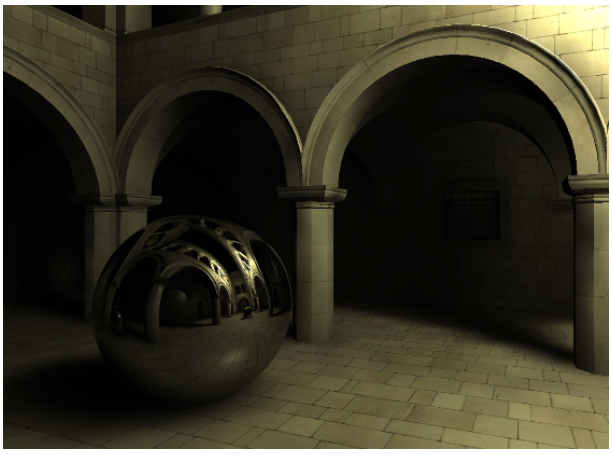
RAY TRACING IN REAL-TIME GAMES



J. Bikker

RAY TRACING IN REAL-TIME GAMES

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft;
op gezag van de Rector Magnificus prof.ir. K.Ch.A.M. Luyben;
voorzitter van het College van Promoties
in het openbaar te verdedigen op maandag 5 november om 12.30 uur
door

Jacobus BIKKER

geboren te Barendrecht

Dit proefschrift is goedgekeurd door de promotor:

Prof.dr.ir. F.W. Jansen

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter
Prof.dr.ir. F.W. Jansen, Technische Universiteit Delft, promotor
Prof.dr. E. Eisemann, Technische Universiteit Delft
Prof.dr. K.L.M. Bertels, Technische Universiteit Delft
Prof.dr. R.C. Veltkamp, Universiteit Utrecht
Prof.dr.ir. P. Dutré, Universiteit Leuven
Prof.Dr.-Ing. P. Slusallek, Universiteit Saarland
Dr.-Ing. I. Wald, Intel Corporation

The research described in this thesis was performed at the Academy of Digital Entertainment of the NHTV University of Applied Sciences, Reduitlaan 41, 4814DC, Breda, The Netherlands.

ISBN 978-90-5335-595-4

And God said, Let there be light: and there was light.
And God saw the light, that it was good:
and God divided the light from the darkness.

Dedicated to the Author of Light.

ABSTRACT

This thesis describes efficient rendering algorithms based on ray tracing, and the application of these algorithms to real-time games. Compared to rasterization-based approaches, rendering based on ray tracing allows elegant and correct simulation of important global effects, such as shadows, reflections and refractions. The price for these benefits is performance: ray tracing is compute-intensive. This is true if we limit ourselves to direct lighting and specular light transport, but even more so if we desire to include diffuse and glossy light transport. Achieving high performance by making optimal use of system resources and validating results in real-life scenarios are central themes in this thesis. We validate, combine and extend existing work into several complete and well-optimized renderers. We apply these to a number of games. We show that ray tracing leads to more realistic graphics, efficient game production, and elegant rendering software. We show that physically-based rendering will be feasible in real-time games within a few years.

SAMENVATTING

Deze thesis beschrijft efficiënte rendering algoritmes gebaseerd op ray tracing, en de toepassing van deze algoritmes in games. Vergeleken met technieken gebaseerd op rasterization stelt ray tracing ons in staat om op een elegante en correcte manier belangrijke globale effecten te berekenen, zoals schaduwen, reflecties en refracties. Ray tracing vergt echter veel rekenkracht. Dit geldt voor directe belichting en perfecte reflectie, maar nog meer voor imperfecte en diffuse reflecties. Centrale thema's in deze thesis zijn het behalen van hoge performance door optimaal gebruik te maken van systeembronnen, en het toepassen van resultaten in realistische scenarios. Wij valideren en combineren bestaand werk en bouwen hierop voort. De resulterende renderers worden toegepast in een aantal games. Wij laten zien dat ray tracing leidt tot realistische beelden, efficiënte game productie, en elegante rendering software. Rendering in games gebaseerd op simulatie van lichttransport is haalbaar binnen enkele jaren.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

J. Bikker and J. van Schijndel, The Brigade Renderer: a Path Tracer for Real-time Games. 2012. Submitted to the International Journal of Game Technology.

J. Bikker, Improving Data Locality for Efficient In-Core Path Tracing. 2012. In: Computer Graphics Forum, Eurographics Association.

J. Bikker and R. Reijerse, A Precalculated Pointset for Caching Shading Information. 2009. In: EG 2009, Short Papers, Eurographics Association.

J. Bikker, Generic Ray Queries using kD-trees. 2008. In: Game Programming Gems 7. Charles River Media.

J. Bikker, Real-time Ray Tracing through the Eyes of a Game Developer. 2007. In: RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing. IEEE Computer Society.

DISSEMINATION

The ideas presented in this thesis have been used in the following articles and products:

Student game "It's About Time". N. Koopman, L. Brailescu, B. de Bree, D. Georgev, T. Verhoeve, S. Verbeek, T. Boone, D. van Wijk, M. Jakobs, K. Ozcan, R. van Kalmhout, J. van Schijndel and J. Bikker, 2012. ADE/IGAD, NHTV, Breda, The Netherlands.

Student game "Reflect". E. Aarts, S. Stroek, M. Pisanu, D. van Wijk, N. van Kaam, A. van der Wijst, D. Shimanovski, S. Vink, J. Knoop, J. van Schijndel and J. Bikker, 2011. ADE/IGAD, NHTV, Breda, The Netherlands.

The Brigade Path Tracer. J. Bikker, J. van Schijndel and D. van Antwerpen, 2010-2012.

Student game “A Time of Light”. M. Peters, B. van de Wetering, W. van Balkom, J. Zavadil, V. Vockel, I. Tomova, M. Goliszec and J. Bikker, 2010. ADE/IGAD, NHTV, Breda, The Netherlands.

Student game “Cycle”. D. de Baets, G. van Houdt, I. Abrossimow, L. Lagidse, N. Ruisch, R. van Duursen, S. Boskma, T. van der Ven and J. Bikker, 2009. ADE/IGAD, NHTV, Breda, The Netherlands.

Student game “Pirates on the Edge”. J. van Schijndel, R. de Bruijne, R. Ezendam, M. van Es, R. van Halteren, C. de Heer, T. van Hoof, K. Baz, S. Dijks, P. Kartner, F. Hoekstra, B. Schutze and J. Bikker, 2008. IGAD/NHTV, Breda, The Netherlands.

Student game “Let there be Light”. K. Baz, M. van Es, T. Van Hoof, D. Hoekstra, B. Schutze, R. de Bruijne, R. Ezendam, Pim Kartner and J. Bikker, 2007. IGAD/NHTV, Breda, The Netherlands.

Ray Tracing Theory and Implementation. J. Bikker, 2006. Seven articles on ray tracing, published on www.flipcode.com and devmaster.net.

Student game “Outbound”. F. K. Kasper, R. Janssen, W. Schroo, M. van der Meide, J. Pijpers, L. Groen, R. Dijkstra, R. de Boer, B. Arents, T. Lunter and J. Bikker, 2006. ADE/IGAD, NHTV, Breda, The Netherlands.

Student game “Proximus Centauri”. M. van Mourik, R. Plaisier, T. Lunter, J. Pijpers, P. van den Hombergh, R. Janssen, E. Verboom, W. Schroo, F. K. Kasper and J. Bikker, 2006. ADE/IGAD, NHTV, Breda, The Netherlands.

The Arauna Real-time Ray Tracer, J. Bikker, 2004-2010.

Interactive Ray Tracing. J. Bikker, 2006. Intel Software Network.

ACKNOWLEDGMENTS

The research described in this thesis was carried out over the course of about eleven years. It started somewhere in 2001, with the discovery of the wonderful world of real-time ray tracing, the challenge I read in Ingo Wald's work, and endless conversations with Thierry Berger-Perrin, which led to the development of the Arauna ray tracer, and the start of the ompf forum. It accelerated when I was invited by Alexander Keller and Carsten Wächter to speak at the RT'07 conference, which in turn led to an incredible summer at Intel in 2008. Many thanks to Jim Hurley, Bill Mark, Ingo Wald, Alexander Reshetov, Ram Nalla, Daniel Pohl, Carsten Benthin and Sven Woop for having me there.

Back in the Netherlands, a guest lecture for Rafaël Bidarra brought me into contact with Professor Erik Jansen, who helped me turn my practical work into scientific form, and allowed me to work with two excellent master students. Roel Reijerse implemented the lightcuts algorithm described in chapter 4. Dietger van Antwerpen worked on the RayGrid algorithm and the CUDA implementation of the path tracer kernels, which influenced greatly the contents of chapters 5 and 6.

This research was carried out in the environment of the IGAD program of the NHTV University of Applied Sciences in Breda. Many programming and visual art students were involved: most of them in one of the GameLab projects, some of them got a little deeper involved. Many thanks to Jeroen van Schijndel for being my research assistant. Thanks to Frans Karel Kasper for representing the 'Arauna team' at the SIGGRAPH'09 conference. Also thanks to all the students and colleagues that patiently heard me out (or not) when I talked too much about ray tracing. IGAD is an incredible environment, and I am proud to be part of it.

Also many thanks to the OTOY people: Alissa Grainger, Jules Urbach and Charlie Wallace, for using Brigade in their cloud rendering products.

Thanks to Samuel Lapère for creating tons of demos based on the Kajiya demo and Brigade source code.

Several people provided advice during this research. Alexander Keller got me through writing my first paper. Ingo Wald provided feedback on early versions of this thesis.

This thesis and the research described in it leans heavily on the creative labor of a large number of talented individuals:

The Modern Room scene that was used in several chapters of this thesis was modeled by students of the IGAD program. The Sponza Atrium and Sibenik Cathedral were modeled by Marko Dabrovic. We also used a version that was heavily modified by Crytek. The Bugback Toad model was modeled by Son Kim. The Lucy Statue and the Stanford Bunny were originally obtained from the Stanford 3D Scanning Repository. The Escher scene was modeled by Simen Stroek.

The games that where produced using Arauna were developed by students of the IGAD program:

"Proximus Centauri" was developed by Mike van Mourik, Ramon Plaisier, Titus Lunter, Jan Pijpers, Pablo van den Hombergh, Rutger Janssen, Erik Verboom, Wilco Schroo and Frans Karel Kasper.

"Outbound" was developed by Frans Karel Kasper, Rutger Janssen, Wilco Schroo, Matthijs van der Meide, Jan Pijpers, Luke Groen, Rients Dijkstra, Ronald de Boer, Benny Arents and Titus Lunter.

"Let there be Light" was developed by Karim Baz, Maikel van Es, Trevor van Hoof, Dimitrie Hoekstra, Bodo Schutze, Rick de Bruijne, Roel Ezendam and Pim Kartner.

"Pirates on the Edge" was developed by Jeroen van Schijndel, Rick de Bruijne, Roel Ezendam, Mikel van Es, Richel van Halteren, Carlo de Heer, Trevor van Hoof, Karim Baz, Sietse Dijks, Pim Kartner, Freek Hoekstra and Bodo Schutze.

"Cycle" was developed by Dieter de Baets, Gabrian van Houdt, Ilja Abrossimow, Lascha Lagidse, Nils Ruisch, Robert van Duursen, Sander Boskma and Tom van der Ven.

"A Time of Light" was developed by Mark Peters, Bram van de Wetering, Wytze van Balkom, Jan Zavadil, Valentin Vockel, Irina Tomova and Marc Goliszec.

Brigade was used for two games:

"Reflect" was developed by Simen Stroek, Marco Pisanu, Dave van Wijk, Elroy Aarts, Nick van Kaam, Astrid van der Wijst, Dimitri Shimanovski, Stefan Vink, Jordy Knoop and Jeroen van Schijndel.

"It's About Time" was developed by Nick Koopman, Lavinia Brailescu, Bart de Bree, Darin Georgev, Tom Verhoeve, Stan Verbeek, Thomas Boone, Dave van Wijk, Martijn Jakobs, Keano Ozcan and Rick van Kalmhout.

Writing a thesis can be taxing for a family. Many thanks to Karin, Anne, Quinten and Fieke for supporting me during isolated vacations and moody hours.

This research was funded in part by two Intel research grants.

CONTENTS

ACRONYMS

AABB Axis-Aligned Bounding Box

AO Ambient Occlusion

AOS Array of Structures

BDPT BiDirectional Path Tracing

BRDF BiDirectional Reflection Distribution Function

BSDF BiDirectional Scattering Distribution Function

BSP Binary Space Partitioning

BTB Branch Target Buffer

BVH Bounding Volume Hierarchy

CDF Cumulative Distribution Function

CPU Central Processing Unit

CSG Combinatorial (or Constructive) Solid Geometry

CUDA Compute Unified Device Architecture

ERPT Energy Redistribution Path Tracing

FPS Frames per Second

GI Global Illumination

GPU Graphics Processing Unit

HDR High Dynamic Range

IS Importance Sampling

IGI Instant Global Illumination

MLT Metropolis Light Transport

MIS Multiple Importance Sampling

MC Monte Carlo

MBVH Multi-branching Bounding Volume Hierarchy

PT Path Tracing

PDF Probability Distribution Function

QMC Quasi-Monte Carlo

RS Ray Streaming

RPU Ray Processing Unit

RMSE Root Mean Squared Error

SAH Surface Area Heuristic

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SM Streaming Multiprocessor

SOA Structure of Arrays

SPP Samples per Pixel

TTI Time To Image

VPL Virtual Point Light

1

INTRODUCTION

Video games have shown a tremendous development over the years, fueled by the increasing performance of graphics hardware. Game developers strive for realistic graphics. Until about a decade ago, this mapped reasonably well to the rasterization algorithm[1], as the focus was on increasing polygon counts and the improvement of the quality of local effects, while retaining real-time performance. Recently, attention has shifted to the simulation of global effects, which do not map well to the rasterization algorithm. Approximating algorithms are available, but are often case-specific, mutually exclusive and labor-intensive. At the same time, an alternative algorithm has become feasible on standard PCs, in the form of ray tracing, which is slower for game graphics but not bound to approximations for global effects. On the contrary; global effects come naturally with this algorithm. However, feasibility of this algorithm for real-time applications completely depends on available processing power.

Graphics for games require a minimum frame rate. Low frame rates mean sluggish responses to player input, which in turn leads to a less immersive experience. The desired frame rate for a game depends on the genre. For non-interactive media, 24 frames per second is generally enough to perceive movement as fluent. However, for interactive media, 24 frames per second means a worst-case response time of $1/12^{th}$ of a second[2]. For this reason, games that require fast reflexes will typically run at very high frame rates, often higher than what the monitor can display[3]. For a game, an acceptable frame rate takes precedence over image quality and accuracy. This explains the preference for the rasterization approach, and also why frame rate has been more or less stable over the past decades, while image quality gradually increased. This also explains why game developers tend to prefer fast approximations over more accurate algorithms.

The desire for realistic, real-time graphics fueled the development of dedicated graphics hardware. This hardware enabled the use of higher resolutions and polygon counts, in particular for the rasterization approach. The new hardware is less efficient for ray tracing approaches. Resolution and polygon count are not the only factors that determine realism however. Global effects such as shadows and reflections also play an important role, but these are not trivially implemented using software rasterization or rasterization hardware.

1 In this thesis, the term *rasterization* is used for both z-buffer scan conversion and the painter's algorithm.

2 User input may occur just after frame rendering started. In this case, the input will be taken into account for the next frame, which is presented 2 frames after the input event. Average response time is 1.5 frame; minimal response time is 1 frame.

3 Some professional players prefer frame rates in excess of 200 for Quake 3 Arena.

When striving for further advances in image quality, we thus face the following problem: within the constraints of computer games, graphics algorithms are reaching the limits of the underlying rasterization algorithm. An alternative algorithm is available in the form of ray tracing, but this algorithm does not map well to specialized graphics hardware, and requires too much processing power to display images at desired frame rates. In this thesis, we want to explore how we can improve the performance of ray tracing on commonly available gaming platforms such as PCs and consoles, to bring ray tracing within the time constraints dictated by gaming.

1.1 GRAPHICS IN GAMES

The level of realism in computer games has increased significantly since the first use of a computer for this purpose [92]. This progress is driven by the desire of players to submerge themselves in a virtual world, for varying reasons. According to Crawford [55], humans use games to compete and to train their skills, alone or in groups, and to find fulfillment for their fantasy. Games also serve as a means to escape social restrictions of the real world.

This competition, fulfillment, and training is not only found in computer games: e.g., a game of chess can fully absorb a player, challenging a worthy opponent, based on equal rules for either player, disregarding stature. Compared to classic games, computer games do however add several elements. A computer game is an interactive simulation in which one or more players partake; it provides artificial opponents, and governs a closed system with objective rules. Increasing realism improves the game: training is more useful when the simulation approaches reality, and bending social rules becomes more satisfying when the virtual world resembles the real world.

Realism in computer games went through several stages before it reached today's level[4]. The first game that used graphics of any kind ran on the 35x16 pixel monochrome display of an EDSAC vacuum-tube computer (figure 1a), and played tic-tac-toe [70]. Color graphics first appeared in the Namco game *Galaxian* [166] (figure 1b). Three-dimensional polygonal graphics first appeared in the Atari arcade game *I, Robot* [236], although 3D games using scaled sprites were available before that [167, 211]. On consumer hardware, basic 3D graphics were available as early as in 1981, in the game *3D Monster Maze*, on the Sinclair ZX-81 [78] (figure 1c). 3D wire-frame graphics appeared shortly after that, in *Elite* [36] on the Acorn Electron home computer. Solid polygons were introduced in 1988, in *Starglider* [154]. Texture mapping first appeared in idSofware's *Catacomb 3D* [42].

Hardware accelerated 3D graphics for gaming consoles and PC's were first introduced by the 3DO company in 1993 [1]and NVidia in 1995 [172], but were

4 A highly detailed time line, not specific to games, is available here: http://www.webbox.org/cgi/_timeline50s.html

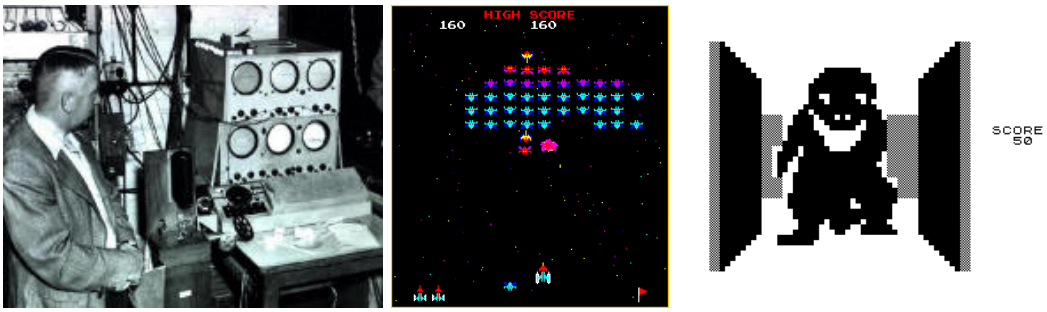


Figure 1: The EDSAC, Galaxian, and 3D Monster Maze.

popularized by 3dfx in 1996 [3][5]. These graphics coprocessors use z-buffer scan conversion for visibility determination. As a result of the availability and subsequent rapid advance of this dedicated hardware, the z-buffer algorithm quickly became the de facto standard for high performance rendering.

Up to this point, real-time graphics were limited to flat shaded or Gouraud-shaded polygons with textures, and no global effects were used. This changed with a number of newer games: In 1996, *Duke Nukem 3D* [2] used reflections and shadows on planar surfaces; in 1997, *Quake II* [43] used precomputed radiosity stored in textures (*lightmaps*) on static geometry; in 2004 both *Half Life 2* [52] and *Far Cry* [56] used refraction for realistic water. Implementing global effects in a z-buffer scan conversion based engine requires the use of approximating algorithms[6]. This leads to high code complexity in the most recent engines: e.g., CryEngine consists of 1 million lines of code, the Unreal 3 engine 2 million [274, 153].

1.2 RAY TRACING VERSUS RASTERIZATION

Current game graphics are based on the rasterization algorithm[7]. Depth- or z-buffer scan conversion (*rasterization*) is the process of projecting a stream of triangles to a 2D raster (color and depth buffer), using associated per-triangle data (figure 2a). During this process, fragments whose depth are greater than or equal to a previously stored depth are discarded. Usually, a limited set of global data is available, such as active light sources. Early GPUs implemented scan conversion in hardware, while the rest of the rendering pipeline remained in software [72, 158, 172, 3]. Modern GPUs implement the full rendering pipeline in hardware [173], with individual parts programmable on the GPU itself, making the GPU

5 The actual start is hazy: Atari used a TMS34010 GSP for the arcade game Hard Drivin' in 1989 [113]. Commodore used a graphics coprocessor in the Commodore Amiga in 1985 [49]. This chip only accelerates span rendering, and does not render polygons.

6 Of all secondary effects, only hard shadows can be considered to be more or less solved, although even the best solutions suffer from rendering artifacts. Up til today, reflections and refractions are approximated in either a highly application-specific way, or with considerable artifacts. Indirect lighting is severely under-sampled, or screen-space based, if present at all.

7 Rasterization: z-buffer scan conversion. Early versions used the painter's algorithm instead.

a more general purpose processor. The rendering pipeline consists of transform and lighting, polygon setup, and z-buffer scan conversion [8]. In a programmable pipeline, vertex shaders are used during the transform and lighting stage, geometry shaders are used during the polygon setup stage, and pixel shaders are used during z-buffer scan conversion. While this makes individual stages programmable, the stages themselves remain in a fixed order. As a consequence, a modern GPU is still a special purpose processor designed for rasterization, rather than general computing.

Although z-buffer scan conversion allows for efficient rendering of 3D scenery, it also has limitations, mainly because of its inherent streaming nature. Shadows, reflections, refractions and indirect lighting all require global knowledge of the scene. Since a rasterizer renders the scene one triangle at a time, this information is not available.

Usually workarounds are available however. For shadows of point light sources, an early solution was to create simplified, flattened shadow geometry, and to draw this geometry under a racing car on the track geometry. Later, shadow volumes were drawn to a stencil buffer in a separate pass. This buffer was then used during triangle stream processing to determine which pixels reside in the shadow. In modern engines, shadows are rendered using shadow maps [266]. These are depth maps, constructed in a separate pass per light source, by rendering the scene from the viewpoint of each light source. During triangle stream processing, pixels are transformed into the space of the light, and tested against the depth map. Shadow map approaches typically suffer from aliasing, but several algorithms are available to alleviate this. For a survey of shadowing techniques, see the survey of Woo et al. [268] and, more recently, Hasenfratz et al. [102].

Approximations for reflection and refraction also exist. Reflections have been used to make cars in racing games more realistic, and for rendering water [122, 159]. Refraction has been used to improve the appearance of water and gems [161]. However, unlike hard shadows, reflections and refractions are quite far from the correct solution. The reflected environment is often infinitely distant and static [31]. Reflections of dynamic environments are achieved by updating the environment in a separate pass. In this case, the reflection is still only correct for distant objects, and self-reflection remains impossible. Since the human eye is not nearly as sensitive to correct reflections as it is to correct shadows [198], convincing results are often achieved, despite these limitations. Artifacts are often most apparent when objects intersect a reflective surface, such as water, in which case obvious discontinuities appear.

Ray tracing, in the context of computer graphics, is the construction of a synthesized image by constructing light transport paths between the camera, through the screen pixels, to the light sources in the scene (figure 2b). The vertices of these paths lie on the surfaces of the scene. Paths or path segments can be traced either forward (starting at light sources) or backward (starting at the camera). Ray tracing can be done deterministically, in which case rendering is limited to perfect specular

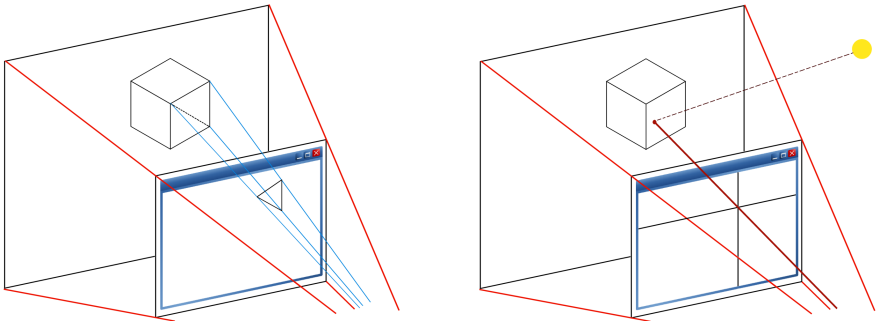


Figure 2: Rasterization and ray tracing. a.) A rendering pipeline based on rasterization iterates over the polygons of the scene, projecting them onto the screen plane, and modifying each covered pixel. b.) A renderer based on ray tracing loops over the pixels of the screen, and finds the nearest object for each of them. A light transport path is then constructed by forming a path to a light source.

surfaces and diffuse surfaces that are lit directly by point lights [265] (figure 3a). This allows rendering of accurate specular reflections, refractions and hard shadows. This deterministic form of ray tracing is referred to as *Whitted-style ray tracing* or *recursive ray tracing*. Cook et al. proposed to extend this with stochastic sampling of certain light paths, in which case soft shadows and diffuse reflections are calculated as the expected value of a random sampling process [51] (figure 3b). This form of ray tracing is referred to as *stochastic ray tracing* or *distribution ray tracing*. Kajiya generalizes the concept of stochastic sampling, by randomly sampling all possible light transport paths [125] (figure 3c). His *path tracing* algorithm is able to render most natural phenomena, including diffuse reflections, diffraction, indirect light and caustics, as well as lens- and film effects such as depth of field and motion blur.

Like rasterization-based rendering algorithms, ray tracing has disadvantages. These are mostly performance related: considering that game developers strive for high frame rates, ray tracing has never been an option. Many games do use ray tracing indirectly however. Cut scenes are often rendered using offline ray tracing software. Some games use ray tracing to bake accurate lighting in light maps. Ray tracing also appears in several demos, where it is used to show off optimization skills and mathematical knowledge. Still, ray tracing never made it beyond the point of being an interesting technical challenge.

Where rasterization-based rendering algorithms struggle to approximate complex light transport, algorithms based on ray tracing generally struggle to achieve sufficient performance. This contrast is further emphasized when global illumination is desired. Approximating glossy and diffuse reflections in rasterization-based renderers requires complex algorithms, which often yield coarse results. When using ray tracing, the correct solution is easily achieved using existing algorithms, but calculating this solution in real-time is currently not possible on consumer hardware.

Figure 3: Three well-known ray traced scenes. a.) Whitted style ray tracing with recursive reflection and refraction. This image is © 1980 ACM, Inc. Included here by permission. b.) Cook's distribution ray tracing with stochastically sampled motion blur and soft shadows. This image is © 1984 Thomas Porter, Pixar. c.) Kajiya's path tracer, with indirect light and caustics. Included here by permission.

Once the performance required to simulate light transport using ray tracing is available, it seems likely that ray tracing will be the prevalent choice for rendering. For the field of games, this is an attractive prospect; one that promises elegant rendering engines, a more efficient content pipeline, and realistic visuals.

1.3 PREVIOUS WORK

Several researchers sought to use the ray tracing algorithm for interactive and real-time rendering.

Initially, this required the use of supercomputers. Muuss deploys a 28 GFLOPS SGI Power Challenge Array to ray trace combinatorial solid geometry (CSG) models of low complexity at 5 frames per second and a resolution of 720x486 pixels [164]. Parker et al. used a 24 GFLOPS SGI Origin 2000 system and achieved up to 20 frames per second at 600x400 pixels [184][8].

On consumer hardware, interactive frame rates were first achieved by Walter et al. using their RenderCache system [258, 259], which uses reprojection (as earlier proposed by Adelson and Hodges [5] and Badt [123]) and progressive refinement [25] to enable interactivity. For their OpenRT ray tracer, Wald et al. use networked consumer PCs to achieve interactive frame rates on complex scenes [248, 250]. Real-time ray tracing on a single consumer PC was first achieved by Reshetov et al. [203]. Like OpenRT, their system is CPU-based. Other interactive and real-time CPU-based ray tracers are the Manta interactive ray tracer [26, 225, 118], the Arauna real-time ray tracer [27], the RTFact system [221], Intel's research group's ray tracer Garfield [204] and Embree [76] and Razor [67].

Concurrently, several GPU-based ray tracers were developed. Building on early work by Purcell et al. [197], Carr et al. [45] and Foley et al. [81], Horn et al., Günther et al. and Zhou et al. propose interactive GPU-based ray tracers [108, 97, 276]. A generic ray tracing system for GPUs, OptiX, was proposed by Parker et al. [185].

8 By contrast, in 1999 a high-end Pentium 3 consumer system achieved 84 MFLOPS.

The potential of ray tracing for games is recognized by several authors (e.g., [207, 244, 33, 196]. Others, such as Oudshoorn and Friedrich et al. studied this more in-depth [177, 209, 82]. The OpenRT ray tracer was applied to two student games [119], as well as walkthroughs of Quake 3, Quake 4, Quake Wars and Wolfenstein scenery [192, 194, 195]. Keller and Wächter replaced the rasterization code of Quake 2 with ray tracing code [135].

Inspired by dedicated rasterization hardware, several authors propose dedicated hardware designs for Whitted-style ray tracing. Schmittler et al. propose the SaarCor hardware architecture for ray tracing [207]. An improved design is prototyped using an FPGA chip [208, 269, 270]. The authors use this hardware to render a number of game scenes, and report a three-fold speed-up, compared to OpenRT.

It was only recently that interactive path tracing on consumer hardware was investigated. Novák et al. proposed a GPU path tracer that renders interactive previews [171]. Van Antwerpen proposed a generic architecture for GPU-based path tracing algorithms, and used this to implement several interactive physically-based renders [238].

1.4 PROBLEM DEFINITION

The desire to use global illumination in games, and the complexity of algorithms that aim to achieve this using rasterization-based rendering, leads to the desire to replace rasterization by ray tracing as the fundamental rendering algorithm in games. The fundamental question discussed in this thesis is how this can be achieved, within the strict constraints of real-time rendering, on consumer hardware.

To answer this question, we validate and combine existing work into several complete, well-optimized renderers, which we apply to practical game applications.

In the first part of this thesis we discuss efficient Whitted-style ray tracing, and its suitability for rendering for games. We further discuss how the basic algorithm can be augmented with diffuse indirect light.

In the second part of this thesis we focus on physically based rendering using path tracing, where computational demands are even higher. We approach this problem first on the CPU, where a data-parallel technique is used to improve performance. We then discuss efficient GPU implementations, and combine these in a single rendering framework.

We validate the developed systems by applying them to several real-time games.

1.5 THESIS OVERVIEW

This thesis is organized as follows:
Chapter 2 provides a theoretical foundation for the subsequent chapters.
Chapter 3 describes the implementation of the Arauna ray tracer. Arauna is currently the fastest CPU-based Whitted-style ray tracer, and has been used for seven

student projects. There are consequences of using a ray tracer as the primary rendering algorithm, for both the game programmer and the game graphics artist. These are outlined in this chapter as well.
Chapter 4 describes a mesh-less algorithm for sparsely sampling expensive shading, such as soft shadows, large sets of lights, ambient occlusion and global illumination. The algorithm is used in Arauna to enhance ray tracing with indirect diffuse reflections, which is approximated spatially using a sparse sampling approach.
In chapter 5 and 6 we describe efficient path tracing on the CPU and the GPU. Chapter 7 describes the Brigade path tracer, which uses multiple GPUs to achieve real-time frame rates for complex scenes, albeit with a limited number of samples per pixel. Despite high variance in the rendered images, the Brigade path tracer enables real-time path tracing in games on current generation consumer hardware for the first time.
Chapter 8 finally summarizes our findings, draws conclusions and summarizes directions for future research.

2

PRELIMINARIES

In this chapter, we lay the foundation for the remainder of this thesis. In section 2.1, we introduce the rendering equation, and rendering algorithms that approximate its solution, with trade-offs typically between performance and accuracy. In section 2.2, we discuss ray / scene intersection, as the fundamental operation of the ray tracing algorithm. Section 2.3 discusses the combination of the two for optimal efficiency in rendering algorithms based on ray tracing. Section 2.4 provides a definition of *real-time* in the context of graphics for games.

2.1 A BRIEF SURVEY OF RENDERING ALGORITHMS

Rendering is the process of generating an image from a virtual model or scene, by means of a computer program. The product of this process is a digital image or raster graphics image file. Rendering can focus on two distinct qualities:

RENDERING QUALITY The first optimizes the fidelity of the final rendered image, while the time needed to render images is of less importance. This approach is typically associated with the ray tracing algorithm and offline rendering.

PERFORMANCE The second makes a fixed or minimum frame rate a constraint, and optimizes the level of realism that can be obtained at this frame rate. This approach is generally associated with rendering algorithms based on the z-buffer scan conversion algorithm (*rasterization*), and is widely used in games.

As compute power increases, rendering techniques that were traditionally reserved for off-line rendering find their way into interactive rendering and real-time rendering. Rasterization has been augmented with algorithms for shadows, reflections and global illumination, and Whitted-style ray tracing has become interactive on mainstream hardware.

Rendering based on rasterization is typically approximative. Improving image fidelity is achieved by combining many algorithms for the various desired phenomena. The cost of image quality is more accurately expressed in terms of code complexity, than required processing power.

Rendering based on ray tracing in principle allows for more straightforward implementation, and higher levels of realism. Renderers based on ray tracing typically accurately implement a subset of all possible light transport paths. Adding additional types of light transport typically requires extra processing power more than algorithmic complexity.

In the chapters three through seven, we will discuss recursive ray tracing, sparsely sampled global illumination and path tracing in the context of real-time graphics for games. This chapter provides the theoretical foundation for this. In section 2.1.1, we first provide a brief review of light transport theory, followed by a description of rendering techniques as approximations of the rendering equation. Physically-based rendering is discussed in section 2.1.4. Biased rendering methods are briefly discussed in section 2.1.9.

2.1.1 *The Rendering Equation*

Physically-based rendering algorithms aim to produce realistic images of virtual worlds by simulating real-world light transport. Light transport is commonly approximated using the rendering equation, introduced by Kajiya in 1986 [125]. We start with the following formulation, which integrates over all surfaces in the scene and includes an explicit visibility term:

$$\begin{aligned} L(p \to r) =& L_e(p \to r) + \int_M L(q \to p)\, f_s(q \to p \to r)\, G(q \leftrightarrow p)\, V(q \leftrightarrow p)\, dA_M(q) \\ G(p \leftrightarrow r) =& \frac{|\cos(\Theta_o)\cos(\Theta_i')|}{\|p - r\|^2} \end{aligned} \tag{2.1}$$

This equation defines the radiance transported from point p to point r recursively as the light emitted by p towards r, plus the incoming light reflected by p, taking into account the visibility of each surface q in the scene. $G(q \leftrightarrow p)$ is the geometric term to convert from unit projected solid angle to unit surface area. In this term, Θ_o and Θ_i' are the angles between the local surface normals and respectively the incoming and outgoing light flow. $V(q \leftrightarrow p)$ is the visibility term, which is 1 if the two surface points are visible from one another and 0 otherwise. The process is illustrated in figure 4.

The equation makes a number of simplifying assumptions: the speed of light is assumed to be infinite, and between surfaces in the scene, light travels in a vacuum, and in straight lines. Furthermore, reflection is instant. The wavelength λ is constant, and p is an infinitely small point. And finally, the wave properties of light are ignored. The consequence is that a number of physical phenomena cannot be described using this equation. These include diffraction, fluorescence, phosphorescence, polarization, and relativistic effects. Various authors suggest extensions to the rendering equation to increase the number of supported phenomena. Smith et al. factor in the speed of light [222], describing irradiant flux as power rather than energy, similar to the radiosity equation proposed by Goral in 1984 [94]. A similar extension is proposed by Siltanen et al., to make the rendering equation suitable for acoustic rendering [217]. They later extended their acoustic rendering equation to support diffraction [216]. Wolff and Kurlander describe a system that supports

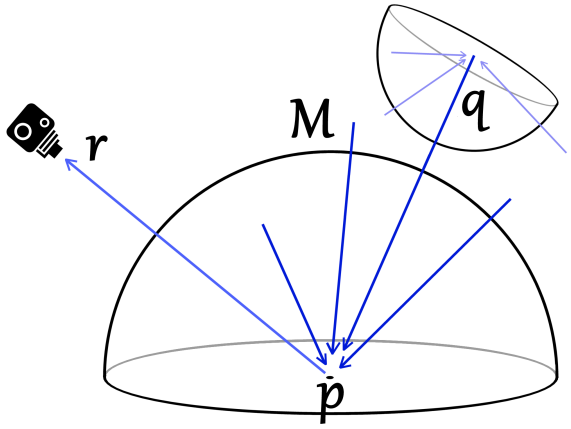


Figure 4: The rendering equation. Light energy emitted by light sources arrives at the camera via one or more scene surfaces.

polarization [267]. Glassner proposes an extension to support fluorescence and phosphorescence [90].

Note that solving the rendering equation by itself does not result in realistic images. Only when the provided data is accurate and sufficiently detailed, the produced images will be accurate.

Despite its limitations, the rendering equation is physically based, since the phenomena that it does support are accurately described, and energy in the system is preserved[1].

2.1.2 *Rasterization-based Rendering*

Z-buffer scan conversion or rasterization [80] is a streaming process, in which the polygons of a scene are processed one by one. Polygons enter the rasterization pipeline in the form of a list of vertices. They are transformed and then used for primitive assembly. Constructed primitives are clipped against the view frustum, and projected onto the view port. The projected primitives are broken up in fragments. Fragments are stored to the output buffer.

This approach has a number of advantages. By operating on a stream, data locality is implicit: processing a single triangle only requires data for that triangle. For the same reason, parallel processing of data is trivial, since elements in the stream are independent. This makes rasterization suitable for dedicated hardware implementations, in which the full rendering pipeline or parts thereof are implemented.

Rasterization by itself is a visibility algorithm: the end result is, for each pixel of the output buffer, the nearest triangle, if any. This result can be used to produce a shaded image. Rasterization-based rendering algorithms are typically interleaved with the visibility determination. In that case, shading happens *on the fly*, as triangles and fragments are processed.

1 Unlike e.g. in the Phong model [189], which is commonly used in real-time graphics.

Single-pass rasterization-based rendering implements the following approximation of the rendering equation:

$$L(p \to r) = L_e(p \to r) + \sum_{i=1}^{N_L} L(q_i \to p)\, f_r(q_i \to p \to r)\, G(q_i \leftrightarrow p) \tag{2.2}$$

In this equation, the integral over the hemisphere is replaced by the sum of the contributions of the individual point light sources, and the visibility factor disappeared. Also, the equation is no longer recursive. Inaccessibility of global data is a fundamental restriction of rasterization. The only part of the above equation that requires access to global data is the iteration over the lights in the scene.

The differences between equation 2.1 and equation 2.2 have several consequences for rendering. Lighting is limited to point lights, but more importantly, all effects that require global data are unsupported. This includes several effects that are important for the correct interpretation of rendered images, such as shadows and reflections. With these limitations however, the rasterization is able to operate using very limited resources.

Rasterization can be augmented with a large number of algorithms that approximate global effects. Most notably, shadows from point light sources (and to some extent, soft shadows) can be rendered convincingly. While this generally requires extra render passes, it effectively implements the visibility factor for the rasterization algorithm. This blurs the line between rasterization and ray tracing, both in terms of supported features and required resources.

2.1.3 *Ray Tracing*

Ray tracing is the process of determining visibility between two points in the scene, or the nearest intersection along a ray[2]. The latter is also referred to as *ray casting*. Ray tracing was first applied to computer graphics in 1968 by Appel [11], who shot rays from the eye (camera) to the pixels of the screen, to determine what geometry should be visible at each pixel. As shown by Whitted in 1980, basic ray casting can be extended to determine shadows, by tracing rays from the first intersection point to light sources. Likewise, reflections are determined by creating a new ray along the reflection vector [265].

Like rasterization, ray tracing is a process that is easily executed in parallel, since rays do not interact. Unlike rasterization however, ray tracing potentially requires access to all scene geometry.

Simple ray casting with shadow rays to point light sources implements the following approximation of the rendering equation:

2 A ray is defined as an infinite line segment, originating at a point in the scene.

$$L(p \to r) = L_e(p \to r) + \sum_{i=1}^{N_L} L(q_i \to p)\, f_r(q_i \to p \to r)\, G(q_i \leftrightarrow p)\, V(q_i \leftrightarrow p) \tag{2.3}$$

Apart from the visibility factor, this is the same equation as 2.2.

Ray casting and rasterization become identical when we limit the ray caster to primary rays only, and add the constraint that the primary ray targets are laid out on a regular grid. Dachsbacher et al. [57] have shown that even this requirement can be relaxed, by extending the commonly used linear edge function approach [191] to 3D, making ray tracing and rasterization nearly identical for all primary rays. This also works the other way round: Hunt and Mark have shown that ray tracing performance can be improved by building specialized acceleration structures *per light*, in the perspective space of each light, effectively turning ray tracing into multi-pass rasterization [110].

For recursive (Whitted-style) ray tracing, equation 2.3 is further extended:

$$\begin{aligned} L(p \to r) = {} & L_e(p \to r) + \sum_{i=1}^{N_L} L(q_i \to p)\, f_r(q_i \to p \to r)\, G(q_i \leftrightarrow p)\, V(q_i \leftrightarrow p) \\ & + L(s \to r)\, f_r(s \to q \to r)\, G(s \leftrightarrow r)\, V(s \leftrightarrow r) \end{aligned} \tag{2.4}$$

Whitted-style ray tracing adds indirect lighting to the direct lighting, but this is limited to pure specular transmissive and reflective surfaces. The BRDF in the recursive part of the above formulation is thus a Dirac function.

This limitation is alleviated in distribution ray tracing[3], introduced by Cook in 1984 [51]. This algorithm approximates glossy reflections using an integral over the surfaces in the scene, and soft shadows using an integral over the surface of each light source:

$$\begin{aligned} L(p \to r) = {} & L_e(p \to r) + \sum_{i=1}^{N_L} \int_M L(q \to p)\, f_r(q \to p \to r)\, G(q \leftrightarrow p)\, V(q \leftrightarrow p)\, dA_M(q) \\ & + \int_N L(s \to r)\, f_r(s \to q \to r)\, G(s \leftrightarrow r)\, V(s \leftrightarrow r)\, dA_N(s) \end{aligned} \tag{2.5}$$

By unifying emissive surfaces and light sources, this reduces to equation 2.1.

2.1.4 *Physically-based Rendering*

In the previous section, we described rasterization-based rendering and rendering algorithms based on ray tracing as partial solutions or approximations of the rendering equation. In this section, we describe rendering algorithms that provide

3 Also known as stochastic ray tracing

a full solution to the rendering equation. We refer to these algorithms as physically based, as they accurately simulate the supported phenomena, and preserve energy equilibrium in the system, when fed with correct data.

Solving the rendering equation can either be done using finite elements methods, such as radiosity [101, 48, 223, 215, 19, 224], or stochastically, using Monte Carlo ray tracing [125, 144, 143, 241, 121], where the recursive rendering equation is evaluated using a Markov chain simulation [243]. This approach is often preferred over finite element methods, as it allows for more complex scenes, procedural geometry, and arbitrary BRDFs [121, 15]. Monte Carlo ray tracing has an algorithmic complexity of $O(\log N)$ (where N is the number of scene elements), whereas the fastest finite elements methods require $O(N \log N)$ [48].

The physical equivalent of the set of Markov chains is a family of light paths that transport light from a light source to the observer, via zero or more diffuse, glossy, or specular surfaces. The class of rendering algorithms that use this approach is called path tracing.

2.1.5 *Monte-Carlo Integration*

The Monte Carlo simulation used in path tracing approximates the integral in the rendering equation by replacing it by the expected value of a random variable:

$$E(x) = \int_{\Omega} L(q \to p)\, f_r(q \to p \to r)\, G(q \leftrightarrow p)\, V(q \leftrightarrow p)\, dA_M(q) \tag{2.6}$$

$$\approx \frac{1}{N} \sum_{i=1}^{N} L(q_i \to p)\, f_r(q_i \to p \to r)\, G(q \leftrightarrow p)\, V(q \leftrightarrow p)\, dA_M(q_i) \tag{2.7}$$

For a sufficiently large N, this yields the correct answer, according the *Law of Large Numbers*:

$$\text{Prob}\left[E(x) = \lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} x_i \right] = 1 \tag{2.8}$$

The variance of the Monte Carlo estimator is $var(x) \equiv E([x - E(x)]^2) = E(x^2) - [E(x)]^2$. Since the variance of the estimate is proportional to $\frac{1}{N}$, the standard deviation is proportional to $\frac{1}{\sqrt{N}}$. Therefore, assuming an even distribution of the random samples is used, we need to quadruple N to halve the error in the estimate.

There are several ways to reduce the variance of the estimator. When using importance sampling, samples are distributed according to a *probability distribution function* (PDF):

$$E(x) \approx \frac{1}{N} \sum_{i=1}^{N} \frac{L(q_i \to p)\, f_r(q_i \to p \to r)\, G(q \leftrightarrow p)\, V(q \leftrightarrow p)\, dA_M(q_i)}{P(q_i)} \tag{2.9}$$

The PDF can be an arbitrary function, as long as $P(q) \geqslant 0$, $\int P(q) = 1$ and $P(q) > 0$ where the integrated function is not zero. For the purpose of variance reduction, the PDF should match the integrated function, so that more samples are taken that contribute significantly to the estimate.

Variance can also be reduced by using evenly distributed random samples. One way to achieve this is using stratification, where the domain of the integrand is divided in multiple strata of equal size [170].

In the context of rendering, a single sample is a path, whose vertices lie on the camera, zero or more scene surfaces, and a light source. The contribution of the light source is scaled at each vertex on the path by $f_r(q_i \to p \to r)\, dA_M(q_i)$.

2.1.6 *Russian Roulette*

The paths that connect the lights to the camera consist of one or more segments. The total number of surface interactions for one path is potentially infinite. Longer paths tend to deliver less energy, since each bounce typically absorbs some of the transmitted energy; however, an artificial maximum on the number of path segments introduces bias in the estimate.

Russian roulette [14, 73] is a technique where a fraction of the paths is terminated with a probability ρ at each encountered surface, while the energy of the remaining paths is scaled by $\frac{1}{\rho}$. Using Russian roulette, paths have a non-zero probability of reaching a certain depth. At the same time, shorter paths are favored over longer paths, and remaining paths maintain their original intensity.

Termination probability ρ is typically locally determined and proportional to one minus the hemispherical reflectance of the material of the surface (increasing termination probability for darker surfaces), but may also be chosen globally, as proposed by Keller [132]. A global termination probability may however cause infinite variance [231].

2.1.7 *Path Tracing and Light Tracing*

Path tracing performs the Markov chain simulation by creating paths backwards from the camera to a light source, via zero or more diffuse, specular, or glossy surfaces. This process is illustrated in figure 5. In this figure, E denotes the eye, L a light source, D a diffuse or glossy surface, and S a specular or dielectric surface. Pseudo code for this process is shown in algorithm 2.1.

The adjoint algorithm for path tracing is *light tracing*. Here, paths start at the light, after which a random walk is executed until the eye is found.

Path tracing may require a large number of bounces until a light source is found, especially when the light sources are small. To some extend, next event estimation (see next subsection) can improve efficiency in this situation. A large number of possible paths may however exist for which next event estimation does not help, e.g. when lights are inside or behind transmissive objects, or visible via specular

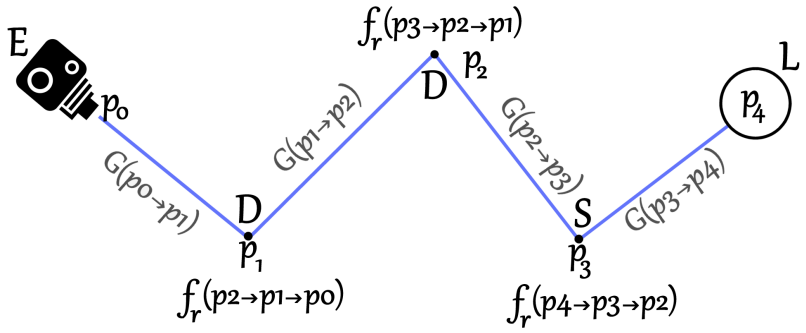


Figure 5: A Markov chain representing a single path connecting a light source and the camera, via three surfaces. At each vertex, the transported energy is scaled by the BRDF. Along each path segment, energy is scaled by the geometry factor.

Algorithm 2.1 The basic recursive path tracing algorithm. The path is extended in direction R until a light source is encountered. The contribution of the light source is then transferred along the path, and scaled by the BRDF and geometry factor at each vertex I.

```
function Trace(O, D)
    // find material, distance and normal along ray
    material, I, N ← find nearest(O, D)
    if (is light(m))
        // path reached light source
        return material.Emissive
    else
        // path vertex: diffuse or specular
        return Trace(I, R) * BRDF(I, R, D) * cos(N, R)
```

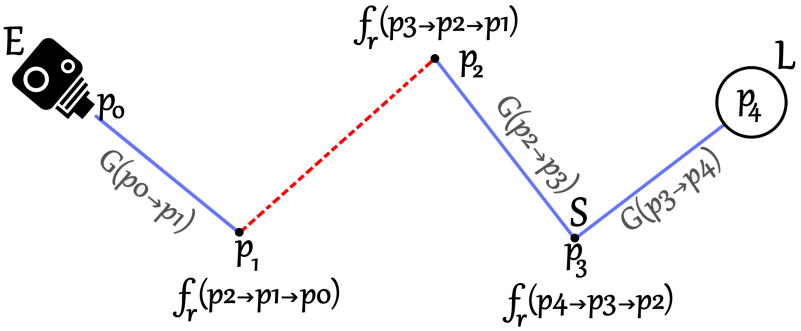


Figure 6: Bidirectional path tracing: a path is generated backward from the camera, and forward from a light source, and connected to form a complete light transport path.

objects. *Bidirectional path tracing* [241, 143] combines path tracing and light tracing. A path is constructed starting from the eye, as well as from a light source. The vertices of the sub-paths are then connected to form complete light transport paths.

