computing (GCC) Technische Universität Darmstadt
2013 – 2014	Forschungspraktikum NVidia, Corp., Santa Clara, CA, USA Entwicklung von Algorithmen zur Multi-View Synthese

Ehrenwörtliche Erklärung

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades „Doktor-Ingenieur“ mit dem Titel *„Foundations and Methods for GPU based Image Synthesis “* selbständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen.

Darmstadt, den 17. Februar 2017

Sven Widmer