The process is illustrated in figure 6.

2.1.8 *Efficiency Considerations*

For many scenes, path tracing and light tracing are not very efficient. In scenes with small light sources, it may take a very large number of path segments to reach the light source, at which point the transported energy is low, as it is scaled by the BRDF and the geometry factor at each surface interaction. Paths that happen to reach a light source in only a few steps will contribute much more to the final estimate. It is thus worthwhile to focus effort on these paths.

IMPORTANCE SAMPLING Importance sampling is a technique that aims to reduce variance in a Monte Carlo estimator by sampling the function of interest according to a probability distribution function (pdf) that approximates the sampled function. In the path tracing algorithm, we use importance sampling to improve the estimate of both indirect and direct illumination. For indirect illumination, the pdf is commonly chosen proportional to the surface BRDF. For the estimation of direct lighting, we chose lights according to potential contribution.

RESAMPLED IMPORTANCE SAMPLING In their 2005 paper, Talbot et al. propose a technique they refer to as Resampled Importance Sampling (RIS) [234]. Their technique uses importance sampling to make a first selection of samples. For this selection, a more accurate pdf is constructed. This pdf is then used to select the final sample from the initial selection. Note that the weight of a sample selected using importance sampling is scaled by the reciprocal of the pdf; therefore, we scale the final sample by the product of the reciprocals of the two pdfs used for the selection process. The time complexity of RIS approach is $O(M)$, where M is the size of set of the initially selected samples.

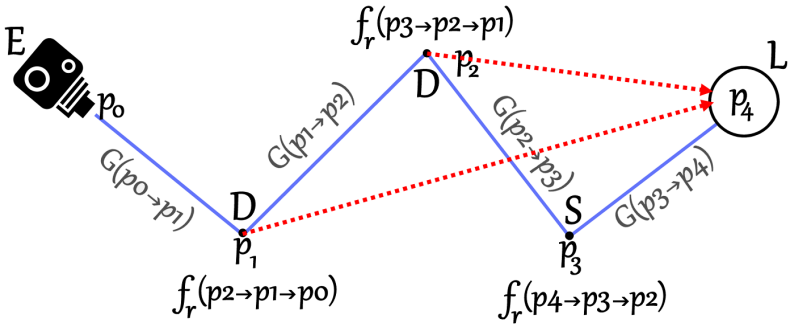


Figure 7: Next event estimation in path tracing: at each diffuse surface interaction, an explicit path to a light source is constructed. This allows reuse of path segments, and strongly decreases the average path length.

MULTIPLE IMPORTANCE SAMPLING Multiple importance sampling (MIS) was proposed as a variance reduction technique for computer graphics by Veach [241]. When using MIS, several sampling strategies are combined using a heuristic, with the aim to keep the strengths of each individual strategy. In a path tracer, MIS is commonly applied to estimate direct lighting. To estimate the direct light contribution, two practical strategies are available. The first is to sample direct light explicitly. In this scenario, a ray is created towards a random light source, either using a uniform random number, or according to some pdf. The second available strategy uses a pdf proportional to the surface BRDF. As shown by Veach in his Ph.D. thesis, certain common lighting conditions are handled considerably better by one of the strategies, but not by the other: light cast by a small light source and reflected by a glossy surface should be sampled using explicit light rays, while a large area light reflected by a nearby diffuse surface exhibits less variance when it is sampled according to the BRDF of the diffuse material. A practical implementation of MIS estimates direct light by creating two rays, one according to each strategy. For each ray, a weight is calculated using the power heuristic: $weight = pa^2/(pa^2 + pb^2)$, where pa is the probability that the chosen strategy would generate this ray, and pb the probability that this ray would have been generated by the alternative strategy.

NEXT EVENT ESTIMATION One way to exploit the higher contribution of short paths is *next event estimation* [73], where an explicit path is created for each non-specular vertex on the path to a light source in the scene[4] (see figure 7). Next event estimation separates indirect from direct illumination, and explicitly handles direct illumination for each surface interaction. This is compensated by omitting direct lighting in cases where a path 'accidentally' encounters an emissive surface.

4 Russian roulette and next event estimation can thus both be considered to be forms of importance sampling.

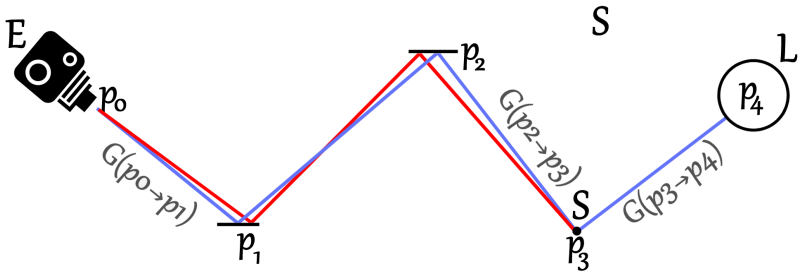


Figure 8: Metropolis light transport: a path that was constructed using a random walk is mutated to explore path space.

METROPOLIS LIGHT TRANSPORT This algorithm combines path tracing or bidirectional path tracing with the Metropolis-Hastings algorithm to make small modifications to the generated paths. This allows the algorithm to explore nearby paths, once a path from the eye to a light has been found. The process is illustrated in figure 8.

2.1.9 *Biased Rendering Methods*

Path tracing and derived algorithms are unbiased approximations to the rendering equation. Unbiasedness is not a strict requirement for a physically based rendering algorithm. For the context of rendering for games, a consistent algorithm may be sufficient, and in many cases, even consistency may not be a strict requirement. In this section we discuss biased rendering methods, which trade unbiasedness or even correctness for rendering performance, while remaining physically based.

An algorithm is consistent, if it is correct in the limit: it approaches the correct solution as computation time increases. It is however not necessarily possible to give a bound for the error at any given time [54], and averaging many renders using the approach does not necessarily converge to the correct solution. An estimator x_i for a quantity I is consistent for ϵ if:

$$\lim_{i \to \infty} P\left[|x_i - I| > \epsilon\right] = 0 \tag{2.10}$$

In other words, given enough time, the error of the estimate will always be less than ϵ. Based on equation 2.8, an estimator x_i is unbiased if:

$$E\left[x_i - I\right] = 0 \tag{2.11}$$

In other words: an algorithm is unbiased, if it is correct on average [53].

In this section, we will provide a brief description of physically-based rendering algorithms that are consistent, but not unbiased. Allowing some bias in the solution often allows for more efficient algorithms. Depending on the context, bias may

or may not be an issue. In the context of realistic graphics for games, some bias is acceptable, and often of less importance than (unbiased) noise. E.g., a post processing filter that removes fire flies in the output of a path tracer introduces bias, but improves image quality for almost all purposes.

PHOTON MAPPING Photon mapping is a two-pass algorithm that uses forward path tracing to create a photon map, and backward ray tracing to create the final image using the information in the photon map [121]. In the first pass, photons are created on the light sources, proportional to the intensity of the light source. The photons propagate flux into the scene, and deposit this in the photon map for each non-specular surface interaction. In the second pass, backward ray tracing is used to construct paths from the camera. At each non-specular surface interaction, the flux of photons within a small radius is added to the direct illumination calculated by the backward ray tracing.

INSTANT RADIOSITY Similar to photon mapping, the instant radiosity algorithm [132] traces light paths until a diffuse surface is encountered, at which point a virtual point light (VPL) is created. In a second pass, the scene is rendered using ray tracing or rasterization, using the set of VPLs to add indirect lighting to the direct lighting.

IRRADIANCE CACHING The irradiance cache algorithm sparsely samples global illumination and uses interpolation to reconstruct global illumination for points where no sample is available [264]. Samples are added on-the-fly if the error bound of the approximation exceeds a specified value. The Irradiance Cache algorithm is discussed in more detail in chapter 4.

2.2 EFFICIENT RAY / SCENE INTERSECTION

The basic underlying operation of all rendering algorithms based on ray tracing is the calculation of the intersection of a ray (or a collection of rays) and the scene. The efficiency of this operation has a great impact on the overall efficiency of the rendering algorithm, and has received extensive attention. In this section, we describe various *divide and conquer* approaches.

2.2.1 *Acceleration Structures for Efficient Ray Tracing*

The time spent in an application can be formally described using the following formula by Hsieh [109]:

$$\mathrm{Total\,time} = \sum_{i=0}^{\#\mathrm{tasks}} \mathrm{time\,of\,task}_i \tag{2.12}$$

where

$$\text{time of task}_i = \frac{\text{work of task}_i}{\text{rate of work of task}_i}$$

Improving the performance of an application can thus be achieved in two ways: we can reduce the algorithmic complexity, by reducing the number of times a specific task is executed, or we can reduce the time it takes to execute a particular task (also known as low-level optimization[5]). Formally expressing algorithmic complexity can be done using the *Big O* notation. Formally describing execution time of a single task is possible, but uncommon: actual timing depends on the hardware architecture that is used, and as a result, it is generally determined empirically. Exceptions are compact tasks that are executed at high frequencies, such as triangle intersection algorithms or traversal kernels, for which operand counts and code path execution probability can be used for platform-independent comparisons. Recent processor technology advances, such as branch prediction and instruction pipelining, reduce the validity of such comparisons however.

A naive ray tracer can be divided in the following major components:

- Ray / primitive intersection;
- Shading.

For N primitives, the cost of intersection is $O(N)$, while the cost of shading is independent of the number of primitives, and thus $O(1)$. Initial optimization therefore should focus on intersection cost, which dominates the total run-time of a ray tracer. For this, acceleration structures are used. Early ray tracers did not use these: although Whitted used bounding spheres for complex objects such as bi-cubic patches, these bounding spheres are not used hierarchically 265. Shortly after that however, Rubin and Whitted proposed a hand-crafted hierarchy of oriented bounding boxes to speed up ray / primitive intersection 205.

Acceleration structures can be divided in two classes: *spatial subdivisions* and *object hierarchies*.

A spatial subdivision subdivides the space in which primitives reside, often recursively. Primitives that overlap an area are stored in these areas. It is thus possible for an object to be stored in multiple areas. It is also possible for an area to be empty. Examples of this class of acceleration structures are:

OCTREES Figure 9a. First introduced for ray tracing in 1984 by Glassner [89]. An octree starts with a bounding cube of the scene, and recursively subdivides this cube into eight cubes, until a termination criterion is met[6]. Octrees are quick to build (with an algorithmic complexity of $O(N)$) and are useful for reducing the number of ray / primitive intersections. They do however not adapt well to varying levels of detail the scene (often referred to as the "teapot in a stadium" problem).

5 Some authors refer to this as the C in the Big O notation.

6 Typically: the number of primitives in each octree node reaches a certain threshold, or a maximum depth is reached

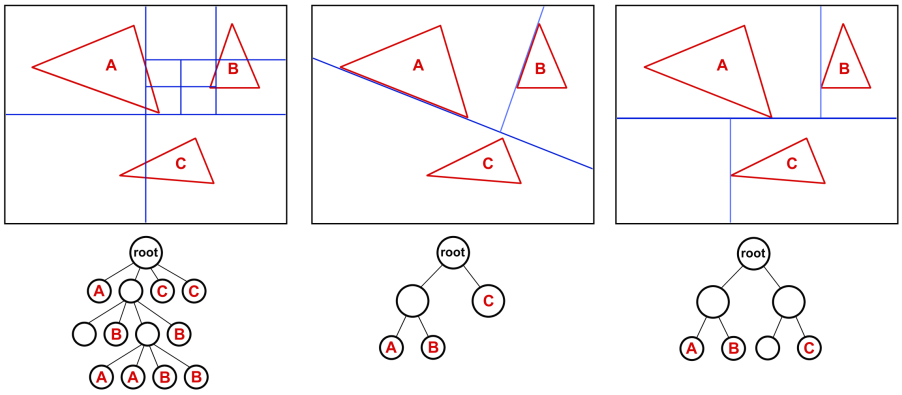


Figure 9: Spatial subdivisions: quadtree (2D equivalent of the octree), BSP, kD-tree.

GRIDS First proposed by Fujimoto and Iwaka in 1986 by Fujimoto et al. [83]. The simple 3D extension to the DDA line algorithm[7] was later improved upon by Amanatides and Woo [9]. Uniform grids can be built in $O(N)$, but like octrees, they do not adapt well to the scene, and construction parameters need to be manually tweaked per scene for optimal performance. Non-uniform and hierarchical grids alleviate this to some extent. Recently, uniform grids where considered for fast construction times in dynamic scenes [115].

BSPS Figure 9b. Binary Space Partitioning (BSP) splits space recursively using a single split plane at a time. Although the orientation of this plane is unrestricted, in practice several authors use axis aligned split planes. The axis-aligned BSP-tree is commonly referred to as *kD-tree* in graphics literature[8] (figure 9c). The use of axis-aligned split planes reduces the complexity of tree construction [228, 104]. In 2008, Ize et al. used an unrestricted BSP tree [117], and showed the resulting trees are often superior to restricted variants, albeit at the expensive of long build times. BSPs adapt well to the scene, and can be efficiently traversed, as shown by Jansen in 1986 [120]. High-quality kD-trees can be automatically constructed, using the *surface area heuristic* (SAH), by Goldsmith and MacDonald [91, 155]. Later, this was further improved by Hurley et al., using the *empty space bonus* [112]. Wald and Havran showed that kD-trees can be efficiently constructed in $O(N \log N)$ [247]. Zhou et al. showed that kD-trees can also be constructed efficiently on the GPU [276].

An object subdivision subdivides the list of primitives, rather than space. Since primitives are not split in such schemes, the space that primitives in different nodes of the hierarchy occupy may overlap. Examples of this class of acceleration structure are:

7 'Digital Differential Analyzer', e.g. the algorithm developed by Bresenham [38].

8 In other branches of computer science, the kD-trees (or k-d tree) is a spatial subdivision used to store points [23]. In a k-d tree, points are typically stored in all nodes, not just in the leafs. In CG, a kD-tree is a restricted form of a BSP, which stores geometry in the leafs. A single primitive may overlap multiple leafs.

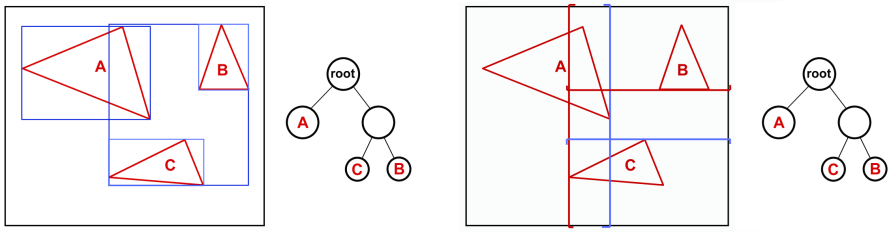


Figure 10: Object hierarchy: BVH and BIH.

BVH Figure 10a. Bounding Volume Hierarchies (BVH) recursively subdivides the list of objects, and stores, at each level of the tree, the bounds of the subtree[9]. The bounds of two nodes at the same level in the tree may overlap. Nodes in the hierarchy cannot be empty. Similar to the kD-tree, good BVHs are obtained by using the SAH to determine locally optimal splits. Most implementations implement the BVH as a binary tree. Some implementations however chose to split nodes in more than two sub-nodes. The QBVH [60]and MBVH [77] use a maximum of four children per node. Wald et al. propose to generalize this to any (a priori set) number of child nodes [257].

BIH Figure 10b. The Bounding Interval Hierarchy proposed by Wächter and Keller [242][10] is similar to the BVH, but rather than storing a full bounding box for each node, it stores intervals along one axis per node.

Blends of the two classes are also possible, and sometimes an acceleration structure of one class is used to assist in the construction of an acceleration structure of the other class. Stich et al. proposed a hybrid of bounding volume hierarchies and kD-trees that combines adaptability of kD-trees to the predictable memory requirements of BVHs [226]. Walter et al. used a kD-tree to speed up the agglomerative construction of BVHs [262].

The selection of the optimal acceleration structure for a specific hardware platform, application or even a specific scene is non-trivial. We discuss this choice in more detail in subsection 2.3.

2.2.2 *Acceleration Structure Traversal*

The suitability of a particular acceleration structure is strongly dependent on the efficiency of acceleration structure traversal. In this section, we describe acceleration structure traversal for kD-trees, BVHs and MBVHs.

9 Objects in a BVH are typically bound by spheres or axis aligned boxes, although oriented boxes (as used in early work by Rubin and Whitted, [205]) and more general convex polyhedra can also be used.

10 Developed earlier but independently in other fields than graphics by Zachmann and Nam et al.[275, 165], and referred to as SKD tree or BoxTrees.

Algorithm 2.2 Recursive kD-tree traversal. The far child and near child are determined based on the sign of the ray direction. Returns distance along ray of the intersection point.

```
function Traverse(node, T_near, T_far)
    if node.isleaf
        IntersectTriangles(node)
        return ray.T_nearest
    d ← node.split − ray.O[node.axis]/ray.D[axis]
    if d ≦ T_near return Traverse(farchild, T_near, T_far)
    if d ≧ T_far return Traverse(nearchild)
    t ← Traverse(nearchild, T_near, d)
    if t ≦ d return d
    return Traverse(farchild, d, T_far)
```

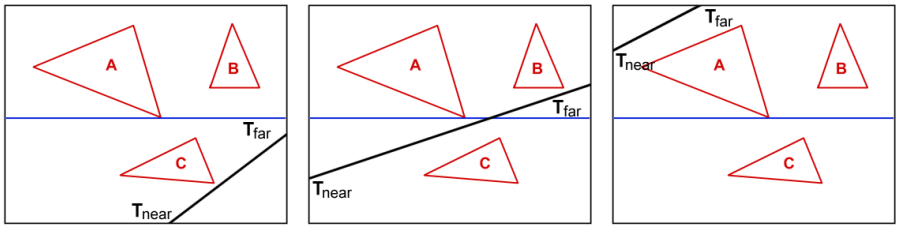


Figure 11: Three cases in kD-tree traversal. Left: the ray visits only the near child node. Center: the ray visits both child nodes. Right: The ray visits only the far child node.

kD-tree Traversal

Traversal of the kD-tree acceleration structure has been studied in-depth by several authors. For a detailed survey, see Havran's Ph.D. thesis [103]. The most commonly used traversal algorithm is a recursive scheme, originally proposed by Jansen [120, 13, 228]. This algorithm is shown in algorithm 2.2, and illustrated in figure 11. In this figure, rays travel diagonally from left to right. The split plane for the kD-tree root node splits the node along the x-axis. For $ray.D.y < 0$, the near child is always the node below the split plane, while the far child is always the node above the split plane. Three situations are possible:

1. the ray misses the far child, if the distance of the intersection point of the ray and the split plane d is greater than T_{far};
2. the ray misses the near child if $d \leqq Tnear$;
3. in all other cases, the ray first visits the near child, and, if no intersection is found, the far child.

This algorithm is typically expressed as an iterative algorithm by using a simple stack mechanism [133].

Algorithm 2.3 Recursive kD-tree packet traversal. The far child and near child are determined based on the sign of the ray direction, which must be the same for all rays in the packet. N is the number of rays in the packet.

```
function Traverse(rays[N])
    T_near[0..N-1] ← 0
    T_far[0..N-1] ← ray[0..N-1].T_max
    node ← root
    do
        if not node.isleaf
            d[0..N-1] = node.split - ray[0..N-1].O[node.axis]/]
                        ray[0..N-1].D[node.axis
            active[0..N-1] = T_near[0..N-1] < T_far[0..N-1]
            if any active d[0..N-1] ≦ T_near[0..N-1]
                node ← nearchild
            else if any active d[0..N-1] ≧ T_far[0..N-1]
                node ← farchild
            else
                push(farchild, max(d[0..N-1],
                        T_near[0..N-1]), T_far[0..N-1])
                node ← nearchild
                T_far[0..N-1] ← min(d[0..N-1], T_near[0..N-1])
        else
            IntersectTriangles(node)
            if all T_far[0..N-1] ≦ ray.T_max return
            if stack is empty return
            pop node, T_near[0..N-1], T_far[0..N-1]
```

The kD-tree can be traversed by multiple rays simultaneously using ray packet traversal, first described by Wald et al. [249]. On systems that support vector operations (such as SSE [235] and AltiVec [65]), this can yield a considerable performance improvement. For ray packet traversal, some modifications are made to the original algorithm:

- each scalar value is replaced by a vector;
- a node is visited if any active ray in the packet wants to visit it.

The iterative packet traversal algorithm is shown in 2.3.

Since the kD-tree traversal scheme depends on strict ordered traversal, and the order of traversal of child nodes depends on the signs of the ray direction, all directions of all rays in a packet must have the same signs. When this is not the case, a packet is split, and two packets traverse the kD-tree independently, both with some rays deactivated.

An important extension to the basic algorithm was proposed by Dmitriev et al. [68]. They propose to bound the rays in a packet by four planes, and use these to

Algorithm 2.4 A typical inner loop for BVH traversal.

```
while stack not empty
    if not leaf
        ray intersects far child ? push far child
        ray intersects near child ? push near child
    else
        intersect primitives in leaf
    pop
```

Algorithm 2.5 Basic ray packet traversal for a BVH.

```
while stack not empty
    if not leaf
        any ray intersects far child ? push far child
        any ray intersects near child ? push near child
    else
        intersect primitives in leaf
    pop
```

cull triangles and nodes of the acceleration structure, similar to the pyramids that where proposed by Zwaan and Jansen [240]. This technique was later successfully applied to BVH traversal. Reshetov extended frustum traversal by creating a transient frustum for the active rays in a packet when a leaf node is visited [202].

BVH Traversal

In 2007, Wald et al. showed that BVH traversal performance can be made competitive by using large packets [254]. Using a BVH as an acceleration structure for ray tracing has important advantages: unlike a kD-tree, a BVH can be changed locally while remaining valid. Also, the directions of the rays in the packet do not have to have the same sign. When using the kD-tree for ray traversal, varying signs require the packet to be split. This is particularly beneficial for secondary ray packets and large ray packets.

The basic algorithm for single-ray BVH traversal is shown in algorithm 2.4. Ray packet traversal of a BVH requires a small modification to this algorithm: instead of visiting a node if a ray intersects it, the node is visited if *any* ray in the packet intersects it. This yields the conceptually simple algorithm 2.5: instead of traversing a node when a single ray intersects it, a node is visited when any ray intersects it. Note that for BVH traversal, strict front-to-back ordering cannot be guaranteed, as the child nodes may overlap. Despite this, choosing an order in which the 'nearest' child is processed first is advantageous in most situations.

A more efficient ray packet traversal scheme was proposed by Wald et al. [254]. Their scheme consists of three stages to determine whether a node needs to be visited or not:

1. Trivial accept: when the first active ray in the packet intersects the node;

Algorithm 2.6 Efficient BVH ray packet traversal using frustum planes, early accept and early reject. N is the number of rays in the ray packet.

```
function FindFirst(rays[N], node, previousFirstActive)
    if ray[previousFirstActive] intersects node
        return previousFirstActive
    if frustum misses node return N
    for rays[previousFirstActive..N − 1]
        if ray intersects node
            return ray index

function Traverse(rays[N])
    node ← root
    firstActive ← 0
    do
        firstActive = FindFirst(ray, node, firstActive)
        if firstActive < N
            if !node.isleaf
                push firstActive, farchild
                node ← nearchild
                continue
            else
                IntersectTriangles(node)
        if stack empty return
        pop node, firstActive
```

2. Trivial reject: when the node is outside the frustum that bounds the rays in the packet;

3. Brute force scan: if all else fails, the rays in the packet are tested individually, starting with the first active ray.

Note that this traversal scheme requires planes that bound the frustum.

The traversal scheme is shown in algorithm 2.6.

In their 2008 paper, Overbeck et al. refer to algorithm 2.6 as *ranged traversal*, referring to the division of active and inactive rays: all rays up until the first active ray are 'inactive', while all subsequent rays are 'active'. Whether this division is effective on average depends on the ray distribution. This is illustrated in figure 12a. The group of rays arriving at the leaf containing triangle B is optimally identified by the first active ray 2. If this node where further partitioned, the set would likely become smaller, but not fragmented. This is not the case when the ray distribution is random (figure 12b). Even though only two rays (2 and 7) reach the node containing triangle C, six rays would traverse further if the node was further partitioned.

To improve ray tracing performance for ray distributions for which ranged traversal performs poorly, Overbeck et al. propose an alternative scheme, which

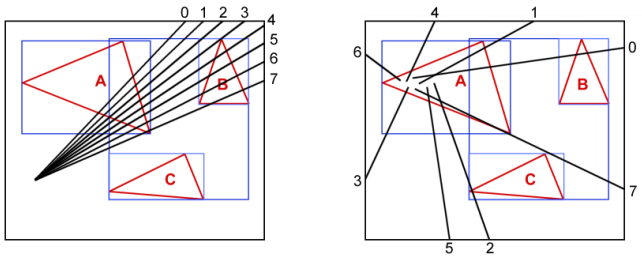


Figure 12: Two ray distributions traversing a BVH. On the left, the highly coherent and ordered ray distribution which is typical for primary rays. On the right, a ray distribution after a diffuse bounce on scene triangle A.

Algorithm 2.7 In-place sorting of the indices of active and inactive rays in the partition traversal scheme. N is the number of rays in the ray packet.

```
function FindFirst( rays[N], rayIndices[N], node, previousFirstInactive )
    if frustum misses node return N
    firstInactive ← 0
    for i = 0 to previousFirstInactive
        if ray[rayIndex[i]] intersects node
            swap rayIndex[firstInactive++], rayIndex[i]
    return firstInactive
```

explicitly partitions the set of rays in an active and inactive set. They refer to this scheme as *partition traversal*. The main component of this algorithm sorts an array of indices of active and inactive rays in-place during the intersection test, as illustrated in algorithm 2.7[11].

MBVH Traversal

kD-tree and BVH traversal schemes are designed for ray packet traversal. For divergent ray tasks, these schemes are not efficient. This led to recent investigation of N-ary BVHs (or MBVHs), where N typically equals SIMD width [257, 60, 77]. Single ray traversal through an MBVH is conceptually identical to BVH traversal (algorithm 2.8).

Using an N-ary BVH instead of a BVH has two advantages:

1. The acceleration structure has less nodes, which reduces the number of node fetches from memory;
2. The bounding boxes of the child nodes can be intersected using SIMD code, leveraging SIMD for single ray traversal.

The basic algorithm does not intersect a single MBVH node with multiple rays, as is done in ray packet traversal schemes for the kD-tree and BVH. Tsakok proposed a scheme that does this [237]. His scheme improves data locality when some

11 For efficiency reasons, partition traversal as described by Overbeck et al. operates on 'SIMD rays', which is a group of N rays, where N is the SIMD width.

Algorithm 2.8 MBVH traversal loop.

```
while stack not empty
    if not leaf
        for each child
            if ray intersects child
                push child
    else
        intersect primitives in leaf
    pop
```

Algorithm 2.9 MBVH/RS traversal.

```
taskStack ← (root, 0, N)
for rayID = 0 to N − 1
    activeRayStack[0] ← rayID
while not taskStack.Empty()
    task ← taskStack.Pop()
    list ← activeRayStack[task.lane].Pop(task.rays)
    if not task.node.IsLeaf()
        active[4] = {0, 0, 0, 0}
        for each rayID in task.list
            result[4] ← Intersect4(rays[rayID], task.node)
            active[4] ← active[4] + result.hit
            for i = 0 to 3 if result[i].hit
                activeRayStack[i].Push(rayID)
        for i = 0 to 3 if active[i] > 0
            taskStack.Push(task.node.child[i], i, active[i])
    else
        for each rayID in list
            for each triangle in node.triangles
                intersect(triangle, ray[rayID])
```

coherence is available, and amortizes the cost of fetching an MBVH node over all the rays in a stream. It falls back to efficient single ray traversal when the size of a stream drops to one. We discuss this scheme in more detail here, since we will use it later in the RayGrid scheme, described in chapter 5.

The MBVH/RS scheme is outlined in algorithm 2.9. MBVH/RS operates on an array of rays. It uses two types of stack: the first is a set of four stacks (one for each SIMD lane), which stores streams of active rays. The second stack is the task stack, which stores tasks consisting of a number of rays and a node pointer. In the traversal loop, a task is obtained from the task stack. The rays in the task are then intersected with four child nodes. When a ray intersects a child node, it is added to the stream of active rays for that node. Once all rays have been processed, a new task is added to the task stack for each output stream that received at least one ray.

Algorithm 2.10 Efficient sorting of the four values in a 128-bit register. Variable v0 contains the values to be sorted. At the end of this code, v0 contains the sorted values. The lowest two bits of v0 contain the original index of each value. This code uses 15 SSE instructions to sort the numbers, and contains no conditional code. Note that the sorted numbers are modified: the lowest two bits of the mantissa are sacrificed. This does not affect the sorting order.

```
1  // values in idxmask4 are set to 0xfffffffc
2  // values in idxadd4 are set to { 0, 1, 2, 3 }
3  __m128 v1, v2, v3, t;
4  v0 = _mm_or_ps( _mm_and_ps( v0, idxmask4 (multicore CPU + GPU)), idxadd4
       );
5  v1 = _mm_movelh_ps( v1, v0 );
6  t = v0;
7  v0 = _mm_min_ps( v0, v1 ), v1 = _mm_max_ps( v1, t );
8  v0 = _mm_movehl_ps( v0, v1 );
9  v1 = _mm_shuffle_ps( v1, v0, 0x88 );
10 t = v0;
11 v0 = _mm_min_ps( v0, v1 ), v1 = _mm_max_ps( v1, t );
12 v2 = _mm_movehl_ps( v2, v1 );
13 v3 = v0;
14 t = v2;
15 v2 = _mm_min_ps( v2, v3 ), v3 = _mm_max_ps( v3, t );
16 v0 = _mm_shuffle_ps( _mm_movelh_ps( v1, v3 ), _mm_shuffle_ps( v0, v2, 0x
     13 ), 0x2d );
```

Adding intersected nodes to the task stack can either be done in random order, or sorted. Although a strict front-to-back traversal order cannot be guaranteed for a stream of rays, some ordering is beneficial, as it increases the number of nodes
the distances at which rays hit the nodes are summed. The nodes are then sorted according to this summed distance.

The implementation of the sorting requires careful attention, as a poor implementation can easily nullify the gains. We base our implementation on the work by Furtak et al. [84], who describe an efficient SIMD implementation of a 4-element sorting network for floating point values in a 128-bit register. We modify their implementation to allow the sorting of MBVH nodes based on the four distances, rather than the distances themselves. For this, we store the original node indices in the lowest two bits of the four floats, prior to sorting them. After sorting, we then extract these indices for the final ordering of the nodes. Our implementation is shown in listing 2.10.

The described algorithm can be efficiently implemented using the SSE2 instruction set. A full implementation is provided in appendix C.

In the previous subsections, we discussed acceleration structures and acceleration structure traversal for efficient ray / scene intersection. When using a hierarchical acceleration structure, the cost of ray traversal for N primitives is $O(\log N)$. This does not take into account the cost of precalculations however. Construction time for an acceleration structure is $O(N)$ at best for a regular grid, or $O(N \log N)$ for hierarchical structures. In an interactive context, this construction time can be considerable, even for moderately complex scenes. For a static scene, this cost is amortized over many frames. In the context of a game however, the scene is often dynamic, and rendering time therefore must include acceleration construction or maintenance. Wächter refers to the total of acceleration structure maintenance plus rendering time as *time to image* (TTI). This terminology was later adopted by others [242, 256, 202].

Ray tracing of dynamic scenes was mentioned as early as 1999, by Parker et al., who propose to keep dynamic objects outside the acceleration structure and intersect them separately [184]. Similarly, Bikker proposed to use a secondary acceleration structure for dynamic objects [27]. Wald et al. propose to refit the BVH for deformable scenes [254]. Ize et al. propose to refit and rebuild the BVH asynchronously [116]. A full solution was proposed by Wald et al. and implemented in the Arauna ray tracer. A top-level BVH is constructed over per-object BVHs, which are either static, rebuilt or refitted (see section 3.2). Several authors assume that games ideally should be able to use fully dynamic environments [82], but this is generally not needed: most games only require a small portion of the game world to be dynamic [256, 27].

When optimizing TTI for a specific rendering algorithm and application, we must take into account the expected scene complexity, the extent to which the scene is dynamic, and the expected summed ray query time. When the portion of the TTI spent on updating the acceleration structure is relatively large, it becomes attractive to reduce this portion, even if this leads to a decrease in ray query performance. This has led to the development of very fast BVH and kD-tree construction algorithms, that sacrifice some quality for build performance, by using a median split [242] or an approximation of the SAH using an approximation of the cost function [111] or a fixed number of discrete split plane candidates (known as *binning*) [245]. Acceleration structure construction can also be improved by leveraging the compute power of the GPU [276, 126, 146, 182]. Construction of the acceleration structure can be sped up further by using regular grids [253]. This has a considerable impact on ray query performance however, and is thus only worthwhile when TTI is strongly dominated by acceleration structure updates.

When TTI is dominated by ray queries, it is important to have a high-quality acceleration structure. A high-quality kD-tree or BVH is obtained using the SAH. Further improvements for BVHs can be realized when using spatial splits [226] and agglomerative construction [262]. Once the BVH is constructed, its total SAH cost can be reduced using tree rotations [137].

In this thesis, we frequently describe a performance level as *real-time*. In computer science, a system is considered *real-time* if it can guarantee a response to an event within a certain amount of time [20]. In the context of graphics for (multicore CPU + GPU)games, real-time can be interpreted in the perceptual sense [130]: a certain frame rate can be considered real-time if the application response to user input is perceived as instantaneous [98], or if the human eye perceives the depicted motion as continuous. In graphics literature, *real-time* is an abstract interval, defined by a certain minimum frame rate. Related to real-time is *interactive*. A frame rate is interactive when frame updates are fast enough to allow the user to operate directly on the rendered image.

Multiple factors determine whether real-time frame rates can be achieved, such as the available hardware, the complexity of the scene, and the rendering quality.

Modern games are typically designed for a broad range of hardware configurations. There are two distinct ways for a renderer to keep within real-time limits. The first is to scale the workload dynamically to enforce a real-time frame rate. The second is to rely on the user to specify rendering features and resolution for which the input data can typically be rendered at real-time frame rates.

There exist several approaches that allow a renderer to scale the workload dynamically. One such approach is referred to as *level of detail* (LOD), as originally proposed by Clark [46]. LOD decouples scene complexity from rendering complexity. It can be implemented using manually crafted meshes of varying complexity, or using dynamically generated meshes [271, 75, 149]. Duchaineau et al. apply LOD to large terrain meshes in their ROAM algorithm [71], based on the CLOD algorithm proposed by Lindstrom et al. [151]. The appropriate LOD is commonly chosen based by comparing an estimated or calculated maximum error to a user-specified error threshold. Image fidelity can be traded for rendering performance by tuning this threshold. Olano et al. propose to apply a similar mechanic to shaders, by automatically simplifying a shader based on distance, size, importance or rendering time budget [176].

Scene complexity is one parameter that affects rendering performance, although ray tracing is less dependent on scene complexity compared to rasterization, as it only visits a subset of all objects once the acceleration structure is built. Another parameter is screen resolution. Binks proposes to make rendering resolution dynamic [30].

Although several methods exist to adjust rendering speed dynamically, in practice, real-time rendering in games is often achieved by providing scene data that will not exceed the capabilities of the renderer. Achieving real-time performance is at least partially the responsibility of the visual artist, who must make sure that for all possible camera views, visible polygon budgets are not exceeded. Assuming high-quality scene data, real-time performance then becomes the responsibility of the end user, who manually selects rendering options and screen resolution. The three most popular rendering engines for games (ID Tech 5 [44], Unreal Engine

3 [229], Unity 3D [10]) all scale with system performance, and require manual feature and detail selection.

For the purpose of this thesis, we assume that a renderer is real-time when there exists a commodity hardware configuration on which the renderer can achieve an average frame rate of 20fps, and never drops below 15fps, for typical input data. For the Whitted-style ray tracer described in chapter 3 and 4, we will show that this can be achieved. For the path tracer described in chapter 5, 6 and 7, we will dynamically scale the number of samples per pixel, trading variance for rendering speed.

2.5 OVERVIEW OF THESIS

In the following chapters, we build on the concepts described in this chapter.

The Arauna ray tracer (chapter 3) implements the recursive ray tracing algorithm proposed by Whitted, embedded in a complete ray tracing based renderer, designed for games. The proposed architecture is able to render scenes that fit in main memory, at real-time frame rates. A secondary acceleration structure is constructed over the light sources, to allow for efficient rendering of scenes with large amounts of point lights. Illumination is limited to direct lighting however.

Chapter 4 describes an approach to add sparsely sampled indirect illumination to the Whitted-style ray tracer. By decoupling shading calculations from rendering, we exploit temporal and spatial coherence in low-frequent indirect illumination, as well as direct illumination from large area light sources. The proposed precalculated pointset adapts to local geometry, can be efficiently queried, and does not suffer from aliasing or temporal low-frequent noise. Being precalculated, it does however assume static geometry.

In the subsequent chapters, we seek to alleviate the limitations of Whitted-style ray tracing by implementing the path tracing algorithm. Efficiently doing so is challenging: in path tracing there is a lack of coherence between object data and ray queries. As a result, caching hardware is not as efficient as with Whitted-style ray tracing.

We study the problem of efficient path tracing, first on the CPU (chapter 5). We propose a data-parallel algorithm that improves data locality for divergent ray queries. We also study the efficiency of divergent ray traversal on the GPU (chapter 6). On these streaming processors, efficiency is determined mostly by execution coherence and utilization of the many hardware threads of these devices. We propose algorithms that improve coherence and utilization, and study the impact of these algorithms on variance.

Finally, in chapter 7 we describe the Brigade path tracing architecture, which provides unbiased rendering for games. In this renderer, variance is traded for rendering speed to achieve real-time performance.

Part I

REAL-TIME RAY TRACING

3

REAL-TIME RAY TRACING

The significance of ray tracing for games has been pointed out by several authors[1][177, 244, 82, 193]. For gamers, ray tracing promises higher levels of realism. Game developers are attracted by efficient game production, intuitive rendering and reduced code complexity, as well as a content creation pipeline that more closely resembles those used for movie production. As shown in the previous chapter, rendering algorithms based on ray tracing support a number of important terms of the rendering equation which are not easily handled by rasterization-based rendering. This allows for correct shadows in an Appel-style ray tracer [11], specular reflection and transmission in a Whitted-style ray tracer [265], glossy reflections and soft shadows in a distribution ray tracer [51] and global illumination in a path tracer [125].

Despite these benefits, ray tracing has so far not been widely adopted for game development. Adoption is held back by dedicated rasterization hardware, rasterization-based legacy code and performance concerns, but also a lack of proof-of-concept games and rendering engines that could alleviate these concerns. To gain more insight in the actual benefits of ray tracing, the implementation of a fast ray tracing based renderer, and the use of it in actual games, is desired. A well-optimized implementation also provides a valuable tool for researchers who wish to investigate new algorithms in a practical context.

In this chapter, we describe the Arauna ray tracer. This renderer implements the Whitted-style ray tracing algorithm, with some small modifications specifically designed for games. Its practical implementation also explicitly targets this context, with a fast *fixed-function* pipeline, an emphasis on low-level optimization, and architecture decisions that target real-time rendering of typical game scenery.

3.1 CONTEXT

The Arauna ray tracer is designed to be a rendering engine for games. This focus brings a number of requirements that affect various components of the renderer.

REAL-TIME Games require real-time rendering. This limits frame time to tens of milliseconds, and makes low-level optimization a necessity. Shading quality, screen resolution and scene complexity must be balanced to meet this requirement.

STANDARD PCS For a valid proof-of-concept renderer, a broad audience must be able to run games produced using the renderer. It is not uncommon for a

1 This chapter is based on the RT'07 keynote speech "Ray Tracing through the Eyes of a Game Developer", and the accompanying paper [27].

game to require high-end 'off-the-shelf' hardware, equipped with a recent CPU and GPU; we therefore target high-end consumer hardware. To make optimal use of the available hardware, the ray tracer uses both high-level parallelism and instruction level parallelism, in order to keep all SIMD units of all CPU cores busy.

RESOLUTION Games are generally played at resolutions of at least 1280x720. The ray tracer is designed for this target resolution, even when a proof-of-concept game uses a lower resolution, in anticipation of sufficient compute power.

IMAGE FIDELITY Image quality competes with (and must ultimately surpass) rasterization based solutions. The ray tracer is able to render scenes with a detail level and material complexity similar to what is used in current games. When necessary or opportune, believable rather than correct graphics are acceptable.

DYNAMIC WORLD A game world is dynamic, although it is common practice to assume that a substantial part of it is static. It may be assumed that the game world is managed using a scene graph. The ray tracer strives for an optimal *Time To Image* (TTI, see chapter 2) for this type of scenery.

Arauna aims to implement an efficient combination of acceleration structure construction and maintenance, ray traversal, primitive intersection, and shading, aiming for an optimal time to image. This section describes each of these elements in more detail.

We conducted our experiments using three scenes. The first scene (figure 13a) is a scene from the student game *Let there be Light*, and consists of 88k triangles, with textures and normal maps. This scene is considered representative for a game scenario. The second one (figure 13b) is the standard architecture model of the Sponza Atrium, by Marko Dabrovic. This model includes consists of 93k triangles, and is included for comparison with existing work. The third scene (figure 13c) is the Sibenik Cathedral, also modeled by Marko Dabrovic. We added a low-polygon version of the Lucy model from the Stanford repository, which brings the polygon count in this scene to 602k. The majority of these polygons are clustered in a small area of the scene. We will use these scenes throughout this thesis.

3.2 ACCELERATION STRUCTURE

Most games use a scene graph to store the object hierarchy and spatial relations between objects in the scene. This scene graph contains valuable information which can be used to improve construction efficiency as well as quality of an acceleration structure [67]. This information can be exploited in two ways. First, the object bounding box planes provide good candidates for split planes near the root of a kD-tree or BVH. The cost of these initial splits can be dramatically reduced by considering only these planes. Secondly, for a BVH we can construct a BVH for

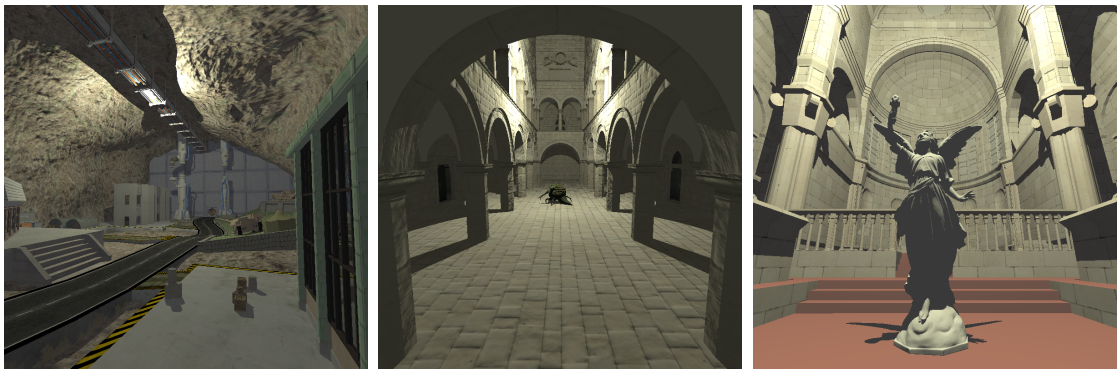


Figure 13: Scenes used for our experiments: *Modern Room* from *Let there be Light*, the Sponza Atrium, and the Lucy statue in a model of the Sibenik Cathedral.

Algorithm 3.1 Updating the scene graph. A BVH per scene graph node is rebuilt or refitted for changed objects.

```
for each node in scenegraph
    if not node.IsSatic()
        node.Transform()
        if node.Changed()
            if node.IsRigid()|node.IsDeforming()
                node.Refit()
            else /* if node.IsDynamic() */
                node.rebuild()
scene.buildtopbvh(scenegraph)
```

each scene graph node. The scene BVH is then constructed using a top-level BVH over the scene graph node BVHs.

Arauna uses this second method to construct a BVH. During construction, properties of the scene graph nodes are used to balance BVH quality and construction performance. Arauna distinguishes static and dynamic objects. For static objects, a scene graph node BVH is constructed once, after which it no longer contributes to frame time. The construction can therefore use a high-quality BVH construction algorithm. Dynamic objects are further classified into rigid, deforming and dynamic. Rigid objects will undergo rigid translation and transformation. The BVHs for these objects do not need reconstruction. Instead, the bounding volumes stored in the nodes of the BVH are refitted to the primitives stored in the leafs. Deforming objects change shape but have a constant polygon count and topology. Dynamic objects undergo arbitrary changes, including addition and deletion of polygons and changes to topology. For these objects, the scene graph node BVH must be rebuilt per frame. Construction speed is important for these BVHs. A faster scheme is therefore used, yielding a lower quality BVH.

Algorithm 3.1 outlines the scene graph update.

Constructing a BVH per scene graph node allows for object-specific BVH construction.

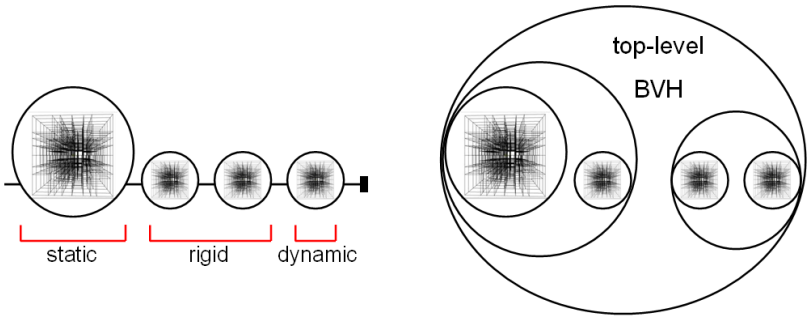


Figure 14: Constructing the top-level BVH. Scene graph node BVHs are constructed using node-specific construction algorithms, and then combined until one root node combines them.

For static objects, we use a high quality BVH builder, which uses the surface area heuristic to determine the optimal split plane position. Our implementation uses a binning approach, where the extent of the bounding box for each axis is subdivided in $N + 1$ discrete regions, separated by N planes. By choosing N to be a multiple of 4, SIMD can be used to increase efficiency of BVH construction, without sacrificing BVH quality. The value of N can then be used to balance quality and construction performance. For a large N, quality improves, but construction time increases. At each separating plane, we consider a spatial split, as proposed by Stich et al. [226].

For dynamic objects, we use a similar BVH builder, with a number of simplifications that reduce construction time. The high-speed builder does not consider the spatial split, and uses a small value for N. It also limits the search for the best split plane to the axis for which the bounding box has the greatest extent. Although this yields a lower quality BVH, in practice this improves TTI. Dynamic object BVHs are typically traversed by a relatively small number of rays. The reduced efficiency of these rays does not outweigh the gains in construction time.

Rigid objects that changed position or orientation reuse the BVH from the previous frame. This BVH is refitted by adjusting the bounding box of each node to the primitives in the node (for leafs) or to the bounds of the child nodes (for interior nodes). Since interior nodes rely on up-to-date bounds of their child nodes, this process must be executed in a bottom-up fashion, using a recursive algorithm. Alternatively, the array of BVH nodes can be refitted *back-to-front*. Assuming that the nodes in the array were allocated in a top-down fashion during BVH construction, each node in the array is guaranteed to be either a leaf node, or lower in the hierarchy than the previous node. In practice, the back-to-front approach is simpler, but not faster than the recursive algorithm, due to the poor memory access pattern.

Table 1 shows timing information for BVH construction and refitting. Refitting takes far less time than rebuilding, but rebuilding benefits more from multi-threaded processing.

Once all scene graph nodes have been updated, a new top level BVH is constructed. For this, we use the agglomerative algorithm described by Walter et al.

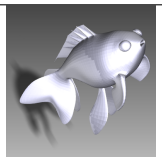
tris	1 thread		4 threads	
tris	rebuild	refit	rebuild	refit
44k 	35.4	2.3	8.4	1.6
88k 	76.5	7.1	19.5	4.0
388k 	397.0	35.7	103.9	19.9

Table 1: BVH construction and refitting time in milliseconds for a group of 4 objects, with a combined triangle count of 44k, 88k and 388k triangles, using 1 thread and 4 threads.

Figure 15: Scenes from the student game *Pirates on the Edge*.

[262] (figure 14). During each iteration of this algorithm, two nearby scene graph node BVHs are combined into one BVH, by creating a new node, and linking the two BVHs to it as child nodes. The bounding volume of the new node becomes the union of the bounding volumes of the two linked BVHs. The new node is now a valid root node for the two BVHs, and will be combined with other scene graph node BVHs in subsequent iterations, until one node remains.

The resulting scene update is highly efficient. For a simple static scene, most steps are skipped, which makes overhead negligible. For typical game scenes, the blend of static, rigid and dynamic objects is effectively exploited. We have applied this approach to render the world of the student game *Pirates on the Edge* (see figure 15). This game features a static world of 363k triangles, an animated ocean with simulated waves consisting of over 129k triangles, and several pirate ships of 7k triangles, each undergoing rigid motion. A maximum of 512 cannonballs may be in flight at any time during the game; these are rendered using a single reflective sphere primitive.

In the previous paragraph, we described the construction of the top-level BVH. The BVH is traversed by ray packets. Arauna uses the traversal scheme proposed by Wald et al. [254] for primary and secondary rays (see section 2.2).

For each visited BVH node, the ranged traversal scheme uses at most three tests:

1. early hit: if the first active ray in the packet intersects the BVH node, the packet traverses the node;
2. early miss: if the BVH node is outside one of the bounding planes of the packet, the BVH node is skipped;
3. brute force scan: the first ray that intersects the BVH node is searched by testing all active rays against the BVH node.

This algorithm allows traversal of BVH nodes at the cost of a single ray/box test, if condition 1 is met. An additional AABB/frustum test must be performed if condition 1 fails. The first two tests are independent of packet size. The third test is not: in a worst case scenario, where all rays miss the BVH node, each ray is tested against the AABB. Similarly, when processing a leaf, a range of rays is tested against primitives. To reduce this cost, the basic algorithm is augmented with a number of low-level modifications that further improve performance.

Packet Layout

The layout of a packet is carefully chosen. Arauna uses 16x16 rays in a primary ray packet, which has been empirically found to be a good size for primary ray packets. Of these 256 rays, the first 64 rays form the first quadrant of the packet (see figure 16). Using this ordering, the packet can be trivially split in four 8x8 packets, which is a better size for secondary rays. After the primary intersection, traversal continues with these 8x8 packets. Within a quadrant of the packet, rays are laid out as 2x2 squares of rays. The 2x2 squares increase ray coherency within a SIMD ray packet of four rays. This increases the probability that four rays traverse the same BVH nodes and hit the same primitive, resulting in more efficient primitive intersection and shading.

Sub-frusta

For a primary ray packet, two extra planes are added. These planes subdivide the frustum in four quadrants or sub-frusta, each of which contains 64 rays. When processing a leaf, each primitive is first tested against the frustum of the packet. If the primitive is not outside the frustum, it is tested against the two extra planes, to determine which quadrants the primitive potentially intersects. This way, sets of 64 rays that are guaranteed to miss the primitive are not tested. The active rays in each quadrant are then tested in groups of four, using a SIMD implementation of the modified Pluecker test proposed by Benthin [21].

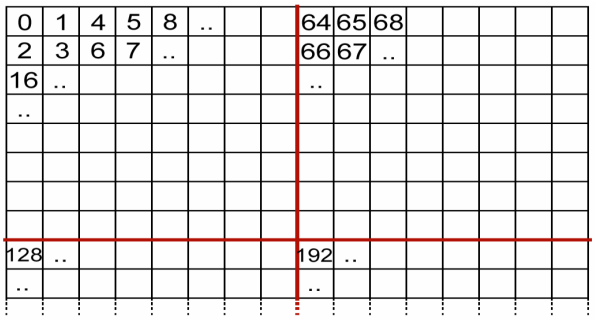


Figure 16: 16x16 packet layout.

Shadow Rays

Shadow rays in Arauna are generated per light source, once packet traversal for primary or secondary rays has completed. The shadow ray packet contains a shadow ray for each intersection point. If no intersection point was found, or if the intersection point did not require a shadow ray (e.g., when a specular material was encountered), the shadow ray is deactivated. The shadow rays start at the point light source, and extend to the intersection point. Note that shadow ray direction is important: when tracing from the intersection point to the light source, occlusion may be missed when a primary or secondary intersection point is on an edge of a shadowed polygon.

To prevent self-shadowing for intersection points, we use a small epsilon value.

Secondary Rays

Packets for primary rays and shadow rays use a common origin for all rays. For secondary rays, such as reflected and refracted rays, this property is typically lost. For these rays, we switch to a more generic triangle intersection test.

3.4 DIVERGENCE

We measured the performance of our ray traversal implementation. Since performance is sensitive to ray divergence, we measured performance for various levels of divergence.

The test setup is as follows: for the three test scenes, three camera views are chosen, with varying occlusion and complexity levels. To minimize shading cost, visualization is limited to depth information. Four levels of divergence are then tested. For an objective measurement, the four levels all consist of primary rays with a common ray origin. Divergence is increased by increasing the angle between the rays in a tile. For divergence levels beyond 1, the tiles thus appear to overlap. Using this approach, all divergence levels access the same geometry (except for the right column and bottom row of tiles). Table 2 shows the impact of ray divergence in the Sponza scene for three viewpoints. Table 3 shows the impact of ray divergence

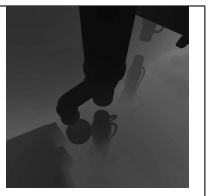
divergence			
1	19.49 (100%)	22.12 (100%)	19.01 (100%)
2	13.59 (70%)	17.01 (77%)	15.06 (79%)
4	8.20 (42%)	11.40 (52%)	10.03 (53%)
8	4.42 (23%)	6.56 (30%)	5.66 (30%)

Table 2: Ray packet performance, in millions of rays per second, for three views of the Modern Room scene and various levels of divergence. Divergence is in multiples of 0.034 degrees between adjacent rays.

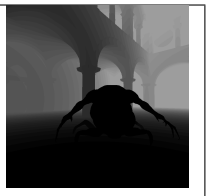
divergence			
1	16.85 (100%)	25.11 (100%)	18.44 (100%)
2	11.61 (69%)	18.83 (75%)	12.93 (70%)
4	6.98 (41%)	12.56 (50%)	7.48 (41%)
8	3.85 (23%)	7.71 (31%)	3.87 (21%)

Table 3: Ray packet performance for three views of the Sponza Atrium.

in the Escher scene. Measurements are performed on a single core of a Intel Xeon processor running at 3.4 Ghz, at a resolution of 1280x800.

It is clear that even for primary rays, ray divergence considerably affects traversal efficiency. The adverse effect depends on the angle between rays, but also on scene complexity. Note that the angle decreases when resolution increases (assuming the field of view remains constant). As mentioned by Wald in his Ph.D. thesis, packet ray tracing performance thus does not linearly scale with screen resolution [244].

3.5 MULTI-THREADED RENDERING

Ray tracing is known to be an "embarrassingly parallel algorithm" [163], since rays for individual pixels do not depend on each other's data and can thus be traced in any order. In practice, a thread granularity of one pixel is inefficient, due to operating system task synchronization overhead. Instead, the workload is split in tiles of pixels, which correspond to the ray packets described in section 3.3. The tiles are placed on a stack by a master thread, and then processed by one or more rendering threads, using a lock-free, wait-free scheme. One rendering thread is

divergence			
1	14.30 (100%)	9.20 (100%)	10.30 (100%)
2	10.13 (71%)	5.19 (56%)	5.78 (56%)
4	6.30 (44%)	2.46 (27%)	3.36 (33%)
8	3.70 (26%)	1.29 (14%)	2.13 (21%)

Table 4: Ray packet performance for three views of Sibenik Cathedral with Lucy.

started for (and locked to) each available physical processor core. A rendering thread executes the code shown in algorithm 3.2.

The rendering thread thus sleeps until it receives a signal from the master thread. At this point, it obtains a rendering task, and renders it. Since all render threads use the same task list, special care must be taken to prevent threads from rendering the same tile, or invalid tile numbers (e.g., -1). For this, an atomic decrement is used (line 8). As it is possible that variable waiting changed between the conditional on line 6 and the atomic decrement on line 8, the obtained value is checked again on line 9. Once no tiles are left to be rendered, the rendering thread notifies the master thread, which wakes up once all rendering threads have run out of work.

In this scheme, the only OS-dependent synchronization happens at line 3 and 13, which are both executed once per frame. Another potential inefficiency is the task granularity: it is possible that a rendering thread starts working on a tile at the same time that all other rendering threads finish their tiles. In that case, rendering is essentially single-threaded for a moment. The impact on rendering efficiency depends on the ratio of tiles to rendering threads, as well as the cost of the final tile: on a many-core architecture, it is important to have many small tiles to reduce this effect.

3.6 SHADING PIPELINE

Arauna uses a fixed-function pipeline, and implements the Phong reflection model [189], augmented with an emissive component and a geometry factor:

$$I_p = k_e + k_a i_a + \sum_{m \in lights} G_m(k_d(L_m \cdot N)i_d + k_s(R_m \cdot V)^{\alpha} i_s) \tag{3.1}$$

In this formulation, k_e is the emissive color of the material, k_a, k_d and k_s denote the ambient, diffuse and specular reflection constants for a material, and α is the shininess constant. Lighting is defined by the diffuse component i_d, the specular

Algorithm 3.2 Code for a lock-free, wait-free processing loop for a rendering thread.

```
 1  while (1)
 2  {
 3     WaitForSignal( goSignal[threadIdx], INFINITE );
 4     while (1)
 5     {
 6        if (waiting > 0)
 7        {
 8           volatile LONG w = AtomicDecrement( &waiting );
 9           if (w >= 0) RenderTile( w );
10        }
11        else
12        {
13           Signal( doneSignal[threadIdx] );
14           break;
15        }
16     }
17  }
```

component i_s, and the ambient lighting i_a. Like in most implementations, the Phong reflectance model is evaluated for red, green and blue. On a four-wide SIMD architecture, it is tempting to use vector operations to operate on 128-bit colors, using 32-bit floats for alpha, red, green, blue. In practice however, this leads to inefficient SSE code; it is much more efficient to operate on four rays in parallel instead.

In the shading pipeline, we perform the following steps:

1. obtain the diffuse material color by sampling the (HDR or 32-bit) texture, or the diffuse color of the material
2. obtain the per-pixel normal by sampling the normal map of the material (if available)
3. calculate the surface normal by combining the interpolated surface normal and the per-pixel normal
4. determine visibility of each relevant light source
5. evaluate the Phong model.

In our implementation, two optimizations are performed to optimize the throughput of the shading pipeline. The first is to minimize the time spent in code that cannot be executed for four rays in parallel. The second is to interleave texture data and normal map data, to reduce the number of cache misses during texture fetches.

Consistent SIMD Shading

By executing the shading pipeline for four rays in parallel, maximum SIMD efficiency is achieved. Arauna evaluates the Phong model for four rays in parallel using SIMD in the same manner as the model would be evaluated using scalar code for a single ray: the color components red, green and blue are calculated separately. A simple color addition would be implemented using a SIMD instruction that operates on the red components of four rays, followed by an instruction that operates on the green components of these rays, and finally an instruction for the blue components. The data layout that is required for this is known as *structure of arrays* (SoA). Reorganizing texture data from *array of structure* (AoS) format to SoA is done during the *gather* operations in steps 1 and 2, where 128-bit SIMD registers are filled with the diffuse color of the material, obtained from the texture map. Normals are processed in the same way.

At the end of the shading pipeline, colors for four rays are converted from SoA RRRR,GGGG,BBBB to four 32-bit integer ARGB results. The overhead of converting to and from SoA is worthwhile: despite the overhead, the shading pipeline executes more than four times faster than the same pipeline in scalar code. The expected limit of 400% is exceeded as a result of the availability of certain SIMD instructions that operate faster on four elements than a corresponding scalar instruction on a single element. Besides this, SIMD allows simple conditional constructs to be converted to branch-free code, which prevents expensive branch mispredictions present in the scalar code.

Consistent use in SIMD code of data organized in a SoA fashion yields efficient and maintainable code, that is easily extended to larger SIMD widths.

Interleaved Normal Map

During the gather operation in steps 1 and 2, cache misses are a significant source of delays. Little can be done to avoid these, apart from tracing coherent packets and deliberate oversampling. However, for every cache miss in the texture data, a cache miss in the normal map data is very likely to occur as well. This can be prevented by interleaving texture data and normal map data. This way, reading the texel also stores the normal in L1 cache, which can then be read without any penalty. This reduces the cost of normal mapping significantly, to the point where the use of normal mapping is essentially free [2]. A consequence of this approach is that the normal map must be the same resolution as the texture map.

3.7 MANY LIGHTS

The most expensive part of the shading pipeline is the visibility determination for light sources. For a typical game scene, tens or even hundreds of lights are required. Since each light potentially affects every point in the scene proportional

2 When using normal mapping on every surface, the overall impact is 1.5%.

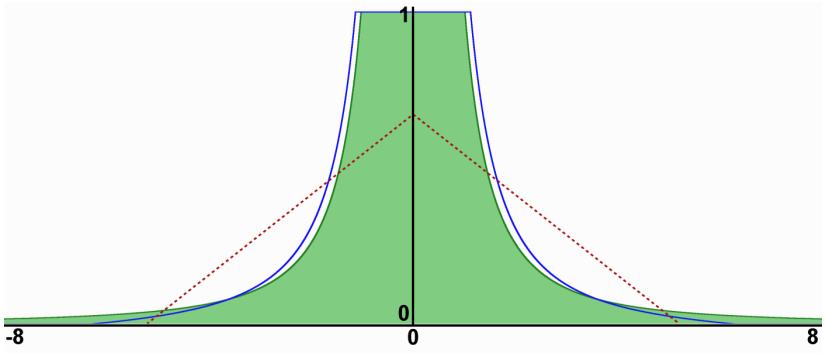


Figure 17: Various lighting models. Dotted red: linear fall-off; blue: quadratic + linear fall-off; solid green: quadratic fall-off (for reference).

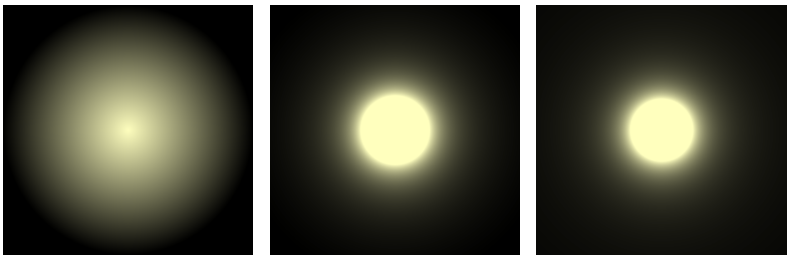


Figure 18: Light spot shapes: a) Linear fall-off. b) Quadratic + linear fall-off. c) Quadratic fall-off (for reference).

to $\frac{1}{r^2}$, tracing shadow rays can easily become a bottleneck. To reduce the impact that a large number of light sources has on frame rate, Arauna uses a simplified fall-off model for lights, where lights have a limited sphere of influence.

Quadratic fall-off is approximated as:

$$f(r) = \max(0, \frac{1}{\alpha r^2} - \beta r) \tag{3.2}$$

where α is a value close to one, and β is a small positive value. A positive value for β ensures that the function reaches 0 at distance $r = 1/(\sqrt[3]{\alpha}\sqrt[3]{\beta})$. Figure 17 shows the graphs for quadratic fall-off, linear fall-off and the approximation of equation 3.2. Note how quadratic fall-off has an infinite sphere of influence, whereas both approximations have a finite range. Figure 18 shows the shape of the light spots.

The approximation of equation 3.2 approximates the original shape reasonably well, and allows us to discard lights that are too far away, reducing the total number of lights that affects a single point in the scene.

Determining the set of lights that affects a point can still become a bottleneck if it is implemented as a loop over all light sources and a distance calculation per light source. For this reason, besides the BVH for the scene primitives, a second BVH is used, which stores the lights in the scene. The light BVH is constructed once per frame, in an agglomerative fashion. Pairs of lights are grouped in an enclosing sphere, until the top level of the BVH is reached, which is a single

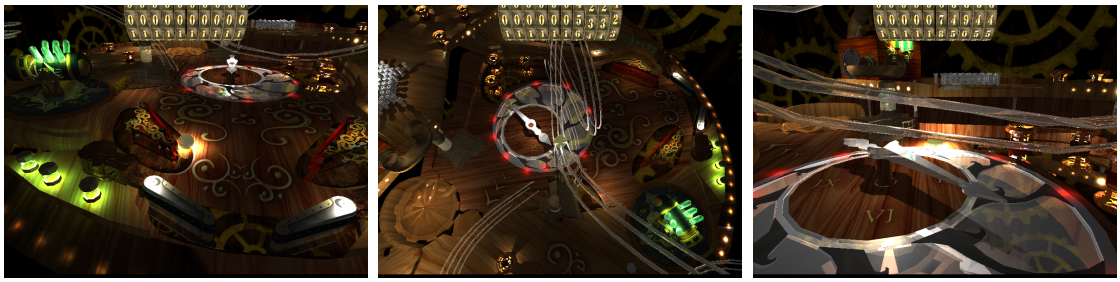


Figure 19: Scenes from the student game *Time of Light*.

sphere, containing all the light sources in the scene. The light sources that affect an intersection point can now be quickly determined by traversing the BVH.

Our approach is similar to the approach proposed by Schmittler et al. [209], who use a kD-tree. Using a BVH solves problems with the subdivision heuristic: Sometimes, lights overlap significantly, in which case it is hard to find a good split plane position. Also, the box shape of kD-tree nodes does not match the sphere of influence of a light well, which leads to considerable overhead for points that will not be lit by a light, yet are in a leaf node containing the light. Spherical BVH nodes by nature enclose the light volumes tightly.

The approximation for fall-off and the light BVH have been used in the student game *Time of Light* (see figure 19). In this pinball game, the scene consists of a detailed table with 60 light sources, most of which are small. The average number of lights that affects a point in the scene is less than 3.

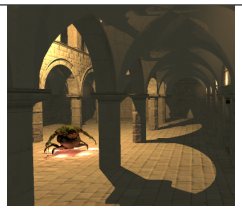
3.8 PERFORMANCE

In this section, we present performance figures for the Arauna ray tracer. We measured performance for the three scenes shown in figure 13. Measurements are performed on a dual-processor system, using two six-core Xeon processors, providing 24 cores (with hyperthreading) running at 3.4 Ghz. To stress the ray tracer, we modified materials and added extra light sources. The Modern Room scene is rendered using 12 light sources along the ceiling, all of which affect all scene surface points. The Sponza scene uses 25 light sources. Most of these have a small range of influence. The Sibenik Cathedral scene uses 4 light sources, all of which affect the entire scene. All images are rendered at a resolution of 1280x720 pixels.

Table 5 shows performance when rendering using 1, 2, 6, 12 or 24 rendering threads. Up to 12 cores, performance scales almost linearly. Beyond this point, the use of hyperthreading results in a modest improvement of performance of about 20%. Peak performance is achieved in the Modern Room scene: for this scene, 549 million rays per second were traced. Note however that many of these are shadow rays: each pixel requires 12 shadow rays.

For the three scenes, at a resolution of 1280x720, performance never drops below 20fps, and can thus be considered real-time.

						
cores	fps	mrays/s	fps	mrays/s	fps	mrays/s
1	1.62	35	1.74	37	2.27	38
2	3.24	70	3.47	74	4.52	75
6	9.77	211	10.34	221	13.45	225
12	18.57	328	19.12	420	26.85	448
24	23.70	506	25.10	534	33.22	549
rpp	23.93		23.62		18.40	

						
cores	fps	mrays/s	fps	mrays/s	fps	mrays/s
1	5.22	25	4.08	31	3.43	28
2	10.31	51	8.09	63	6.80	56
6	30.1	151	24.29	188	20.02	168
12	59.37	297	48.02	365	39.36	331
24	74.91	379	51.98	394	46.60	398
rpp	5.50		8.25		9.23	

						
cores	fps	mrays/s	fps	mrays/s	fps	mrays/s
1	1.71	18	2.09	19	2.06	18
2	3.27	35	4.23	35	4.08	35
6	9.78	106	12.17	101	12.47	108
12	19.28	207	24.29	200	24.23	213
24	25.1	269	31.2	258	30.5	263
rpp	11.87		9.13		9.56	

Table 5: Absolute rendering performance in frames per second and millions of rays per second for three scenes and three camera views per scene. Measured using 1, 2, 6, 12 and 24 rendering threads. The last row shows the average number of rays per pixel. A single primary ray per pixel is used.

In this chapter, we described the Arauna ray tracer, which combines efficient acceleration structure maintenance and ray packet traversal for primary and secondary rays. We proposed an alternative to quadratic fall-off for light sources that bounds their radius of influence. By storing the light sources in a secondary BVH, the lights that affect a surface point are efficiently determined during rendering. For typical scenes, the light BVH can be reconstructed for each frame. We measured the impact of divergence of rays in a packet, which allows us to compare the efficiency of CPU ray tracing and GPU ray tracing, where this impact is much lower (see chapter 6). We proposed a ray packet layout that allows efficient subdivision for secondary ray traversal. The proposed layout maximizes coherence for rays that are traced using the same vector operations. We proposed to interleave texture data and normal map data. A texture fetch will always bring the corresponding normal into L1 cache, which greatly reduces the cost of normal mapping, to the point where this is essentially free. We presented a number of ray traced games, and achieved real-time performance for these on modern CPUs.

The real-time performance of these games indicates that Whitted-style ray tracing has reached the point where it is a viable technique for games. Although Whitted-style ray tracing does not provide a full solution to the rendering equation, it is far less restricted than rasterization: where rasterization has to revert to more or less accurate approximations to global effects such as shadows and reflections, Whitted-style ray tracing supports these elegantly.

The performance level of today's mainstream PCs and the absence of mainstream graphics hardware with support for the ray tracing algorithm necessitates careful optimization of the ray tracing algorithm, both on a high level and on a low level. Efforts in this regard have been very successful over the past decade. Optimizations did lead to a reduction of generality however: where performance levels for coherent primary rays and shadow rays benefited greatly, divergent ray distributions are lagging behind. For this reason, reflections and refractions in Whitted-style ray tracing only recently reached a performance level acceptable for games.

4

SPARSE SAMPLING OF GLOBAL ILLUMINATION

In the previous chapter, we have described the Arauna real-time Whitted-style ray tracer. We have shown that real-time ray tracing has reached a point where it has become a viable option for rendering virtual worlds for games. For the field of games, ray tracing promises an intuitive approach to rendering, making many of the approximations used to augment rasterization unnecessary. Whitted-style ray tracing enables correct visualization of shadows, reflections and refractions; these are hard to do well using rasterization. Distribution ray tracing adds soft shadows and glossy reflections, and retains low algorithmic complexity. With similar low algorithmic complexity, path tracing enables full global illumination.

This elegance comes at a price however. Rendering based on ray tracing is well-known for its high computational cost. Many features that are trivial to implement in a ray tracer require a performance level that is not available on commodity hardware, at least for now. The high cost of secondary effects and the almost linear dependency of rendering time on the total number of rays force us to limit the use of non-diffuse materials and complex illumination. While it is possible to augment Whitted-style ray tracing with coarse approximations of complex secondary effects similar to those used in rasterization-based rendering, we would rather not revert to this approach, as it reintroduces restrictions and code complexity. Instead, we propose to use approaches that converge to the correct solution, so that we can scale up with future advances in hardware technology. One such approximation used for low-frequent shading is the irradiance cache [264]. We propose a variation on this algorithm, which makes it more suitable for real-time rendering in a multi-threaded environment.

We use a three-stage scheme, which decouples shading calculations from actual rendering. In a pre-process step, we build a static point set. Then, prior to rendering each frame, we fill this set with shading information. Finally, during rendering, we query this shading information using interpolation.

The presented scheme is an extension of earlier work [29].

4.1 PREVIOUS WORK

Caching shading data is commonly used to enable real-time rendering of scenes with complex lighting. Even when lighting for the scene is not static, caching can be used to exploit temporal and spatial coherence of the (often low-frequent) shading information.

In the context of rasterization, precalculated shading information is typically linked to scene geometry. An early approach was *vertex shading*, where each vertex of the mesh stores shading information. This shading information is then

reconstructed using bilinear interpolation (Gouraud shading, [95]) over the polygon. Using this scheme, shading calculations are limited to vertex positions, and thus, for most scenes, sparsely sampled. One disadvantage of this approach is that the density of the samples is directly coupled to mesh resolution. Vertex shading was implemented in hardware[1] by SGI for the Nintendo 64 console [218], and used extensively in games such as Super Mario 64 [160] and GoldenEye 007 [107].

Recording shading information in bitmaps allows decoupling sample density from geometric resolution. Techniques that use this approach include shadow maps [266, 199], illumination maps [12] and directional light maps [106]. Since these techniques store samples on a raster, shading reconstruction suffers from aliasing [6].

In the context of ray tracing, mesh-less schemes have been employed. Photon mapping [121] uses the locations of particles that hit scene geometry. The density of the photon map depends on the amount of incident light; areas that receive little light also receive few photons. The irradiance cache [264] exploits the fact that indirect light is typically low-frequent, and calculates samples with a density that adapts to local scene complexity, as well as details in the lighting. Few samples are created for areas with slowly varying illumination, while areas with high-frequent details receive many samples. The samples are created on the fly. In a multi-threaded environment, this leads to excessive synchronization. Neither the photon map nor the irradiance cache target real-time performance.

A recent paper by Lehtinen et al. [148] describes a mesh-less approach using a precalculated point set. Their scheme is designed for GPUs, and relies on a hierarchical point set with a high density, rather than adaptive density. Furthermore, their scheme targets precomputed light transport (PRT, [220]) rather than direct storage of irradiance.

4.2 THE IRRADIANCE CACHE

Since our scheme bears significant resemblance to the irradiance cache, we will discuss this approach in more detail.

The irradiance cache was first used in the Radiance system, which is a physically-based rendering system, aiming to deliver a "reasonably accurate result in a reasonable time"[2], by using a hybrid approach of Monte Carlo and deterministic ray tracing. In this system, the irradiance cache is used to accelerate the calculation of indirect diffuse light. The approach that the irradiance cache implements is based on the premise that indirect light is mostly low frequent, and can thus be sampled sparsely.

The irradiance cache is a mesh-less structure that consists of an octree, used to store irradiance values. Each sample stores the position and normal of a surface point, a range over which the sample is potentially valid, and the actual irradiance

1 Actually, Nintendo's RCP chip which was used in the N64 was programmable on a low level, and vertex shading was implemented in micro code for this chip.

2 From: http://radsite.lbl.gov/radiance/refer/long.html

value. To calculate the irradiance for an arbitrary surface point, nearby samples are selected from the cache and used to calculate an interpolated or extrapolated irradiance value. If insufficient samples are available, a new one is created on the fly.

The range over which a sample is valid is calculated by taking the harmonic mean of the lengths of the rays used for the irradiance estimate:

$$H = (\frac{1}{n}\sum_{i=0}^{n}\frac{1}{d_i})^{-1} \tag{4.1}$$

This links the range of individual samples (and thus sample density) to occlusion.

As other schemes that use interpolation to reconstruct illumination from a sparse set of samples, the irradiance cache is a biased algorithm (see section 2.1.9).

The irradiance cache, in its initial form, suffers from a number of problems:

- The range calculation for a sample involves a stochastic process, which may occasionally result in a density that is too sparse, leading to leaks and other artifacts.
- Adding samples on-the-fly may lead to new samples that would have affected the estimated radiance arriving at pixels that already have been finalized.
- In a multi-threaded environment, adding samples from several rendering threads simultaneously requires extensive synchronization.
- For dynamic light sources, the samples need to be recreated for each frame, leading to temporal noise.

Most of these have been addressed in more recent work. Křivánek et al. propose to use neighbor clamping to prevent leaks [141]. Progressive refinement or an irradiance gathering pass can be used to prevent samples from affecting already rendered pixels [142]. Several methods have been proposed to make the irradiance cache more suitable for a multi-threaded environment [232, 64]. In an off-line environment, temporal noise is solved by refining the irradiance cache until a predefined maximum error is guaranteed to be not exceeded. This illustrates an important design constraint of the irradiance cache: where necessary, it trades rendering time for image fidelity.

In the remainder of this chapter, we present a scheme that is suitable for scenarios where rendering time is the main constraint. Our scheme maintains the benefits of mesh-less schemes. It uses a sampling density that adapts itself to local requirements. In contrast to irradiance caching, we do not create this point set on the fly. Instead, the point set is created once, during a preprocess that estimates optimal positions for point sampling low-frequent shading information. We use a variation of the Poisson-disk process for this, while the density of the points is steered using ambient occlusion [277]. This provides a good estimate of local scene

complexity. The static point set is then used to store shading information. During rendering, the shading data stored in the point set is interpolated to obtain the final shading for any surface point in the scene. Calculating shading information is thus decoupled from actual rendering. Finally, shading reconstruction is done in real-time.

4.3 POINT SET

The reconstruction of a continuous function based on a discrete set of samples is a well-studied problem in computer graphics. Sampling the domain of an unknown continuous function can be done by taking N uniformly distributed random samples. While it is also possible to use evenly spaced samples, this limits us to sample counts of $N = x^{dim}$ (where dim is the dimensionality, and $x \in \mathbb{N}$, $x \geqslant 1$). A deterministic sampling pattern would also result in aliasing artifacts, which are far more objectionable to the human visual system than random noise [273]. There are several ways to obtain a uniform random set of sample points. A particularly good distribution is the Poisson-disk distribution, in which samples are uniformly distributed on the domain of an n-dimensional space, based on a minimum distance criterion between samples. In their 1985 paper, Dippé and Wold propose to use a Poisson-disk distribution for anti-aliasing [66]. In 1986, Cook suggested that the Poisson-disk distribution would be useful for distribution ray tracing [50]. He reverted to jittered sampling however, considering the inefficiency of Poisson-disk generation methods at the time. Hachisuka et al. extended the results of Cook by sampling directly in the multidimensional space of the rendering equation. They show that this reduces the required number of samples [99]. For an extensive review of the Poisson-disk distribution, we refer the reader to [85].

The reversed problem is the representation of a continuous function by a set of discrete samples. An optimal set of samples minimizes both the number of samples and the error after reconstruction. The optimal distribution is non-uniform, unless the represented continuous function is predictable[3]. More specific, samples will be placed on discontinuities, and clustered in high-frequent areas.

For the construction of a point set that will be used to cache irradiance in changing lighting conditions, we face an unknown irradiance function f, of which some characteristics are known a priori. Discontinuities in f are likely to coincide with sharp features in the underlying geometry, while high frequency changes typically occur in areas of high geometric occlusion. The optimal set of samples is thus one that has evenly spaced points on the known discontinuities of f, and a random distribution that adapts to geometric occlusion and surface curvature elsewhere, but is otherwise uniform.

An estimate of geometric occlusion is also used in ambient occlusion [277]. This technique is commonly used in real-time graphics as a rough estimate of

3 Predictable: Constant, when no interpolation is used during reconstruction, e.g. during visualization of a voxel set; constant derivative or constant higher order derivative, when interpolation is used.

global illumination arriving at a point in the scene. Ambient occlusion can be calculated in several ways: one is to estimate the visible area of the sky dome over the hemisphere of the surface. As a more generic alternative, the average distance (typically with a predefined cap) that rays can travel before hitting scene geometry is used.

One final consideration for a precalculated point set is *potential visibility*. In most scenes, there will be closed volumes, unreachable by light particles. By simulating light particles bouncing around the scene, the potentially visible surfaces can be found. This excludes the aforementioned closed volumes, but also the 'outside' of scenes that are not fully enclosed, further reducing the final sample count. Note that this is achieved without any explicit knowledge about solid volumes in the scene.

We combine the described ingredients in the final point set construction:

- In a first step, sample points are evenly distributed over the sharp edges of the scene;
- In a second step, sample points are added to the visible surfaces of the scene, using a Poisson-disk distribution, taking into consideration the points that have already been placed on the discontinuities;
- The radii of the disks are adapted to surface curvature and to local geometric density using ambient occlusion.

The construction of the point set is controlled by two user defined parameters: A minimum and maximum disk radius. The minimum disk radius is also used on discontinuity edges. The maximum disk radius guarantees a minimum sample point density on large open surfaces.

4.3.1 *Points on Sharp Edges*

In 3D modeling software, the polygons of a scene are typically grouped in smoothing groups. In a group, vertex normals are used to calculate a smoothly varying surface normal, creating the illusion of a smooth transition from one polygon to the other. The edges on the boundaries of the smoothing group represent discontinuities. In the absence of smoothing groups, discontinuity edges can be defined as edges between polygons of which the normals differ more than a predefined threshold.

Discontinuity edges can be found efficiently using a winged edge data structure [17], which stores for each edge the polygons connected to that edge. This structure can be constructed in $O(N)$, after which discontinuity edges are also found in $O(N)$. Once discontinuity edges have been found, samples are created on them using the minimal disk radius. Note that samples need to be created on both 'sides' of the edge, i.e. once with the normal of each connected polygon. By placing the sample points on identical positions, correct interpolation is guaranteed.

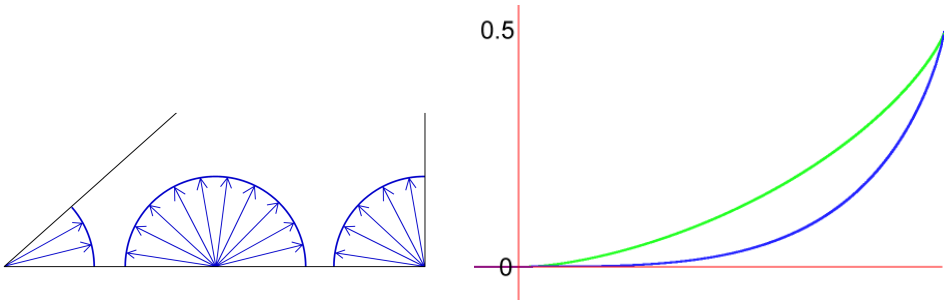


Figure 20: Ambient occlusion and its relation to point density. a) Ambient occlusion is 0 when all rays over the hemisphere can travel a predefined distance. b) Graph of ambient occlusion for a point on a horizontal surface, approaching a vertical wall. Green: ambient occlusion; blue: ambient occlusion squared.

4.3.2 *Dart Throwing*

The Poisson-disk distribution is created by tracing particles. Particles start at one or more points in the scene that are known to be outside closed volumes. Natural locations for these points are the light sources in the scene. Particles are terminated after a predefined number of random bounces, after which a new particle is created at one of the light sources. The result is essentially a *flood fill*. Note that although the process is similar to photon shooting, the particle tracing does not contribute to the shading itself. Therefore, the number of bounces, energy absorption, or a specific distribution over the light sources are irrelevant. Also note that while it is possible to miss scene regions that require many bounces, in practice this is not an issue, since the light received by these regions will be negligible.

Each vertex of the path that the particle travels is considered as a potential new sample position. At the vertex position, the ambient occlusion is determined, and then used to define the radius of a disk. To determine ambient occlusion, a fixed number of rays, over the hemisphere of the surface point, with a predefined length, are cast (see figure 20a). The radius of the disk is then scaled by the square of this value (see figure 20b). Finally, the disc is checked against previously created sample points.

The process is terminated when a predetermined number of subsequent samples overlapped existing sample points.

Calculating ambient occlusion for new samples is a relatively expensive part of the process. To prevent this calculation for samples that will be rejected anyway, we first check if a sample could be inserted with a minimal search radius. If this is not the case, the sample is immediately rejected.

Per sample point we store its position, the surface normal, and a radius. This radius will be used during reconstruction (see paragraph 4.4.8).

The resulting point set is shown in figure 21.

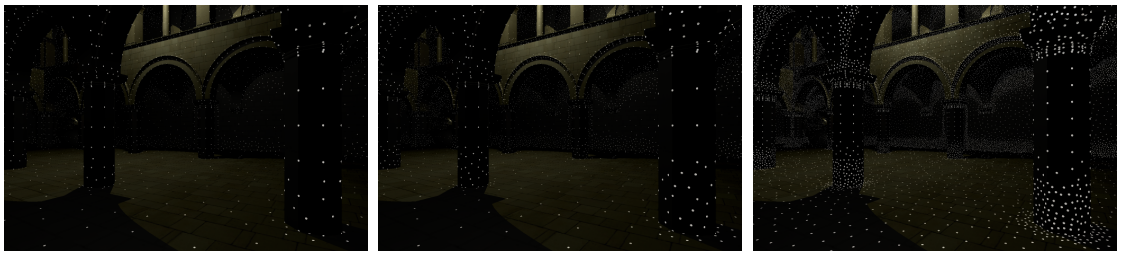


Figure 21: The Sponza Atrium, and the point set for this scene, at three densities: 48k, 105k and 270k points.

4.3.3 *Discussion*

Using the point set to store shading information rather than calculating this shading per pixel has several benefits. First of all, the size of the point set is typically much smaller than the amount of rendered pixels. Besides this, calculating the shading information can now be decoupled from the rendering process. Shading can e.g. be updated incrementally, or in a view dependent manner. One particularly interesting application is to store ambient occlusion in the sample points: by scaling stored values to 90% and sending out 10% new rays per sample point, updating ambient occlusion is effectively sparsely sampled both spatially and temporally.

4.4 SHADING THE POINTS

The point set constructed using the approach outlined in the previous section is primarily useful for storing low-frequent shading information, such as indirect lighting. Since the density of the point set does not adapt itself during rendering, it does not handle sharp shadow boundaries from point lights well. Contact shadows for area lights also pose a problem, unless locally a very high density is used. For diffuse indirect illumination however, even a very small set of points gives plausible results. In the context of real-time graphics for games, it is therefore worthwhile to decouple direct and indirect illumination. In this section, we will discuss the efficient calculation of indirect illumination for static and dynamic light sources. Our implementation adds a single bounce of indirect light, which is sufficient for games [233]. Direct lighting is calculated separately by a Whitted-style ray tracer.

4.4.1 *Previous Work*

In his 1997 paper on *instant radiosity* [132], Keller proposes to render global illumination to keep within real-time limits using hemispherical point light sources, placed on the vertices of light paths obtained by performing a deterministic, quasi-random walk [131]. The outgoing directions of light paths are evenly distributed; the local density of the VPLs is thus proportional to the light received from the light source. The instant radiosity approach is primarily aimed at GPUs, which can efficiently render many point light sources with shadow maps and interleaved sampling [134].

The problem of the weak singularity, commonly present in implementations of this algorithm, was later addressed by Kollig and Keller [140].

Instant radiosity was further improved by Laine et al. to support dynamic light sources, in a real-time context [145]. This is done by updating the set of VPLs generated for the first frame only partially for subsequent frames: VPLs that are no longer visible from the light sources are replaced by new ones, in such a way that the dispersion of the total set is minimized [170, 147]. Despite this, the resulting set is not guaranteed to have an equal distribution over the sphere of the light source. The intensity of individual VPLs is therefore scaled by the area of the hemisphere represented by the VPL.

VPLs have also been used in the context of ray tracing by Wald et al. [251, 252, 244]. In their *instant global illumination* approach, fast ray tracing is combined with instant radiosity, photon mapping, and interleaved sampling to achieve a full global illumination on a cluster of PCs. Due to low sample rates and full reconstruction of the set of VPLs for each frame, their method suffers from temporal low-frequent noise.

The instant global illumination algorithm uses interleaved sampling to reduce the cost of evaluating many lights. Various alternatives have been proposed to solve this problem.

Ward proposes to only test visibility for lights which potential contribution exceeds a predefined threshold [263]. The remaining lights also contribute, but their visibility is estimated using earlier visibility queries for those lights.

Shirley proposes an unbiased method for sampling direct lighting from many area light sources [214], by designing a probability density function over all luminaries.

Paquette uses an octree to store the lights in a scene [183], and approximates their collective contribution at each level of the octree, with a bound on the error of this approximation. The method does not account for occlusion however.

In [105], the many-light problem is formulated as a matrix of light-sample interactions; the sum of the matrix columns is the ideal final image. An approximation of this ideal solution is calculated by sparsely computing full rows and full columns.

Like the instant radiosity algorithm, the *lightcuts* algorithm by Walter et al. [260, 261] discretizes direct illumination and indirect illumination into the summed contribution of many point lights, and approximates the illumination from these point lights at a strongly sub linear cost. The point lights are organized into a binary tree. Each node in the tree contains a single light source that represents the cluster of lights in the subtree. During rendering, clusters are adaptively selected, based on a perceptual metric and conservative bounds on the error in estimating the contribution of a cluster.

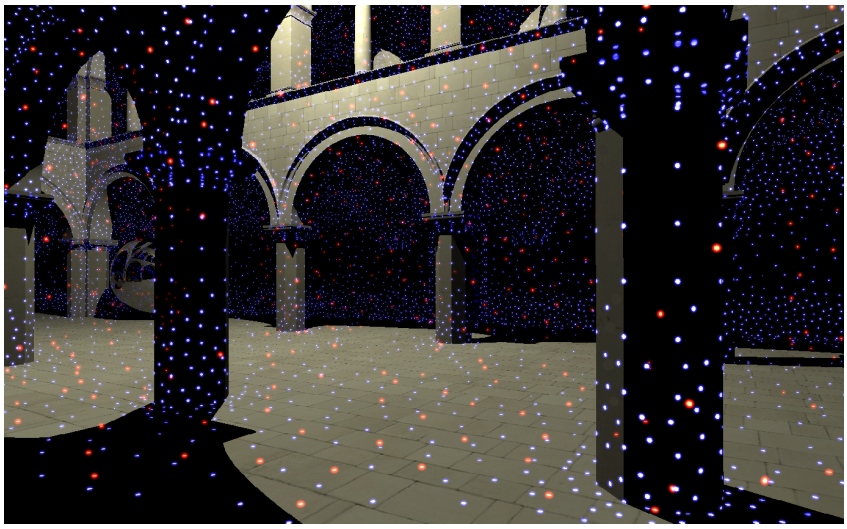


Figure 22: The static set of VPLs for the Sponza Atrium. The yellow dots represent the VPLs. The blue dots represent the sample points. For this image, 8k VPLs where used, and 105k sample points.

4.4.2 *Algorithm Overview*

Within our framework, we use VPLs to add a diffuse bounce to our Whitted-style real-time ray tracer [27], using the point set of which the construction has been detailed in the previous section. Our scheme differs from existing approaches:

- A static set of VPLs is used rather than regenerating or updating the set for each frame;
- Visibility between VPLs and the sample points is precalculated;
- Before rendering a frame, illumination for the sample points is estimated using the lightcuts algorithm;
- During rendering, indirect illumination is reconstructed using the shading stored in the sample points.

The reconstructed indirect illumination is then added to direct illumination calculated by the ray tracer. The scheme runs in real-time.

4.4.3 *Constructing the Set of VPLs*

In the absence of a known set of lights on which the distribution of VPLs can be based, the best distribution is a uniform one. We generate a static set of VPLs using the same dart throwing process used to create the set of sample points. For the VPLs, we do not steer density using ambient occlusion. Also, the density of the set of VPLs is lower than the set of sample points. Typically, for N sample points, N / 10 VPLs are sufficient. The result is shown in figure 22.

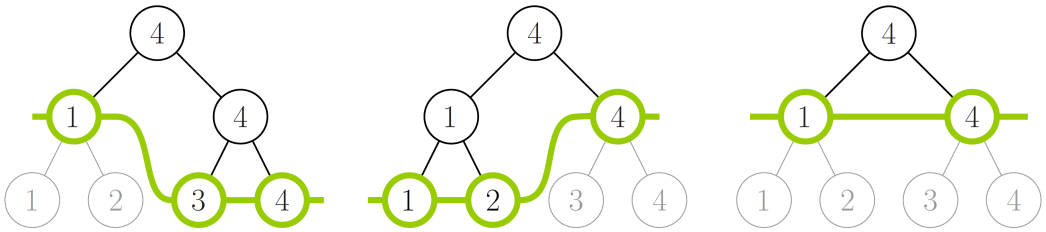


Figure 23: Three cuts through a light tree for four lights. In the first cut, lights 1 and 2 are represented by a cluster. Light 1 is the representative light of the cluster. In the second cut, lights 1 and 2 are used without clustering. Lights 3 and 4 are clustered, and represented by light 4. In the third cut, all lights are represented by clusters. Lights 1 and 4 are not approximated, since they represent the two clusters.

4.4.4 *Shading using the Set of VPLs*

Per frame, the intensity of all VPLs is updated, by evaluating the light sources in the scene. Each VPL covers a certain area of a sphere around the light source. The energy that the light sends to that area is re-emitted by that VPL (scaled with the surface BRDF), as if the surface that the VPL represents was reflecting it. Similar to the distribution in the incremental instant radiosity algorithm by Laine et al., the density of the VPLs is not equally distributed over the sphere of the light sources. When calculating the intensity of each VPL, the contribution of each light source is thus scaled by the area of the sphere that the VPL represents (eq. 4.2).

$$I_p = \sum_{i=1}^{N} L_i G(p,q) V(p,q) dA_q \tag{4.2}$$

The VPLs in turn illuminate the sample points.

4.4.5 *Precalculated Visibility*

The cost of this process is dominated by visibility queries between the VPLs and the sample points. Since both the VPLs and the sample points are static, visibility can be cached. The size of this cache is proportional to the product of the number of VPLs and the number of sample points. The data is however highly coherent; neighboring sample points are likely to "see" a similar set of VPLs. As a result, visibility data compresses well, using e.g. *run-length encoding* [93]. To further reduce the size of the data, we use the following approach. First, we group nearby sample points in *clusters.* For each cluster, a base-vector is stored, containing visibility data for the first sample point in the cluster. For the remaining sample points, a delta vector is constructed, by calculating the XOR between the visibility data of the sample point and the base vector. The delta vector is then compressed using run-length encoding and stored to disk.

4.4.6 *The Lightcuts Algorithm*

We use a modified version of the lightcuts algorithm to reduce the cost of transferring illumination from the VPLs to the sample points. We first summarize the original algorithm, before we describe the modifications.

Given a set of point light sources S, the radiance L caused by their direct illumination at a surface point x viewed from direction ω is a product of each light's material, geometric, visibility and intensity terms summed over all the lights:

$$L_S(x,\omega) = \sum_{i \in S} M_i(x,\omega) G_i(x) V_i(x) L_i \tag{4.3}$$

In the lightcuts algorithm, lights are stored in the leafs of a binary tree. The interior nodes of the tree group these lights into clusters. The root node thus represents all lights in the scene. A cluster is represented by one of the lights in the cluster. Besides this position, the bounds, an orientation bounding cone, and the summed intensity of the lights in the cluster are stored. The tree is constructed in a bottom-up fashion, by grouping pairs of lights or clusters, based on a cluster size metric $\|C\| = I_C({\alpha_C}^2 + c^2(1 - \cos\beta_C)^2)$. The naive approach has an algorithmic complexity of $O(N^3)$; a more efficient approach is described by Walter et al. [262].

Using a cluster rather than the individual lights reduces the number of calculations, but introduces error. The radiance caused by the lights in the cluster is:

$$L_C(x,\omega) = \sum_{i \in C} M_i(x,\omega) G_i(x) V_i(x) L_i \tag{4.4}$$

By using the representative light and the cluster intensity instead, this is approximated as

$$L_C(x,\omega) \approx M_i(x,\omega) G_i(x) V_i(x) \sum_{i \in C} L_i \tag{4.5}$$

The error introduced by this approximation is the difference between equation 4.4 and equation 4.5. This error can be bound by calculating the upper bound $\hat{L}_C$:

$$\hat{L}_C(x,\omega) \approx \hat{M}_i(x,\omega) \hat{G}_i(x) \hat{V}_i(x) \sum_{i \in C} L_i \tag{4.6}$$

It is clear that $0 \leqslant L_C \leqslant \hat{L}_C$, and since both L_C and $\hat{L}_C$ are positive, the error ϵ_C is bound by $0 \leqslant \epsilon_C \leqslant \hat{L}_C$.

The upper bound on the error is thus determined by the upper bounds on the material term, the geometric term, and the visibility term. These are determined individually as follows:

The upper bound on the visibility term $\hat{V_C}$ is zero, if all lights in the cluster are invisible from x, or one otherwise. Since this is hard to bound conservatively, an upper bound of one is used.

The geometry term for oriented (hemispherical) point lights is $G_i = \frac{\max(\overline{\cos}\theta_i, 0)}{\|y_i - x\|^2}$. To compute the bounds of this term, the minimum distance to the cluster for the denominator is determined, as well as the minimum angle between a vector from x and any orientation vector of lights inside the cluster for the numerator. For this, the cluster is first transformed into the tangent space of x. Then, the upper bound for $\overline{\cos}\theta$ over the transformed points p is calculated:

$$\overline{\cos}\theta \leqslant \begin{cases} \frac{\max(p_z)}{\sqrt{\min(p_x^2)+\min(p_y^2)+(\max(p_z^2))^2}}, & \text{if } \max(p_z) \geqslant 0 \\ 0, & \text{otherwise} \end{cases}. \tag{4.7}$$

The material term is calculated as $M_i(x, \omega) = \frac{\max(\overline{\cos}\theta_i, 0)}{\pi}$. The upper bound for this term is calculated similarly to the geometry term.

The bounds on the error of clusters are used to construct *lightcuts*. A lightcut is a set of clusters that represents each light source exactly once (figure 23). The clusters in the cut are refined by replacing them with their child nodes, as long as the maximum error of the lightcut is above a predefined threshold. Refinement of clusters happens in order of contribution to the total error. During each refinement step, a cluster in the lightcut is replaced by its two children.

4.4.7 *Modifications to Lightcuts*

We propose a number of modifications to the original lightcuts algorithm that make it more suitable for our problem.

As described in subsection 4.4.5, we precalculate visibility between sample points and virtual point lights. The precalculated visibility data replaces the expensive visibility tests during cluster refinement. By determining visibility either between one VPL and many sample points, or one sample point and many VPLs, coherent sets of rays are created, which can be efficiently traced using coherent ray packet tracing [249]. This optimization is not possible in the original algorithm, since the result of each visibility test is needed before the decision is made to further refine a cluster. Precalculated visibility also allows us to estimate the upper bound on the visibility term, which is estimated to be one in the original algorithm. With precalculated visibility, the test to see if all lights are invisible from a sample point x is now a feasible one. This allows us to skip entire clusters. It also allows us to refine clusters faster, when exactly one of the children of a cluster is invisible. In that case, replacing the cluster by its visible child is always better, as the error bound on the child node is smaller.

A second modification to the original algorithm is the use of spherical bounds, rather than axis aligned bounding boxes (AABBs). The spherical bounds simplify

the bounding volume transform of the original algorithm that is needed to determine the numerator of the material term and the geometry term. This transform is expensive: it involves transforming all the lights in the cluster to determine an accurate spatial bound for the cluster, or, alternatively, a transform of the vertices of the bounding box, which leads to a larger transformed bounding box, and thus an overestimated upper bound on the error. For spherical bounds, this is reduced to a trivial transform of the center of the bounding sphere.

Using spheres rather than AABBs requires modifications to the calculation of the material term and the geometry term.

To calculate $\hat{M}_C$ for spherical bounds, we distinguish three cases. The first one is when x is inside the sphere: in this case, $\hat{M}_C = 1$. $\hat{M}_C$ is also one if the normal at x intersects the sphere. In all other cases, $\hat{M}_C = \frac{\mathbf{n}\cdot\mathbf{y}}{\|\mathbf{y}\|}$, where $\mathbf{y} = u\mathbf{n} + v\mathbf{c}$, $u = \sqrt{\frac{\mathbf{r}^2(\mathbf{c}\cdot\mathbf{c}-\mathbf{r}^2)}{\mathbf{c}\cdot\mathbf{c}-(\mathbf{n}\cdot\mathbf{c})^2}}$, $v = \frac{\mathbf{c}\cdot\mathbf{c}-r^2-u\mathbf{c}\cdot\mathbf{n}}{\mathbf{c}\cdot\mathbf{c}}$ and $\mathbf{c}$ is the position of x relative to the sphere.

Calculating $\hat{G}_C$ for spherical bounds requires the distance of x to the sphere, which is $\| \mathbf{c} \| - r$.

The total upper bound on the cluster error is summarized in algorithm 4.1.

For our application, the light tree is constructed once, instead of per frame. This requires a final modification to the original algorithm. The light tree is constructed by clustering lights, grouping them by similarity. For the lightcuts algorithm, the following metric is used:

$$\|\mathbb{C}\| = I_C(\alpha_C{}^2 + c^2(1 - \cos\beta_C)^2) \tag{4.8}$$

Considering the unknown light intensities, we use the following metric instead:

$$\|\mathbb{C}\|' = \alpha_C{}^2 + c^2(1 - \cos\beta_C)^2 \tag{4.9}$$

4.4.8 *Reconstruction*

Before the point set can be used for rendering, a structure is assembled for efficiently querying the point set. Reconstruction is done in a real-time context; high performance is therefore paramount. In the irradiance cache algorithm, samples are stored in an octree. Each sample is stored once, in one of the leafs of the octree. The octree adapts itself well to local scene complexity, and ensures compact data storage.

We implemented a similar octree for our point set. Besides this octree, we also implemented a *loose grid* data structure. In this structure, grid cells overlap, and samples are inserted in all grid cells that contain the sample position. One sample thus potentially ends up in multiple grid cells. By adapting the overlap between grid cells to the maximum search radius during reconstruction, this data structure ensures that only a single grid cell is accessed for each query. We found that shading reconstruction using the grid is up to an order of magnitude faster than

Algorithm 4.1 Calculating the error bound for a cluster bounded by a sphere of radius r at position c_{sphere}, with a cone axis m and the cosine of its half-angle $\cos\beta_C$, illuminating surface point $\mathbf{x}$ with surface normal $\mathbf{n}$.

$\mathbf{c} \leftarrow \mathbf{c}_{sphere} - \mathbf{x}$
$a \leftarrow \mathbf{c} \cdot \mathbf{c} - r^2$
if $a \leqslant 0$ **return** 1
// compute geometric upper bound
$c_m \leftarrow \mathbf{c} \cdot \mathbf{m}$
$t_G \leftarrow \mathbf{c} \cdot \mathbf{c} - c_m^2$
$\hat{G}_{num} \leftarrow 1$
if $(c_m > 0 \wedge \mathbf{c} \cdot \mathbf{c} > r^2) \vee (t_G > r^2) \wedge (\cos\beta_C > -1)$
    $u \leftarrow \sqrt{a \cdot r^2 / t_G}$
    $v \leftarrow (a + u \cdot c_m)/\mathbf{c} \cdot \mathbf{c}$
    $\mathbf{y}_G \leftarrow u\mathbf{m} - v\mathbf{c}$
    $c_\phi \leftarrow (u - v \cdot c_m)/ \parallel \mathbf{y}_G \parallel$
    if $(c_\phi < \cos\beta_C)$
        if $(\cos\beta_C > 1 - \epsilon \wedge c_\phi \leqslant 0)$ **return** 0
        $\hat{G}_{num} \leftarrow \overline{\cos}(\arccos\, c_\phi - \beta_C)$
    end
end
$d^2 \leftarrow (\sqrt{\mathbf{c} \cdot \mathbf{c}} - r)^2$
// compute material upper bound
$c_n \leftarrow \mathbf{c} \cdot \mathbf{n}$
$t_M \leftarrow \mathbf{c} \cdot \mathbf{c} - c_n^2$
if $(c_n \leqslant 0 \vee \mathbf{c} \cdot \mathbf{c} \leqslant r^2)$ **return** $\hat{G}_{num}/d^2$
$u \leftarrow \sqrt{a \cdot r^2 / t_M}$
$v \leftarrow (a + u \cdot c_n)/(\mathbf{c} \cdot \mathbf{c})$
$\mathbf{y}_M \leftarrow u\mathbf{n} - v\mathbf{c}$
$c_\theta \leftarrow u - v \cdot c_n$
if $c_\theta \leqslant 0$ **return** 0
$\hat{M} \leftarrow c_\theta / \parallel \mathbf{y}_M \parallel$
return $(\hat{G}_{num} \cdot \hat{M})/d^2$

reconstruction using the octree. Despite its larger memory footprint, it is therefore the preferred structure.

Direct visualization of the shading information stored in the point set by searching for the nearest point yields a Voronoi diagram, with discontinuities between the Voronoi cells. A more visually pleasing result is obtained by adding a random number to the calculated distances to the points. If enough samples are taken, this results in a smoothed Voronoi diagram. Alternatively, the smoothed solution can be calculated directly:

$$\begin{aligned} C_p &= searchradius - distance_p \\ shade &= \frac{\sum_{p=1}^{N} shade_p C_p}{\sum_{p=1}^{N} C_p} \end{aligned} \tag{4.10}$$

where N is the number of points within the search radius, C_p is the contribution of a single point, $shade_p$ is the shading value stored in that point, and $shade$ is the final interpolated result.

This will cause lone points to have a linear gradient from the point itself towards the edge of the search radius. Instead of a linear gradient, a quadratic or cubic gradient can be used, to make the shape follow the Voronoi edges more closely.

Note that this algorithm implements part of the Shepard approximation [213]. The interpolation scheme proposed by Shepard assumes 2D or 3D interpolation in unconstrained 2D or 3D space. In our case, reconstruction takes place on the surfaces of the scene geometry. This puts additional constraints on the set of sample points that can be used to reconstruct shading. We use the following criteria for sample P to reconstruct shading for point X:

1. The contribution C_p of sample P is scaled by the dot product of the surface normal at X and the surface normal at P;

2. P is rejected if its distance to the plane of X exceeds a specified threshold.

The first criterion ensures that samples with a similar normal as X have the greatest influence, while samples on opposing and perpendicular surfaces are not included in the estimate. The second criterion prevents that nearby samples that lie on a different surface are not included, even when normals are equal.

The reconstruction algorithm is shown in algorithm 4.2. An efficient implementation of this algorithm is provided in appendix A.

The cost of reconstruction is similar to the cost of a single shadow ray. This cost is dominated by cache misses. With minimal loss of quality, these can be reduced. When four primary rays (arranged in a 2x2 square) hit the same primitive, it can be assumed that the sample points contributing to the shading of the four surface points are the same. In this case, the grid is queried once, and the obtained data is used for all four rays, taking into account the surface normals for the four primary intersection points. For typical scenes, this reduces the cost of reconstruction by 50%, without visible artifacts.

Algorithm 4.2 The reconstruction algorithm. X is the surface point for which shading is reconstructed, P is a sample point. N_x and N_p are the surface normals at these locations.

$bbox \leftarrow scene.GetBounds()$
$G_{x,y,z} \leftarrow (X - bbox.p1) * gridsize/bbox.extent$
$cell \leftarrow grid[G_x, G_y, G_z]$
$sum \leftarrow 0$
$rgb \leftarrow 0$
for each P **in** cell
    $scale_{angle} \leftarrow max(0, N_X \cdot N_P)$
    $scale_{distance} \leftarrow max(0, P_{radius} - |X - P|)$
    $sum \leftarrow sum + scale_{angle} * scale_{distance}$
    $rgb \leftarrow rgb + Prgb * scale_{angle} * scale_{distance}$
$rgb \leftarrow rgb/sum$

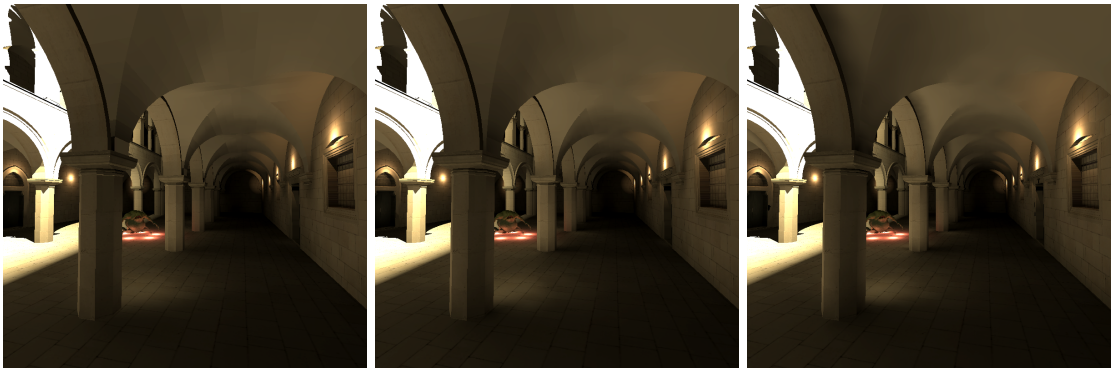


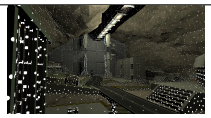
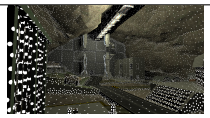
Figure 24: Comparison of image quality for 20k, 40k and 80k sample points.

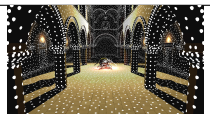
4.5 RESULTS

We implemented the caching scheme in the Arauna real-time ray tracer (see chapter 3). The point set and the grid have been applied to store indirect lighting based on a set of VPLs for a wide range of scenes, three of which are shown in figure 6. All scenes where rendered at a resolution of 1280x720 pixels. The renderer used 24 rendering threads on 12 physical cores of a hyperthreaded dual Intel Xeon platform, running at 3.4 Ghz.

These results show that reconstruction of the indirect illumination adds little overhead to the overall rendering cost. Real-time performance is achieved for the Sponza Atrium and Sibenik Cathedral scenes. The reconstruction cost in the Modern Room scene is relatively high, which leads to somewhat lower frame rates for this scene. Reconstruction speed is mostly independent of sample point count.

Figure 24 shows the relation between pointset density and image quality. At 20k sample points, the pointset is too sparse to find sufficient points on the ceiling. At 40k sample points, this is improved. For 80k samples, a mostly smooth result is obtained.

				
samples/vpls	0/0	40k/1k	80k/1k	80k/2k
ms	41.7	67.8	65.4	65.4
fps	24.0	14.7	15.3	15.3
update	0	69.1	122.9	170.8
fps	24.0	7.3	5.3	4.2

				
samples/vpls	0/0	40k/1k	80k/1k	80k/2k
ms	20.3	30.6	29.5	29.5
fps	49.3	32.7	33.9	33.9
update	0	56.5	107.1	149.7
fps	49.3	11.5	7.3	5.9

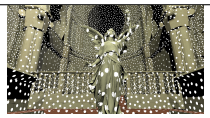
				
samples/vpls	0/0	40k/1k	80k/1k	80k/2k
ms	31.1	39.8	40.4	40.4
fps	32.2	25.1	24.8	24.8
update	0	68.2	133.6	206.6
fps	32.2	9.3	5.7	4.0

Table 6: Absolute rendering performance in milliseconds per frame and frames per second for three scenes, for various combinations of sample point counts and VPL counts. The *ms* rows shows rendering time, including shading reconstruction. The *update* rows show the time spent on updating the shading information, using the VPLs and the lightcuts algorithm. The screenshots show the scene without indirect light, two pointset densities, and the scene with indirect light.

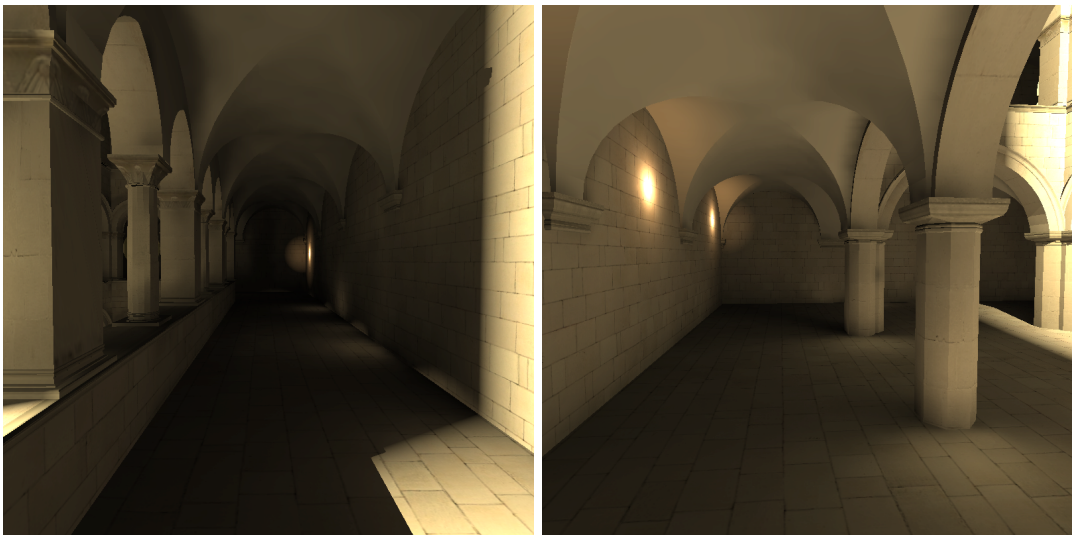


Figure 25: Two typical problematic situations for the pointset. Left image: light leaking from a brightly lit area below the floor. Sample points on the vertical wall below the floor are included in the estimate for the area above the floor. Right image: contact shadows for indirect light near the foot of the pillar appear detached from geometry, due to low sample point density.

The results further show that updating the indirect illumination for each frame using the lightcuts algorithm can only be done at interactive frame rates. Although multiple frames per second can be rendered, a full update of the indirect illumination is currently not feasible in real-time.

4.5.1 *Conclusion*

We presented a caching scheme for indirect illumination. The indirect light is added to the direct light contribution evaluated by a Whitted-style ray tracer. Our implementation achieves interactive frame rates on commodity hardware, at a resolution comparable to what is common in modern games.

As expected, sparse sampling of indirect illumination works well for scenes where changes in indirect lighting are low frequent. For these scenes, relatively small sets of sample points and VPLs are sufficient for visually pleasing results.

Our system is limited to static scenes. Also, since the sample points are stored in a regular grid for rapid reconstruction, for scenes that occupy only a small subset of the cells of such a grid, our scheme may not be efficient. Our scheme has some problems with indirect shadows near occluders, and fails to achieve good results in scenes that are mostly lit indirectly.

4.6 FUTURE WORK

There are a number of topics that would benefit from further research to improve the applicability of the described scheme. One limitation is the lack of support for

dynamic scenes. Another limitation is the preprocessing time of the point set and the visibility data. Adding multiple bounces to the indirect illumination would improve the lighting quality in scenes that are mostly lit indirectly.

4.6.1 *Dynamic Meshes*

The presented approach focuses on static geometry with moving light sources. This can be extended to dynamic geometry in several ways. For rigid motion, sample points can be transformed along with the geometry. This requires an update of the visibility information, but in most cases, visibility will not undergo rapid changes, and calculations can be spread over multiple frames. Due to our choice for the loose grid, removing and reinserting sample points is quite expensive however, and so this method is not suitable for extensive scene changes.

For deformable objects, an alternative approach is practical: In this case, sample points can be generated directly on the vertices of the geometry. Large triangles (if present) can be further subdivided by placing extra sample points along edges and at the center of the triangle. Generating the point set in this deterministic way is efficient, and prevents temporal artifacts. We did however find that this method tends to produce artifacts where the deformable object touches static geometry, as points on the dynamic objects get close to this static geometry, and 'leak light' on it. Finding a robust solution for this, and for unstructured motion, is an interesting direction for future research.

4.6.2 *Point Set Construction*

Generating the point set is currently a time consuming process. The dart throwing requires many ray queries, of which many are in vain, as darts get rejected due to the presence of existing sample points. There are more efficient ways to create Poisson-disk distributions on the surfaces of a scene. We would like to investigate the possibility of decoupling the flood fill from generating points on the surfaces. A limited particle trace could be used to find polygons that can be reached from the light sources, after which the set can be extended by traversing the winged edge structure. Polygon sides marked as visible could then be populated by sample points directly. This would significantly reduce time, and perhaps also enable real-time reconstruction of the point set for dynamic sections of the scene.

Once particle tracing is not directly coupled to Poisson-disk dart throwing anymore, Lloyd's relaxation can be used to further improve the quality of the point set.

4.7 DISCUSSION

To add diffuse indirect light to the Whitted-style Arauna ray tracer, we proposed to use a precalculated set of points on the surfaces of the scene, distributed according

Figure 26: The Sponza Atrium, rendered with specular reflections and a single indirect bounce, sparsely sampled using the pointset.

to a Poisson-Disk distribution, with an adaptive density based on ambient occlusion. The pointset is detailed near occluders, and sparse on large surfaces. We use the pointset to store indirect illumination, which we calculate using the Instant Radiosity approach. The set of VPLs is evaluated using the Lightcuts algorithm, and precalculated visibility between VPLs and the pointset. Although the pointset is static, we proposed several approaches to handle dynamic geometry.

The resulting renderer is able to produce high quality images (see figure 26). This image quality is somewhat deceptive however. The renderer requires significant preprocessing of the (static) geometry. It cannot efficiently handle area lights, unless we store both direct and indirect illumination in the pointset. Glossy materials can only be handled if we update the pointset for each rendered frame. The shading detail is coarse for smaller pointsets, leading to artifacts around sharp polygon edges, and low-quality contact shadows. In practice, this requires manual parameter tuning to achieve acceptable results. The preprocessing, manual tuning and limitations nullify many of the advantages of ray tracing over rasterization.

In the second part of this thesis, we turn to path tracing to alleviate these problems. Where Whitted-style ray tracing offers intuitive and elegant rendering of direct light and specular transport, path tracing provides the same elegance for a broader set of light transport paths, including glossy and diffuse reflection and area lights.

Part II

REAL-TIME PATH TRACING

5

CPU PATH TRACING

[1]In the following chapters, we will investigate the feasibility of real-time path tracing on consumer hardware. In this chapter, as well as the next, we lay the foundation for this.

We first investigate the efficiency of ray queries on the CPU in the context of path tracing, where ray distributions are mostly random. We show that existing schemes that aim to improve the efficiency of ray tracing on the CPU fail to do so beyond the first diffuse bounce. We then present an alternative scheme inspired by the work by Pharr et al. [188] in which we improve data locality by using a data-centric breadth-first approach. We show that our scheme improves on state-of-the-art performance for ray distributions in a path tracer.

5.1 DATA LOCALITY IN RAY TRACING

Over the past decade, ray tracing performance on CPU's has greatly increased[2] due to SIMD-extensions and multi-core architectures. The algorithms that benefitted the most of these advances depend strongly on coherent ray distributions, where rays have similar origins and directions, to work well. In the context of Monte Carlo path tracing, this coherency is mostly unavailable: directional coherency is typically already lost for the first diffuse bounce, and although these rays still have similar ray origins, this is not the case for the second diffuse bounce. Beyond this point, ray distributions are essentially random. This leads to a highly random access pattern of scene data, and, as a consequence, poor utilization of caches and SIMD hardware.

In this chapter, we investigate the work that has been done to improve data locality in the context of ray tracing. We show that the extent to which existing approaches improve data locality is limited in the context of path tracing, and analyze the cause of this. For divergent rays, we propose a traversal scheme that uses breadth-first traversal and batching to improve the performance of a ray packet traversal scheme. Our scheme is based on the work by Pharr et al., which targeted out-of-core rendering. Unlike their system, our scheme targets the top of the memory hierarchy. Our system consistently outperforms single-ray traversal through a multi-branching BVH.

1 This chapter is based on an article to appear in CGForum, "Memory-Coherent Path Tracing" [28].

2 In their 1999 paper [184], Parker et al. report 1M rays per second (for figure 16) on a 24Gflops SGI Origin 2000 system. Boulos et al. achieve 3M for (almost) the same scene [34], on a 2Gflops machine, a 36x improvement.

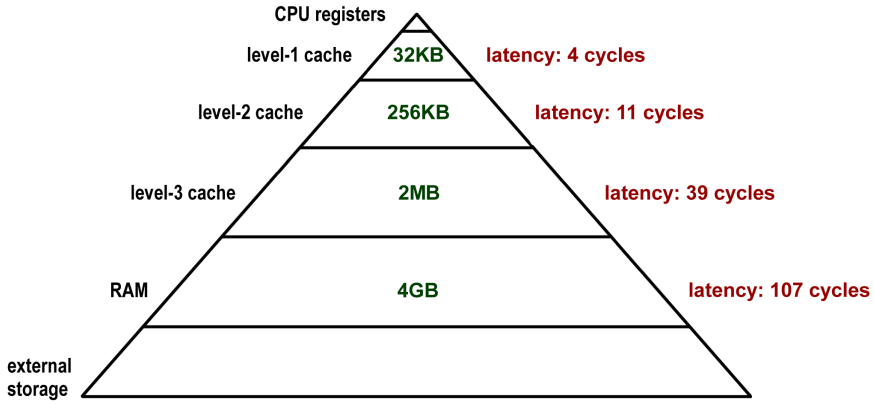


Figure 27: The memory hierarchy: smaller, but faster caches hide the latency of larger but slower memories. Shown cache sizes and latencies are for our test system (Intel Xeon X5670), per core.

5.2 PATH TRACING AND DATA LOCALITY

The game developer and optimization specialist Terje Mathisen once stated that "almost all programming can be viewed as an exercise in caching" [4]. With this remark, he points out the importance of caches on modern computer architectures, especially when algorithms deal with large amounts of data. Considering the vast gap between the rate at which a processor can execute commands, and the rate at which memory can supply data for these commands, caching is usually the optimization with the greatest effect.

Caches are part of the memory hierarchy [40] (see figure 27). Small, but fast caches hide the latency of larger, but slower memories, assuming a certain level of data locality exists [136][3]. Optimal data locality in an algorithm is achieved when the number of times that the same datum is loaded into the caches is one. In other words: all the work that involves a particular datum is carried out, after which the datum will not be accessed again. Note that uniform streaming algorithms, where one kernel is applied to all elements of an input stream, naturally reach this optimum.

For algorithms that perform tree traversal, data locality tends to decrease with tree node depth. While nodes high in the tree are repeatedly accessed for successive queries, deeper levels are often evicted before being accessed again. This is shown in figure 28. The graph shows how the cost of accessing the various levels of the memory hierarchy is distributed over the levels of a 4-wide multi-branching BVH, traversed by the rays of an incoherent ray distribution[4]. The L1 cache is able to handle the majority of the transfers. At deeper levels, more queries fall through to L2, L3 and memory. While L1 is mostly ineffective for the deepest levels, the total

3 In computer science, data locality is subdivided in temporal locality, spatial locality and sequential locality. In modern systems, caches benefit from the first two, while the latter is exploited by instruction prefetching.

4 This data was gathered using a cache simulator, which is described in section 5.4.

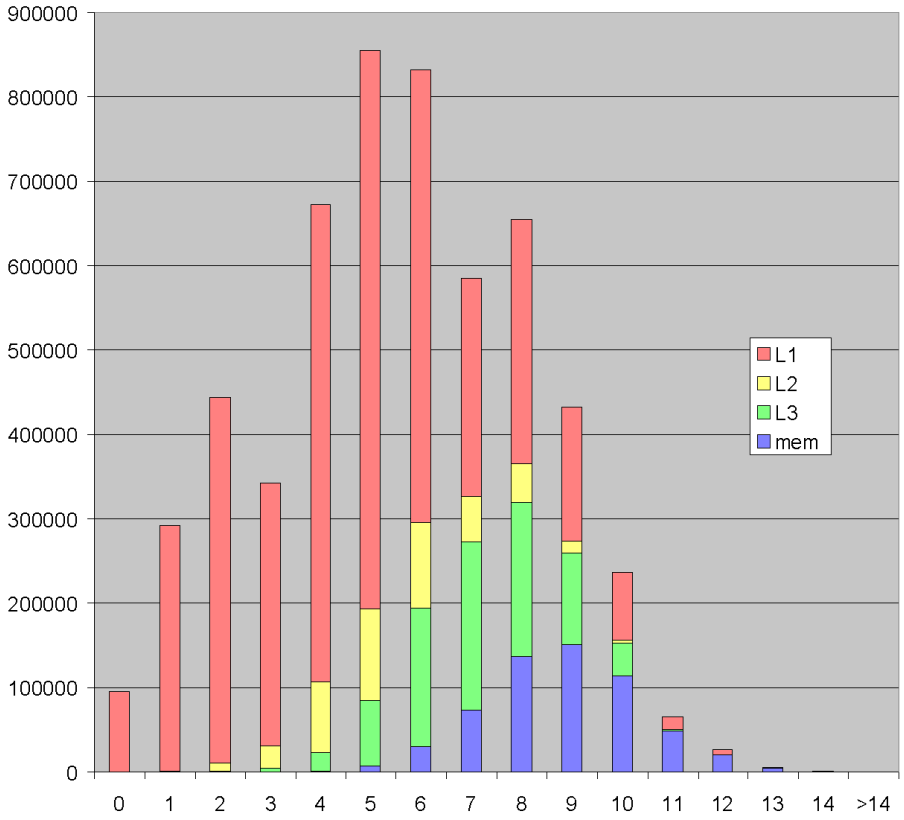


Figure 28: Access of the levels of the memory hierarchy, scaled by access cost (in CPU cycles), for the first 14 levels of an acceleration structure. Measured for single ray traversal, for the Soda Hall scene.

amount of traffic at these levels is small, and contributes little to overall memory access cost.

The overall cost of memory access can be reduced by improving data locality. Better data locality keeps data higher in the cache hierarchy, and reduces the total number of memory transfers, by using the same data for more rays.

5.2.1 *SIMD Efficiency and Data Locality*

Modern CPUs strongly depend on SIMD technology to achieve optimal compute efficiency. SIMD operates on vectors, rather than on scalars. Assuming that multiple streams of data are available for which the same instructions are to be executed, SIMD hardware processes these streams in parallel. The elements of the vectors used to operate on these streams are typically referred to as *lanes*. CPUs operate on four lanes (Intel/SSE [235], AltiVec [65]), eight lanes (Intel/AVX [152]) or sixteen lanes (Intel Many Integrated Core (MIC) [114]/Larrabee [212]). Similar technology on the GPU simultaneously processes 32 lanes [69, 88].

SIMD is effective when all lanes require the same instructions. When this is not the case (e.g. due to conditional code), operations can be masked, or processed sequentially. In both cases, SIMD utilization decreases.

SIMD efficiency is also affected by scatter / gather operations: loading data into vector registers is faster if the required addresses are sequential[5]. At the same time, sequential data access reduces the total number of cache lines that is read from memory, as sequential data typically resides in the same cache line.

Efficiency of the memory hierarchy and SIMD efficiency are tightly coupled: optimizations that aim to improve data locality will often also lead to better SIMD utilization.

5.2.2 *Previous work on Improving Data Locality in Ray Tracing*

Several authors recognize the importance of data locality for the performance of the ray tracing algorithm.

Ray Packets Zwaan, Reinhard and Jansen use ray packet traversal to quickly select object data from the spatial data structure needed for the bundle of rays [240, 201]. By traversing ray packets (referred to as *pyramids* in their papers) rather than single rays, acceleration structure nodes are fetched once for a number of rays. The authors report improved data locality for coherent ray distributions. Wald et al. uses SIMD to traverse a kD-tree with a ray packet containing four rays [249], and achieves interactive frame rates on a cluster of PCs. Smittler et al. propose a custom hardware architecture, SaarCOR [207], that traces packets of 64 rays. They hide the latency of cache misses by swapping between ray packets, using a technique similar to *multi-threading* [186]. Later, the concept of ray packet traversal is generalized to arbitrarily sized ray packets by Reshetov [202] and to other acceleration structures [253, 254].

Reordering Based on the observation that packets of secondary rays often exhibit little coherence, reordering schemes aims to regain coherence by reordering the secondary rays from multiple packets into more coherent sets. Mansson et al. [156] investigated several reordering methods for secondary rays. They aim to create coherent packets of secondary rays by batching and reordering these rays. They conclude that due to the cost of reordering none of the heuristics improves efficiency when compared to secondary ray performance of the Arauna system [27], which does not attempt to reorder secondary rays. Overbeck et al. propose a ray packet traversal scheme that is less sensitive to degrading coherence in a ray packet [178]. Their partition traversal scheme reorders the rays in the packet in-place by swapping inactive rays for active rays and by keeping track of the last active ray. This scheme is less efficient for primary rays, but performs better for secondary rays.

Hybrid schemes Taking into account the inefficiency of ray packets for divergent ray distributions, Benthin et al. proposed a hybrid scheme for the Intel MIC architecture [114] that traces packets until rays diverge, after which it switches to efficient single ray traversal [22].

5 In fact, on many SIMD architectures, this is a requirement; sequential code is used when this requirement is not met.

Breadth-first A typical traversal scheme uses an outer loop that iterates over a set of rays, and an inner loop that processes acceleration structure nodes. Hanrahan proposed to swap these loops [100]. By using the inner loop to iterate over rays rather than objects, access to objects stored on disk is minimized. In their 2007 study, Wald et al. investigated breadth first ray tracing with reordering at every step [255]. They conclude that breadth-first ray tracing reduces the number of acceleration structure nodes that is visited, but also that the high reordering cost may not justify this. Boulos et al. continue this work [35]. In their paper, they show that the performance gains of demand-driven reordering out-weight the overhead. For diffuse bounces, these gains drop below **2x** however. On the GPU, Garanzha and Loop propose breadth-first traversal of rays [86]. Their scheme sorts the set of rays into coherent packets and then performs a breadth-first traversal of a BVH. On the GPU, they claim a 3x improvement over depth-first implementations for soft shadows cast by large area lights.

Batching Several authors propose to use a form of batching to improve data locality. In these schemes, traversal of a single ray is broken up in parts; rays are batched in nodes of the acceleration structure, and advanced when such a node is scheduled for processing. Pharr et al. [188] describe a system, Toro, for out-of-core ray tracing where objects are subdivided using regular grids. Rays are batched in the voxels of a secondary regular grid. This system is discussed in more detail in section 5.3. Budge et al. [41] perform out-of-core rendering on hybrid systems by breaking up data and algorithmic elements into modular components, and queuing tasks until a critical mass of work is reached. Navratil et al. propose a system that actively manages ray and geometry states to provide better cache utilization and lower bandwidth requirements [168]. As in the Toro system, rays in their system progress asynchronously. While Pharr et al. apply ray scheduling at the bottom of the memory hierarchy, Navratil et al. aim to reduce RAM-to-cache data transport. They claim a reduction of RAM-to-L2 cache transport up to a factor 7.8 compared to depth-first packet traversal.

Streaming Breadth-first ray traversal combined with a filtering operation that partitions the set of rays into active and inactive subsets effectively transforms ray traversal into a streaming process, where one traversal step provides the input stream for the next traversal step. Gribble and Ramani [96] propose an approach that during traversal sorts a stream of rays into a set of active rays (rays that intersect the current node) and inactive rays. They implement this on a custom hardware architecture that supports wide SIMD processing. For a stream of rays, their approach bears resemblance to breadth-first ray traversal [100]. Tsakok [237] proposes a streaming scheme, MBVH/RS, which benefits from coherency if this is present, and falls back to efficient single-ray traversal using an multi-branching BVH for divergent rays. For divergent ray tasks on x86/x64 CPUs, this scheme is currently the best performing approach.

5.2.3 *Interactive Rendering*

Interactive ray tracing poses specific challenges for efficient ray tracing schemes. In an interactive context, many schemes exhibit overhead that exceeds the gains. Because of this, schemes developed for out-of-core and offline rendering often do not transfer to interactive rendering.

Of the approaches targeted at improving data locality in ray tracing, ray packets have been by far the most successful for interactive ray tracing. Using ray packets, the cost of data access is amortized over multiple rays. To work efficiently, ray packet traversal schemes require that the rays in the set visit a similar set of acceleration structure nodes. Benthin defines this *traversal coherence* in his Ph.D. thesis as "the ratio between the number of spatial cells traversed by all rays and the sum of cells traversed by any ray" [21]. This ratio is low when rays travel in a similar direction, and have a similar origin. Without this coherence, ray packet traversal schemes fail to improve on naive, depth-first single ray traversal.

Ray packet schemes, which have proven to be successful for interactive rendering of primary rays and shadow rays, show degrading efficiency for secondary rays. Although some authors report reasonable results for interactive Whitted-style ray tracing [178], for path tracing, the overhead of these schemes makes them slower than single ray traversal. Some authors therefore suggest to abandon ray packets altogether [257], and to focus on efficient single ray traversal [60, 77].

To better understand the impact of ray coherence and the overhead of schemes, we have implemented three schemes that target interactive performance, which we compare against base-line performance of single ray traversal. We use the terminology of Overbeck et al. [178] for the naming of masked traversal, ranged traversal and partition traversal. We refer the reader to their paper for a detailed description of these schemes.

SINGLE RAY TRAVERSAL For baseline performance, we chose single-ray, depth-first traversal of a *multi-branching BVH* (MBVH or QBVH [60, 77, 257]), rather than the more commonly used 2-ary BVH. Compared to a 2-ary BVH, the 4-ary BVH performs 1.6 to 2.0 times better [60].

RANGED TRAVERSAL This scheme is based on the packet traversal scheme introduced by Wald et al. (*masked traversal*, [249]), where a node of the acceleration structure is traversed if any ray in the packet intersects it. Ranged traversal improves on masked traversal by storing the first and last active ray in a packet. Rays outside this range are not tested against the nodes of the acceleration structure, reducing the number of ray-AABB tests. Like masked traversal, this scheme performs best for primary rays. For secondary rays, the range may contain a considerable amount of inactive rays, reducing efficiency.

PARTITION TRAVERSAL Designed for secondary rays in a Whitted-style ray tracer, this scheme partitions the rays in the packet in-place by swapping inactive rays for active rays and by keeping track of the last active ray. Compared to

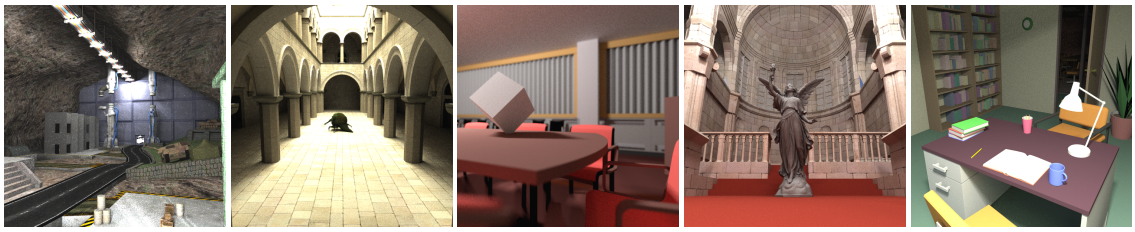


Figure 29: The five scenes used in our experiments: Modern Room from *Let there be Light* (88k triangles), the Sponza Atrium (93k), Conference Room (273k), Sibenik Cathedral with the Lucy statue (603k), and Soda Hall (2.1M). Rendered with up to 1k samples per pixel, and a maximum of 6 diffuse bounces.

ranged traversal, this scheme is less efficient for primary rays, but it performs better for secondary rays, assuming some coherence is still available. Partition traversal operates on groups of N rays (where N is the SIMD width) to reduce overhead.

MBVH/RS Tsakok's Multi-BVH Ray Stream tracing scheme, designed for divergent ray distributions. For each MBVH node, the scheme intersects a list of rays with the four child nodes, generating new lists for each of them. Like other packet traversal schemes, MBVH/RS amortizes the cost of fetching a node over all active rays. Unlike in partition traversal, the generated lists do not contain any inactive rays. This makes MBVH/RS more efficient for divergent ray distributions, where many MBVH nodes are traversed by a small number of rays.

All traversal scheme implementations are hand-tuned for optimal performance.

Table 7 shows performance figures for the five scenes shown in figure 29. Using a 4-ary BVH, baseline performance slowly degrades for each diffuse bounce. For primary rays, ranged traversal outperforms all other traversal methods by a significant margin. After one diffuse bounce, only MBVH/RS outperforms single ray traversal, for some scenes, and by a small margin. After three diffuse bounces, ranged traversal only achieves 17-26% of single ray traversal performance, while partition traversal achieves 27-60% of single ray traversal performance. MBVH/RS achieves between 79% and 106%.

To understand why the traversal schemes perform so poorly for divergent ray distributions, we measured how many active rays visit the nodes of the acceleration structure. Table 8 shows the average number of rays (of the original packet) that intersects each visited node. For a traversal scheme to work well, this number should be high. However, for the Modern Room scene (figure 29a), this number drops rapidly after only one diffuse bounce. Note that this number is an average: even for random ray distributions, all rays will intersect the root node of the BVH (assuming the camera is within the scene bounds). We therefore also measured the average number of rays that intersects the leafs of the acceleration structure. After a few diffuse bounces, this number approaches one. The MBVH/RS algorithm performs better, as it uses a relatively shallow BVH.

scheme	scene	primary	1st	2nd	3rd
Single	Modern	3.032	**1.874**	**1.608**	**1.531**
Partition		6.400	1.157	0.790	0.723
Ranged		**9.102**	0.707	0.448	0.399
MBVH/RS		3.539	1.793	1.421	1.324
Single	Sponza	2.788	1.973	**1.926**	**1.890**
Partition		5.595	1.231	0.941	0.856
Ranged		**7.897**	0.800	0.520	0.476
MBVH/RS		3.424	**2.174**	1.765	1.673
Single	Lucy	2.899	**1.746**	**1.645**	**1.591**
Partition		4.145	0.879	0.636	0.605
Ranged		**5.310**	0.516	0.341	0.313
MBVH/RS		3.180	1.626	1.282	1.251
Single	Conference	3.528	2.141	1.718	1.583
Partition		5.937	1.435	1.102	0.956
Ranged		**6.976**	1.037	0.802	0.715
MBVH/RS		4.959	**2.492**	**1.887**	**1.631**
Single	Soda Hall	3.542	2.787	2.527	2.477
Partition		5.349	1.098	0.755	0.682
Ranged		**8.821**	0.788	0.459	0.430
MBVH/RS		4.151	**3.058**	**2.734**	**2.631**

Table 7: Performance of four traversal schemes, in 10^6 rays per second: single ray traversal through a MBVH, partition traversal, ranged traversal, and MBVH/RS. Measured for five scenes, for primary rays and 1, 2 and 3 diffuse bounces, on a single core of a 3.8 Ghz Intel Xeon processor. Bold figures denote the best performing scheme for each scene and depth.

	Depth	primary	1st	2nd	3rd
Ranged/partition	Interior	94.38	9.12	4.71	3.84
	Leaf	37.88	3.34	1.63	1.30
MBVH/RS	Interior	170.17	13.25	7.01	5.70
	Leaf	97.57	6.26	3.20	2.60

Table 8: Average number of rays per visited leaf / interior node of the BVH, per recursion depth, out of the original 256 rays in a 16x16 ray packet. Measured for the Modern Room scene.

5.2.4 *Discussion*

Ray tracing efficiency for divergent ray distributions is affected by the low average number of rays that is active when visiting the nodes of the acceleration structure. The result of this is that the cost of fetching data from L3 cache and memory is shared by a small number of rays. The low number of active rays also leads to poor SIMD utilization.

The low average number is caused by the packet sizes for which existing schemes perform optimally. Ranged and partition traversal, as well as MBVH/RS, perform best for packets of 64 to 1024 rays [178, 237]. Although larger ray packets would lead to higher active ray counts, in practice this reduces overall efficiency of these schemes.

A scheme that is able to traverse very large ray packets, on the other hand, would exhaust the L1 cache for the ray data alone in the first nodes of the acceleration structure.

An optimal traversal scheme would operate on the same number of rays at each level of the acceleration structure. This requires batching of rays at all levels.

Although batching could improve data locality, it has some disadvantages: the batching itself may introduce considerable overhead. Batched rays must store their full state. For BVH traversal, this includes a traversal stack.

5.3 DATA-PARALLEL RAY TRACING

In the previous section, we have shown that schemes that are designed to improve data locality in ray tracing work well for coherent ray distributions, such as those found in Whitted-style ray tracing, but fail to improve data locality for divergent ray distributions, as found in path tracing. Where Whitted-style ray tracing benefits from a task-centric approach (where a task is a ray query or a ray packet query), path tracing may benefit more from a data-centric approach.

In this section we describe a scheme, RayGrid, which locally batches rays, until enough work is available to amortize the cost of cache misses over many rays. Our scheme is similar to the scheme described by Pharr et al. [188], but targets the top of the memory hierarchy. We analyze the characteristics of this scheme in the context of interactive, in-core rendering.

5.3.1 *Algorithm Overview*

We will first describe the scheme developed by Pharr et al., which was designed for out-of-core rendering in the Toro system. We will then describe our RayGrid system, which borrows from the original scheme and makes it suitable for in-core rendering.

Data structures used in Toro

The Toro system uses a number of data structures. The first is a set of static regular voxel grids, which stores the scene geometry: the *geometry grid*s. One such grid is created per geometric object in the scene[6]. Geometry inside a grid cell is stored sequentially in memory, so that spatial coherence in 3D equals memory coherence. The static grids do not reside in main memory, and are accessed via a caching mechanism that loads and evicts entire grid cells at once. For this to be efficient, the grid cells must store thousands of primitives. This requires a secondary acceleration structure inside each grid cell, for which Pharr et al. propose another regular grid. This grid is referred to as the *acceleration grid*. Finally, rays traverse a third regular grid, the *scheduling grid*.

The data structures are shown in figure 30. For clarity, the acceleration grid has been omitted in this figure.

Batching for Out-of-core Rendering in Toro

Newly created rays are queued in the cells of the scheduling grid. Rays are advanced by processing grid cells from the scheduling grid. When a grid cell is scheduled for processing, each queued ray in it is tested for intersection with the geometry inside each overlapping geometry voxel, and, if no intersection occurred, advanced to the nearest neighboring grid cell, where it awaits further processing.

The system schedules grid cells in the following order: ray queues in the scheduling grid for which all geometry data is cached are processed first. Once these are depleted, the system loads geometry into the cache for the largest ray queue. This way, the cost of loading data into the cache is amortized over as many queued rays as possible.

In this system, processing of an individual ray does not necessarily lead to completion: rays are merely advanced to the next voxel. The implication of this is that ray traversal is asynchronous: the order in which rays arrive is undefined,

6 Primitives that stride voxel boundaries are stored in all grid cells they overlap.

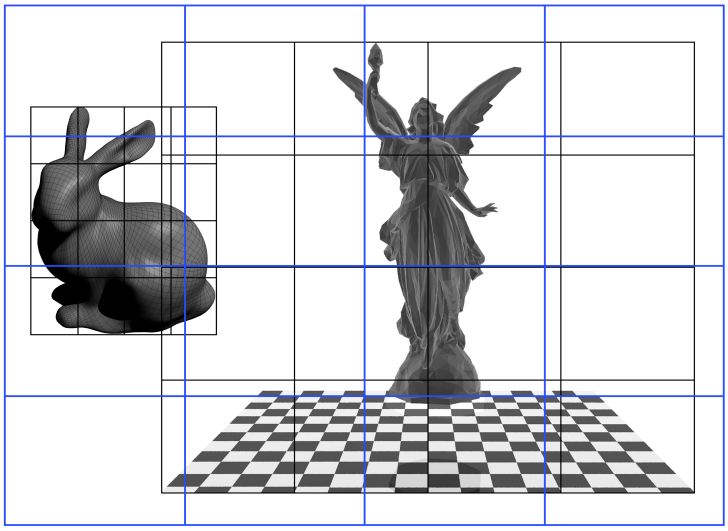


Figure 30: Data structures used in the Toro system: *geometry grids* enclosing two geometric objects (black), and the *scheduling grid*, used for advancing rays through the scene (blue). The cells of the geometry grid typically contain thousands of primitives, which are stored in another grid, the *acceleration grid* (not shown).

and the potential contribution of each individual ray to the final image must be explicitly stored with the ray.

Batching for In-Core Rendering

In the Toro system, the cost of loading geometry into the cache is determined by file I/O, patch tessellation, generating procedural geometry and displacement mapping. Compared to in-core rendering, where caching is used to reduce the cost of RAM to L3/L2/L1 data transfer, these costs are high, and justify considerable overhead. This explains why a similar approach has not been considered for in-core rendering, where overhead can easily nullify the potential gains of a scheme.

Like the Toro system, our RayGrid scheme uses a coarse spatial subdivision for geometry that is used to improve data locality for geometry data. Instead of a regular grid, we use a shallow octree structure, which adapts itself to local scene complexity[7]. We queue rays directly in the voxels of the octree, rather than in a separate structure. Like the Toro system, we store thousands of polygons in octree nodes. To intersect these efficiently, we use an MBVH per octree node.

New rays are added to the system by placing them in ray queues, associated with octree nodes. The system then processes ray queues ordered by size.

Overhead in RayGrid has been reduced by careful data layout and code optimization. The resulting system performs significantly better than existing approaches for divergent rays.

7 This is proposed by Pharr et al., but not implemented in the Toro system.

5.3.2 *Data structures*

This subsection discusses the main data structures used in RayGrid. The scheme uses a hybrid data structure, consisting of a shallow octree, which contain MBVHs for the geometry in each octree leaf.

Octree

Our system uses a shallow octree to subdivide the scene geometry. The octree is extended with spatial bounds and neighbor links, to allow for stack-less ray propagation: rays that leave an octree node through a boundary plane are added to the octree node that the plane links to. If this is an interior node, the ray descents to the leaf node that contains the entry point of the ray. The octree adapts itself to local scene complexity, while the stack-less traversal maintains the benefits of regular grid traversal.

The actual size of an octree node in memory is of little importance for the efficiency of the scheme: compared to overall memory usage for geometry, the octree node size is negligible.

MBVH

An octree node typically stores thousands of primitives. We further subdivide this geometry using an MBVH, which is traversed using the MBVH/RS algorithm. The MBVH is constructed by collapsing a 2-ary BVH, as described by Dammertz et al. [60]. Since each octree node containing geometry has its own MBVH, we refer to these as *mini-MBVHs*.

Ray Queues

Newly added rays are stored in ray queues. A ray queue is a container of a fixed size, which stores rays by value. Ray queues are stored in three linked lists: one for empty ray queues, one for partially filled ray queues, and one for full ray queues. Initially, all ray queues are stored in the empty ray queue list. When a ray is added to an octree node that does not yet contain any rays, the system assigns one ray queue from the empty ray queue list to the octree node, and adds the ray to this list. If the octree node already has a ray queue, the ray is added to that queue. If the queue is full after adding the ray, it is decoupled from the octree node, and added to the list of full queues.

The somewhat elaborate system for storing rays has a number of advantages:

- No run-time memory management is required to store arbitrary amounts of rays per octree node;
- Many processed ray queues are full ray queues. This amortizes the cost of fetching geometry data into the hardware caches over a large number of rays, and leads to efficient traversal using the MBVH/RS algorithm.

A non-empty ray queue stores a pointer to the octree node it belongs to. Multiple full ray queues can belong to the same octree node. Effectively, this allows us to store an arbitrary amount of rays per octree node, divided over zero or more full ray queues, and zero or one partially filled ray queues.

Rays are stored by value in the ray queues, in SoA (*"structure of arrays"*) format[8]. Although ray queues could also store ray indices, this requires one level of indirection, which in practice proves to have a small impact on performance. We also measured the impact of moving a ray from one queue to another in SoA rather than the AoS (*"array of structures"*) format. Despite the less coherent memory writes for the SoA format (each stored float is written to a different cache line), this is not slower in practice. The SoA format does allow us to intersect the rays with the boundary planes of an octree node using SIMD code, which makes this layout the preferred one.

The optimal size for a ray queue is determined by balancing the optimal stream size for the MBVH/RS algorithm (256-1024) and the cost of having many partially filled ray queues. The cost of exchanging a full ray queue for an empty one is negligible.

Ray Data

Our scheme depends on the availability of large numbers of rays to run efficiently. With many rays in flight, the size of a single ray record is important. We store a ray in 48 bytes (11 floats and an integer), by storing the ray origin and direction, the nearest intersection distance, the potential contribution of the ray, and the index of the image pixel the ray contributes to. Since the contribution information is only needed during shading, this data can be stored separately per pixel, which reduces the amount of data copied per ray to 36 bytes (8 floats and an integer).

Since the ray direction is always normalized, it is possible to store only two components of the vector, and derive the third whenever it is needed. Although this reduces the ray record to only 32 bytes, this did not result in improved performance.

5.3.3 *Ray Traversal*

Once all rays have been added into the system, typically a large number of full ray queues is available in the list of full ray queues. Ray traversal then proceeds, starting with full ray queues. Once these are depleted, partially filled ray queues are processed, until no active rays remain in the system. Each processed ray queue is returned to the list of empty ray queues. At the end of the process, this list once again contains all ray queues.

Processing a ray queue consists of two steps:

1. intersection of the rays with the primitives in the octree node, and

8 The queue stores the x-coordinates of all ray origins consecutively in memory, then the y-coordinates, and so on. This data layout is more suitable for SIMD processing than the "array of structures", where full ray records are stored consecutively.

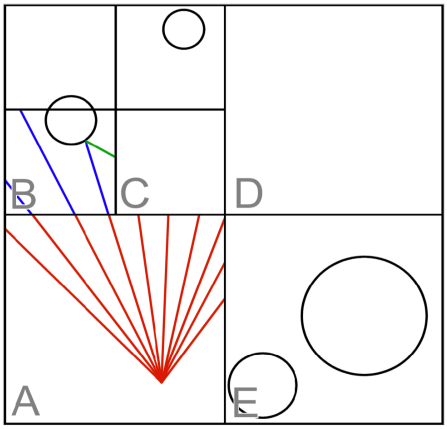


Figure 31: Advancing rays through the octree structure.

2. advancing rays to neighboring nodes, if no intersection was found.

To intersect the rays with geometry, the rays in the queue are converted to the AoS format, suitable for the MBVH/RS algorithm, and then traversed through the mini-MBVH associated with the current octree node. For this, we use an unmodified version of the MBVH/RS algorithm.

To advance a ray to a neighboring node, we first determine the boundary plane through which the ray leaves the current octree node. The neighbor link of this plane then determines the destination for the ray. Since we use an octree, it is possible that the neighbor node is not a leaf node. If this is the case, the ray is recursively added to the child node that contains the ray entry point, until we reach a leaf.

Ray traversal in RayGrid is shown in pseudo-code in algorithm 5.1.

The high-level octree traversal is illustrated in figure 31. Rays are added in octree node A, which contains the camera. Once all rays have been added to the system, the ray queue for node A is processed. Rays propagate from node A to node B. In node B, a new ray is added by the shading code. This secondary ray is advanced together with the primary rays, and arrives in node C. Once node C is scheduled for processing, this ray will be processed together with the primary rays from node A.

In a practical implementation, RayGrid will not handle primary rays: as discussed in section 5.2.3, ray packet traversal is more efficient in this case. Secondary rays however are efficiently handled by our scheme.

5.3.4 *Efficiency Characteristics*

As mentioned in section 5.3.3, full ray queues are always processed first. Doing so may saturate partially filled ray queues belonging to neighboring octree nodes. At some point, there are no full ray queues left, and the system starts processing partially filled ray queues. This reduces the efficiency of the algorithm: the cost of

Algorithm 5.1 Octree traversal in RayGrid. Rays are added to the octree leaf nodes that contain the ray origins. Once all rays have been added, ray queues are processed, starting with full queues, until all rays have terminated.

```
queue_full ← {}
queue_partial ← {}
queue_empty ← allocatequeues()
node_cam ← octree.findleaf(camera.getpos())
for each ray in rays[0..N − 1]
    add( node_cam.getqueue(), ray )
do
    if not queue_full.empty()
        process(queue_full.head())
    else if not queue_partial.empty()
        process(queue_partial.head())
    else break
end

function process( Queue q )
    for each ray in q
        if intersect( ray, q.node.mbvh )
            finalize( ray )
    for each link in neighbourlinks
        for each active ray in q
            if ray.intersect( link.getplane() )
                neighbor ← link.getneighbor()
                if (neighbor.getqueue() = null)
                    neighbor.setqueue( queue_empty.head() )
                    queue_partial.add( neighbor.getqueue() )
                full ← neighbor.getqueue().add( ray )
                if( full )
                    queue_full.add( neighbor.getqueue() )
                    neighbor.setqueue( null )
    queue_empty.add( q )
end function
```

fetching geometry for an octree leaf node is amortized over fewer rays, and the MBVH is traversed with a smaller ray packet.

This tail effect can be reduced by feeding more rays in the system. There are two possible strategies for this:

- By feeding a large amount of rays before processing ray queues, the tail will be relatively short. This increases the memory requirements of the algorithm.
- Alternatively, feeding rays may be coupled to queue processing: by feeding new rays only when no full ray queues are available, memory requirements stay low. This does however require a tight coupling between the code that generates the new rays, and the ray queue scheduler.

In the context of a path tracer, where typically many passes are processed in succession to reduce variance, it is also possible to stop processing ray queues when no full ray queues are available. The partially processed rays remain in the system, and add to subsequent passes. Only when the last pass of the path tracer has completed, the tail needs to be processed. In this scenario, the scheme almost exclusively operates on full ray queues.

5.3.5 *Memory Use*

The requirement to have many rays in flight leads to relatively high memory requirements for the proposed traversal scheme.

The size of a single ray in memory is 36 bytes. As discussed in subsection 5.3.2, rays are stored by value in the ray queues. For a set of N active rays and a queue size of M, a minimum of M/N full queues is required. However, a partially filled queue requires the same amount of memory as a full one. Since each octree leaf node may contain zero or one partially filled queues, the memory required by RayGrid is $(O \cdot M + N) \cdot 36$, where O is the number of octree leaf nodes.

We found a queue size $M = 384$ and a maximum number of primitives per octree leaf of 4096 to be a good choice for most scenes. For the Conference Room, this results in $O = 386$ octree leaf nodes. For $N = 1024^2$ rays, the memory use is 41.09MB.

5.3.6 *Cache Use*

Perhaps equally or more important is the use of cache memory for the algorithm. We measured the average amount of memory that is accessed while processing an octree leaf for the Conference Room scene. For this scene, using the proposed parameters, an octree leaf node contains 818 primitives on average. Processing a leaf involves accessing $818 \cdot 32$ bytes for the primitives, $384 \cdot 36$ bytes for a full queue, and $384 \cdot 36$ bytes in the queues of the neighboring nodes. The 818 primitives are stored in a mini-MBVH of (on average) 175 nodes of 112 bytes each. The total

amount of memory accessed for an octree leaf is thus 73KB on average, which is well below the size of the L2 cache in our system.

5.4 RESULTS

In this section, we discuss the performance of the RayGrid scheme.

5.4.1 *Performance*

We have measured the performance of our scheme. In table 9, we compare the performance of RayGrid against base-line single-ray performance and the original MBVH/RS algorithm.

As can be seen from these figures, our RayGrid scheme consistently outperforms MBVH single ray traversal and MBVH/RS. In many cases this is only by a small margin. However, compared to the MBVH/RS algorithm, the margin is larger.

Table 10 provides more insight in the improved efficiency. Except for primary rays, the MBVH/RS scheme is able to use a significantly larger number of rays in the visited nodes in the RayGrid algorithm.

To gain more insight in the performance characteristics, we gathered several statistics. For all scenes, optimal or near-optimal performance is achieved for a ray queue size of 384 and a maximum number of primitives per octree leaf node of 4096. For the Conference Room scene, these parameters result in an octree of 441 nodes (of which 386 nodes are leaf nodes). The average depth of an octree leaf node is 4.15. On average, octree leafs contain 818 primitives, for which mini-MBVHs are constructed with an average size of 175 nodes and an average leaf depth of 7.0. For comparison, we constructed an MBVH for the same scene, without the shallow octree: the average depth of leaf nodes in this structure is 10.9, which means that for this scene, the octree replaces the first 3.9 levels of the MBVH.

Profiling indicates that in the RayGrid algorithm, 47.3% is spent on octree traversal, versus 26.3% on MBVH traversal, indicating that octree traversal is considerably more expensive than MBVH traversal.

To measure the characteristics of the RayGrid algorithm in terms of caching behavior we implemented a cache simulator. The simulator mimics the cache hierarchy of our test system, with a 32KB L1, 256KB L2 cache, and 2MB L3 cache[9]. The L1 and L2 caches are 8-way set associative, L3 is 16-way set associative. The caches use a pseudo-LRU eviction scheme. We use code instrumentation to record reads and writes.

In table 11, cache behavior of the RayGrid algorithm and MBVH/RS is compared for the Conference Room scene. We estimate the total cost of memory access by

9 Our test system uses a 12MB shared L3 cache for six CPU cores. The simulated 2MB L3 cache is an approximation of the per-core L3 capacity when all cores run the RayGrid algorithm.

scheme	scene	1st	2nd	3rd	4th
Single	Modern	1.874	1.608	1.531	1.497
MBVH/RS		1.793	1.421	1.324	1.275
		-4.3%	*-11.6%*	*-13.5%*	*-14.8%*
RayGrid		2.392	2.136	2.115	2.113
		+27.6%	***+32.9%***	***+38.1%***	***+41.1%***
Single	Sponza	1.973	1.926	1.890	1.869
MBVH/RS		2.174	1.765	1.673	1.636
		+10.2%	*-8.4%*	*-11.5%*	*-12.5%*
RayGrid		2.311	2.325	2.310	2.315
		+17.1%	***+20.7%***	***+22.2%***	***+23.7%***
Single	Lucy	1.746	1.645	1.591	1.472
MBVH/RS		1.626	1.282	1.251	1.122
		-6.9%	*-22.1%*	*-21.4%*	*-23.8%*
RayGrid		1.855	1.840	1.806	1.763
		+6.2%	***+11.9%***	***+13.55%***	***+19.8%***
Single	Conf.	2.141	1.718	1.583	1.467
MBVH/RS		2.492	1.887	1.631	1.499
		+16.4%	*+9.9%*	*+3.0%*	*+2.2%*
RayGrid		2.355	2.055	1.910	1.838
		+10.0%	***+19.6%***	***+20.6%***	***+25.3%***
Single	Soda	2.787	2.527	2.477	2.450
MBVH/RS		3.058	2.734	2.631	2.577
		+9.7%	*+8.2%*	*+6.2%*	*+5.2%*
RayGrid		3.482	3.230	3.188	3.101
		+24.9%	***+27.8%***	***+28.7%***	***+26.6%***

Table 9: Performance of our scheme compared to base-line single-ray MBVH traversal and MBVH/RS. Measured for five scenes, on a single core of a 3.8 Ghz Intel Xeon processor, in 10^6 rays per second.

	Depth	primary	1st	2nd	3rd
RayGrid	Interior	149.50	22.33	14.25	12.51
	Leaf	77.87	12.26	7.39	6.36

Table 10: Average number of rays per visited node of a mini-MBVH in an octree leaf when using the RayGrid algorithm. Measured for the Modern Room scene.

	64	128	256	512	1024	MBVH/RS
L1 read hit	907.1	856.8	760.1	665.2	615.2	1023.0
L2 read hit	10.9	24.7	89.9	158.6	185.1	78.5
L3 read hit	9.7	8.2	6.7	6.5	11.0	31.0
mem read	11.2	11.4	11.7	11.7	11.6	6.6
L1 write hit	428.8	399.2	351.5	341.0	338.2	276.1
L2 write hit	63.0	124.9	279.6	361.5	369.4	111.7
L3 write hit	40.5	33.5	27.7	36.9	95.3	110.2
mem write	41.9	43.7	45.7	43.9	42.1	21.4
cost (est.)	5328.8	5234.6	5536.5	5915.1	6168.0	6872.6
L1%	68.1	65.5	54.9	45.0	39.9	54.1
L2%	2.3	5.2	17.9	29.5	33.0	10.1
L3%	7.1	6.1	4.7	4.3	7.0	12.9
mem%	22.6	23.2	22.5	21.2	20.1	22.9

Table 11: Detailed cache behavior for the Conference Room scene, rendered using the RayGrid algorithm, measured for different batch sizes. Hit counts are in 10^6 hits for a 512x512 image. Estimated cost in 10^9 cycles, assuming 4:11:39:107 cycle latencies for L1:L2:L3:memory access. For comparison, the last column contains figures for the original MBVH/RS algorithm (without octree traversal).

summing L1, L2, L3 and RAM accesses, multiplied by the respective latencies of each level (4, 11, 39, 107 cycles on our test platform).

Compared to the MBVH/RS, the RayGrid algorithm achieves a significant reduction in L3 cache access. Although the number of RAM transfers increased, the (estimated) overall cost of memory access is reduced.

5.5 Conclusion and future work

In this chapter, we have investigated efficient ray tracing in the context of a path tracer.

To improve the performance of CPU path tracing, we proposed a scheme that improves data locality through breadth-first ray traversal. Our scheme is similar to a scheme proposed by Pharr, but rather than reducing the cost of disk I/O, we target the top of the memory hierarchy. Our algorithm improves on state-of-the-art performance for ray distributions beyond the first diffuse bounce in a path tracer, by batching rays in the leafs of a shallow octree. The improved data locality leads to improved L2 cache efficiency and SIMD utilization.

We would like to further investigate the use of batching for improved data locality, perhaps without relying on an octree data structure, which is currently required for efficient leaf-to-leaf ray propagation. Alternatively, an implementation

on an architecture that allows for efficient gather / scatter would allow for more efficient octree traversal, which might even lead to improved L1 cache efficiency.

6

GPU PATH TRACING

In this chapter, we investigate GPU path tracing in the context of real-time rendering for games.

The performance of a game rendering engine is typically expressed in terms of *frames per second* at a certain resolution. In a path tracer, this metric is not very useful, since rendering time in a path tracer is strongly dependent on the number of samples used to estimate the color of each pixel, and thus on the variance in the final image. It makes more sense to express the performance of a path tracer in terms of achievable quality within a certain amount of time. For games, the time budget is game-type dependent, but must be considered to be less than 100ms.

This performance level may seem hard to achieve on today's hardware. However, there are mitigating aspects: the fact that the images are usually animated allows for more variance than stills, and, although physically-based rendering has strong advantages for games and game production, a somewhat loose interpretation of 'physically-based' may be possible: some bias in our algorithms is acceptable, if this leads to less variance in the produced frames. And finally, the production of game scenery is typically a compromise between rendering engine capabilities and limitations, and the desired art style. In an ideal situation, level art emphasizes strong points of a renderer, and hides weak aspects.

In this chapter, we discuss efficient GPU path tracing, which we will combine with CPU path tracing in the next chapter. We start with a discussion of previous work. We then discuss efficiency issues of path tracing on streaming processors, and variance reduction techniques suitable for a small time budget and low sampling rates.

6.1 PREVIOUS WORK

Recently, GPUs have rapidly evolved from fixed-function graphical co-processors to general purpose streaming processors (for an overview, the reader is referred to surveys by Borgo et al. [32] and Owens et al. [180]). The compute power of GPUs makes them an attractive, albeit somewhat unapproachable platform for general computing, including graphics research. Recently, several authors investigated the efficiency of the path tracing algorithm on the GPU. In this section, we first discuss work that deals with the fundamental underlying operation of ray / scene intersection. We then review work that aims to map the path tracing algorithm itself to the GPU. We conclude this section with an overview of the CUDA programming model, which we used for our experiments.

6.1.1 *GPU Ray / Scene Intersection*

Since the introduction of *general purpose graphics processing unit* (GPGPU) programming, many researchers have attempted to implement efficient ray traversal algorithms on the GPU. Of these, Purcell et al. were the first to publish an efficient GPU traversal algorithm [197]. Their algorithm ran in multiple passes and used a uniform grid as underlying spatial structure.

Because uniform grids are not well suited to handle non-uniform geometry, several researchers proposed traversal algorithms using a kD-tree as acceleration structure. Foley and Sugerman proposed two stack-less multi-pass kD-tree traversal algorithms, called kD-restart and kD-backtrack [81]. Horn and Sugerman improved on this work and implemented the kD-restart algorithm as a single-pass algorithm [108]. Popov et al. proposed an alternate stack-less kD-tree traversal algorithm using links between adjacent tree nodes to steer traversal [196].

The introduction of NVidia's GPGPU framework CUDA allowed for the implementation of efficient stack-based GPU traversal algorithms. Günther et al. proposed a packet traversal algorithm, using the BVH as a spatial data structure [97]. In the algorithm, all rays in the packet share a single stack. Similar to CPU-based ray packet traversal, all rays in a packet should have a similar origin and direction for reasonable efficiency. Aila and Laine elaborately studied the SIMT efficiency of stack based BVH traversal on the GPU and improved the efficiency through the use of persistent GPU threads [7]. Their ray traversal kernels represent state-of-the-art. Garanzha and Loop proposed a stack-less breadth-first packet traversal algorithm. The algorithm traverses packet frusta through a BVH to locate ray-leaf intersections. The rays are then tested against all triangles in these leafs [86].

All packet traversal algorithms require packets of reasonably coherent rays in order to achieve high performance [178]. Garanzha proposed to construct coherent ray packets on the GPU by spatially sorting the rays and grouping them into packets [86]. Aila and Laine found that even though their BVH traversal algorithm does not use packets, it still benefits significantly from ray coherence. When the rays traced by different threads in a GPU warp are relatively coherent, SIMT efficiency is increased and the GPU caches become more effective, increasing traversal performance [7].

Several GPU algorithms are developed to construct spatial structures. The most notable of these are the kD-tree construction algorithm by Zhou [276] and the BVH construction algorithm by Lauterbach [146].

6.1.2 *GPU Path Tracing*

Although a lot of research has been dedicated to efficient ray traversal algorithms on the GPU, relatively little research has been targeted at the development of complete unbiased rendering solutions on the GPU. Novák et al. proposed a GPU path tracer which improves SIMT efficiency by regenerating paths that are

stochastically terminated by Russian roulette [171]. We will discuss this approach in more detail in the next section.

Pajot et al. proposed a CPU/GPU hybrid BDPT approach, where the CPU generates camera and light paths and sends them to the GPU [181]. The GPU then connects every camera path with every light path. The large number of connections ensures high GPU utilization. Their algorithm does not target interactive performance, and does not scale well for multiple GPUs.

In his master thesis, Van Antwerpen presents a number of GPU-based unbiased renderers [238]. The core idea is a data-centric system that operates on a stream of samplers. The system switches between two phases: in phase one, the samplers either create a primary ray, or extend an existing path. In phase two, the samplers create explicit rays to the light sources. Ray/scene intersection happens between phases, and the results of these queries are handled in opposing phases: phase one handles the results of the explicit ray queries for the light sources (shadow rays); phase two handles primary and extension rays. The two-phase system is general enough to support BDPT [143, 241] and ERPT [47]. For the basic the path tracing algorithm, it can be simplified by collapsing the two phases. Van Antwerpen refers to this algorithm as *streaming path tracing*. We describe this algorithm in section 6.3.3.

Like Novák et al. and Van Antwerpen, Wald identifies SIMT efficiency as an important factor in GPU path tracing performance [246]. He proposes to increase SIMT efficiency by performing active thread compaction, turning multiple partially filled warps into fewer, but fully-utilized warps. Although the method successfully increases SIMT efficiency, only a modest performance improvement is reported.

Outside academia, several applications implement the path tracing algorithm on the GPU, such as Octane [200], SmallPT [18], TokaSPT [24], SmallLuxGPU [124] and NVidia's Design Garage demo [175], which is based on the OptiX framework [185]. Some of these achieve interactive frame rates for simple scenes and / or low numbers of samples.

6.1.3 *The CUDA Programming Model*

Modern GPUs are stream processors [138, 59]. In its most basic form (*uniform streaming*), a stream processor takes a data stream as input, applies a series of operations (a kernel) to all elements in the data stream and produces an output data stream. Data elements are processed independently, and can thus be processed in arbitrary order, or in parallel. Modern GPUs achieve high throughput by operating on data using many cores: as many as 1600 on recent models [128].

For our experiments, we used the CUDA parallel computing architecture [174], although the concepts presented in this chapter do not use any CUDA-specific features, and apply to other GPGPU architectures as well. This includes OpenCL [139], Microsoft's DirectCompute [157], and ATIs Stream [16].

At the highest level, a CUDA device consists of one or more *Streaming Multiprocessors* (SMs), comparable to CPU cores. Each SM executes blocks of *warps*. Each

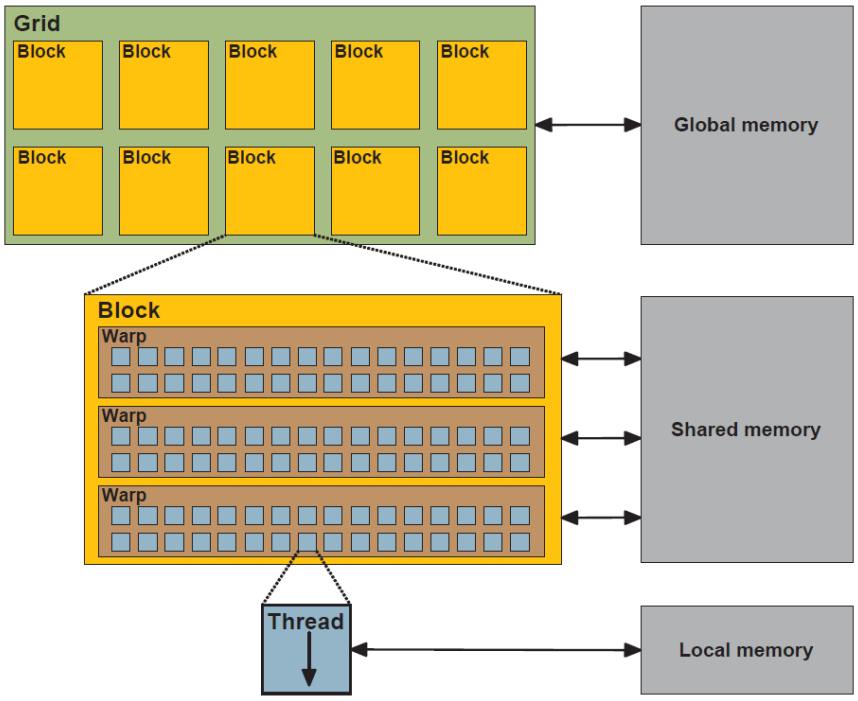


Figure 32: The CUDA thread and memory hierarchy.

warp consists of 32 CUDA threads. The hardware executing one of these threads is referred to as a *CUDA core*. The threads in a warp are executed in lock-step: all threads in the warp must either execute the same instruction, or wait for the next instruction. A single CUDA core thus bears resemblance to a SIMD lane, and an SM bears resemblance to a CPU core executing 32-wide SIMD (*Single Instruction Multiple Data*) instructions. This model is referred to as *SIMT*: Single Instruction Multiple Threads, and can be considered an implicit form of SIMD, handled by the hardware [150].

Like modern CPU cores, an SM can use a form of hyper-threading. Ideally, each SM is assigned more than a single warp. The group of warps (up to 48) assigned to a single SM is referred to as a *block*. The SM can switch between warps in a block. This happens when one warp is stalled, e.g. awaiting a memory transfer. Switching always happens for full warps, enforcing the strict lockstep between threads in each warp. It is important to assign sufficient warps to each SM to allow it to hide memory latencies as much as possible.

The CUDA memory hierarchy matches the thread hierarchy. At the lowest level, each thread can access a number of *registers*, as well as cached, off-chip *local memory*. All threads in a block share on-chip *shared memory*. Finally, all threads on all SMs have access to off-chip, cached *global memory*[1]. The CUDA thread and memory model is illustrated in figure 32.

1 On a hardware level, there is no difference between local memory and global memory: local memory is mapped to global memory by the compiler.

A CUDA device is thus first of all a massively parallel compute device. Application performance strongly depends on the utilization of all processing units, as well as the availability of threads to switch to in case of stalls due to memory latency.

Secondly, at the warp level, the lock-step model requires control flow coherence. Threads in a warp are executed in parallel only if they agree on control flow. If this is not the case, the threads are executed sequentially.

And finally, since potentially many threads are fighting for memory bandwidth, it is far more likely for an application to be memory-bound than to be compute bound. Threads should thus minimize memory access (especially incoherent access) to prevent stalls.

6.2 EFFICIENCY CONSIDERATIONS ON STREAMING PROCESSORS

Compared to the sequential execution paradigm, the streaming paradigm is more restricted, but for suitable algorithms, it allows for very efficient code execution. As discussed in the previous section, an algorithm is suitable for the streaming model if it can be executed as many parallel tasks, that exhibit high flow coherence, and use little memory bandwidth. An important metric for GPU efficiency is *utilization*, which is defined as the average number of threads that is active at any point in time, or: the average number of active threads per warp, scaled by average control flow coherence per warp. As discussed in chapter 5, the path tracing algorithm exhibits poor data locality. In this section, we show that, running on a streaming processor, the path tracing algorithm also exhibits low parallelism. We investigate the impact of data locality and parallelism on GPU path tracing efficiency. In the next section, we discuss several ways to improve this efficiency.

6.2.1 *Divergent Ray Traversal on the GPU*

In chapter 3, we showed that CPU ray tracing performance strongly depends on ray coherence. To better understand the relation between ray coherence and GPU traversal performance, we replicate this test here for the GPU. For four scenes, we chose a camera view, for which primary rays are traced. Per tile of 8^2 pixels, we increase the angle between primary rays by scaling the tile by a factor D, where $D = \{1, 2, 4, 8, 16, 32\}$. We also test the performance of primary rays, created randomly over a hemisphere on the camera plane, but with a common origin. We use a hemisphere to prevent these rays from going backwards, since two of the scenes have a blind wall behind the camera, which would influence the measurements in an undesirable way. And finally, we measure performance for a ray distribution in which rays have a random direction and origin.

The results are shown in table 12.

The results show that, in terms of absolute performance, for coherent primary rays, the GPU is comparable to the CPU. However, the GPU is far less sensitive

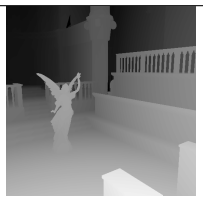
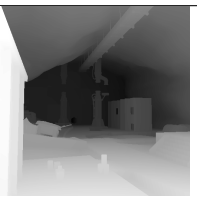
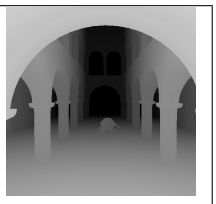
				
D=1	79.9	112.2	109.7	95.6
	100.0%	**100.0%**	**100.0%**	**100.0%**
D=2	79.5	116.5	108.9	95.9
	99.5%	**103.8%**	**99.4%**	**100.3%**
D=4	78.2	116.1	107.2	94.8
	97.9	**103.5**	**97.8**	**99.2**
D=8	76.6	112.2	103.6	92.7
	95.9%	**100.0%**	**94.4%**	**97.0%**
D=16	65.4	97.3	90.7	81.2
	81.9%	**86.7%**	**82.7%**	**85.0%**
D=32	55.8	78.6	76.6	67.8
	69.9%	**70.0%**	**69.9%**	**70.9%**
rnd	51.5	63.1	73.9	64.5
	64.4%	**56.2%**	**67.4%**	**67.5%**
rnd2	35.4	42.8	47.5	40.8
	44.3%	**38.1%**	**43.3%**	**42.7%**

Table 12: Ray traversal performance for primary rays, with varying levels of divergence: prior to traversal, the angle between primary rays for adjacent pixels is scaled by a factor D. The rows labeled 'rnd' are for random rays with a common origin. Rows labeled 'rnd2' are random rays with different origins on scene surfaces. Measured in MRays/s. Figures in bold show relative performance compared to D=1. Measured on a single NVidia GTX470 GPU, in millions of rays per second.

to ray divergence than the CPU: where CPU ray traversal efficiency drops to 30% or less for D=8, at this level of divergence the GPU maintains more than 94% of baseline performance. For D=32, this is still 70%. For random rays with a common origin, performance does not drop below 50%, and for rays with both a random direction and a random origin, performance is still at least 38% of baseline performance. This is a promising result that suggests that the GPU should be a suitable platform for path tracing, where ray coherence typically is very low.

Note that these measurements do not take into account memory bandwidth required for shading.

6.2.2 *Utilization and Path Tracing*

The second factor that determines the efficiency of an algorithm running on a streaming processor is utilization. In this subsection we investigate utilization during the execution of the path tracing algorithm on a streaming processor.

The unbiased path tracing algorithm with Russian roulette is shown in algorithm 6.1. The algorithm aims to find a number of paths that connect the camera to light sources, via zero or more scene surfaces, by performing a random walk. The expected value of the average energy transported via these paths is the solution to the rendering equation (see chapter 2). To improve the efficiency of this process, two extensions are commonly used: Russian roulette is used to reduce the number of paths that transport little energy[2], and at each surface interaction, direct light is explicitly sampled[3].

Utilization during path tracing is illustrated in figure 33. In this figure, two iterations of the while loop that extends a path are shown (lines 4-17 and 18-31). The left half of the image shows the activity of the threads in a warp. *If* statements result in a temporary stall of the threads for which the condition is not true. This happens on line 8: in this example, only a single path encounters a light source, and as a consequence, the SM temporarily works exclusively on this thread. On line 14, some paths encounter a specular surface. For these paths, no explicit connection to a light source is created. The expensive ray / scene intersection that is part of function `sampleDirect` (line 15) is thus executed for only a portion of the threads, while the others stall. Paths that leave the scene, as well as paths that reach a light source, are terminated (line 7 and 11). These threads will remain inactive until control flow reaches line 33. After the first iteration, a number of paths is terminated by Russian roulette (line 17-18). As a consequence, utilization during the second iteration is significantly lower.

Utilization may be further reduced by control flow divergence in acceleration structure traversal code.

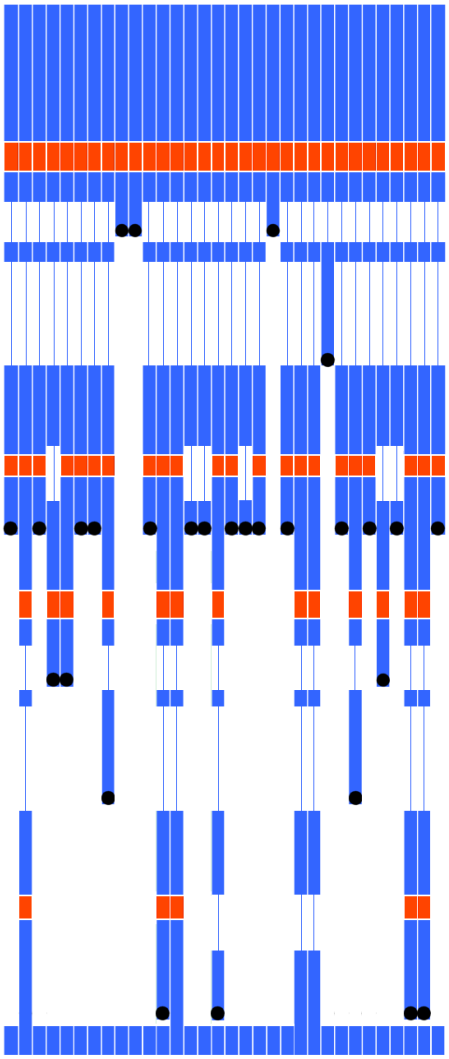
When executed *as-is* on a streaming processor, the path tracing algorithm exhibits poor utilization.

2 This process is explained in more detail in subsection 2.1.6.

3 This approach is discussed in section 2.1.8.

Algorithm 6.1 The path tracing algorithm with Russian roulette and explicit light sampling, in a format suitable for sequential execution. The final image is scaled by $\frac{1}{passes}$.

```
for each pass
    for each pixel
        c_rgb ← 0, scale_rgb ← 1
        hitDiffuse ← false
        D⃗, O ← ray through pixel()
        do
            // find material, intersection point and normal along ray
            m, I, N⃗ ← find nearest(O, D⃗)
            if (is empty(m))
                break      // path left scene
            else if (is light(m))
                if not hitDiffuse c_rgb ← c_rgb + scale_rgb * getEmissive(m)
                break      // path hit light
            else
                O ← I
                if is diffuse(m)
                    c_rgb ← c_rgb + sample direct()
                    hitDiffuse ← true
                D⃗, scale_rgb ← evalBRDF(m, I, D⃗, N⃗)
            p ← RR(m)
            if rnd() < p) break      // russian roulette
            scale_rgb ← scale_rgb * (1 − p)
        while (true)
        pixel[x, y]←pixel[x, y]+c_rgb
```



```
1  c_rgb ← 0, scale_rgb ← 1
2  D⃗, O ← ray through pixel()
3  do
4    // find material, distance and normal along ray
5    m, I, N⃗ ← find nearest(O, D⃗)
6    if (is empty(m))
7      break        // path left scene
8    else if (is light(m))
9      if not hitDiffuse
10       c ← c + scale_rgb * getEmissive(m)
11     break        // path reached light
12   else
13     O ← I
14     if is diffuse(m)
15       c_rgb ← c_rgb + sampleDirect()
16       hitDiffuse ← true
17     D⃗, scale_rgb ← evalBRDF(m, I, D⃗, N⃗)
18   // find material, distance and normal along ray
19   m, I, N⃗ ← find nearest(O, D⃗)
20   if (is empty(m))
21     break        // path left scene
22   else if (is light(m))
23     if not hitDiffuse
24       c ← c + scale_rgb * getEmissive(m)
25     break        // path reached light
26   else
27     O ← I
28     if is diffuse(m)
29       c_rgb ← c_rgb + sampleDirect()
30       hitDiffuse ← true
31     D⃗, scale_rgb ← evalBRDF(m, I, D⃗, N⃗)
32 while (not RR(m))     // Russian roulette
33 pixel[x, y]←pixel[x, y]+c_rgb
```

Figure 33: Utilization of a streaming processor executing the path tracing algorithm. Two iterations of the while loop are shown. Path termination is indicated by a black dot. After one iteration, on average 50% of the paths is terminated by Russian roulette. Additional paths are terminated when a light source is encountered, and when rays leave the scene. Ray / scene intersection is marked in red in this figure, as this typically dominates run-time of the algorithm.

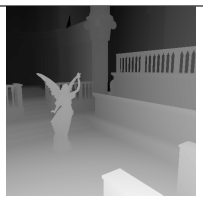
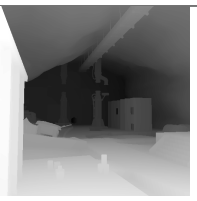
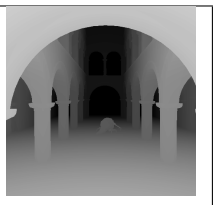
				
O=1	4.8	5.3	5.9	5.4
	1.0x	**1.0x**	**1.0x**	**1.0x**
O=2	7.1	7.9	8.9	7.9
	1.5x	**1.5x**	**1.5x**	**1.5x**
O=4	10.5	11.9	13.5	11.7
	2.2x	**2.2x**	**2.3x**	**2.2x**
O=8	15.7	18.2	20.4	17.6
	3.2x	**3.4x**	**3.5x**	**3.3x**
O=16	23.6	28.1	31.1	26.8
	4.9x	**5.3x**	**5.3x**	**5.0x**
O=32	35.4	42.8	47.5	40.8
	7.3x	**8.1x**	**8.1x**	**7.6x**

Table 13: Ray traversal performance for primary rays, with varying numbers of active threads per warp. For O=32, all threads are active. Numbers in bold show the relative performance compared to a single active thread. Measured on a single NVidia GTX470 GPU, in millions of rays per second.

6.2.3 *Relation between Utilization and Performance*

Although the path tracing algorithm is 'trivially parallel' due to the large number of independent samples that is required for each animation frame, the path tracing algorithm suffers from poor utilization when executed on the streaming multiprocessors of a modern GPU, due to control flow divergence. This notion by itself does not mean that we should focus on improving utilization. Considering that the path tracing algorithm suffers from poor data locality, it is possible that low utilization is (partially) hidden by memory latency. We have therefore tested the impact of low utilization by tracing divergent primary rays using a limited set of active threads per warp. The results are shown in table 13.

When using only a single thread per SM for path tracing, performance lies between 4.8M and 5.9M. In this scenario, a thread will never stall because of memory transfer initiated by another thread. As soon as more threads are executed on the same SM, performance of individual threads is reduced: two threads realize only 1.5x of the performance of a single thread, reducing the performance of individual threads to 75%; when 32 threads are active, per-thread efficiency drops to 25%.

Figure 34: The five scenes used in our experiments: Sponza Atrium, modified by Crytek; Escher; Lucy in the Sibenik Cathedral; Aztec; and MIS Test.

6.2.4 *Discussion*

On recent GPUs, inefficiency due to under-utilization is partially masked by limited memory bandwidth. Equal to CPU path tracing, GPU path tracing requires random access of scene data. Despite this, increasing utilization improves performance, even for very divergent ray distributions: doubling the number of active threads typically yields ~50% more rays per second. Interestingly, these gains hardly decline as the number of active threads increases. This suggests that path tracing should benefit from improved utilization: mostly for the primary rays, but also for path segments beyond the first diffuse bounce.

Another factor that affects GPU ray tracing efficiency is coherence in the ray distribution, albeit to a lesser extent than is the case on the CPU. The lower sensitivity of the GPU to coherence (and thus data locality) is a result of the much higher available memory bandwidth on the GPU: where the CPU used in chapter 5 has a bandwidth of 32GB per second, the GTX470 GPU has a bandwidth of 133.9GB per second.

Our experiments suggests that path tracing performance should benefit from tracing all primary rays together to exploit their coherence. This should however not be done at the expense of GPU utilization.

6.2.5 *Test Scenes*

For the remainder of this chapter, we will use the five test scenes shown in figure 34.

SPONZA ATRIUM, CRYTEK VERSION This model is a modified version of the original model by Marco Dabrovic. Crytek increased the polygon count, and added more colorful materials, to make this scene representative of scenery used in a modern game. The model consists of 270k triangles.

ESCHER The Escher scene is rendered using a camera that hovers above a reflecting sphere. Illumination is provided by many area lights. The scene is heavily occluded, and bounded by reflective planes, which give the scene a recursive, infinite appearance. The model uses a small polygon budget of 10k triangles.

LUCY IN THE SIBENIK CATHEDRAL Also modeled by Marco Dabrovic, the Sibenik Cathedral is a standard benchmark scene for ray tracing. We added an optimized version of the Lucy statue with a glass material, to be able to test complex dielectrics.

AZTEC SCENE Scene from the student game *It's About Time*. This outdoor scene consists of 150k triangles and has very little occlusion.

GLASS HORSE Scene designed to test multiple importance sampling, using large area lights that intersect the walls. A glass chess piece was added to test dielectrics.

In the following sections, we calculate the error of a rendered image by comparing it to a converged image produced using the same technique. We used the *root mean squared error* (rmse) as an error metric, calculated separately over the red, green, and blue components of the image. For an unbiased estimator, the rmse is the square root of the variance (i.e., the standard deviation).

6.3 IMPROVING GPU UTILIZATION

In this section, we discuss three approaches that aim to improve GPU utilization. The first is an existing scheme by Novák et al., which increases utilization, but at the same time mixes primary rays and subsequent path segments, resulting in reduced data locality. The second scheme is a novel scheme, which trades variance for utilization, but does not affect data locality. The third scheme is an existing scheme by Van Antwerpen. This data-centric scheme improves utilization, and does not affect variance nor data locality.

We first discuss the three schemes and their characteristics. In subsection 6.3.4 we evaluate the effect on efficiency of the schemes.

6.3.1 *Path Regeneration*

In their 2011 paper, Novák et al. propose to improve SIMT utilization by replacing terminated paths with new paths [171]. The new paths can either be full paths that start at the camera, or paths that start at the primary intersection point, in which case they help to reduce variance in indirect lighting. Path regeneration is motivated by the observation that threads for which paths have been terminated are idling, which means that any work they carry out instead of idling is essentially for free.

The path regeneration algorithm and its effect on utilization is shown in figure 35. The first iteration of the loop is identical to algorithm 6.1. At the start of each subsequent loop iteration, terminated paths are replaced by new paths (shown in green). Once the requested number of paths for a pixel has been started, no new paths are generated. The active paths continue until all paths either left the scene,

hit a light source, or have been terminated by Russian roulette. The authors refer to this final stage as the *closing phase*. This phase is needed to avoid bias in the final estimate. During this stage, the algorithm behaves as the original path tracing algorithm, and GPU utilization decreases.

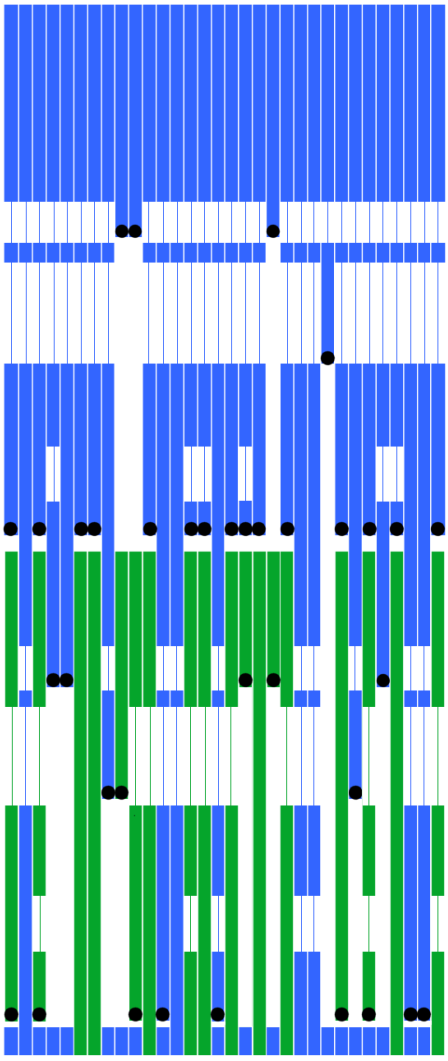
Figure 35 shows that utilization in the path regeneration algorithm is significantly higher than in the reference path tracer. The path regeneration algorithm does not yield full GPU utilization:

- New paths are started at the beginning of a loop iteration. Therefore, threads that terminated early in the previous iteration will idle until this point is reached. One example of this is shown on line 8 of figure 35: a path that leaves the scene will cause the associated thread to be inactive while the other threads execute most of the remaining loop code, including the expensive direct light estimation (which involves a ray / scene intersection). Only paths terminated by Russian roulette are replaced instantly.
- The code for the actual path restart is typically executed for a subset of the threads in a warp. During this process, GPU utilization decreases. For most scenes this is only a minor issue.
- When the number of samples per pixel is low, the cool-down period may take up a considerable portion of the total run time. This is especially true for scenes in which paths consist of many ray segments. During this cool-down period, utilization may be considerably lower.
- When using a predetermined number of samples per pixel, it is still possible for a thread to finish the work well before other threads. This may result in extended low GPU utilization during the cool down period.

The last issue can be evaded if we do not enforce a fixed amount of samples per pixel. Instead, we define a *segment budget*, which we estimate as the product of the desired number of samples per pixel and the average path length. Instead of looping over paths (as in the original algorithm), we now loop over segments. Once the budget is depleted, all threads in the warp simultaneously enter the closing phase.

6.3.2 *Deterministic Path Termination*

Russian roulette is commonly used to focus computation on paths that contribute most to the final estimate. When an integral is approximated through Monte Carlo integration using N samples, we can stochastically remove samples from this set with probability p without introducing bias by multiplying surviving samples by $\frac{1}{p}$. In this process, p can be chosen arbitrarily, as long as $0 < p < 1$. Furthermore, p is typically chosen per sample, based on an estimate of the contribution of the sample. Arvo and Kirk propose to base p on surface albedo (i.e, the probability that an incident particle will be re-radiated after collision [14]). Pharr et al. base



```
1  c_rgb ← 0,  scale_rgb ← 1
2  D⃗, O ← ray through pixel()
3  do
4      // find material, distance and normal along ray
5      m, I, N⃗ ← find nearest(O, D⃗)
6      if (is empty(m))
7          break        // path left scene
8      else if (is light(m))
9          if not hitDiffuse
10             c ← c + scale_rgb * getEmissive(m)
11         break        // path reached light
12     else
13         O ← I
14         if is diffuse(m)
15             c_rgb ← c_rgb + sampleDirect()
16             hitDiffuse ← true
17         D⃗, scale_rgb ← evalBRDF(m, I, D⃗, N⃗)
18     // find material, distance and normal along ray
19     m, I, N⃗ ← find nearest(O, D⃗)
20     if (is empty(m))
21         break        // path left scene
22     else if (is light(m))
23         if not hitDiffuse
24             c ← c + scale_rgb * getEmissive(m)
25         break        // path reached light
26     else
27         O ← I
28         if is diffuse(m)
29             c_rgb ← c_rgb + sampleDirect()
30             hitDiffuse ← true
31         D⃗, scale_rgb ← evalBRDF(m, I, D⃗, N⃗)
32 while (not RR(m))     // Russian roulette
33 pixel[x, y]←pixel[x, y]+c_rgb
```

Figure 35: Utilization of a streaming processor executing the path regeneration algorithm. Two iterations of the while loop are shown. After one iteration, the inactive threads restart paths at the camera or the primary intersection point. During the second iteration of the loop, utilization is significantly higher than in the original algorithm.

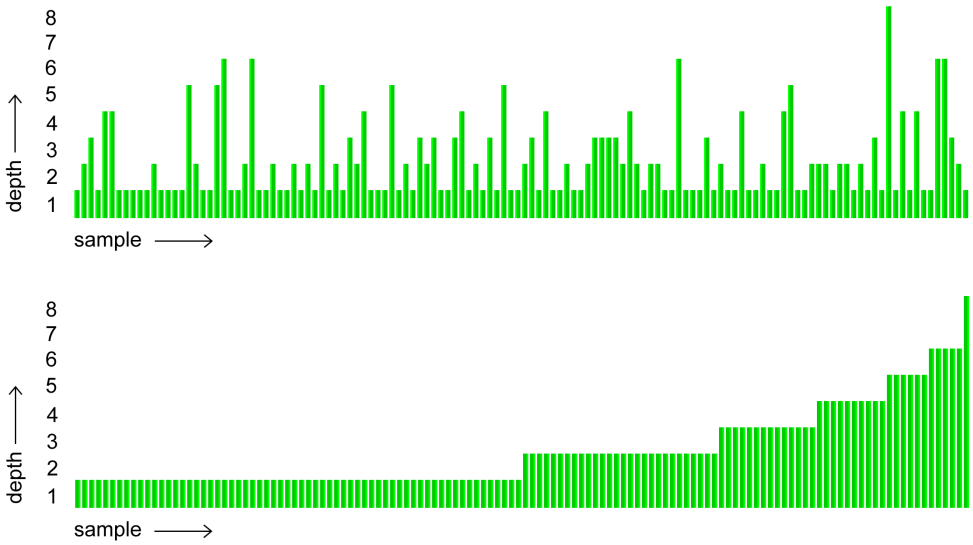


Figure 36: Russian roulette applied to a set of 128 paths, where each path is terminated with a 50% probability at each vertex. The top graph shows the depth reached by each path. The lower graph shows the same paths, sorted by reached depth.

p on the green component of the current path throughput [187]. Keller proposes to use a global termination probability, and states that this is a reasonable choice for most scenes [132]. A global termination probability may however cause infinite variance [231].

Regardless of the termination probability p, reducing the number of samples increases variance. However, removing samples also reduces overall sampling cost. By balancing variance and sampling cost, Russian roulette aims to improve the efficiency of the estimator [241].

In this subsection, we describe an algorithm that relies on a fixed termination probability of 50%.

Consider the following situation: we render a closed scene with diffuse materials only. At each surface interaction, we use Russian roulette to terminate paths with a probability of 50%. In this setup, 100% of the paths will be active for the first path segment. For the second segment, on average 50% of the paths is still active. For N paths, the expected number of active paths drops below one for path segment $\log_2 N$.

This case is illustrated in figure 36. The top graph shows one possible outcome for a set of 128 paths. In this experiment, each path reached a depth between 1 and 7. Note that there is no maximum depth: the probability of at least one path reaching depth N is greater than zero for any $0 < N < \infty$. The bottom graph shows the same samples, sorted by the depth they reached. Approximately 50% of all paths is terminated at the first surface interaction, and 25% at the second (reaching a depth of 2), and so on.

Since there is no dependency between paths, the order in which paths are constructed does not affect the estimate. If it were possible to know beforehand the

number of rays that will reach a certain depth, we could group these rays. For a fixed termination probability, this is possible, if we directly use the expected size of each group, rather than relying on the result of a stochastic experiment. For a fixed termination probability of 50%, for a total of N paths, this will yield a group of N/2 paths that consist of one segment, a group of N/4 paths of two segments, and so on. Using this approximation, N samples per pixel correspond to a maximum recursion depth of $\log_2 N$. When N approaches infinity, the maximum recursion depth approaches infinity as well, removing all bias from the approximation. Until this point, the approximation is biased. The proposed grouping is obtained when we express the maximum depth for a path as a function of the current sample index: $d_{max}(idx) = floor(\log_2(spp - idx)) + 1$. For $spp = 16$, this function yields the following output: 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 4, 5. When the number of samples per pixel is a power of 2, this can be efficiently expressed in CUDA using the `__clz` function, which returns the number of leading zeroes of a binary number: `maxdepth=(__clz((spp-1)-idx)-__clz(spp))+1`.

The reformulated path tracing algorithm with deterministic path termination (DPT), where Russian roulette is replaced by a loop of a predetermined number of iterations, is shown in algorithm 6.2.

Like the path regeneration algorithm, DPT does not yield full utilization. In fact, utilization will be lower: although Russian roulette no longer effects utilization, paths that leave the scene or reach a light source will still terminate. Unlike the path regeneration algorithm, DPT does not replace these paths. DPT has an important benefit over path regeneration: at any point in time, all active threads in a warp will process path segments of the same depth.

The coarse approximation of the Russian roulette termination probability will result in increased variance. Whether the increased utilization outweighs the increase in variance depends on the scene. For scenes with specular surfaces, the 50% termination probability is a poor choice. Scenes that are dominated by diffuse BRDFs generally exhibit little increase in variance.

6.3.3 *Streaming Path Tracing*

Where the previously described algorithms either trade data locality for GPU utilization (section 6.3.1) or variance for GPU utilization (section 6.3.2), the algorithm described in this subsection aims to maintain both. Our implementation is based on the work by Van Antwerpen, and referred to as s *streaming path tracing* (SPT) [239].

The SPT algorithm uses a fixed thread pool to work on a stream of N samples (see figure 37). Initially, the stream is filled with N primary rays (green). During each iteration the rays are advanced, and new path segments are created. The new path segments are stored in two output streams. The first output stream receives explicit light rays that are used to estimate direct light (red). The second output stream receives path extension rays, which sample indirect illumination (blue).

Algorithm 6.2 The path tracing algorithm with deterministic path termination (DPT). The outer loop (over the screen pixels) is not part of the actual GPU kernel; i.e., one kernel execution determines the final color of one pixel.

```
for each pixel
    c_rgb ← 0
    for pass = 0 to passes
        D⃗, O ← ray through pixel()
        scale_rgb ← 1
        maxdepth ← log_2(pass + 1) + 1
        for d = 0 to maxdepth
            // find material, distance and normal along ray
            m, I, N⃗ ← find nearest(O, D⃗)
            if is empty(m)
                break      // path left scene
            else if is light(m))
                c_rgb ← c_rgb + scale_rgb * getEmissive(m)
                break      // path hit light
            else
                O ← I
                if is diffuse(m)
                    c_rgb ← c_rgb + sampleDirect()
                    hitDiffuse ← true
                D⃗, scale_rgb ← evalBRDF(m, I, D⃗, N⃗)
            scale_rgb ← scale_rgb/(1 − 0.5)
    pixel[x, y]←c_rgb/passes
```

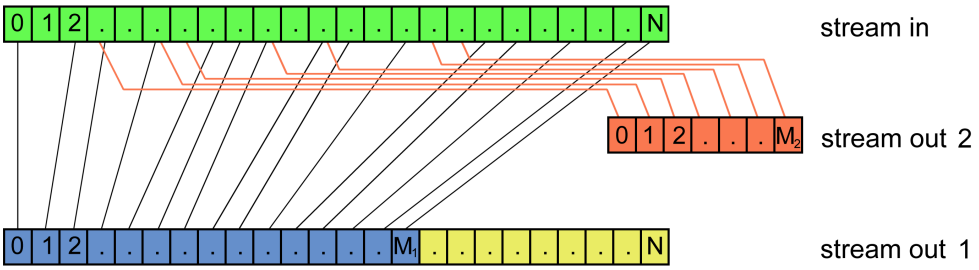


Figure 37: Russian roulette applied to a set of 128 paths, where each path is terminated with a 50% probability at each vertex. The top graph shows the depth reached by each path. The lower graph shows the same paths, sorted by reached depth.

Both output streams are compacted on-the-fly, by storing emitted segments to the first empty slot in the output stream.

When the N threads finish processing the input stream, both output streams contain 0..N path segments. The M_2 rays that were stored in output stream 2 (the direct light rays) are now traced by the first M_2 threads. Output stream 1, which contains M_1 rays, is appended with $N - M_1$ new primary rays (yellow), so that it contains exactly N active samples. These samples are the input for the next loop iteration.

Since the SPT algorithm uses path regeneration, each loop iteration beyond the first processes path segments of different depths. However, because of the compaction, the set of samples is strictly partitioned in sets of path segments of the same depth. When the stream length is sufficiently large, the majority of warps of 32 threads will thus be working on path segments with similar coherence.

Note that the SPT algorithm is data centric. Between processing steps, all data for each sample is explicitly stored to global memory. A path is thus not associated with a single thread, as in the previously described algorithms. This is illustrated in figure 37. Here, N samples are advanced, emitting N new samples. Output stream 1 consists of extended paths (green) and newly generated paths (yellow). Some explicit light rays are also generated (red) and stored in output stream 2. Note that the samples in output stream 1 are implicitly sorted into extended paths and newly generated paths. Also note that no thread will be inactive, until no new samples are left to generate.

6.3.4 *Results*

We have implemented the three algorithms described in this section, and measured their performance. The results are shown in table 14.

The table shows that path regeneration successfully increases the raw ray throughput for most scenes, albeit often by a small margin. For the Lucy scene, the gains are as high as 30% at 32spp. The improved throughput comes at a price however: the error of the produced image at 32spp exceeds the error of the reference path tracer. This happens because the path regeneration algorithm must

method	scene	8spp			16spp			32spp		
		ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays
Ref	Sponza	77	27.39	72.2	153	20.72	71.8	305	15.37	71.6
PR		73	27.79	74.5	154	20.87	70.5	319	15.49	67.8
DPT		66	31.95	77.0	138	24.90	74.9	282	18.90	74.0
SPT		66	23.01	78.6	129	17.68	76.6	255	13.63	78.0
Ref	Escher	123	28.63	75.4	245	22.88	75.2	488	18.07	75.2
PR		78	33.55	83.4	157	27.17	80.4	316	21.69	78.6
DPT		51	38.18	111.1	107	31.44	107.8	220	25.30	105.8
SPT		93	29.35	91.9	182	23.08	93.3	360	18.06	93.7
Ref	Lucy	142	32.04	47.9	281	22.99	47.9	560	16.71	47.9
PR		99	34.73	62.2	194	25.38	62.2	382	18.85	62.3
DPT		70	38.27	80.9	146	28.68	79.1	296	21.66	78.7
SPT		95	34.50	69.5	183	24.55	72.1	360	17.93	73.1
Ref	Aztec	21	12.81	206.9	42	8.63	200.4	82	6.03	202.1
PR		21	14.12	240.1	43	9.07	228.7	88	6.21	220.4
DPT		17	16.20	220.1	34	10.82	212.8	69	7.52	206.1
SPT		21	12.56	204.9	41	8.46	199.4	81	5.91	196.8
Ref	Horse	78	39.88	110.7	154	31.16	110.9	306	24.42	111.1
PR		62	43.47	108.9	380	34.04	34.5	206	26.64	125.8
DPT		29	46.04	195.0	59	36.84	193.1	119	29.20	192.3
SPT		58	41.01	133.1	112	31.64	136.0	220	24.66	138.2

Table 14: Timings in milliseconds, root mean squared error and raw ray throughput (in millions of rays per second) for the reference path tracer (Ref), path regeneration (PR), deterministic path termination (DPT) and streaming path tracing (SPT) for the five test scenes. Performance in millions of rays per second, measured on two GTX470 GPUs.

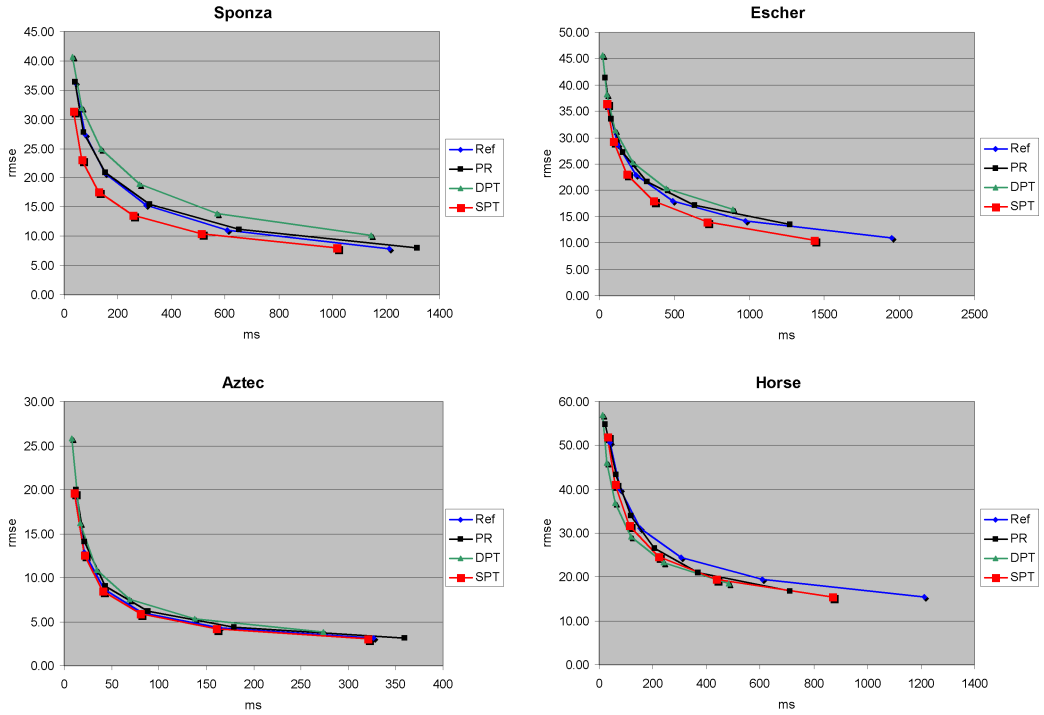


Figure 38: Rendering time versus error for Sponza, Escher, Lucy and Glass Horse, for the reference path tracer (Ref), path regeneration (PR), deterministic path termination (DPT) and streaming path tracing (SPT).

estimate a segment budget based on the requested number of samples per pixel. If this estimate is too low, variance will increase. This is compensated somewhat by the restarted paths, but this does not lead to improved efficiency in all cases.

Figure 38 shows graphs of rendering time versus error for four scenes. The graphs reveal that in fact for none of the four scenes path regeneration improves efficiency.

Deterministic path termination shows a similar problem. The algorithm increases ray tracing performance, in excess of 70% for the Glass Horse scene, but this comes with a significant increase in variance. Still, DPT outperforms the other schemes for this scene. For all other scenes, SPT is the fastest algorithm. Note that for the Aztec scene, none of the algorithms perform significantly better than the reference path tracer. Most paths are very short for this scene, which makes the reference path tracer an efficient algorithm.

Using two standard GPUs, all scenes can be rendered with at least 8spp at 10 fps. For the Aztec scene, we achieve 32spp at a frame rate that exceeds 10fps. For all sample counts, variance for the Aztec scene is also significantly lower than for the other scenes.

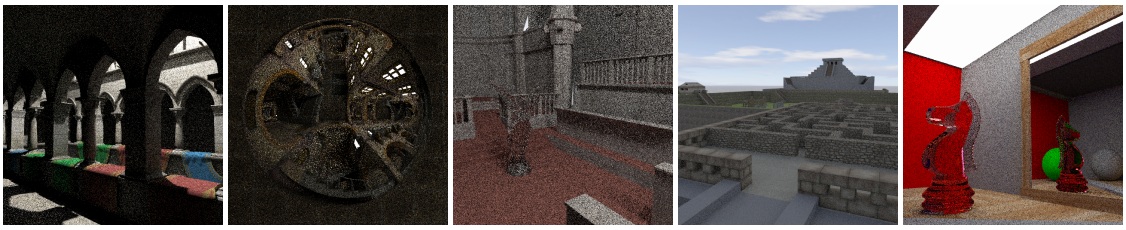


Figure 39: The five test scenes rendered at 10fps on two GTX470 GPUs. At a resolution of 512x512 pixels, all scenes can be rendered with 8spp, resulting in considerable noise. The Aztec scene still runs at 10fps for 32spp. At this point, the generated images are almost converged.

6.4 IMPROVING EFFICIENCY THROUGH VARIANCE REDUCTION

In the previous sections, we sought to improve the efficiency of the path tracing algorithm by improving GPU execution of the algorithm on a low level. In this section, we evaluate the effect of common variance reduction techniques on efficiency, in the context of interactive rendering of game scenes.

6.4.1 *Resampled Importance Sampling*

Resampled Importance Sampling (RIS), as discussed in chapter 2, is a technique that aims to improve the pdf used in importance sampling. The pdf ideally is proportional to the function that we are sampling. According to the rendering equation, direct light reflected by a surface point is the sum of light arriving from all light sources, scaled by the BRDF of the surface material. The light arriving from one light source is the product of its intensity, its projected solid angle, and a visibility factor.

The pdf used to sample light sources could thus be based on the intensity and projected solid angle of the light source and the BRDF of the material. This is not efficient: the pdf must be determined for each surface interaction, and requires evaluation of all light sources. The time complexity of this approach is $O(N)$, where N is the number of lights. If we instead base the pdf on location-invariant data, we can use the same pdf for all surface points. The data that is available for constructing this pdf is limited: we can only use the intensity and the area of the light sources. As a result, the pdf is a crude approximation of the sampled function.

When using RIS for the direct light estimate, we use the location-invariant pdf to make a first selection of light sources. We then calculate the more accurate potential contribution of the lights in this set, taking into account the surface BRDF and projected solid angle of the light source. We then chose one light from the set, with a probability proportional to the potential contribution.

Algorithm 6.3 Efficient implementation of Resampled Importance Sampling.

$L[0..X-1] \leftarrow RandomPointOnLight(lightArray[rnd() * M]) - P$
$cost[0..X-1] \leftarrow BRDF * ProjectedSolidAngle(P, L[0..X-1])$
$r \leftarrow rnd() * \sum cost$
$sum \leftarrow 0$
for $i = 0$ to $X-1$
    $sum \leftarrow sum + cost[i]$
    $if\ (r \leqslant sum)\ return\ L[i]$

6.4.2 *Implementing RIS*

We propose the following efficient implementation of RIS.

Prior to rendering a frame, we construct the pdf based on the location invariant properties of the set of light sources. We use this pdf to fill an array of a fixed size M. For N lights, this array is partitioned in N parts. All array elements in one part point to the same light. Without any scaled probability, each light would thus use M/N array elements. By scaling the parts, we scale the probability of selecting a particular light source. A light can now be selected from this array using a single random variable, using a single array look-up. During direct light estimation, we use the array to fill a small array of X elements with lights, randomly selected according to the precalculated pdf. For each of the X lights, we estimate the contribution, this time taking into account the BRDF of the material and the projected solid angle of the light source. The estimated contributions are used to construct a cdf for the X lights. Finally, the cdf is used to select a single light source. The process is summarized in algorithm 6.3.

Note that the variance of the estimator reaches a minimum when X equals M. In this case, all lights are considered using the accurate contribution estimate. The cost of the algorithm will be equal to $O(N)$. Instead of minimizing variance for a set of samples, we would like to maximize efficiency. In practice, a substantial reduction of variance is obtained for a relatively small value for X. We found that for most scenes, it is worthwhile to always pick two points on each light source. For larger light sources, we get the second sample at a very low cost, because the additional calculations use the same cached data. For small light sources, the extra samples do not improve the pdf, but the wasted calculations are negligible.

6.4.3 *Multiple Importance Sampling*

Multiple Importance Sampling (MIS) uses two rays to improve the direct light estimate (see chapter 2): one is created using a pdf proportional to the surface BRDF; the other proportional to the light pdf. The values of these samples are then weighted using a heuristic, based on the pdfs.

In practical scenes, rays created according to the surface BRDF seldom reach the same light source as rays created by explicit light sampling. In such scenes, MIS

method	scene	8spp			16spp			32spp		
		ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays
Ref	Sponza	77	27.4	72.2	153	20.7	71.8	305	15.4	71.6
RIS		95	18.7	70.4	188	13.9	70.6	374	10.1	70.6
RIS/MIS		98	18.7	71.3	191	13.9	70.9	377	10.1	70.8
Ref	Escher	123	28.6	75.4	245	22.9	75.2	488	18.1	75.2
RIS		149	20.8	70.2	296	15.9	70.1	588	12.1	70.4
RIS/MIS		151	18.2	70.9	298	13.4	70.5	592	9.6	70.4
Ref	Lucy	142	32.0	47.9	281	23.0	47.9	560	16.7	47.9
RIS		169	18.4	50.8	335	13.7	50.9	664	10.3	51.2
RIS/MIS		173	18.3	51.2	338	13.6	51.2	667	10.2	51.3
Ref	Aztec	21	12.8	206.9	42	8.6	200.4	82	6.0	202.1
RIS		28	12.7	179.7	55	8.6	176.7	108	6.0	177.0
RIS/MIS		28	12.7	184.0	55	8.6	179.2	109	6.0	176.8
Ref	Horse	78	39.9	110.7	154	31.2	110.9	306	24.4	111.1
RIS		92	24.3	98.9	183	18.5	98.5	363	14.3	98.8
RIS/MIS		94	16.3	99.1	185	11.3	98.5	365	8.1	98.8

Table 15: Timings in milliseconds, root mean squared error and raw ray throughput (in millions of rays per second) for the reference path tracer, with and without RIS and MIS.

would almost double the number of rays used for direct light sampling. We can prevent this if we reuse the BRDF ray for the indirect light estimate.

6.4.4 *Results*

We have implemented RIS and MIS as modifications to the reference path tracer, and measured the efficiency of the algorithms for the five test scenes. The results are shown in table 15.

6.5 DISCUSSION

In this chapter, we investigated factors that determine GPU path tracing performance, with the aim to maximize efficiency at low sampling rates, where 'efficiency' can be defined as variance reduction over time, and 'low sampling rates' are those that can be achieved in real-time, in the context of a game. We found that utilization (the number of active threads at any point in time) is one such factor. Another factor is ray coherence.

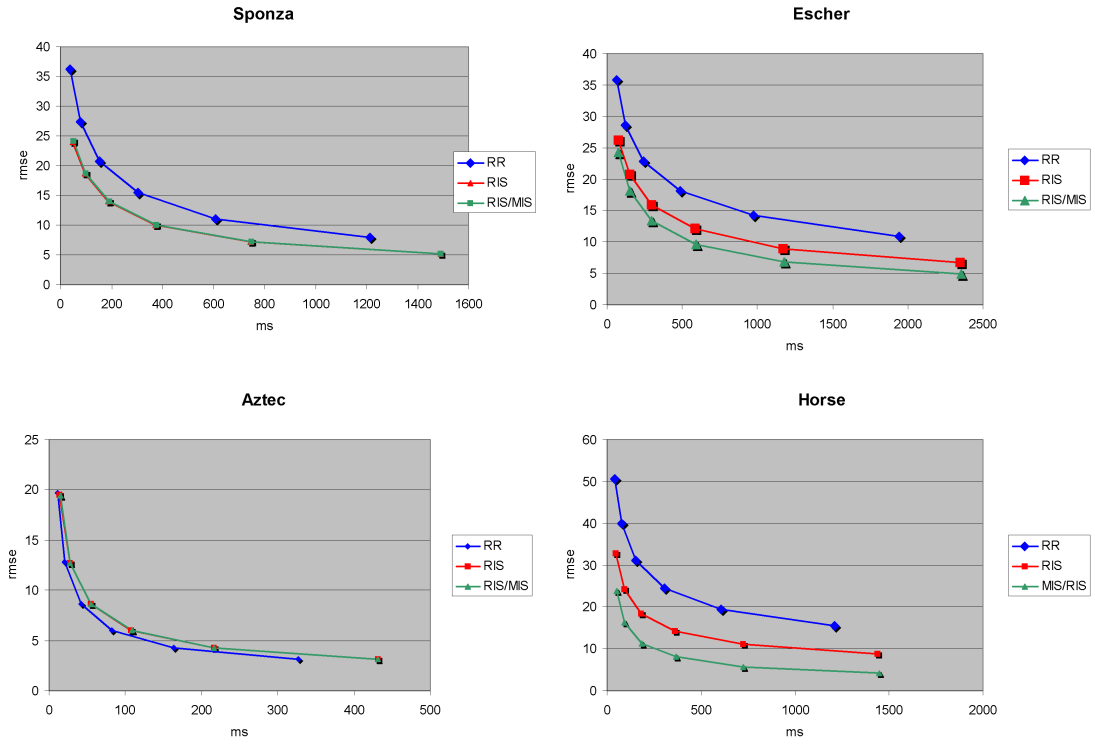


Figure 40: Rendering time versus error for Sponza, Escher, Aztec and Glass Horse, for the reference path tracer, with and without RIS and MIS.

We proposed a novel deterministic path termination scheme (DPT) that improves GPU utilization at the cost of variance, by using a fixed termination probability for Russian Roulette. Besides this scheme, two other algorithms that aim to improve utilization were discussed. Although utilization did increase, in many cases efficiency did not improve: in the case of the path regeneration algorithm, this is caused by reduced coherence in the ray set; for the DPT algorithm, the increase in variance adversely affects efficiency. We found that the SPT algorithm, which does not sacrifice variance nor coherence, performs best for many scenes. In the Glass Horse scene, DPT performs better than SPT. In all cases, the improvement over the reference path tracer is small: on average, an improvement of only 20% is achieved.

At low sample rates, variance reduction techniques such as RIS and MIS further improve efficiency. Especially MIS, which requires extra computation but not extra data transfer, is suitable for GPU path tracing. RIS and MIS do not significantly improve efficiency for the outdoor Aztec scenes, where global illumination is dominated by direct light, and paths are short.

During our tests, we found that certain scenes perform significantly better than others, both in terms of ray throughput and variance. The outdoor Aztec scene exhibits little variance, even at low sampling rates, and can be rendered in real-time on modern consumer hardware. The reason for this is the low occlusion between the scenery and the light source, and the low average path length. Game design

should take this into account, by focusing on scenes with large light sources, and little occlusion.

In the next chapter, we will use the GPU path tracing building blocks discussed in this chapter to build a physically-based renderer for games.

7

THE BRIGADE RENDERER

In this chapter, we present the Brigade renderer, which uses the path tracing algorithm to produce images for real-time games.

The typical target platform for a modern game is a heterogeneous system, consisting of a multi-core CPU and one or more GPUs. Considering the computational demands of the path tracing algorithm, resources should ideally be used to their full potential.

The three main subtasks in a path tracer are acceleration structure maintenance, ray traversal, and shading. When these tasks are not executed by a single processor (either CPU or GPU), the synchronization of data between the processors becomes a fourth significant task. Execution efficiency now depends on two factors. The first is specialization: subtasks should be assigned to the processor that executes them in the least amount of time. The second is system utilization: execution efficiency can only be optimal when none of the processors is idling at any time during rendering.

The architecture described in this chapter assigns acceleration structure maintenance to the CPU cores, while one or more GPUs trace paths and perform shading. By double-buffering the acceleration structure, data synchronization can be executed in parallel to the GPU tasks, and the time between GPU tasks is minimized.

Although this work division does employ both the CPU and GPU, by itself this does not guarantee full utilization of the system. Full utilization is achieved when we implement acceleration structure maintenance and path tracing on both processors, and allow each processor to use idle time to contribute to the task of the other processor. This comes at the expense of a considerable engineering effort, which may be worthwhile only if the amount of idle time is significant.

In the following sections, we describe the architecture of the Brigade renderer. We discuss workload balancing, as well as approaches to maintain a real-time frame rate. At the end of this chapter, we describe two games that have been created using Brigade.

7.1 BACKGROUND

Historically, games have been an important driving force in the advance of graphics hardware and rendering algorithms. Effort has evolved from striving for abstract, visually pleasing results, to more plausible realistic rendering. In the former, a distinct visual style is chosen, which does not necessarily require realism. Instead, *over the top* animation styles and matching graphics are used. Examples of this

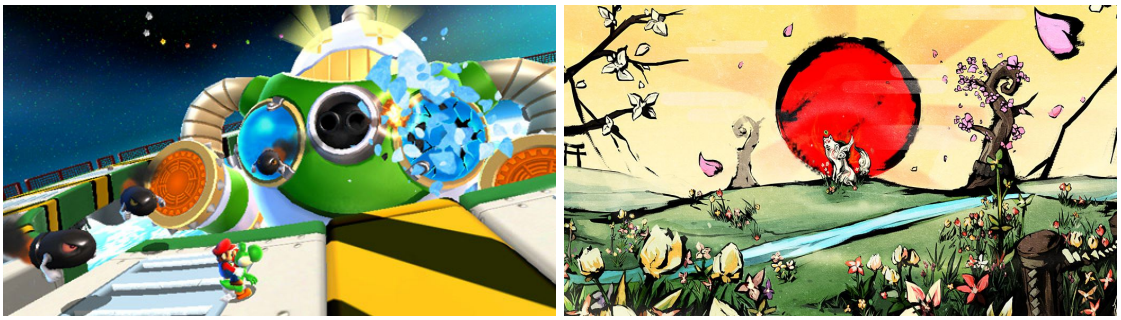


Figure 41: Two examples of modern games that use a non-realistic, distinct visual style. Left: Super Mario Galaxy, right: Okami.

Figure 42: Two examples of modern games that aim for a high level of realism. Left: Tom Clancy's H.A.W.X., right: Gran Tourismo 5.

approach are most early 2D computer games, as well as more recent titles such as Super Mario Galaxy [161] and Okami [127] (figure 41).

Many modern games strive for realistic graphics, where the goal is to convince the player that the result is (or could be) realistic. Examples are racing games such as the Gran Tourismo series [272] and flight simulators such as Tom Clancy's H.A.W.X. [219], which augment the rasterization algorithm with various algorithms to add secondary effects such as shadows, reflections and indirect illumination (figure 42).

Recently, efforts are being made towards physically correct results. For static scenery and a static light configuration, this can be achieved by precalculating global illumination, or by coarsely calculating radiosity. Examples of this are games based on the Unreal 3 engine [230] (figure 43).

The fundamental algorithm used in renderers for games is the z-buffer scan conversion algorithm. The availability of dedicated hardware for z-buffer scan conversion has created a strong dependence on this algorithm, and has led to a huge efficiency gap between z-buffer scan conversion and the more versatile ray tracing algorithm.

Recently, Whitted-style ray tracing and distribution ray tracing have been shown to run in real-time, or at least at interactive frame rates, on CPUs and GPUs, as well as the streaming processors of modern consoles (see e.g. [248, 27, 34, 178]; [197, 81, 108, 7]; [21, 227]).

Figure 43: Precalculated global illumination, calculated using Unreal technology. Left: Mirror's edge, lit by Beäst. Right: scene lit by Lightmass.

Despite the availability of real-time ray tracing, games predominantly use rasterization for rendering virtual worlds[1]. This complicates the advance towards *physically based rendering*, where realism is not approximated empirically, but accurately simulated.

Physically based rendering of virtual worlds has strong advantages. The obvious advantage is image fidelity. Perhaps of equal importance however is production efficiency. Whereas lighting for a level in a rasterization-based engine typically requires a designer to work around technical limitations of the renderer to make the lighting *look right*, physically based rendering naturally leads to correct lighting, which limits the design effort to a creative process alone.

Of the available physically based rendering algorithms, stochastic ray-tracing based methods (path tracing and derived methods) are favored over finite element methods, due to their elegance and efficient handling of large scenes. Unlike rasterization based approaches, path tracing scales up to photo realism with minimal algorithmic complexity: the only dependencies are compute power and memory bandwidth. Both increase over time. Moore's law states that the number of transistors that can be placed inexpensively on an integrated circuit rises exponentially over time [162]. Although the link between transistor count and application performance is complex, the latter follows the same pattern, with compute power increasing at 71% per year on average, and DRAM bandwidth at 25% per year [179].

Assuming that all other factors remain constant (e.g., scene complexity, screen resolution), it can thus be assumed that there will be a point where physically based rendering is feasible on consumer hardware.

7.2 PREVIOUS WORK

Interactive path tracing was first mentioned in 1999 by Walter et al. as a possible application of their Render Cache system [258]. Using their system and a sixty-

1 Exceptions exist, e.g. [79, 58, 190, 74, 63]. These games reached only a small audience, or where not developed with commercial intentions.

core machine, a scene can be navigated at interactive frame rates. During camera movement, samples are cached and reprojected to construct an approximation for the new camera view point. New samples are created for pixels with a relatively large error. The image converges to the correct solution when the camera is stationary.

Sadeghi et al. use ray packets for their path tracer [206]. Coherence between rays on the paths of block of pixels is obtained by using the same random numbers for all pixels in the block. This introduces structural noise, but remains unbiased. The system is CPU based, and achieves about 1.2M rays per second per core of an Intel Core 2 Quad running at 2.83 Ghz.

In their 2009 paper, Aila and Laine evaluate the performance of various ray traversal kernels on the GPU [7]. Although they did not aim to implement a full path tracer, their measurements include a diffuse bounce, for which they report nearly 50M rays per second on an NVidia GTX285, not including shading.

More recently, Novák et al. used GPU path tracing with path regeneration to double the performance of the path tracing algorithm on stream processors [171]. Their system is able to render interactive previews on commodity hardware, achieving 13M rays per second on an NVidia GTX285 on moderately complex scenes, and is claimed to be "the first efficient (bidirectional) path tracer on the GPU". Van Antwerpen proposed a generic streaming approach for GPU path tracing algorithms, and used this to implement three streaming GPU-only unbiased rendering algorithms: a path tracer, a bidirectional path tracer, and an energy redistribution path tracer.

Outside academia, several applications implement interactive path tracing. Examples are Octane [200], SmallPT [18], TokaSPT [24], SmallLuxGPU [124] and NVidia's Design Garage demo [175].

7.3 THE BRIGADE SYSTEM

A renderer for games has specific requirements, which differ significantly from other applications. Of these, the requirement of real-time performance probably has the greatest overall impact on the design of a renderer. A modern game runs at 60 fps or more. For certain genres, a lower frame rate is acceptable. For the game Doom 4, a fixed frame rate of 30 fps is enforced by the renderer [37].

Frame rate translates to a strict millisecond budget, which must be divided over all subsystems. Note that if we chose to run the subsystems in order, the budget available to rendering decreases. If, on the other hand, we run the subsystems and rendering in parallel, we introduce input lag: in a worst-case scenario, user input that occurred at the beginning of frame N will be rendered in frame $N + 1$, and presented to the user just before frame $N + 2$ starts.

Apart from real-time performance, rendering for games requires dynamic scenery. Scene elements may undergo complex movement due to physics as well as hand-crafted animations and procedural effects such as explosions. Contrary to popular

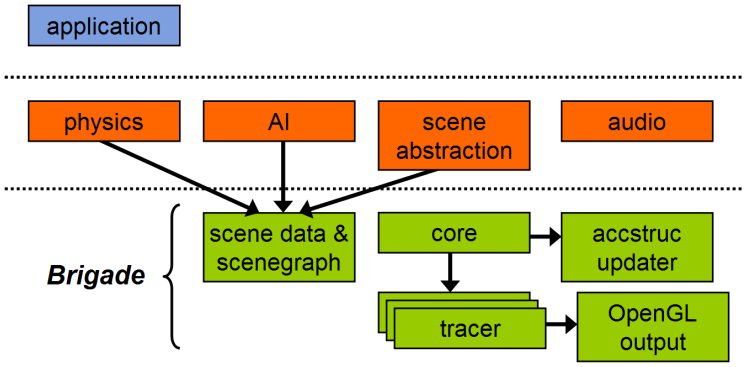


Figure 44: Functional overview of the Brigade renderer, combined with a generic game engine.

belief, global changes to scenery are uncommon in games. Typically, large portions of the scenery are static, to avoid game states in which the user cannot progress.

Tightly coupled to the real-time requirement is the fact that games are interactive applications. The renderer must produce correct results for all possible user input, and cannot predict any scenery changes that depend on user interaction.

On top of the generic requirements, there are requirements that evolve over time, most notably rendering resolution and scene complexity. At the time of writing, a typical game renders at a resolution of at least 1280x720 (HD 720). A typical scene consists of hundreds of thousands of polygons.

The Brigade rendering system is designed specifically for games, and applies and encapsulates the technology of chapter 6 in this context. The rendering system achieves high performance by fully utilizing all compute devices in a heterogeneous architecture (subsection 7.3.2). It implements a synchronization-free balancing scheme to divide the workload over these compute devices (subsection 7.3.3). Adaptive converging (subsection 7.3.5) and dynamic workload scaling (subsection 7.3.7) are used to ensure a real-time frame rate at high-definition resolutions.

7.3.1 *Functional Overview*

Figure 44 provides a functional overview of the Brigade renderer. In a typical setup, Brigade is combined with a more generic game engine that provides components not specific to the rendering algorithm, such as artificial intelligence and physics libraries.

The main components of Brigade are:

SCENE GRAPH The scene and hierarchical scene graph contain all data required for rendering. This includes the object hierarchy, mesh data, materials, textures, cameras and lights. The object decomposition represented by the scene graph is used to steer acceleration structure construction, which makes the scene graph an essential data structure within the system. For convenience, the scene graph object implements keyframe- and bone animation.

CORE The core implements the `Render()` method, initiates acceleration structure updates, synchronizes scene data changes with the compute devices and divides work over the tracers, if there is more than one.

ACCELERATION STRUCTURE UPDATER The acceleration structure updater maintains the BVH, by selectively rebuilding parts of the acceleration structure based on changes in the scene graph.

TRACERS A *tracer* is an abstract representation of a compute device or group of similar compute devices. A 'compute device' in this context can be a GPU, the set of available CPU cores, or a compute device connected over a network. The tracer holds a copy of the scene data and the acceleration structure, and implements the path tracing algorithm with next event estimation and multiple importance sampling. Tracers are assumed to produce identical output for identical input[2]. Each tracer is responsible for a part of the screen, and emits pixels to an integer or floating point output surface.

The acceleration structure used by the tracers is the only cached data structure that is derived from scene data. All other data can be modified *on-the-fly*. This includes (all properties of) materials, lights and cameras.

In this system, the governing processes run on the CPU, and tracers (which in a typical setup primarily run on the GPUs) function as workers.

7.3.2 *Rendering on a Heterogeneous System*

A modern PC is a heterogeneous architecture, which typically consists of a CPU with multiple cores, and at least one GPU.

To efficiently use the available compute power, several options are available:

1. The rendering algorithm is implemented completely on either the CPU or the GPU;

2 Except for non-deterministic aspects of the rendering algorithm.

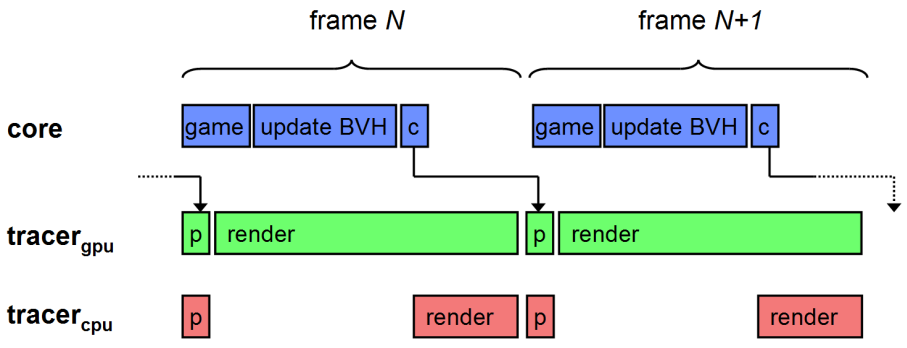


Figure 45: Double buffering the BVH. The CPU updates the BVH and sends changes to the tracers. Each tracer processes the changes in the commit buffer before rendering the next frame.

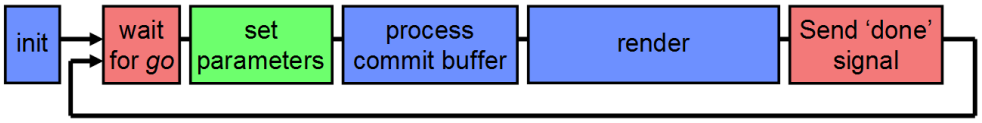


Figure 46: Tracer thread initialization and main loop.

2. The rendering algorithm is implemented on both the CPU and the GPU;
3. Tasks are divided over CPU and GPU.

Each of these options has advantages and disadvantages. A renderer that runs entirely on either the CPU or on the GPU may result in under-utilization of the other compute device. An algorithm that is implemented on both the CPU and the GPU will use all resources, but requires more implementation effort. Dividing tasks over CPU and GPU seems the most attractive option. This is however only efficient when CPU and GPU spend equal amounts of time on their assigned tasks.

A fourth option is to use a hybrid solution, where the CPU has specific tasks, and uses the frame time that remains to assist the GPU. This is the approach implemented in our system. The CPU is responsible for game logic and acceleration structure maintenance, while the tracers perform the actual rendering. Assuming a CPU tracer is available, this system is able to keep compute devices fully occupied. The process is illustrated in figure 45.

For each frame, the CPU updates the game state. The resulting changes to the scene graph are then translated to a new BVH. The changes to the BVH, as well as any other scene changes, are sent to the tracers, where they are placed in a commit buffer, which the tracers use to iteratively update a local copy of the scene.

Parallel to these activities, the tracers render using the data that was prepared in the previous frame. A tracer starts a frame by processing the changes in the commit buffer, and then renders a part of the frame. CPU tracers are handled slightly differently than GPU tracers, by postponing rendering until the acceleration structure has been updated. This prevents that rendering interferes with acceleration structure maintenance.

When no CPU tracer is available, the CPU can execute game code that does not affect the scene graph after copying scene changes to the commit buffers of the tracers.

7.3.3 *Workload Balancing*

The Tracer flow is shown in figure 46. Upon instantiation, the tracer spawns a thread that executes the worker loop. This loop waits for a signal from the core, renders a number of pixels, and signals the core, before going to sleep until the next frame.

When more than a single tracer is available, the core estimates an optimal workload division prior to rendering each frame. The advantage of this approach is that no communication between the tracers and the core is required once rendering

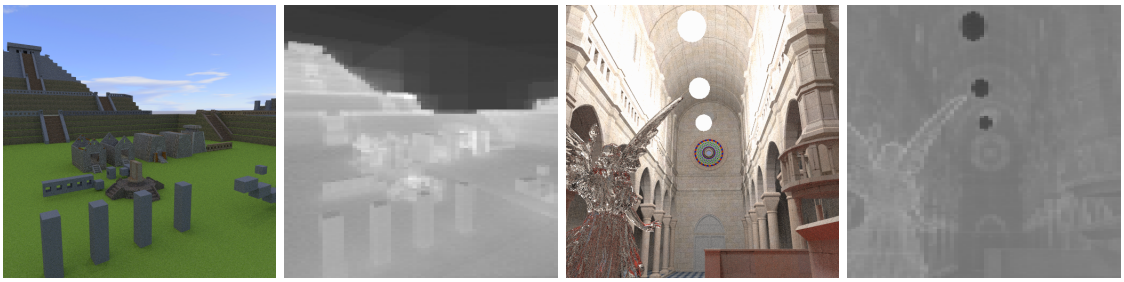


Figure 47: Render cost visualized: pixels representing the sky dome or light sources have a significant lower cost than other pixels. Cost is represented by greyscale values (brighter is higher cost), per 32 pixels (a full warp). Measured using the CUDA tracer.

has commenced, which greatly reduces communication overhead for GPU and network tracers. Dividing the work is non-trivial however: not every compute device may have the same rendering capacity, and not every line of pixels has the same rendering cost (see figure 47).

In a game, a typical camera moves in a somewhat smooth fashion. A good workload division for one frame will thus be at least reasonable for the next frame. We exploit this by adjusting the workload balance in an iterative manner.

We implemented four schemes to divide work over the tracers:

DON'T BALANCE In this naive scheme, the work is evenly distributed over the tracers; no balancing is performed. This scheme is included for reference.

ROBIN HOOD In this scheme, the tracer that finished last passes one work unit to the tracer that finished first. When the work is poorly distributed, it may take some number of frames to properly balance.

PERFECT Calculates the exact amount of work a tracer can handle based on the previous frame, but without considering differences in cost between lines of pixels. This may result in hick-ups, when many expensive lines are assigned to a tracer at once.

PERFECT SMOOTH Same as "Perfect", but this time, the workload per tracer is smoothed over multiple frames.

Figure 48 shows the efficiency of the four schemes, for a spinning camera in the Aztec scene. For a slow moving camera, the workload in two subsequent frames is similar. All schemes except the overcompensating "Perfect" balancer work well. The "Robin Hood" balancer exhibits poor efficiency for the first frames. For a faster camera, "Robin Hood" is not able to keep up. For this situation, the aggressive "Perfect" balancer outperforms even the "Perfect Smooth" balancer. When more GPUs are used, "Perfect Smooth" is clearly the optimal scheme.

Table 16 shows the average efficiency of the four balancers over 128 frames, for a slow and a faster moving camera. This table confirms that the "Perfect" and

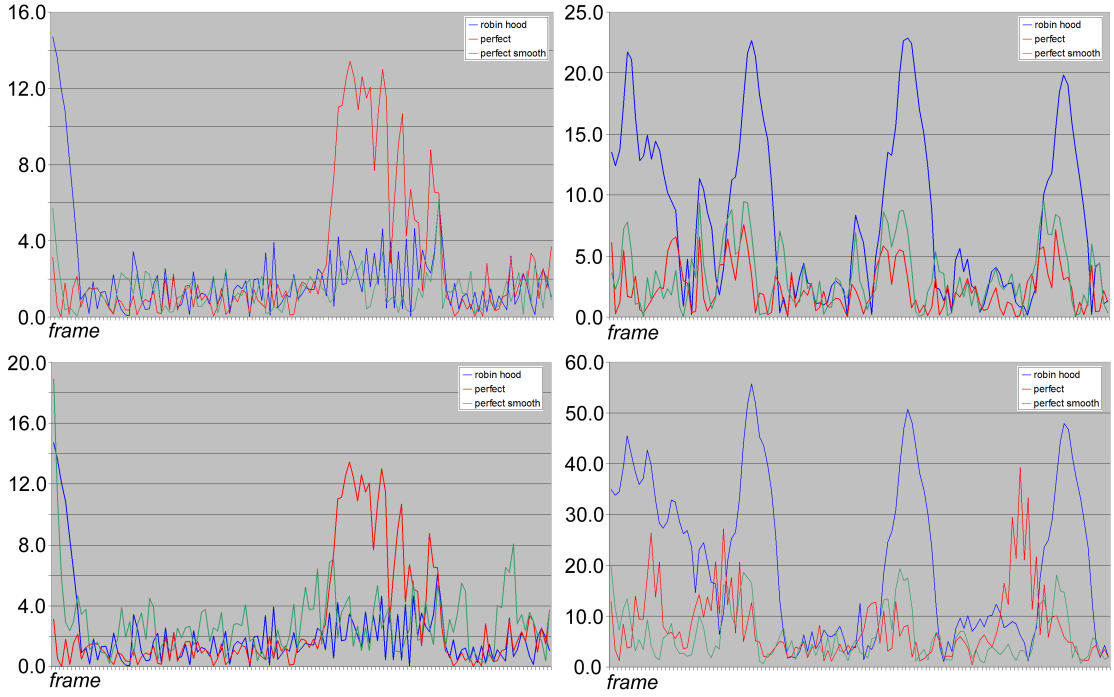


Figure 48: Efficiency of three workload balancing schemes, for two GPUs (top row) and three GPUs (bottom row), and small camera movements (left column) and larger camera movements (right column). Values are percentage of rendering time spent idling due to early completion.

"Perfect Smooth" schemes are similar in terms of average efficiency. The table does however not show the spikes that are visible in the graphs.

7.3.4 *Double-buffering Scene Data*

For acceleration structure maintenance, we use the following assumptions:

1. A game world may consist of millions of polygons.
2. A small portion of these polygons is dynamic.
3. Several tracers will use the same acceleration structure.

Based on these assumptions, a full rebuild of the BVH for each frame is neither required nor desired, as it would put a cap on maximum scene complexity, even when very few changes occur. We reuse the system described in chapter 3, where each scene graph node has its own BVH, and a top-level BVH is constructed per frame over these BVHs. Each changed scene graph node is updated, using either full reconstruction or refitting.

Brigade uses a double-buffered approach for BVH maintenance. During a single frame, the CPU updates the BVH based on modifications of the scene graph. The resulting changes to the BVH are sent to the tracers, where they are placed in a commit buffer. At the start of the next frame, the commit buffer is processed, which

	2 GPUs		3 GPUs	
	slow	fast	slow	fast
none	46.4	30.2	45.1	47.2
robin hood	2.1	8.2	4.9	20.7
perfect	2.8	2.4	12.2	8.0
perfect smooth	1.4	3.4	2.8	6.2

Table 16: Average percentage of rendering time spent idling due to early completion, for the four balancing schemes, over 128 frames, for a slow and a faster moving camera.

results in an up-to-date BVH for each of the tracers. This process is illustrated in figure 45.

Each frame is thus rendered using the BVH constructed during the previous frame. Acceleration maintenance construction thus only becomes a bottleneck when the time it requires exceeds the duration of a frame.

7.3.5 *Converging*

To improve the quality of the rendered image, several frames can be blended. Each pixel of the final image is calculated as $C_{final} = C_{prev} * (1 - f) + C_{new} * f$, where $f \epsilon (0, 1]$. For stationary views, this effectively results in a higher number of samples per pixel. For non-stationary views, this results in an incorrect image. The result can be improved by linking f to camera movement. For a stationary camera, a small value of f allows the renderer to blend many frames. For a moving camera, a value of f close to 1 minimizes ghosting.

Note that even though the camera may be static, objects in the scene may not be. It is therefore important to limit the minimum value of f to keep the ghosting for dynamic objects within acceptable bounds.

7.3.6 *CPU Single Ray Queries*

Brigade exposes a CPU-based synchronous single ray query that uses the BVH from the previous frame, to provide the game engine with a fast single-ray query. This query is useful for a number of typical game situations, such as line-of-sight queries for weapons and AI, collision queries for physics, and general object picking. The single ray query uses the full detail scene (rather than e.g. a coarse collision mesh), including animated objects.

7.3.7 *Dynamically Scaling Workload*

Maintaining a sufficient frame rate is of paramount importance to a game application. In this subsection, we propose several approaches to scale the workload.

ADJUSTING SAMPLES PER PIXEL The relation between frames per second and samples per pixel is almost linear. Brigade adjusts the rendered number of samples per pixel when the frame rate drops below a specified minimum, and increases this value when the frame rate exceeds a specified maximum.

BALANCING PRIMARY RAYS AND SECONDARY RAYS By balancing the ratio of primary and secondary rays, the quality of anti-aliasing and depth of field blurring can be traded for secondary effects. The primary rays are faster; increasing their ratio will also improve frame rate.

SCALE RUSSIAN ROULETTE TERMINATION PROBABILITY As discussed in section 6.3.2, changing the termination probability of Russian Roulette does not introduce bias, but only more variance. Altering the termination probability does however affect the number of deeper path segments, and thus frame rate. Unlike the previous approach, scaling the termination probability using a factor which is based on frame rate does not distinguish between primary and secondary rays, and allows smooth scaling of performance.

Alternatively, the workload can be reduced by reducing rendering resolution, or limiting trace depth. Limiting the maximum recursion depth of the path tracer introduces bias, but also improves performance. In practice, due to Russian Roulette deep rays are rare, which limits the effect of a recursion depth cap on performance.

For game development, the scalability of a renderer based on path tracing is an attractive characteristic. A relatively slow system is able to run the path tracer at an acceptable frame rate, albeit not at an acceptable level of variance. Faster systems benefit from the additional performance by producing more samples per pixel, and thus a smoother image.

7.3.8 *Discussion*

The rendering system described in this section is relatively simple. To a large extend, this simplicity is the result of the chosen rendering algorithm. The path tracer does not rely on any precalculated data, which greatly reduces data dependencies. There are two exceptions, and these are also the most complex parts of the system. The first is the acceleration structure, which is cached and updated iteratively, in a double-buffered fashion. As a result, games cannot make arbitrary changes to the scene graph. The second is the data synchronization between the renderer core and the tracers, which generally run on the GPU(s). Using a commit buffer system, Brigade makes this virtually invisible to the application, and very few restrictions apply.

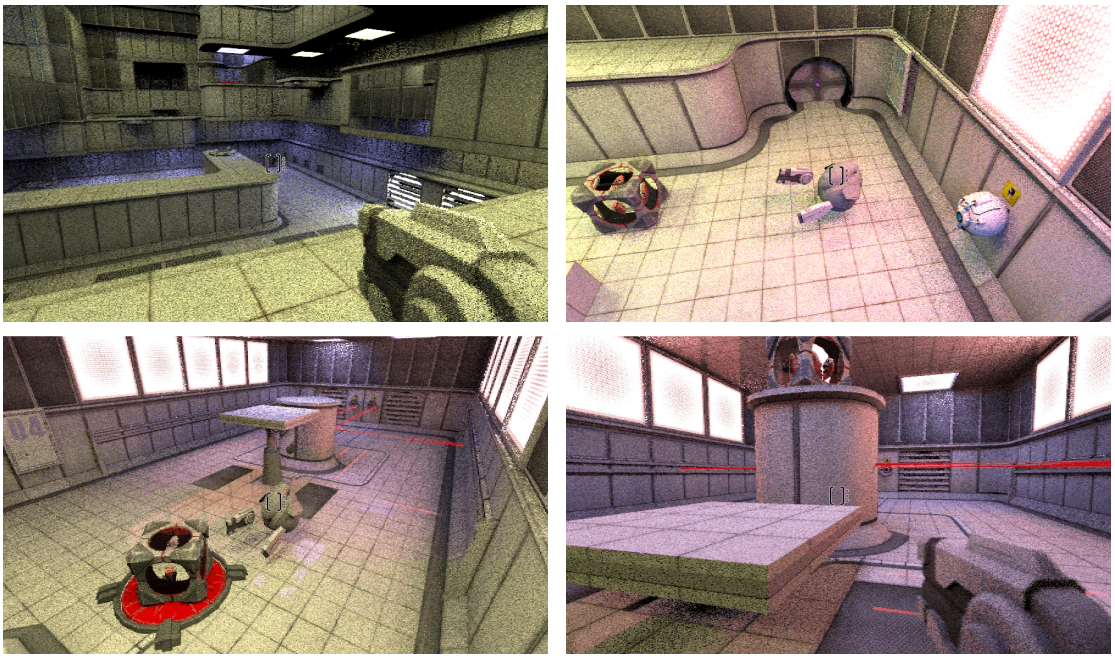


Figure 49: Two views from the "Reflect" game, rendered at 448x576 pixels using 16spp, scaled up to 896x576.

Apart from the tracers, Brigade is a platform-independent system. The tracers abstract away vendor-specific APIs for GPGPU, and allow the implementation of networked tracers and CPU-based tracers. When using a CPU tracer, the system is able to achieve full system utilization, with very little overhead.

7.4 APPLIED

To validate our results, we have applied the renderer to two student game projects.

Both games have been produced in approximately 14 working days.

7.4.1 *Demo Project "Reflect"*

The Reflect game application is a student game that was developed using an early version of the Brigade engine. The game scenery is designed to simulate actual requirements for game development, and purposely mimics the graphical style of a well-known modern game (Portal 2 [169]).

- scenery consists of approx. 250k triangles, divided over multiple, separated rooms;
- the scene is illuminated by thousands of area light sources, many of which are dynamic;
- the game world is populated by dozens of dynamic objects.

Art assets for the game were created in Alias Wavefront Maya 2011, and directly imported into the game.

Like Portal 2, Reflect is a puzzle game, where the player advances by activating triggers that in turn open doors or activate elevators. A "mirror gun" is available to the player to transform flat and curved wall sections into mirrors. These mirrors, as well as glass cube objects, can be used to divert lasers that block the way.

Configuration

Reflect was developed for a dual-CPU / dual-GPU machine (2 hexacore Intel Xeon processors, 2 NVidia GTX470 GPUs). We implemented a CPU tracer as well as a CUDA GPU tracer. For performance reasons, we limited the path tracers to a single diffuse bounce.

Game-specific Optimizations

The scenery of the game consists of many rooms, separated by doors. A common optimization in rasterization-based renderers is to disable geometry that cannot be visible. For a path tracer this does not significantly improve performance. We did find however that turning off lights in those rooms reduces variance, as the path tracer will no longer sample those light sources.

Performance and Variance

Figure 49 shows two scenes from the game running on a dual-CPU / dual GPU machine. At 16 spp, the game runs at 10-12fps. At this sample count, brightly lit scenes are close to acceptable. Darker regions, such as the area under the platform in the right image, show significant temporal noise. Careful level layout helps to reduce objectionable noise levels. To the visual artist, this is counter-intuitive: where rasterization-based renderers tend to use small amounts of point light sources, a path tracer benefits from large area lights, and incurs no slowdown when those lights are animated.

Observations

"Reflect" struggles to achieve an acceptable frame rate, at a low resolution, on a high-end system. The project does however show the potential of using path tracing for games. The art for this game was produced in Maya 2011, and directly imported into the game, leading to very short development cycles, and usable art on the second day of the project. Within the same time span, the programmers implemented a basic physics engine using ray queries that allowed them to navigate the rooms.

The freedom in lighting setup led to a final level that contains approximately 10k light emitting polygons. Direct and indirect illumination 'simply works', and results in subtle global illumination, both for static and dynamic objects.

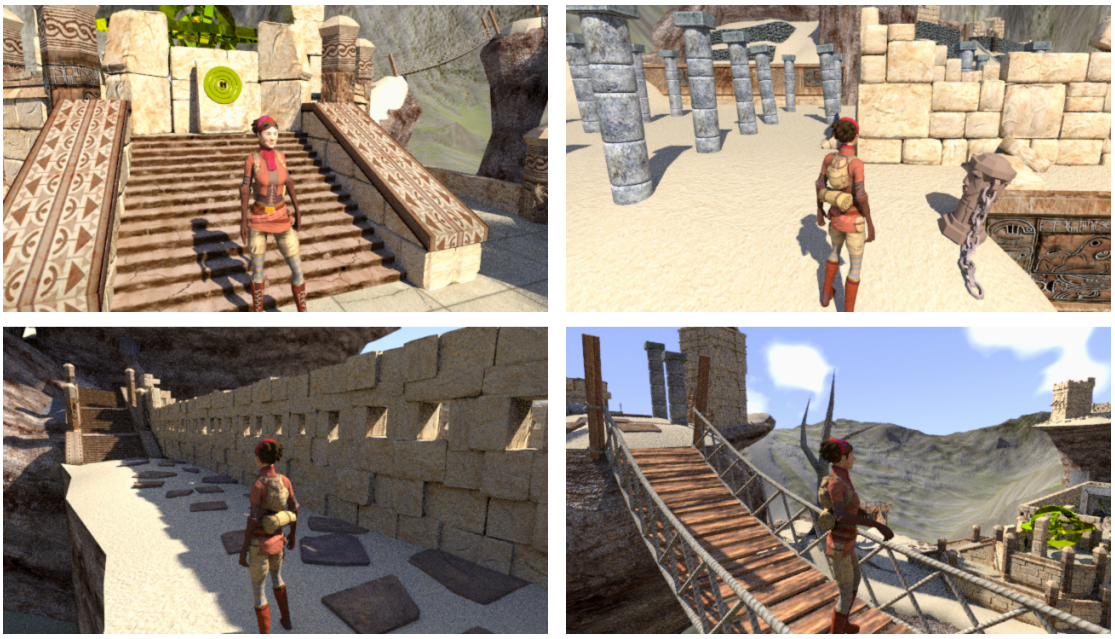


Figure 50: Four views from "It's About Time".

The focus on using level design to reduce variance, rather than to optimizing frame rate, proved to be challenging.

The CPU tracer that was implemented for this project proved to be problematic: keeping the CPU and GPU tracers in sync required significant engineering effort, while the overall contribution of the CPU is quite small.

7.4.2 *Demo Project "It's About Time"*

The student game "It's About Time" was created using a recent version of the Brigade renderer. Four views from the game are shown in figure 50.

"It's about time" is a third-person puzzle game that takes place in deserted Aztec ruins. The player must collect a number of artifacts by solving a number of puzzles, located in several areas in an open outdoor world.

Configuration

"It's About Time" is designed for a standard high-end system, using a single hexacore CPU and one or more current-generation NVidia or AMD GPUs. The game renders to standard HD resolution. This resolution can be halved to improve performance. We developed an updated CUDA tracer that roughly doubles the performance of the first iteration (as used in "Reflect"), as well as an OpenCL tracer, which produces identical images. A CPU tracer was not used; the CPU is reserved for acceleration structure maintenance and game logic. The implemented path tracers are unbiased.

Project-specific Features

One of the puzzles features an animated water surface that responds to the player, consisting of tens of thousands of polygons. For the player character, a detailed set of skinned animations is used. The puzzles make extensive use of rigid animation. As a result, acceleration structure maintenance requires considerable processing. A detailed day-night cycle and an animated cloud system (with shadowing) were implemented to emphasize the strength of the path tracer for dynamic lighting. A standard physics engine was integrated to provide high quality physics simulation. The level is detailed, and consists of 1.4M triangles. The artists used a small set of sand stones to construct most of the buildings and ruins.

Game-specific Optimizations

The game world is illuminated by a sun (or the moon), and some smaller light sources. To reduce variance, we modified the path tracer to always sample two light sources per diffuse surface interaction. One of these rays always probes the primary light source. This significantly reduces variance in most areas. Adaptive converging is used to improve image quality when the camera is (almost) stationary.

System utilization

Figure 51 shows system utilization for the four views of figure 50, rendered at 4spp.

For the first two views, the CPU is underutilized, as both acceleration structure maintenance and game logic require little processing time. For the other two views, the camera is near a simulated water surface that consists of 18k polygons. Both the simulation itself and the resulting acceleration structure maintenance require considerable processing time. This also affects the GPU tracers, which use more time to transfer and process the modified scene data.

Performance and Variance

Figure 52 shows a single scene from the game, rendered using varying sample counts. As in "Reflect", areas that are directly illuminated converge quickly, while shadowed areas exhibit more noise. For the outdoor scenery of "It's About Time", an acceptable quality for most camera views is obtained with 8 or 16spp. On a system with two NVidia GTX470 GPUs, we achieve 2 to 4 spp at real-time frame rates, at a quarter of 720p HD resolution (640x360). This lets us quantify the remaining performance gap: real-time frame rates at 720p require 8 to 16 times the achieved performance.

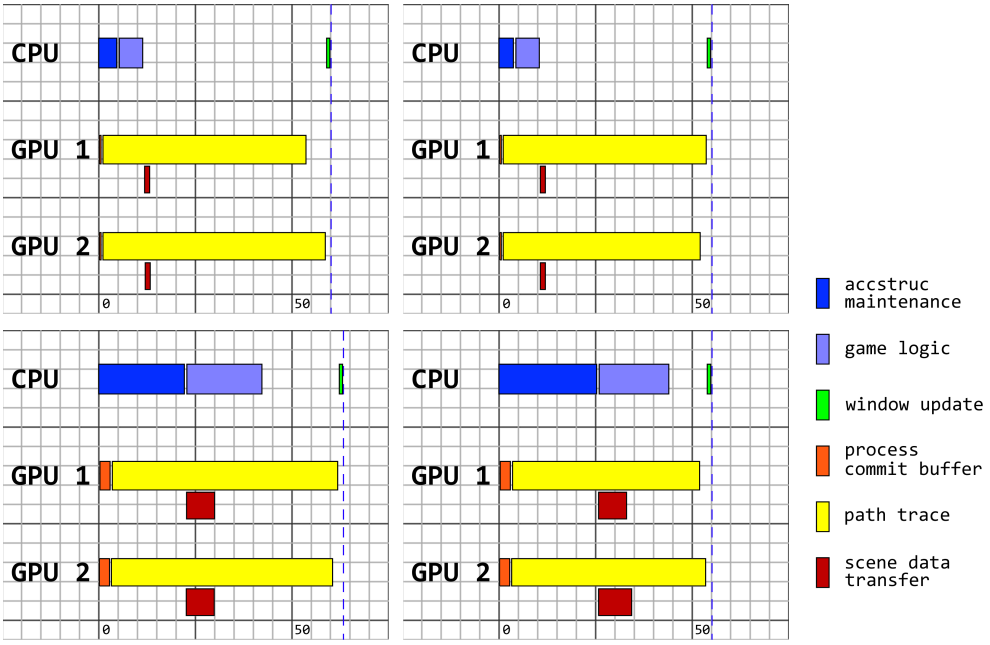


Figure 51: System utilization for the four views shown in figure 50.

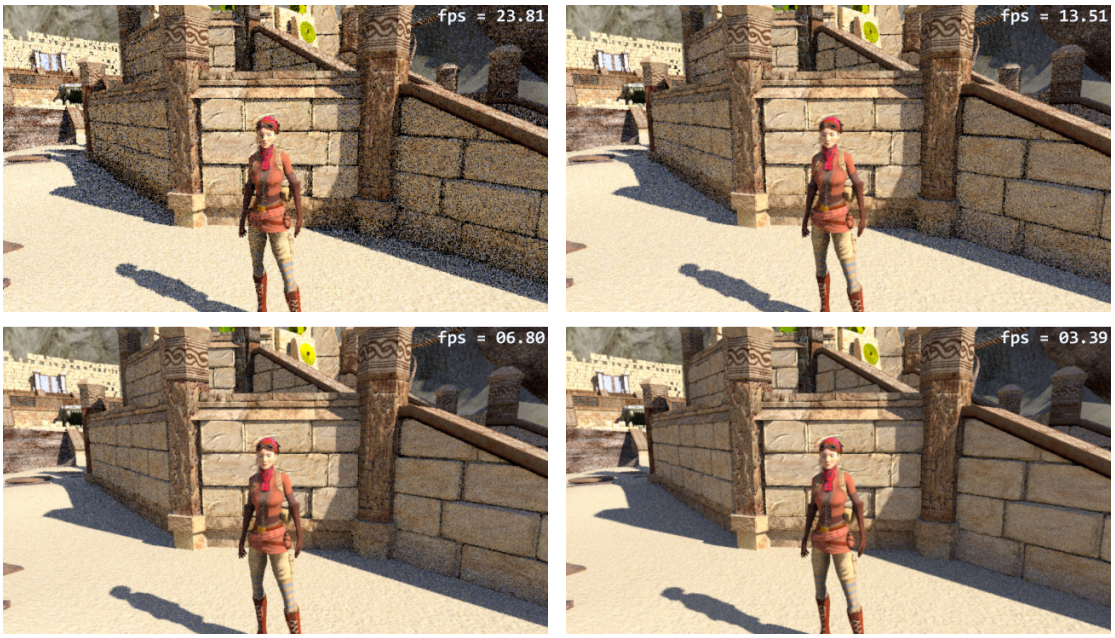


Figure 52: Noise level and performance at 2spp, 4spp, 8spp and 16spp. Rendering resolution is 640x360. Measured on a system with a 6-core CPU and two NVidia GTX470 GPUs.

The Brigade renderer implements the path tracing algorithm on a heterogeneous architecture of CPUs and GPUs. In this system, the CPU is responsible for acceleration structure maintenance. A set of *tracers* implements the core path tracing algorithm on all compute devices (e.g. CPU, GPU, and remote systems over a network connection). Assuming availability of a CPU tracer, the system achieves full system utilization. We proposed to use double buffering of scene data to enable parallel execution of acceleration structure maintenance and path tracing. We applied path tracing to two games, and achieve real-time performance for non-trivial scenes, albeit at low resolutions and high variance.

The development of a game using path tracing for rendering simplifies game development. This affects both software engineering and art asset development. Since Brigade does not distinguish static and dynamic light sources, and does not impose any limitations on the number or size of light sources, lighting design requires little knowledge beyond discipline-specific skills. The fact that polygon counts and material properties have only a small impact on rendering performance provides level designers and graphical artists with a high level of freedom in the design of the game. This reduces the number of iterations level art goes through, and allows a team to have game assets in the engine early on in the project.

Despite these positive experiences, real-time path tracing in commercial games is not feasible yet on current generation high-end hardware. Acceptable variance at HD resolution and real-time frame rates requires 8x to 16x the performance that can be achieved on our test system. Without further algorithmic improvements, this level may be reached in a few years. We do believe this can be accelerated. Already GPU ray tracing performance is benefiting from architectural improvements, on top of steady performance improvements. Another way to partially solve the rendering performance problem is to use cloud rendering, where dedicated servers are used for rendering images, which are then transferred over the internet to the client. At the time of writing, the Brigade system is being integrated into the OTOY cloud service, specifically for this purpose, and will be able to use 8 or more GPUs in parallel, which should allow us to reach real-time performance. The cloud rendering service will be made available to indie game developers in the near future, and will allow them to use path tracing without the need of owning sufficiently powerful hardware.

Apart from raw performance, we should address the issue of variance. While low sample rates already result in reasonably converged images in our experiments, this will not be sufficient for more complex materials. Techniques like bidirectional path tracing (BDPT) and energy redistribution path tracing (ERPT) may solve this to some extent. However, not all of these techniques produce acceptable images at low sample rates; therefore, a minimum performance level is required before this can be considered for real-time graphics.

A temporary solution to the rendering performance problem is to use post processing on the path traced image. Although some work has been conducted in

this area, it typically does not consider all the data that is available in a path tracer, which leaves room for improvement.

8

CONCLUSIONS AND FUTURE WORK

In this thesis, we set out to investigate ray tracing and path tracing as alternatives to rasterization-based rendering for real-time games. The desire to replace the rasterization algorithm stems from fundamental limitations of this algorithm when dealing with global effects. These limitations led to the addition of approximating algorithms, and, consequently, the development of highly complex renderers, which often place restrictions on various aspects of rendering, such as the use of lights and dynamic geometry. This complicates the game development process as a whole. The construction of renderers as a complex ecosystem of sub-algorithms also affects realism in modern games. Generally speaking, the level of realism achieved by today's games is bound by the availability of approximating algorithms and the size of the development budget, rather than compute power.

For the ray tracing algorithm, the opposite is true. Algorithms for physically based light transport have been available for nearly three decades, but the required compute power did make these algorithms unsuitable for real-time games, where frame times are typically limited to tens of milliseconds. Over the past decade, this has changed: real-time ray tracing became feasible on consumer hardware. The promise of realistic images produced by an elegant renderer, and the engineering challenge this represents, triggered our initial research.

In the first part of this thesis, we focused on Whitted-style ray tracing, and validation and combination of existing work.

We implemented a carefully optimized renderer for games. This served a number of purposes. First of all, existing work often focused only on parts of a practical renderer. The lack of a realistic context may reduce the validity of findings. Furthermore, existing work showed varying levels of absolute performance for similar functionality, which suggests that performance figures may be related to engineering quality as much as algorithmic quality. And finally, the validation of work in an actual game development environment let us observe the impact of ray tracing on graphical style, and the efficiency of the development process.

To add diffuse interreflection to the Whitted-style ray tracer, we replaced the constant ambient color by an approximation of diffuse indirect light, using a cache, and a secondary rendering algorithm. However, by doing so, we reintroduced the drawbacks of stacked rendering algorithms.

In the second part of this thesis, we studied physically based rendering using path tracing. Path tracing improves on Whitted-style ray tracing, by enabling the full set of light transport paths described by the rendering equation, which results in highly realistic images. Unlike most rasterization-based rendering systems,

the path tracing algorithm requires no scene preprocessing (apart from building the acceleration structure). The algorithm does not require manual tuning of parameters. This is not the case for Whitted-style ray tracing, which generally requires a global ambient light color, and rasterization based algorithms, which typically require the user to specify many variables, e.g., shadow map resolution and detail settings.

From a production point of view, the use of path tracing in games is highly desirable. At the same time, the performance requirements of the path tracing algorithm are daunting. For that reason, we aimed for a *proof-of-concept* renderer and an estimate of the performance level required for a practical application.

Path tracing is able to outperform rasterization-based rendering in terms of perceived realism, without sacrificing the elegance of the underlying algorithm. The compute power required for this will be available in a typical consumer system in the near future. Using cloud rendering, this power is available today. Relying on compute power alone for further advances in image fidelity is an important change, which will significantly affect the future of graphics in games.

Our work focuses on practical implementation and application of rendering algorithms. The harsh performance requirements of the algorithms demand optimal use of system resources, and an emphasis on code optimization. We believe our implementations are of high quality, which makes them suitable for comparisons in future work.

SUMMARY

Whitted-Style Ray Tracing Ray tracing in a real-time context requires careful engineering. The development of acceleration structures and acceleration structure traversal using single rays and later bundles of rays can be seen as optimization efforts, each targeting a bottleneck in rendering performance. When *time to image* includes acceleration structure maintenance, as is the case for games, we need to balance time spent on this maintenance, and time spent tracing rays. This suggests that no single 'ultimate renderer' can be constructed; as has always been the case, renderers for games (and probably for a broader context) are tailor-made, if we desire optimal performance of the application.

The Arauna real-time ray tracer was implemented as a proof-of-concept renderer, and serves as a state-of-the-art ray tracer which combines existing algorithms and data structures in a highly optimized manner. It implements the Whitted-style ray tracing algorithm, with one modification: to facilitate the large number of lights used in most games, Arauna uses an approximation of quadratic fall-off that limits the radius of influence of lights, so that a light BVH can be used to reduce the cost of scenes with many lights. Arauna was used by other researchers as a reference renderer, as well as for a number of student games, including a fast-paced pinball game and a multiplayer deathmatch game. For almost all developed titles, custom features were added to the renderer, such as a multi-beam spotlight, normal

mapping and volumetric fog. This reiterates the assumption that a renderer is tailor-made for a game, even when using ray tracing.

The development of several games using the Arauna engine showed that production benefits from ray tracing. Several teams implemented basic physics using ray queries, and visual artists were pleased with the forgiving art pipeline, resulting in early playable builds of all games.

Sparse Sampling of Global Illumination The development of the Arauna ray tracer, and its use in games, also made clear that ray tracing is efficient mainly for primary rays and shadow rays. Although the algorithm remains elegant when reflections and refractions are added, performance quickly degrades. In practice, this discourages the use of these features in games, and leads to somewhat bland graphics that do not improve on the visual quality achieved by rasterization-based renderers. Particularly the absence of indirect light, which is approximated quite convincingly by some modern renderers, is disappointing.

While extensions of Whitted-style ray tracing such as path tracing would solve this, the computational burden of these algorithms is prohibitive even for modern CPUs. As an intermediate solution, we proposed to calculate global illumination on the CPU within the time available for one frame by sampling global illumination sparsely. To do this, we create a set of random points distributed over the surfaces of the scene, with a density that adapts to ambient occlusion, which we use as an estimate of the frequency of the global illumination. For this set of points we then calculate accurate illumination. By sparsely sampling global illumination, we can scale the cost to match the available computational power. As the density of the point set increases, the result converges to the correct solution. For the illumination itself, we thus keep the generality and elegance of ray tracing. The approach does however complicate illumination in dynamic environments.

Path Tracing The final goal of rendering game scenes using path tracing comes within reach if we combine the computational power of the CPU and the GPU. With the relative inefficiency of secondary effects in Arauna in mind, we further investigated performance of divergent ray queries on the CPU. We adapted an existing scheme aimed at out-of-core rendering to improve data locality for in-core ray tracing. We improved on state-of-the-art performance, using breadth-first traversal of a shallow octree with MBVHs in the leaf nodes. Despite the promising results, the efficiency of the CPU for path tracing remains low.

Measuring path tracing performance is not straight-forward. Frame rate has little meaning, and neither has raw ray throughput. We therefore compare efficiency of algorithms by measuring variance over time. We use this to evaluate several GPU path tracing algorithms that aim to improve SIMT utilization. Using the same measure, we evaluate algorithms that aim to reduce variance, at the cost of extra computations.

Equipped with optimized implementations of techniques for fast path tracing for a variety of scenes on both the CPU and the GPU, we built the Brigade path tracing system. Like Arauna, the Brigade system was developed specifically for games, which is a new application of physically based rendering. This specific context

involves a number of harsh requirements, of which a small time budget is the most challenging. Brigade implements an architecture that divides the rendering workload over abstract 'tracers', which implement the path tracing algorithm for specific compute units. The Brigade system has been used for two more games and is used commercially by OTOY Inc. for cloud rendering.

We have shown that physically-based rendering algorithms based on ray tracing can be applied to games using consumer hardware. These algorithms do not rely on precomputed data or configurable parameters (apart from a global quality setting). This has strong advantages for game production. Several long-standing issues in rendering are solved in an elegant and straight-forward way. Rendering becomes compute bound, rather than dependent on 'hack after hack' for the support of individual phenomena. The computational power required to render games using path tracing is not available yet, but not far off either: path tracing will be a viable alternative to rasterization based rendering within a few years. Once this point is reached, legacy is perhaps the only remaining hurdle.

Future Work

In this section, we conclude with an overview of possible directions for future work.

In a practical game application, considerable amount of processing is required to maintain the acceleration structure. Based on the assumption that large portions of typical game scenery are static, we chose an approach of incremental updates, carried out by the CPU, in parallel with rendering on the GPU. For the games presented in chapter 7, this did not exceed CPU capacity, and we did not seek out the boundaries of acceleration structure maintenance performance. For certain games, requirements in terms of number of dynamic object may be much higher. We would like to investigate ways to increase performance under these circumstances. Recent work shows that the GPU significantly outperforms the CPU for full updates of the acceleration structure [276, 182]. We would like to investigate incremental updates of the acceleration structure on the GPU to combine this higher performance level with the scalability of our approach.

Several authors point out that shading cost is a concern in a renderer. In our system, rendering time is dominated by ray traversal. This is partially due to the relatively simple shaders in our system: a programmable shader system such as the one as proposed by Karrenberg et al. [129] would increase the versatility of our system, but also the impact of shading on overall rendering time: rays that are processed in parallel may not just traverse different scene regions, but also encounter different materials, in which case efficiency is reduced due to sequential processing of shader code. This is similar to the data locality problem that was discussed in chapter 5. We believe that a GPU implementation of the algorithm described in chapter 5 may improve execution coherence for shader code: by batching rays that traverse the same scene regions, the chance that these rays will encounter the same materials and thus execute the same shading code increases.

To further increase the performance of our system, we would like to reduce the variance of our path tracing samplers, using bidirectional path tracing, and careful sample placement. Considering the context of rendering for games, we would also like to investigate the possibility of a real-time hybrid system, similar to the system presented by Dammertz et al. [62]. This system uses a combination of path tracing and biased algorithms, where each method handles light transport paths for which it is efficient. Although such a system would be more elaborate than a single-method renderer, it would not sacrifice generality.

In section 7.3.7 we discussed mechanisms that aim to maintain real-time frame rate by dynamically adjusting the workload. Although a path tracer generally does not benefit from level of detail algorithms, we do want to investigate the possibility of adjusting shader cost based on distance and recursion depth. This could be done by reducing material specularity and ignoring normal maps after one non-specular bounce.

We believe our system would further benefit from postprocessing. Although this would also introduce bias, noise reduction algorithms would improve the perceived quality of the produced images. Although recently quite some research has been done in this area (e.g. [61, 210, 87]), we believe a further investigation of real-time filtering techniques that make use of all relevant data in a path tracer would be worthwhile.

In this work, we have applied renderers based on ray tracing to a variety of game projects of increasing complexity. The next step is to apply these renderers to commercial games.

Part III

APPENDIX

A

APPENDIX

A.1 SHADING RECONSTRUCTION IMPLEMENTATION

This appendix contains source code for reconstruction of shading from the point set described in chapter 4.

```
// float ix, iy, iz contain surface point location
// float NX, NY, NZ contain surface point normal
int gx = (ix - scenebounds.x1) / scenebounds.xext;
int gy = (iy - scenebounds.y1) / scenebounds.yext;
int gz = (iz - scenebounds.z1) / scenebounds.zext;
int igx = gx * GRIDXSIZE;
int igy = gy * GRIDYSIZE;
int igz = gz * GRIDZSIZE;
Cell* c = &grid[igx +
igy * GRIDXSIZE + igz * GRIDXSIZE * GRIDYSIZE];
const int count = c->GetSampleCount();
__m128 gi4 = zero;
float totalscale = 0.0001f;
float sdreci = 1.0f / (1 - MINDOT);
SamplingPoint** plist = (const SamplingPoint**)c->GetSamples();
for ( int j = 0; j < count; j++ )
{
   float dx = plist[j]->pos.x - ix;
   float dy = plist[j]->pos.y - iy;
   float dz = plist[j]->pos.z - iz;
   float dist = dx * dx + dy * dy + dz * dz;
   if (dist < (plist[j]->radius * plist[j]->radius))
   {
      __m128 pgi = plist[j]->value;
      float dot = NX * plist[j]->N.x +
      NY * plist[j]->N.y + NZ * plist[j]->N.z;
      if (dot > MINDOT)
      {
         float d1 = ix * NX + iy * NY + iz * NZ;
         float d2 = plist[j]->pos.x * NX +
         plist[j]->pos.y * NY +
            plist[j]->pos.z * NZ;
         float dotscale = (dot - MINDOT) * sdreci;
         float diff = fabs( d1 - d2 );
         if (diff > MAXSURFACEDIST) continue;
         __m128 dist4 = _mm_set_ps1( dist );
         __m128 scale4 = _mm_sub_ps( _mm_set_ps1( plist[j]->radius ),
            _mm_mul_ps( dist4, _mm_rsqrt_ps( dist4 ) ) );
         __m128 scale4a = _mm_mul_ps( _mm_mul_ps( scale4, scale4 ),
            _mm_set_ps1( dot - m_SampleDot ) );
```

```
        totalscale += *reinterpret_cast<const float*>(&scale4a);
        gi4 = _mm_add_ps( gi4, _mm_mul_ps( scale4a, pgi ) );
      }
   }
}
const __m128 invscale4 = _mm_mul_ps( _mm_set_ps1( 0.003f ),
   _mm_rcp_ps( _mm_set_ps1( totalscale ) ) );
float* gi = (float*)&gi4;
addc4[p1 * 4 + 0] = _mm_add_ps( addc4[p1 * 4 + 0],
   _mm_mul_ps( _mm_mul_ps(
   m_ID->colip[p1 * 4 + 0].rgba, invscale4 ),
   _mm_set_ps1( gi[2] ) ) ); // rrrr
addc4[p1 * 4 + 1] = _mm_add_ps( addc4[p1 * 4 + 1],
   _mm_mul_ps( _mm_mul_ps(
   m_ID->colip[p1 * 4 + 1].rgba, invscale4 ),
   _mm_set_ps1( gi[1] ) ) ); // gggg
addc4[p1 * 4 + 2] = _mm_add_ps( addc4[p1 * 4 + 2],
   _mm_mul_ps( _mm_mul_ps(
   m_ID->colip[p1 * 4 + 2].rgba, invscale4 ),
   _mm_set_ps1( gi[0] ) ) ); // bbbb
```

B

APPENDIX

B.1 REFERENCE PATH TRACER

This CUDA code implements a basic path tracer, as described in chapter 6. GPU hardware occupation will be low when using this kernel, due to the break statements, which are encountered when a path leaves the scene, or encounters a light source. In these cases, the thread becomes inactive, and is only replaced by an active thread when all threads in the warp have completed.

```
extern "C" __global__ void TracePixelReference()
{
   // setup path
   int numRays = context.width * context.height;
   int idx0 = threadIdx.y + blockDim.y *
      (blockIdx.x + gridDim.x * blockIdx.y) +
      ((context.firstline * context.width) >> 5);
   int tx = threadIdx.x & 7, ty = threadIdx.x >> 3;
   int tilesperline = context.width >> 3;
   int xt = idx0 % tilesperline, yt = idx0 / tilesperline;
   int px = (xt << 3) + tx, py = (yt << 2) + ty;
   int pidx = numRays - 1 - (px + py * context.width);
   RNG genrand( pidx, (clock() * pidx * 8191) ^ 140167 );
   int spp = context.SampleCount;
   float rcpw = 1.0f / context.width;
   float u = (float)px * rcpw - 0.5f;
   float v = (float)(py + (context.width - context.height) *
      0.5f) * rcpw - 0.5f;
   float3 E = make_float3( 0, 0, 0 );
   // trace path
   for( int sample = 0; sample < spp; sample++ )
   {
      // construct primary ray
      float3 O, D;
      CreatePrimaryRay( O, D );
      // trace path
      float3 throughput = make_float3( 1, 1, 1 );
      int depth = 0;
      while (1)
      {
         int prim = 0;
         float2 BC, UV = make_float2( 0, 0 );
         float dist = 1000000;
         bool backfaced = false;
         intersect<false,true>(O,D,dist,BC,prim,backfaced);
         O += D * dist;
```

```
            if (prim == -1)
            {
               E += throughput * GetSkySample( D );
               break;
            }
            Triangle& tri = context.Triangles[prim];
            TracerMaterial mat = context.Materials[tri.GetMaterialIdx()];
            if (mat.flags & TracerMaterial::EMITTER) // light
            {
               E += throughput * mat.EmissiveColor;
               break;
            }
            else // diffuse reflection
            {
               float3 matcol = tri.GetMaterialColor( mat, BC, UV );
               float3 N = tri.GetNormal( mat, BC, UV ) * (backfaced ? -1 : 1 );
               D = normalize( RandomReflection( genrand, N ) );
               throughput *= matcol * dot( D, N );
            }
            O += D * EPSILON; // prevent intersection at dist = 0
            depth++;
            if (depth > 3)
            {
               if (genrand() > 0.5f) break;
               throughput *= 2.0f;
            }
         }
      }
      context.RenderTarget[pidx]=make_float4( E / (float)spp, 1 );
   }
```

B.2 PATH RESTART

This CUDA code implements Novak's path restart algorithm, where terminated paths are replaced by new paths to prevent GPU under-utilization. This implementation also includes Russian Roulette, explicit light sampling and multiple importance sampling to reduce the variance of the estimate.

```
#define TERMINATE { restart = true; continue; }
extern "C" __global__ void TracePixelSegment()
{
   // setup path
   int numRays = context.width * context.height;
   int idx0 = threadIdx.y + blockDim.y *
      (blockIdx.x + gridDim.x * blockIdx.y) +
      ((context.firstline * context.width) >> 5);
   int tx = threadIdx.x & 7, ty = threadIdx.x >> 3;
   int tilesperline = context.width >> 3;
   int xt = idx0 % tilesperline, yt = idx0 / tilesperline;
   int px = (xt << 3) + tx, py = (yt << 2) + ty;
   int pidx = numRays - 1 - (px + py * context.width);
```

```
RNG genrand( pidx, (clock() * pidx * 8191) ^ 140167 );
int spp = context.SampleCount;
float rcpw = 1.0f / context.width;
float u = (float)px * rcpw - 0.5f;
float v = (float)(py + (context.width - context.height)
   * 0.5f) * rcpw - 0.5f;
float3 E = make_float3( 0, 0, 0 ), throughput, O, D;
bool restart = true, firsthit = true;
int paths = 0, curdepth = 0;
// trace path
for( int segment = 0; ((segment < spp * 2) || (!restart)); segment++ )
{
   if (restart)
   {
      // construct primary ray
      CreatePrimaryRay( O, D );
      firsthit = true, restart = false;
      throughput = make_float3( 1, 1, 1 );
      curdepth = 0, paths++;
   }
   // trace path segment
   int prim = 0; float2 UV, BC;
   float dist = 1000000;
   bool backfaced = false;
   O += D * EPSILON; // prevent intersection at dist = 0
   intersect<false,true>( O, D, dist, BC, prim, backfaced );
   O += D * dist;
   if (prim == -1)
   {
      // path left scene
      E += throughput * GetSkySample( D );
      TERMINATE;
   }
   Triangle& tri = context.Triangles[prim];
   TracerMaterial mat = context.Materials[tri.GetMaterialIdx()];
   if (mat.flags & TracerMaterial::EMITTER)
   {
      // path arrived at light
      if (firsthit & (!backfaced))
         E += throughput * mat.EmissiveColor;
      TERMINATE;
   }
   float3 matcol = tri.GetMaterialColor( mat, BC, UV );
   float3 N = tri.GetNormal( mat, BC, UV ) * (backfaced ? -1 : 1 );
   float3 wo = D * -1.0f;
   // sample direct lighting using next event estimation
   float3 L, LN, LColor;
   float area;
   RandomPointOnLight( L, LN, LColor, genrand, area );
   L -= O;
   float sqdist = dot( L, L ), ldist = sqrtf( sqdist );
   L *= 1.0f / ldist;
   float NdotL = dot( N, L ), LNdotL = -dot( LN, L );
```

```
      if ((NdotL > 0) && (LNdotL > 0))
      {
         bool backface; int sprim; float2 SBC; ldist *= 0.99f;
         intersect<true,false>( O + L * EPSILON, L,
            ldist, SBC, sprim, backface );
         if (sprim == -1)
         {
            float lightPdf = (LNdotL > EPSILON) ? (sqdist /
               (LNdotL * area * context.lightcount)) : 0.0f;
            if (lightPdf > 0) E += throughput * matcol *
               INVPI * 0.5f * LColor * NdotL / lightPdf;
         }
      }
      // russian roulette
      if (curdepth > 1)
      {
         float p = max( EPSILON, min( 0.5f, (throughput.x +
            throughput.y + throughput.z) * 0.333f ) );
         if (genrand() > p) TERMINATE;
         throughput /= p;
      }
      // do a lambert reflection
      D = DiffuseReflection( genrand, N );
      float bsdfPdf = LambertPdf( D, N );
      float3 f = matcol * INVPI * 0.5f;
      if (bsdfPdf < EPSILON) TERMINATE;
      throughput *= f * dot( D, N ) / bsdfPdf;
      firsthit = false;
      curdepth++;
   }
   context.RenderTarget[pidx]=make_float4( E * (1.0f / (float)paths), 1.0f );
}
```

B.3 COMBINED

This CUDA kernel combines deterministic path termination and path regeneration.

```
#define TERMINATE { restart = true; continue; }
extern "C" __global__ void TracePixelCombined()
{
   // setup path
   int idx0 = threadIdx.y+blockDim.y*
      (blockIdx.x+gridDim.x*blockIdx.y)+
      ((context.firstline*context.width) >> 5);
   int tx = threadIdx.x & 7,ty = threadIdx.x >> 3;
   int tilesperline = context.width >> 3;
   int xt = idx0 % tilesperline,yt = idx0/tilesperline;
   int px = (xt << 3)+tx,py = (yt << 2)+ty;
   int pidx =(px+py*context.width);
   px = context.width-px;
   py = context.height-py;
```

```
RNG genrand(pidx,(clock()*pidx*8191) ^ 140167);
int spp = context.SampleCount;
float rcpw = 1.0f/context.width;
float u = (float)px*rcpw-0.5f;
float v = (float)(py+(context.width-context.height)*
   0.5f)*rcpw-0.5f;
float3 E = make_float3(0,0,0),throughput,O,D;
bool firsthit = true;
int rays = 0;
float3 prevabs;
int paths = 0,curdepth = 0;
// path loop
for (int p = 0; p < spp; p++)
{
   int maxdepth = (__clz((spp-1)-p)-__clz(spp))+1;
   bool restart = true;
   // trace path segment
   for (int depth = 0; depth < maxdepth; depth++)
   {
      if (restart)
      {
         // construct primary ray
         CreatePrimary(O,D);
         prevabs = make_float3(0,0,0);
         firsthit = true,restart = false;
         throughput = make_float3(1,1,1);
         curdepth = 0,paths++;
      }
      int prim = 0;
      float2 UV,BC;
      float dist = 1000000;
      bool backfaced = false;
      O += D*EPSILON; // prevent intersection at dist = 0
      intersect<false,true>(O,D,dist,BC,prim,backfaced);
      rays++;
      O += D*dist;
      if (prim == -1)
      {
         // path left scene
         E += throughput*GetSkySample(D);
         TERMINATE;
      }
      // absorbance
      if (prevabs.x || prevabs.y || prevabs.z)
      {
         throughput *= make_float3(
            __expf(prevabs.x*-dist),
            __expf(prevabs.y*-dist),
            __expf(prevabs.z*-dist));
         prevabs = make_float3(0,0,0);
      }
      Triangle& tri = context.Triangles[prim];
      const TracerMaterial mat =
```

```
   context.Materials[tri.GetMaterialIdx()];
if (mat.flags & TracerMaterial::EMITTER)
{
   if (firsthit & (!backfaced)) E += throughput*mat.EmissiveColor;
   TERMINATE;
}
float3 matcol = tri.GetMaterialColor(mat,BC,UV);
float3 N = tri.GetNormal(mat,BC,UV)*
   (backfaced ? -1 : 1);
// handle diffuse materials
float3 wo = D*-1.0f;
float3 L,LN,LColor;
float area;
RandomPointOnLight(L,LN,LColor,genrand,area);
L -= O;
float sqdist = dot(L,L),ldist = sqrtf(sqdist);
L *= 1.0f/ldist;
float NdotL = dot(N,L),LNdotL = -dot(LN,L);
if ((NdotL>0) && (LNdotL>0) && context.lightcount>0)
{
   bool backface;
   int sprim;
   float2 SBC;
   ldist *= 0.99f;
   intersect<true,false>(O+L*EPSILON,L,ldist,SBC,
      sprim,backface);
   rays++;
   if (sprim == -1)
   {
      float lightPdf = (LNdotL > EPSILON) ?
         (sqdist/(LNdotL*area*context.lightcount)) : 0.0f;
      if (lightPdf > 0)
      {
         float3 f = matcol*INVPI*0.5f;
         E += throughput*f*LColor*NdotL/lightPdf;
      }
   }
}
// bsdf sampling
float3 f;
float bsdfPdf;
D = DiffuseReflection(genrand,N);
bsdfPdf = LambertPdf(D,N);
f = matcol/PI*0.5f;
if (bsdfPdf <= 0) TERMINATE;
// russian roulette
if (curdepth > 1)
{
   float p = max(EPSILON,min(0.5f,
      (throughput.x+throughput.y+throughput.z)*0.333f));
   if (genrand() > p) TERMINATE;
   throughput /= p;
}
```

```
            throughput *= f*dot(D,N)/bsdfPdf;
            firsthit = false;
         }
      }
      context.RenderTarget[pidx]=make_float4(E*(1.0f/(float)paths),*(float*)&rays);
   }
```

C

APPENDIX

C.1 MBVH/RS TRAVERSAL

This appendix contains the source code for the kernel of the streaming path tracer presented in chapter 5.

```
void MBVHTracer::Traverse(MBVHRaylist* packet)
{
   struct ERay { float4 rdx4,rdy4,rdz4,ox4,oy4,oz4; };
   ERay ray[400];
   Ray* rays=packet->ray;
   int numrays=packet->GetRayCount();
   uint16 rayStack[4][ACTIVE_RAY_STACK_SIZE];
   union { uint rayHead[4]; float4i rayHead4; };
   rayHead[0]=numrays;
   rayHead[1]=rayHead[2]=rayHead[3]=0;
   MBVHTask taskStack[1024];
   int tasks=1;
   MBVH::Node* root=mbvh->GetRoot();
   taskStack[0].node=root;
   taskStack[0].data=numrays<<8;
   for(int i=0; i<numrays; ++i)
   {
      rayStack[0][i]=i;
      ray[i].rdx4=set4(packet->ray[i].rDx);
      ray[i].rdy4=set4(packet->ray[i].rDy);
      ray[i].rdz4=set4(packet->ray[i].rDz);
      ray[i].ox4=set4(packet->ray[i].O.x);
      ray[i].oy4=set4(packet->ray[i].O.y);
      ray[i].oz4=set4(packet->ray[i].O.z);
   }
   float4 mintmin=set4(0.00001f);
   float4i one=set4i(1);
   float4i izero=zero4i();
   union { float4 idxmask4; uint idxmask[4]; };
   union { float4 idxadd4; uint idxadd[4]; };
   idxmask[0]=idxmask[1]=idxmask[2]=idxmask[3]=0xfffffffc;
   idxadd[0]=0,idxadd[1]=1,idxadd[2]=2,idxadd[3]=3;
   while (tasks)
   {
      MBVH::Node* node=taskStack[--tasks].node;
      int SIMDlane=taskStack[tasks].data&255;
      int numRays=taskStack[tasks].data>>8;
      if (node->child != 0)
      {
         rayHead[SIMDlane] -= numRays;
         uint16* list=&rayStack[SIMDlane][rayHead[SIMDlane]];
```

```
union{ float4i numActive4; int numActive[4]; };
numActive4=izero;
float4 tdist4=zero4();
for(int i=0 ; i<numRays ; i++)
{
   int rayid=list[i];
   ERay* r=&ray[rayid];
   float4 tx0=mul4(sub4(node->bmin[0].v4,
      r->ox4),r->rdx4);
   float4 tx1=mul4(sub4(node->bmax[0].v4,
      r->ox4),r->rdx4);
   float4 txn=min4(tx0,tx1);
   float4 txf=max4(tx0,tx1);
   float4 tmin1=max4(mintmin,txn);
   float4 tmax1=min4(set4(packet->ray[rayid].dist),
      txf);
   float4 ty0=mul4(sub4(node->bmin[1].v4,r->oy4),
      r->rdy4);
   float4 ty1=mul4(sub4(node->bmax[1].v4,r->oy4),
      r->rdy4);
   float4 tyn=min4(ty0,ty1);
   float4 tyf=max4(ty0,ty1);
   float4 tmin2=max4(tmin1,tyn);
   float4 tmax2=min4(tmax1,tyf);
   float4 tz0=mul4(sub4(node->bmin[2].v4,r->oz4),
      r->rdz4);
   float4 tz1=mul4(sub4(node->bmax[2].v4,r->oz4),
      r->rdz4);
   float4 tzn=min4(tz0,tz1);
   float4 tzf=max4(tz0,tz1);
   float4 tmin3=max4(tmin2,tzn);
   float4 tmax3=min4(tmax2,tzf);
   float4 result4=cmple4(tmin3,tmax3);
   tdist4=add4(tdist4,and4(tmin3,result4));
   float4i add4=and4i(*((float4i*)&result4),one);
   numActive4=add4i(numActive4,add4);
   for (int j=0; j<4; j++)
      rayStack[j][rayHead[j]]=rayid;
   rayHead4=add4i(rayHead4,add4);
}
// sort intersected nodes
union { float4i amask; float4 cmask; };
amask=cmpgt4i(numActive4,izero);
int mask=movemask4(cmask);
int count=(mask&1)+((mask>>1)&1) +
   ((mask>>2)&1)+((mask>>3)&1);
if (count==1)
{
   for (int j=0; j<4; j++,mask>>=1) if (mask&1)
   {
      taskStack[tasks].node=&node->child[j];
      taskStack[tasks++].data=(numActive[j]<<8)+j;
   }
```

```
   }
   else
   {
      // sort
      float4 v0,v1,v2,v3,t;
      v1=v2=zero4();
      v0=or4(and4(cmask,tdist4),
         andnot4(cmask,set4(FLT_MAX)));
      v0=or4(and4(v0,idxmask4),idxadd4);
      v1=movelh4(v1,v0),t=v0;
      v0=min4(v0,v1),v1=max4(v1,t);
      v0=movehl4(v0,v1);
      v1=shuff(v1,v0,0x88),t=v0;
      v0=min4(v0,v1),v1=max4(v1,t);
      v2=movehl4(v2,v1);
      v3=v0,t=v2;
      v2=min4(v2,v3),v3=max4(v3,t);
      v0=shuff(movelh4(v1,v3),shuff(v0,v2,0x13),0x2d);
      for (int j=(count-1); j >= 0; j--)
      {
         int idx=((uint*)&v0)[3-j]&3;
         taskStack[tasks].node=&node->child[idx];
         taskStack[tasks++].data=(numActive[idx]<<8)+idx;
      }
   }
}
else
{
   rayHead[SIMDlane] -= numRays;
   uint16* list=&rayStack[SIMDlane][rayHead[SIMDlane]];
   int i=0;
   for(; i<numRays>>2; i++)
   {
      int rayid=i<<2;
      Ray* r[4]={ &rays[list[rayid+0]],
         &rays[list[rayid+1]],
         &rays[list[rayid+2]],
         &rays[list[rayid+3]] };
      // AoS => SoA conversion
      float4 v1=shuff(r[0]->O.xyzw, r[1]->O.xyzw,68);
      float4 v2=shuff(r[0]->O.xyzw, r[1]->O.xyzw,238);
      float4 v3=shuff(r[2]->O.xyzw, r[3]->O.xyzw,68);
      float4 v4=shuff(r[2]->O.xyzw, r[3]->O.xyzw,238);
      float4 ox4=shuff(v1,v3,136);
      float4 oy4=shuff(v1,v3,221);
      float4 oz4=shuff(v2,v4,136);
      float4 v5=shuff(r[0]->D.xyzw, r[1]->D.xyzw,68);
      float4 v6=shuff(r[0]->D.xyzw, r[1]->D.xyzw,238);
      float4 v7=shuff(r[2]->D.xyzw, r[3]->D.xyzw,68);
      float4 v8=shuff(r[2]->D.xyzw, r[3]->D.xyzw,238);
      float4 dx4=shuff(v5,v7,136);
      float4 dy4=shuff(v5,v7,221);
      float4 dz4=shuff(v6,v8,136);
```

```
   for(int j=0 ; j<node->pcount ; j++)
   {
      Triangle* p=&prim[node->pidx[j]];
      float4 n4x=set4(p->N.x), n4y=set4(p->N.y);
      float4 n4z=set4(p->N.z), nw4=set4(p->N.w);
      float4 d4=DOT128(dx4,dy4,dz4,n4x,n4y,n4z);
      float4 d24=sub4(nw4,
      DOT128(ox4,oy4,oz4,n4x,n4y,n4z));
      float4 fres1=or4(cmpgt4(d4,zero4()),
         xor4(d24,d4));
      if (movemask4(fres1)==15) continue;
      float4 ux4=set4(p->u.x), uy4=set4(p->u.y);
      float4 uz4=set4(p->u.z), uw4=set4(p->u.w);
      float4 vx4=set4(p->v.x), vy4=set4(p->v.y);
      float4 vz4=set4(p->v.z), vw4=set4(p->v.w);
      float4 p4x=add4(mul4(d4,ox4), mul4(d24,dx4));
      float4 p4y=add4(mul4(d4,oy4), mul4(d24,dy4));
      float4 p4z=add4(mul4(d4,oz4), mul4(d24,dz4));
      float4 vu4=add4(DOT128(p4x,p4y,p4z,ux4,uy4,uz4),
         mul4(d4,uw4));
      float4 v24=sub4(d4,vu4);
      float4 fres2=or4(xor4(vu4,v24),fres1);
      float4 vv4=add4(DOT128(p4x,p4y,p4z,vx4,vy4,vz4),
         mul4(d4,vw4));
      float4 v34=sub4(sub4(d4,vu4),vv4);
      float4 fres3=or4(fres2,xor4(vv4,v34));
      if (movemask4(fres3)==15) continue;
      float4 T4=mul4(d24,fastrcp(d4));
      float4i mask=cmpgt4i(*(float4i*)&fres3,izero);
      for (int k=0; k<4; k++)
      {
         float dist=((float*)&T4)[k];
         if ((dist<r[k]->dist) && (((uint*)&mask)[k]))
         {
            r[k]->dist=dist;
            r[k]->tri=p;
         }
      }
   }
}
for(i<<=2; i<numRays ; i++)
{
   Ray* r=&rays[list[i]];
   for(int j=0; j<node->pcount; j++)
   {
      Triangle* p=&prim[node->pidx[j]];
      float d=r->D.Dot(p->N);
      if (d > 0) continue;
      float d2=p->N.w-(r->O.Dot(p->N));
      float v1=d * r->dist;
      if (((*(uint*)&d2)>>31) != ((*(uint*)&v1)>>31))
         continue;
      vector3 P=d * r->O+d2 * r->D;
```

```
            float u=P.Dot(p->u)+d * p->u.w,v2=d-u;
            if (((*(uint*)&u)>>31) != ((*(uint*)&v2)>>31))
               continue;
            float v=P.Dot(p->v)+d * p->v.w,v3=d-u-v;
            if (((*(uint*)&v)>>31) != ((*(uint*)&v3)>>31))
               continue;
            float T=d2/d;
            if (T<r->dist) r->dist=T, r->tri=p;
         }
      }
   }
}
```

D

APPENDIX

D.1 GPU PATH TRACER DATA

These tables contain the raw data used in chapter 6. The left column contains brief codes that represent the rendering techniques. They are:

RR Reference path tracer with Russian roulette and direct light estimation.

DPT Deterministic path termination algorithm.

RR/RIS Reference path tracer with improved direct light estimation using RIS.

SEG Generalized path regeneration algorithm using the segment loop.

SEG/RIS Combination of SEG and RIS.

RR/SNG Single diffuse bounce version of the reference path tracer.

MIS/RIS Multiple importance sampling combined with RIS.

MIS/RIS/NOP Version of the MIS algorithm that does not always complete the last path (resulting in some bias).

For each scene and technique, we measured performance in terms of millions of ray segments per second (the most objective performance measurement) as well as milliseconds per generated frame. Both figures are the average of multiple rendered frames. We also measured the average error of each render, compared to a reference frame that was rendered using the same technique, using many samples per pixel (tens of thousands). The used error metric is the root mean squared error metric (rmse).

Scene 1: Sponza	4spp			8spp			16spp			32spp			64spp			128spp		
	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays
RR	39	36.22	72.6	77	27.39	72.2	153	20.72	71.8	305	15.37	71.6	608	11.06	71.6	1212	7.91	71.8
DPT	30	40.69	83.0	66	31.95	77.0	138	24.90	74.9	282	18.90	74.0	570	13.88	73.5	1144	10.11	73.4
RR/RIS	48	24.02	71.4	95	18.71	70.4	188	13.92	70.6	374	10.06	70.6	745	7.20	70.7	1489	5.14	70.7
DPT/RIS	36	29.55	82.9	80	23.63	76.4	167	18.02	74.3	341	13.24	73.6	688	9.59	73.3	1385	6.89	73.0
SEG	39	36.43	77.3	73	27.79	74.5	154	20.87	70.5	319	15.49	67.8	652	11.15	66.1	1317	8.00	65.3
SEG/RIS	46	24.21	80.2	87	19.16	74.2	185	14.15	70.5	382	10.19	68.3	780	7.25	66.8	1577	5.17	66.0
RR/SNG	35	36.12	80.1	69	27.45	79.5	138	20.71	78.6	274	15.41	78.7	546	11.09	78.8	1090	7.94	78.8
RR/RIS/SNG	43	24.11	78.6	84	18.73	78.5	167	13.96	78.4	333	10.09	78.3	664	7.22	78.4	1327	5.17	78.3
RR/SPT	34	31.42	82.8	66	23.01	78.6	129	17.68	76.6	255	13.63	78.0	510	10.52	77.8	1017	7.97	78.0
MIS/RIS	52	24.14	72.0	98	18.66	71.3	191	13.94	70.9	377	10.06	70.8	749	7.20	70.7	1492	5.13	70.7
MIS/RIS/NOP	48	24.04	71.4	95	18.66	70.4	188	13.97	70.6	374	10.05	70.6	745	7.19	70.7	1487	5.14	70.8

Scene 2: Escher	4spp			8spp			16spp			32spp			64spp			128spp		
	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays
RR	62	35.76	75.8	123	28.63	75.4	245	22.88	75.2	488	18.07	75.2	976	14.17	75.1	1946	10.93	75.1
DPT	23	45.73	121.3	51	38.18	111.1	107	31.44	107.8	220	25.30	105.8	444	20.43	105.3	891	16.35	105.2
RR/RIS	75	26.26	71.2	149	20.82	70.2	296	15.92	70.1	588	12.13	70.4	1174	8.88	70.4	2348	6.69	69.9
DPT/RIS	27	35.79	110.4	60	28.61	103.5	126	22.76	100.3	256	18.17	99.7	518	13.94	99.0	1042	10.71	98.7
SEG	39	41.43	88.1	78	33.55	83.4	157	27.17	80.4	316	21.69	78.6	634	17.22	77.7	1268	13.48	77.4
SEG/RIS	46	31.02	84.0	91	24.52	79.4	185	19.45	76.3	370	15.09	75.3	742	11.41	74.6	1486	8.45	74.2
RR/SNG	60	35.64	77.2	118	28.64	77.6	235	22.81	77.3	468	18.16	77.4	936	14.18	77.2	1870	10.93	77.1
RR/RIS/SNG	72	26.20	73.3	142	20.57	72.6	281	16.03	72.8	561	12.08	72.7	1122	8.90	72.6	2238	6.65	72.3
RR/SPT	48	36.47	90.9	93	29.35	91.9	182	23.08	93.3	360	18.06	93.7	717	14.04	93.9	1434	10.53	93.3
MIS/RIS	77	24.23	72.3	151	18.15	70.9	298	13.42	70.5	592	9.64	70.4	1178	6.84	70.4	2353	4.91	69.9
MIS/RIS/NOP	75	24.08	71.2	149	18.00	70.2	296	13.28	70.2	590	9.59	70.2	1177	6.86	70.2	2351	4.93	69.8

Scene 3: Lucy	4spp			8spp			16spp			32spp			64spp			128spp		
	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays
RR	73	45.24	47.5	142	32.04	47.9	281	22.99	47.9	560	16.71	47.9	1110	12.42	48.2	2211	9.52	48.3
DPT	32	52.27	84.3	70	38.27	80.9	146	28.68	79.1	296	21.66	78.7	597	16.56	78.4	1199	12.87	78.2
RR/RIS	86	25.13	50.6	169	18.42	50.8	335	13.68	50.9	664	10.33	51.2	1321	8.07	51.3	2648	6.68	51.1
DPT/RIS	38	32.58	92.3	84	24.73	86.2	175	18.88	84.2	357	14.52	83.4	720	11.37	83.1	1445	9.24	83.0
SEG	48	49.24	68.4	99	34.73	62.2	194	25.38	62.2	382	18.85	62.3	760	14.33	62.2	1512	11.12	62.3
SEG/RIS	56	29.18	73.2	115	20.64	67.4	228	15.73	67.0	448	12.26	67.5	893	9.88	67.3	1774	8.24	67.6
RR/SNG	64	45.42	51.5	125	32.26	51.7	245	23.08	52.3	485	16.76	52.6	964	12.43	52.7	1925	9.48	52.8
RR/RIS/SNG	74	25.19	55.6	145	18.34	56.2	286	13.73	56.7	568	10.38	56.9	1132	7.98	56.9	2255	6.55	57.1
RR/SPT	50	48.43	67.9	95	34.50	69.5	183	24.55	72.1	360	17.93	73.1	711	13.40	73.7	1416	10.21	73.9
MIS/RIS	89	25.19	52.3	173	18.33	51.2	338	13.55	51.2	667	10.18	51.3	1327	7.98	51.3	2644	6.45	51.3
MIS/RIS/NOP	86	25.08	50.6	169	18.24	50.8	335	13.58	50.9	665	10.27	51.1	1325	8.01	51.2	2637	6.73	51.3

Scene 4: Aztec	4spp			8spp			16spp			32spp			64spp			128spp		
	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays
RR	11	19.69	211.4	21	12.81	206.9	42	8.63	200.4	82	6.03	202.1	164	4.30	200.5	327	3.14	200.3
DPT	8	25.80	246.9	17	16.20	220.1	34	10.82	212.8	69	7.52	206.1	138	5.34	204.2	274	3.85	204.8
RR/RIS	14	19.50	191.1	28	12.68	179.7	55	8.63	176.7	108	6.00	177.0	216	4.28	175.7	431	3.13	175.4
DPT/RIS	11	25.42	201.9	23	16.09	188.0	45	10.81	185.3	91	7.51	180.1	181	5.32	179.5	361	3.85	179.3
SEG	12	19.99	231.8	21	14.12	240.1	43	9.07	228.7	88	6.21	220.4	179	4.37	215.1	359	3.17	213.6
SEG/RIS	15	19.78	208.9	27	14.08	211.2	56	9.05	198.2	112	6.17	195.6	228	4.38	190.8	456	3.16	190.2
RR/SNG	10	19.98	231.4	19	13.01	226.8	38	8.70	219.8	74	6.07	222.1	147	4.34	221.8	293	3.16	221.6
RR/RIS/SNG	13	19.78	204.2	25	12.91	200.0	49	8.70	196.9	98	6.07	193.5	194	4.33	194.0	387	3.15	193.8
RR/SPT	11	19.58	216.6	21	12.56	204.9	41	8.46	199.4	81	5.91	196.8	160	4.23	197.0	320	3.08	196.2
MIS/RIS	15	19.44	186.8	28	12.70	184.0	55	8.61	179.2	109	6.00	176.8	216	4.31	176.4	431	3.14	175.8
MIS/RIS/NOP	15	19.44	178.4	28	12.72	179.8	55	8.60	176.7	108	6.01	177.0	215	4.29	176.5	430	3.13	175.8

Scene 5: MISTest	4spp			8spp			16spp			32spp			64spp			128spp		
	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays	ms	rmse	mrays
RR	40	50.64	109.7	78	39.88	110.7	154	31.16	110.9	306	24.42	111.1	608	19.49	111.5	1212	15.44	111.3
DPT	13	56.85	213.8	29	46.04	195.0	59	36.84	193.1	119	29.20	192.3	241	23.25	190.2	482	18.54	190.3
RR/RIS	47	32.73	99.2	92	24.34	98.9	183	18.50	98.5	363	14.29	98.8	722	11.13	99.1	1439	8.73	98.6
DPT/RIS	16	40.79	177.1	34	31.76	174.0	71	24.62	168.4	145	19.25	165.5	292	15.16	164.7	584	12.04	164.9
SEG	22	54.78	165.2	62	43.47	108.9	380	34.04	34.5	206	26.64	125.8	369	21.02	139.2	712	16.73	143.7
SEG/RIS	26	38.11	148.7	68	27.97	103.4	128	21.35	107.6	236	16.35	115.5	443	12.63	122.0	842	9.89	127.9
RR/SNG	39	50.78	111.0	75	40.05	113.6	149	31.23	113.2	295	24.45	113.8	586	19.50	114.3	1170	15.62	113.9
RR/RIS/SNG	45	32.84	102.5	88	24.39	102.1	176	18.44	101.1	348	14.29	101.8	693	11.15	102.0	1383	8.84	101.3
RR/SPT	31	51.98	128.9	58	41.01	133.1	112	31.64	136.0	220	24.66	138.2	437	19.49	139.0	870	15.57	138.6
MIS/RIS	49	23.84	99.6	94	16.33	99.1	185	11.33	98.5	365	8.07	98.8	726	5.71	98.8	1448	4.16	98.1
MIS/RIS/NOP	47	23.80	99.2	93	16.37	97.9	184	11.37	98.0	364	8.02	98.6	725	5.76	98.7	1447	4.15	98.0

BIBLIOGRAPHY

[1] The 3DO Company, 1991. (Cited on page 2.)

[2] 3D Realms. Duke Nukem 3D, 1996. URL http://www.3drealms.com. (Cited on page 3.)

[3] 3dfx Interactive. VooDoo Graphics. San Jose, CA, USA, 1996. URL http://www.3dfx.com. (Cited on page 3.)

[4] M. Abrash. *Michael Abrash's Graphics Programming Black Book (Special Edition)*. Coriolis Group Books, 1997. ISBN 1576101746. (Cited on page 76.)

[5] S. J. Adelson and L. F. Hodges. Generating Exact Ray-Traced Animation Frames by Reprojection. *IEEE Computer Graphics Applications*, 15(3):43–52, May 1995. ISSN 0272-1716. (Cited on page 6.)

[6] T. Aila and S. Laine. Alias-Free Shadow Maps. In *Proc. Eurographics Symposium on Rendering 2004*, pages 161–166. Eurographics Association, 2004. (Cited on page 54.)

[7] T. Aila and S. Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-603-8. (Cited on pages 96, 122, and 124.)

[8] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008. ISBN 987-1-56881-424-7. (Cited on page 4.)

[9] J. Amanatides and A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *In Eurographics '87*, pages 3–10, 1987. (Cited on page 22.)

[10] J. Ante. Unity 3D. http://www.unity3d.com, 2005. (Cited on page 33.)

[11] A. Appel. Some Techniques for Shading Machine Renderings of Solids. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45, New York, NY, USA, 1968. ACM. (Cited on pages 12 and 37.)

[12] J. Arvo. Backward Ray Tracing. In *In ACM SIGGRAPH '86 Course Notes - Developments in Ray Tracing*, pages 259–263, 1986. (Cited on page 54.)

[13] J. Arvo. Linear-time Voxel Walking for Octrees. Ray Tracing News 12(1), 1988. (Cited on page 24.)

[14] J. Arvo and D. Kirk. Particle Transport and Image Synthesis. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, pages 63–66, New York, NY, USA, 1990. ACM. ISBN 0-89791-344-2. (Cited on pages 15 and 107.)

[15] J. Arvo, P. Hanrahan, H. W. Jensen, D. Mitchell, M. Pharr, P. Shirley, and M. Fajardo. State of the Art in Monte Carlo Ray Tracing for Realistic Image Synthesis. Siggraph 2001 course, 2001. (Cited on page 14.)

[16] ATI. Stream Technology, 2010. URL www.amd.com/stream. (Cited on page 97.)

[17] B. G. Baumgart. Winged Edge Polyhedron Representation. Technical report, Stanford University, Stanford, CA, USA, 1972. (Cited on page 57.)

[18] K. Beason. SmallPT, 2007. URL www.kevinbeason.com/smallpt. (Cited on pages 97 and 124.)

[19] P. Bekaert. *Hierarchical and Stochastic Algorithms for Radiosity*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, 1999. (Cited on page 14.)

[20] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. ISBN 0-13-711821-X. (Cited on page 32.)

[21] C. Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, Saarbrücken, Germany, January 2006. (Cited on pages 42, 80, and 122.)

[22] C. Benthin, I. Wald, S. Woop, M. Ernst, and W. R. Mark. Combining Single and Packet Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics*, 99 (PrePrints), 2011. ISSN 1077-2626. (Cited on page 78.)

[23] J. L. Bentley. Multidimensional Binary Search Trees used for Associative Searching. *Commun. ACM*, 18(9):509–517, September 1975. ISSN 0001-0782. (Cited on page 22.)

[24] T. Berger-Perrin. The Once Known As SmallPT, 2009. URL http://code.google.com/p/tokaspt. (Cited on pages 97 and 124.)

[25] L. Bergman, H. Fuchs, E. Grant, and S. Spach. Image Rendering by Adaptive Refinement. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 29–37, New York, NY, USA, 1986. ACM. ISBN 0-89791-196-2. (Cited on page 6.)

[26] J. Bigler, A. Stephens, and S. G. Parker. Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*,

pages 187–196, Los Alamitos, CA, USA, 2006. IEEE Computer Society. ISBN 1-4244-0693-5. (Cited on page 6.)

[27] J. Bikker. Real-time Ray Tracing through the Eyes of a Game Developer. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 1–10, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1629-5. (Cited on pages 6, 31, 37, 61, 78, and 122.)

[28] J. Bikker. Improving Data Locality for Efficient In-Core Path Tracing. *Computer Graphics Forum*, 2012. (Cited on page 75.)

[29] J. Bikker and R. Reijerse. A Precalculated Point Set for Caching Shading Information. In *Eurographics 2009 - Short Papers*, pages 65–68, Munich, Germany, 2009. Eurographics Association. (Cited on page 53.)

[30] D. Binks. Dynamic Resolution Rendering, 2011. URL `http://software.intel.com/file/35451`. (Cited on page 32.)

[31] J. F. Blinn and M. E. Newell. Texture and Reflection in Computer Generated Images. *Commun. ACM*, 19(10):542–547, October 1976. ISSN 0001-0782. (Cited on page 4.)

[32] R. Borgo and K. Brodlie. State of the Art Report on GPU Visualization. *Virtual Reality*, 2009. (Cited on page 95.)

[33] S. Boulos, D. Edwards, J. Dylan Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald. Interactive Distribution Ray Tracing. *Technical Report, SCI Institute, University of Utah, No UUSCI-2006-022*, 2006. (Cited on page 7.)

[34] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald. Packet-based Whitted and Distribution Ray Tracing. In *GI '07: Proceedings of Graphics Interface 2007*, pages 177–184, New York, NY, USA, 2007. ACM. ISBN 978-1-56881-337-0. (Cited on pages 75 and 122.)

[35] S. Boulos, I. Wald, and C. Benthin. Adaptive Ray Packet Reordering. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2008*, pages 131–138, Los Alamitos, CA, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2741-3. (Cited on page 79.)

[36] D. Braben and I. Bell. Elite, 1984. URL `http://www.elite.com`. (Cited on page 2.)

[37] T. Bramwell. Doom 4 Three Times Rage Visual Quality, 2011. URL `http://www.eurogamer.net/articles/doom-4-three-times-rage-visual-quality`. (Cited on page 124.)

[38] J. Bresenham. Algorithm for Computer Control of a Digital Plotter. *IBM Systems Journal*, 4(1):25–30, 1965. (Cited on page 22.)

[39] R. Bringhurst. *The Elements of Typographic Style*. Version 2.5. Hartley & Marks, Publishers, Point Roberts, WA, USA, 2002. (Cited on page 195.)

[40] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010. ISBN 0136108040, 9780136108047. (Cited on page 76.)

[41] B. Budge, T. Bernardin, J. A. Stuart, S. Sengupta, K. I. Joy, and J. D. Owens. Out-of-core Data Management for Path Tracing on Hybrid Resources. *Computer Graphics Forum*, 28(2):385–396, 2009. (Cited on page 79.)

[42] J. Carmack. Catacomb 3D, 1991. URL `http://www.id-software.com`. (Cited on page 2.)

[43] J. Carmack. Quake II, 1997. URL `http://www.quake.com`. (Cited on page 3.)

[44] J. Carmack. ID Software Debuts ID Tech 5 Press Release. http://www.idsoftware.com/business/press, 2007. (Cited on page 32.)

[45] N. A. Carr, J. D. Hall, and J. C. Hart. The Ray Engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 37–46, Aire-la-Ville, Switzerland, 2002. Eurographics Association. ISBN 1-58113-580-7. (Cited on page 6.)

[46] J. H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Commun. ACM*, 19(10):547–554, October 1976. ISSN 0001-0782. (Cited on page 32.)

[47] D. Cline, J. Talbot, and P. Egbert. Energy Redistribution Path Tracing. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 1186–1195, New York, NY, USA, 2005. ACM. (Cited on page 97.)

[48] M. F. Cohen, J. Wallace, and P. Hanrahan. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Inc., San Diego, CA, USA, 1993. ISBN 0-12-178270-0. (Cited on page 14.)

[49] Commodore. Commodore Amiga 1000, 1985. (Cited on page 3.)

[50] R. L. Cook. Stochastic Sampling in Computer Graphics. *ACM Trans. Graph.*, 5(1):51–72, January 1986. ISSN 0730-0301. (Cited on page 56.)

[51] R. L. Cook, T. Porter, and L. Carpenter. Distributed Ray Tracing. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 137–145, New York, NY, USA, 1984. ACM. ISBN 0-89791-138-5. (Cited on pages 5, 13, and 37.)

[52] Valve Corporation. Half-Life 2, 2004. URL `http://www.valve-software.com`. (Cited on page 3.)

[53] K. Crane. Importance Sampling for Monte Carlo Ray Tracing. California Institute of Technology, 2006. (Cited on page 19.)

[54] K. Crane. Bias in Rendering. California Institute of Technology, 2006. (Cited on page 19.)

[55] Chris Crawford. *The Art of Computer Game Design*. Osborne/McGraw-Hill, Berkeley, CA, USA, 1984. ISBN 0881341177. (Cited on page 2.)

[56] Crytek. Far Cry, 2004. (Cited on page 3.)

[57] C. Dachsbacher, P. Slusallek, T. Davidovic, T. Engelhardt, M. Phillips, and I. Georgiev. 3D Rasterization - Unifying Rasterization and Ray Casting. *Technical Report*, 2009. (Cited on page 13.)

[58] T. Dahmen, C. Vogelgesang, J. Guenther, J. Schmittler, and C. Benthin. Oasen, 2004. (Cited on page 123.)

[59] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with Streams. In *Proceedings of the 2003 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'03)*, November 2003. (Cited on page 97.)

[60] H. Dammertz, J. Hanika, and A. Keller. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Computer Graphics Forum*, 27(4):1225–1233, 2008. (Cited on pages 23, 28, 80, and 86.)

[61] H. Dammertz, D. Sewtz, J. Hanika, and H. P. A. Lensch. Edge-avoiding A-Trous wavelet transform for fast global illumination filtering. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 67–75, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association. (Cited on page 143.)

[62] Holger Dammertz, Alexander Keller, and Hendrik Lensch. Progressive point-light-based global illumination. *Computer Graphics Forum*, 29(8):2504–2515, 2010. (Cited on page 143.)

[63] Dann-Ball. Ray Trace Fighter, 2011. URL `http://dan-ball.jp/en/javagame/raytracing`. (Cited on page 123.)

[64] K. Debattista, P. Dubla, L. Santos, and A. Chalmers. Wait-Free Shared-Memory Irradiance Caching. *IEEE Comput. Graph. Appl.*, 31(5):66–78, September 2011. ISSN 0272-1716. (Cited on page 55.)

[65] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20:85–95, March 2000. ISSN 0272-1732. (Cited on pages 25 and 77.)

[66] M. A. Z. Dippé and E. H. Wold. Antialiasing through Stochastic Sampling. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '85, pages 69–78, New York, NY, USA, 1985. ACM. ISBN 0-89791-166-0. (Cited on page 56.)

[67] P. Djeu, W. Hunt, R. Wang, I. Elhassan, G. Stoll, and W. R. Mark. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. *ACM Trans. Graph.*, 30(5):115:1–115:26, October 2011. ISSN 0730-0301. (Cited on pages 6 and 38.)

[68] K. Dmitriev, V. Havran, and H. P. Seidel. Faster Ray Tracing with SIMD Shaft Culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut fi¿œr Informatik, Saarbrücken, Germany, December 2004. (Cited on page 25.)

[69] M. Doggett. AMD's Radeon HD 2900. *Graphics Hardware 2007*, August 2007. (Cited on page 77.)

[70] A. S. Douglas. Noughts and Crosses. EDSAC, 1952. (Cited on page 2.)

[71] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Alrich, and M. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. *Tech. Rep. UCRL-JC-127870*, August 1997. (Cited on page 32.)

[72] K. A. Duke and W. A. Wall. A Professional Graphics Controller. *IBM Systems Journal*, 24(1):14–25, March 1985. ISSN 0018-8670. (Cited on page 3.)

[73] P. DutrÃ©. Global Illumination Compendium, 2001. (Cited on pages 15 and 18.)

[74] L. Dymchenko. AntiPlanet2.0, 2009. (Cited on page 123.)

[75] C. M. Erikson. *Hierarchical Levels of Detail to Accelerate the Rendering of Large Static and Dynamic Polygonal Environments*. PhD thesis, The University of North Carolina at Chapel Hill, 2000. (Cited on page 32.)

[76] M. Ernst. Embree - Photo-Realistic Ray Tracing Kernels, 2012. (Cited on page 6.)

[77] M. Ernst and G. Greiner. Multi Bounding Volume Hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 35–40, 2008. (Cited on pages 23, 28, and 80.)

[78] M. Evans. 3D Monster Maze, 1981. URL `http://www.monstermaze.com`. (Cited on page 2.)

[79] F.A.N. BowlXTreme, 2003. (Cited on page 123.)

[80] J. D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982. ISBN 0-201-14468-9. (Cited on page 11.)

[81] T. Foley and J. Sugerman. KD-tree Acceleration Structures for a GPU Raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM. ISBN 1-59593-086-8. (Cited on pages 6, 96, and 122.)

[82] H. Friedrich, J. Günther, A. Dietrich, M. Scherbaum, H. P. Seidel, and P. Slusallek. Exploring the Use of Ray Tracing for Future Games. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 41–50, New York, NY, USA, 2006. ACM. ISBN 1-59593-386-7. (Cited on pages 7, 31, and 37.)

[83] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated Ray-Tracing System. *IEEE Computer Graphics and Applications*, 6:16–26, 1986. ISSN 0272-1716. (Cited on page 22.)

[84] T. Furtak, J. Amaral, N. José, and R. Niewiadomski. Using SIMD Registers and Instructions to Enable Instruction-Level Parallelism in Sorting Algorithms. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 348–357, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-667-7. (Cited on page 30.)

[85] M. N. Gamito and S. C. Maddock. Accurate Multidimensional Poisson-disk Sampling. *ACM Trans. Graph.*, 29:8:1–8:19, December 2009. ISSN 0730-0301. (Cited on page 56.)

[86] K. Garanzha and C. Loop. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum*, 29:289–298, 2010. (Cited on pages 79 and 96.)

[87] Eduardo S. L. Gastal and Manuel M. Oliveira. Adaptive Manifolds for Real-Time High-Dimensional Filtering. *ACM TOG*, 31(4):33:1–33:13, 2012. Proceedings of SIGGRAPH 2012. (Cited on page 143.)

[88] P. N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. Technical report, NVidia, 2009. (Cited on page 77.)

[89] A. S. Glassner. Space Subdivision for Fast Ray Tracing. *IEEE Computer Graphics and Applications*, 4:15–22, 1984. (Cited on page 21.)

[90] A. S. Glassner. A Model for Fluorescence and Phosphorescence. *In Proceedings Fifth Eurographics Workshop on Rendering (1994)*, pages 57–68, 1994. (Cited on page 11.)

[91] J. Goldsmith and J. Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics Applications*, 7(5):14–20, 1987. ISSN 0272-1716. (Cited on page 22.)

[92] T. T. Goldsmith and E. R. Mann. Cathode Ray Amusement Device. U.S. Patent no. 2 455 992, 1948. (Cited on page 2.)

[93] S. W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, 12:399–401, September 1966. (Cited on page 62.)

[94] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the Interaction of Light between Diffuse Surfaces. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 213–222, New York, NY, USA, 1984. ACM. ISBN 0-89791-138-5. (Cited on page 10.)

[95] H. Gouraud. Continuous Shading of Curved Surfaces. *IEEE Trans. Comput.*, 20(6):623–629, June 1971. ISSN 0018-9340. (Cited on page 54.)

[96] C. Gribble and K. Ramani. Coherent Ray Tracing via Stream Filtering. In *Interactive Ray Tracing (Aug. 2008), no. 3*, pages 59–66, 2008. (Cited on page 79.)

[97] J. Günther, S. Popov, H. P. Seidel, and P. Slusallek. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118, September 2007. (Cited on pages 6 and 96.)

[98] N. Guy. Photonotes Dictionary of Photography, 2004. URL `http://www.photonotes.org`. (Cited on page 32.)

[99] T. Hachisuka, W. Jarosz, R. P. Weistroffer, K. Dale, G. Humphreys, M. Zwicker, and H. W. Jensen. Multidimensional Adaptive Sampling and Reconstruction for Ray Tracing. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 33:1–33:10, New York, NY, USA, 2008. ACM. ISBN 978-1-4503-0112-1. (Cited on page 56.)

[100] P. Hanrahan. Using Caching and Breadth-first Search to Speed up Ray-tracing. In *Proceedings on Graphics Interface '86/Vision Interface '86*, pages 56–61, Toronto, Ont., Canada, Canada, 1986. Canadian Information Processing Society. (Cited on page 79.)

[101] P. Hanrahan, D. Salzman, and L. Aupperle. A Rapid Hierarchical Radiosity Algorithm. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '91, pages 197–206, New York, NY, USA, 1991. ACM. ISBN 0-89791-436-8. (Cited on page 14.)

[102] J. M. Hasenfratz, M. Lapierre, N. Holzschuch, and F. Sillion. A survey of Real-Time Soft Shadows Algorithms. *Computer Graphics Forum*, 22(4):753–774, dec 2003. (Cited on page 4.)

[103] V. Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech

Technical University in Prague, November 2000. URL http://www.cgg.cvut.cz/~havran/phdthesis.html. (Cited on page 24.)

[104] V. Havran, T. Kopal, J. Bittner, and J. Žára. Fast Robust BSP Tree Traversal Algorithm for Ray Tracing. *Journal of Graphics Tools*, 2(4):15–23, January 1998. ISSN 1086-7651. (Cited on page 22.)

[105] M. Hašan, F. Pellacini, and K. Bala. Matrix Row-Column Sampling for the Many-Light Problem. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM. (Cited on page 60.)

[106] H. Hey and W. Purgathofer. Real-Time Rendering of Globally Illuminated Soft Glossy Scenes With Directional Light Maps. Technical Report TR-186-2-02-05, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, March 2002. (Cited on page 54.)

[107] M. Hollis, D. Doak, and D. Botwood. GoldenEye 007. Published by Nintendo, 1997. (Cited on page 54.)

[108] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-D Tree GPU Raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-628-8. (Cited on pages 6, 96, and 122.)

[109] P. Hsieh. Programming Optimization, 2004. URL http://www.azillionmonkeys.com/qed/optimize.html. (Cited on page 20.)

[110] W. Hunt and W. R. Mark. Adaptive Acceleration Structures in Perspective Space. In *In 2008 IEEE Symposium on Interactive Ray Tracing. IEEE*, 2008. (Cited on page 13.)

[111] W. Hunt, W. R. Mark, and G. Stoll. Fast kD-tree Construction with an Adaptive Error-Bounded Heuristic. *Symposium on Interactive Ray Tracing*, 0: 81–88, 2006. (Cited on page 31.)

[112] J. Hurley, E. Kapustin, A. Reshetov, and A. Soupikov. Fast Ray Tracing for Modern General Purpose CPU. In *In Proceedings of Graphicon*, 2002. (Cited on page 22.)

[113] Atari Inc. Hard Drivin', 1989. URL http://www.atari.com. (Cited on page 3.)

[114] Intel. Intel Many Integrated Core Architecture, 2010. URL http://download.intel.com/pressroom/archive/reference/ISC2010Skaugenkeynote.pdf. (Cited on pages 77 and 78.)

[115] T. Ize, I. Wald, C. Robertson, and S. G Parker. An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 27–55, 2006. (Cited on page 22.)

[116] T. Ize, I. Wald, and S. G. Parker. Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In Jean M. Favre, Paulo Santos, and Dirk Reiners, editors, *Eurographics Symposium on Parallel Graphics and Visualization, EGPGV 2007, Lugano, Switzerland*, pages 101–108. Eurographics Association, 2007. ISBN 978-3-905673-50-0. (Cited on page 31.)

[117] T. Ize, I. Wald, and S. G. Parker. Ray Tracing with the BSP Tree. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, Washington, DC, USA, 2008. IEEE Computer Society. (Cited on page 22.)

[118] T. Ize, C. Brownlee, and C. D. Hansen. Real-Time Ray Tracer for Visualizing Massive Models on a Cluster. In Torsten Kuhlen, Renato Pajarola, and Kun Zhou, editors, *EGPGV*, pages 61–69. Eurographics Association, 2011. ISBN 978-3-905674-32-3. (Cited on page 6.)

[119] D. Pohl C. Vogelsang J. Schmittler, T. Dahmen and P. Slusallek. Ray Tracing for Current and Future Games. In *Proceedings of 34. Jahrestagung der Gesellschaft fur Informatik*, 2004. (Cited on page 7.)

[120] F. W. Jansen. Data structures for Ray Tracing. In *Proceedings of a workshop (Eurographics Seminars on Data structures for raster graphics*, pages 57–73, New York, NY, USA, 1986. Springer-Verlag New York, Inc. ISBN 0-387-16310-7. (Cited on pages 22 and 24.)

[121] H. W. Jensen. Global Illumination using Photon Maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag. ISBN 3-211-82883-4. (Cited on pages 14, 20, and 54.)

[122] L. S. Jensen and R. Golias. Deep-Water Animation and Rendering, 2001. URL http://www.gamasutra.com/gdce/2001/jensen/jensen_03.htm. (Cited on page 4.)

[123] S. Badt Jr. Two Algorithms Taking Advantage of Temporal Coherence in Ray Tracing. *The Visual Computer*, 4:123–132, September 1988. (Cited on page 6.)

[124] Jromang. SmallLuxGPU, 2009. URL http://www.luxrender.net/wiki/SLG. (Cited on pages 97 and 124.)

[125] J. T. Kajiya. The Rendering Equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM. ISBN 0-89791-196-2. (Cited on pages 5, 10, 14, and 37.)

[126] J. Kalojanov and P. Slusallek. A Parallel Algorithm for Construction of Uniform Grids. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 23–28, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-603-8. (Cited on page 31.)

[127] H. Kamiya and A. Inaba. Okami, 2006. (Cited on page 122.)

[128] D. Kanter. AMD's Cayman GPU Architecture, 2010. URL http://www.realworldtech.com/page.cfm?ArticleID=RWT121410213827. (Cited on page 97.)

[129] R. Karrenberg, D. Rubinstein, P. Slusallek, and S. Hack. AnySL: Efficient and Portable Shading for Ray Tracing. In *HPG '10: Proceedings of the Conference on High Performance Graphics*, 2010. (Cited on page 142.)

[130] N. D. Kehtarnavaz and M. Gamadia. *Real-Time Image and Video Processing: From Research to Reality*. Synthesis Lectures on Image, Video, and Multimedia Processing. Morgan & Claypool Publishers, 2006. (Cited on page 32.)

[131] A. Keller. Quasi-Monte Carlo Radiosity. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 101–110, London, UK, 1996. Springer-Verlag. ISBN 3-211-82883-4. (Cited on page 59.)

[132] A. Keller. Instant Radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0-89791-896-7. (Cited on pages 15, 20, 59, and 109.)

[133] A. Keller. *Quasi-Monte Carlo Methods for Realistic Image Synthesis*. PhD thesis, University of Kaiserslautern, Germany, 1998. (Cited on page 24.)

[134] A. Keller and W. Heidrich. Interleaved Sampling. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 269–276, London, UK, 2001. Springer-Verlag. ISBN 3-211-83709-4. (Cited on page 59.)

[135] A. Keller and C. Waechter. To Trace or Not To Trace, That is the Question. Presentation for the Breakpoint 2005 demo party seminar, 2005. (Cited on page 7.)

[136] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: a Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0. (Cited on page 76.)

[137] A. Kensler. Tree Rotations for Improving Bounding Volume Hierarchies. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 73–76, Aug 2008. (Cited on page 31.)

[138] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21:35–46, 2001. ISSN 0272-1732. (Cited on page 97.)

[139] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 2008. URL http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf. (Cited on page 97.)

[140] T. Kollig and A. Keller. Illumination in the Presence of Weak Singularities. In *Monte Carlo And Quasi-monte Carlo Methods*, pages 245–257, 2004. (Cited on page 60.)

[141] J. Křivánek, K. Bouatouch, S. Pattanaik, and J. Zara. Making Radiance and Irradiance Caching Practical: Adaptive Caching and Neighbor Clamping. In *Rendering Techniques*, pages 127–138. Eurographics Association, 2006. ISBN 3-905673-35-5. (Cited on page 55.)

[142] J. Křivánek, P. Gautron, G. Ward, H. W. Jensen, P. H. Christensen, and E. Tabellion. Practical Global Illumination with Irradiance Caching. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 60:1–60:20, New York, NY, USA, 2008. ACM. (Cited on page 55.)

[143] E. Lafortune. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering*. PhD thesis, Katholieke Universiteit Leuven, 1996. (Cited on pages 14, 17, and 97.)

[144] E. Lafortune and Y. Willems. Bidirectional Path Tracing. In *Proc. 3rd International Conference on Computational Graphics and Visualization Techniques (Compugraphics)*, pages 145–153, 1993. (Cited on page 14.)

[145] S. Laine, H. Saransaari, J. Kontkanen, J. Lehtinen, and T. Aila. Incremental Instant Radiosity for Real-Time Indirect Illumination. In *Proceedings of Eurographics Symposium on Rendering 2007*, pages 277–286. Eurographics Association, 2007. (Cited on page 60.)

[146] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009. (Cited on pages 31 and 96.)

[147] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, UK, 2006. Available at http://planning.cs.uiuc.edu/. (Cited on page 60.)

[148] J. Lehtinen, M. Zwicker, E. Turquin, J. Kontkanen, F. Durand, F. Sillion, and T. Aila. A Meshless Hierarchical Representation for Light Transport. *ACM Trans. Graph.*, 27(3), 2008. (Cited on page 54.)

[149] D. Leubke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Elsevier Science, San Francisco, CA, USA, 2003. ISBN 1-55860-838-9. (Cited on page 32.)

[150] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVidia Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28:39–55, March 2008. ISSN 0272-1732. (Cited on page 98.)

[151] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. Turner. Real-time, Continuous Level of Detail Rendering of Height Fields. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'96)*, pages 109–118, August 1996. (Cited on page 32.)

[152] C. Lomont. Introduction to Intel Advanced Vector Extensions. Intel Software Network, 2011. URL http://software.intel.com/en-us/avx/. (Cited on page 77.)

[153] J. Lowensohn. How Epic Fit the Unreal Engine into the iPhone, 2010. URL http://news.cnet.com/8301-27076_3-20000214-248.html. (Cited on page 3.)

[154] Argonaut Software ltd. Starglider 2, 1988. URL http://www.starglider2.com. (Cited on page 2.)

[155] J. D. MacDonald and K. S. Booth. Heuristics for Ray Tracing using Space Subdivision. *The Visual Computer*, 6:153–166, 1990. ISSN 0178-2789. (Cited on page 22.)

[156] E. Mansson, J. Munkberg, and T. Akenine-Möller. Deep Coherent Ray Tracing. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 79–85, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1629-5. (Cited on page 78.)

[157] Microsoft. Microsoft DirectCompute, 2010. URL http://msdn.microsoft.com/directx. (Cited on page 97.)

[158] J. G. Miner, D. Dean, J. C. Decuir, R. H. Nicholson, and A. Tanaka. Personal Computer Apparatus for Block Transfer of Bit-Mapped Image Data. US Patent 4,874,164, 1999. (Cited on page 3.)

[159] M. Mittring. Finding Next Gen: CryEngine 2. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pages 97–121, New York, NY, USA, 2007. ACM. (Cited on page 4.)

[160] S. Miyamoto. Super Mario 64. Published by Nintendo, 1996. (Cited on page 54.)

[161] S. Miyamoto. Super Mario Galaxy. Published by Nintendo, 2007. (Cited on pages 4 and 122.)

[162] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), April 1965. (Cited on page 123.)

[163] M. J. Muuss. RT & REMRT: Shared Memory Parallel and Network Distributed Ray-Tracing Programs. In *USENIX: Proceedings of the Fourth Computer Graphics Workshop*, pages 86–97, October 1987. (Cited on page 44.)

[164] M. J. Muuss. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium*, 1995. (Cited on page 6.)

[165] B. Nam and A. Sussman. A Comparative Study of Spatial Indexing Techniques for Multidimensional Scientific Datasets. In *SSDBM '04: Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, page 171, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2146-0. (Cited on page 23.)

[166] Namco. Galaxian, 1979. URL http://www.galaxian.com. (Cited on page 2.)

[167] Namco. Pole Position, 1982. URL http://www.namco.com. (Cited on page 2.)

[168] P. A. Navratil, D. S. Fussell, C. Lin, and W. R. Mark. Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 95–104, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1629-5. (Cited on page 79.)

[169] G. Newell and J. Weier. Portal 2. Valve Corporation, 2011. (Cited on page 132.)

[170] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992. ISBN 0-89871-295-5. (Cited on pages 15 and 60.)

[171] J. Novák, V. Havran, and C. Daschbacher. Path Regeneration for Interactive Path Tracing. In *The European Association for Computer Graphics 28th Annual Conference: EUROGRAPHICS 2007, short papers*, pages 61–64. The European Association for Computer Graphics, 2010. (Cited on pages 7, 97, 106, and 124.)

[172] NVidia. NVidia NV1, 1995. URL http://www.nvidia.com. (Cited on pages 2 and 3.)

[173] nVIDIA. GeForce 256, 1999. URL http://www.nvidia.com/page/geforce256.html. (Cited on page 3.)

[174] NVidia. Fermi: NVidia's Next Generation CUDA Compute Architecture, 2009. URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. (Cited on page 97.)

[175] NVidia. Design Garage, 2010. URL http://www.nvidia.com. (Cited on pages 97 and 124.)

[176] M. Olano, B. Kuehne, and M. Simmons. Automatic Shader Level of Detail. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 7–14, Aire-la-Ville, Switzerland, 2003. Eurographics Association. ISBN 1-58113-739-7. (Cited on page 32.)

[177] J. A. Oudshoorn. *Ray Tracing as the Future of Computer Games*. PhD thesis, Department of Computer Science, University of Utrecht, 1999. (Cited on pages 7 and 37.)

[178] R. Overbeck, R. Ramamoorthi, and W. R. Mark. Large Ray Packets for Real-time Whitted Ray Tracing. In *IEEE/EG Symposium on Interactive Ray Tracing (IRT)*, pages 41–48, Aug 2008. (Cited on pages 78, 80, 83, 96, and 122.)

[179] J. Owens. Streaming Architectures and Technology Trends. *GPU Gems 2*, 2005. (Cited on page 123.)

[180] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krï¿œger, A. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. (Cited on page 95.)

[181] A. Pajot, L. Barthe, M. Paulin, and P. Poulin. Combinatorial Bidirectional Path-Tracing for Efficient Hybrid CPU/GPU Rendering. *Computer Graphics Forum*, 30(2):315–324, April 2011. (Cited on page 97.)

[182] J. Pantaleoni and D. Luebke. HLBVH - Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry. *Proceedings of HPG 2010*, pages 87–95, 2010. (Cited on pages 31 and 142.)

[183] E. Paquette, P. Poulin, and G. Drettakis. A Light Hierarchy for Fast Rendering of Scenes with Many Lights. In *Computer Graphics Forum (Proceedings of the Eurographics conference)*, pages 63–74. Eurographics, Sep 1998. (Cited on page 60.)

[184] S. Parker, W. Martin, P. P. J. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive Ray Tracing. In *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 119–126, New York, NY, USA, 1999. ACM. ISBN 1-58113-082-1. (Cited on pages 6, 31, and 75.)

[185] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberockand D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. OptiX: a General Purpose Ray Tracing Engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, July 2010. ISSN 0730-0301. (Cited on pages 6 and 97.)

[186] W. J. Paul, P. Bach, M. Bosch, J. Fischer, C. Lichtenau, and J. Röhrig. Real PRAM Programming. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 522–531, London, UK, 2002. Springer-Verlag. ISBN 3-540-44049-6. (Cited on page 78.)

[187] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. ISBN 012553180X. (Cited on page 109.)

[188] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering Complex Scenes with Memory-coherent Ray Tracing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0-89791-896-7. (Cited on pages 75, 79, and 84.)

[189] B. T. Phong. Illumination for Computer Generated Pictures. *Commun. ACM*, 18(6):311–317, June 1975. ISSN 0001-0782. (Cited on pages 11 and 45.)

[190] PhotonStudios. Let There Be Light, 2008. (Cited on page 123.)

[191] J. Pineda. A Parallel Algorithm for Polygon Rasterization. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 17–20, New York, NY, USA, 1988. ACM. ISBN 0-89791-275-6. (Cited on page 13.)

[192] D. Pohl. Ray Tracing and Gaming - Quake 4: Ray Traced Project. http://pcper.com, 2007. (Cited on page 7.)

[193] D. Pohl. Ray Tracing and Gaming - One Year Later. *PC Perspective*, 0, 2008. (Cited on page 37.)

[194] D. Pohl. Light It Up! Quake Wars Gets Ray Traced. *Visual Adrenaline*, (2): 34–39, 2009. (Cited on page 7.)

[195] D. Pohl. Experimental Cloud-based Ray Tracing Using Intel MIC Architecture for Highly Parallel Visual Processing. Intel Software Network, February 2011. (Cited on page 7.)

[196] S. Popov, J. Günther, H. P. Seidel, and P. Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 26(3): 415–424, September 2007. (Proceedings of Eurographics). (Cited on pages 7 and 96.)

[197] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002). (Cited on pages 6, 96, and 122.)

[198] G. Ramanarayanan, J. Ferwerda, B. Walter, and K. Bala. Visual Equivalence: Towards a New Standard for Image Fidelity. *ACM Trans. Graph.*, 26, July 2007. ISSN 0730-0301. (Cited on page 4.)

[199] W. T. Reeves, D. H. Salesin, and R. L. Cook. Rendering Antialiased Shadows with Depth Maps. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 283–291, New York, NY, USA, 1987. ACM. ISBN 0-89791-227-6. (Cited on page 54.)

[200] Refractive. Octane Renderer, 2010. URL www.refractivesoftware.com. (Cited on pages 97 and 124.)

[201] E. Reinhard and F. W. Jansen. Rendering Large Scenes using Parallel Ray Tracing. *Parallel Computer*, 23:873–885, July 1997. ISSN 0167-8191. (Cited on page 78.)

[202] A. Reshetov. Faster Ray Packets - Triangle Intersection through Vertex Culling. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 105–112, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1629-5. (Cited on pages 26, 31, and 78.)

[203] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level Ray Tracing Algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1176–1185, New York, NY, USA, 2005. ACM. (Cited on page 6.)

[204] A. Reshetov, A. Soupikov, and W. R. Mark. Consistent Normal Interpolation. In *ACM SIGGRAPH Asia 2010 papers*, SIGGRAPH ASIA '10, pages 142:1–142:8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0439-9. (Cited on page 6.)

[205] S. M. Rubin and T. Whitted. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 110–116, New York, NY, USA, 1980. ACM. ISBN 0-89791-021-4. (Cited on pages 21 and 23.)

[206] I. Sadeghi, B. Chen, and H. W. Jensen. Coherent Path Tracing. *Journal of Graphics, GPU, and Game Tools*, 14(2):33–43, 2009. (Cited on page 124.)

[207] J. Schmittler, I. Wald, and P. Slusallek. SaarCOR: a Hardware Architecture for Ray Tracing. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 27–36, Aire-la-Ville, Switzerland, 2002. Eurographics Association. ISBN 1-58113-580-7. (Cited on pages 7 and 78.)

[208] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '04, pages 95–106, New York, NY, USA, 2004. ACM. ISBN 3-905673-15-0. (Cited on page 7.)

[209] J. Schmittler, D. Pohl, T. Dahmen, C. Vogelgesang, and P. Slusallek. Realtime Ray Tracing for Current and Future Games. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM. (Cited on pages 7 and 49.)

[210] K. Schwenk, A. Kuijper, J. Behr, and D. Fellner. Practical Noise Reduction for Progressive Stochastic Ray Tracing with Perceptual Control. *IEEE Computer*

Graphics and Applications, 99(PrePrints), 2012. ISSN 0272-1716. (Cited on page 143.)

[211] Sega. Turbo, 1981. URL http://www.sega.com. (Cited on page 2.)

[212] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a Many-Core x86 Architecture for Visual Computing. In *ACM SIGGRAPH 2008 papers*, pages 18:1–18:15, New York, NY, USA, 2008. ACM. ISBN 978-1-4503-0112-1. (Cited on page 77.)

[213] D. Shephard. A Two-Dimensional Interpolation Function for Irregularly-Spaced Data. In *Proceedings of the 1968 23rd ACM national conference*, pages 517–524, New York, NY,USA, 1968. ACM. (Cited on page 67.)

[214] P. Shirley, C. Wang, and K. Zimmerman. Monte Carlo Techniques for Direct Lighting Calculations. *ACM Trans. Graph.*, 15:1–36, January 1996. ISSN 0730-0301. (Cited on page 60.)

[215] F. X. Sillion and C. Puech. *Radiosity and Global Illumination*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994. ISBN 1558602771. (Cited on page 14.)

[216] S. Siltanen and T. Lokki. Diffraction Modeling in Acoustic Radiance Transfer Method. *Acoustical Society of America Journal*, 123(5):3759, 2008. ISSN 1520-8524. (Cited on page 10.)

[217] S. Siltanen, T. Lokki, S. Kiminki, and L. Savioja. The Room Acoustic Rendering Equation. *Journal of the Acoustical Society of America*, 122(3):1624–1635, 2007. (Cited on page 10.)

[218] S. Simha. Super Mario Chip - Inside the 64-bit RISC Processor that Powers the New Nintendo Game Machine. *j-BYTE*, 21(12):59–65, dec 1996. ISSN 0360-5280. (Cited on page 54.)

[219] T. Simon. Tom Clancy's H.A.W.X., 2009. (Cited on page 122.)

[220] P. P. Sloan, J. Kautz, and J. Snyder. Precomputed Radiance Transfer for Real-time Rendering in Dynamic, Low-Frequency Lighting Environments. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 527–536, New York, NY, USA, 2002. ACM. ISBN 1-58113-521-1. (Cited on page 54.)

[221] P. Slusallek and I. Georgiev. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In R. J. Trew, editor, *Proceedings of the IEEE / EG Symposium on Interactive Ray Tracing 2008*, pages 115–122, Marina del Rey, CA, USA, 2008. IEEE Computer Society, Eurographics Association, IEEE. (Cited on page 6.)

[222] A. Smith, J. Skorupski, and J. Davis. Transient Rendering. Technical Report UCSC-SOE-08-26, School of Engineering, University of California, Santa Cruz, February 2008. (Cited on page 10.)

[223] B. Smits, J. Arvo, and D. P. Greenberg. A Clustering Algorithm for Radiosity in Complex Environments. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 435–442, New York, NY, USA, 1994. ACM. ISBN 0-89791-667-0. (Cited on page 14.)

[224] M. Stamminger. *Finite Element Methods for Global Illumination Computations*. Herbert Utz Verlag, 1999. ISBN 3896756613. (Cited on page 14.)

[225] A. Stephens, S. Boulos, J. Bigler, I. Wald, and S. G. Parker. An Application of Scalable Massive Model Interaction using Shared-Memory Systems. In Alan Heirich, Bruno Raffin, and Luís Paulo Peixoto dos Santos, editors, *EGPGV*, pages 19–26. Eurographics Association, 2006. ISBN 3-905673-40-1. (Cited on page 6.)

[226] M. Stich, H. Friedrich, and A. Dietrich. Spatial Splits in Bounding Volume Hierarchies. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 7–13, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-603-8. (Cited on pages 23, 31, and 40.)

[227] J. Sugerman, T. Foley, S. Yoshioka, and P. Hanrahan. Ray Tracing on a CELL Processor with Software Caching. Poster at The 2006 IEEE Symposium on Interactive Ray Tracing, 2006. (Cited on page 122.)

[228] K. Sung and P. Shirley. Ray Tracing with the BSP Tree. In *Graphics Gems III*, pages 271–274. Academic Press, 1992. (Cited on pages 22 and 24.)

[229] T. Sweeney. Unreal Engine 3. http://www.unrealengine.com/platforms, 2009. (Cited on page 33.)

[230] T. Sweeny. Unreal Engine 3, 2008. (Cited on page 122.)

[231] László Szirmay-Kalos, György Antal, and Mateu Sbert. Go with the winners strategy in path tracing. In *WSCG (Journal Papers)*, pages 49–56, 2005. (Cited on pages 15 and 109.)

[232] E. Tabellion. Irradiance Caching at DreamWorks. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 69:1–69:47, New York, NY, USA, 2008. ACM. (Cited on page 55.)

[233] E. Tabellion and A. Lamorlette. An Approximate Global Illumination System for Computer Generated Films. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 469–476, New York, NY, USA, 2004. ACM. (Cited on page 59.)

[234] J. Talbot, D. Cline, and P. K. Egbert. Importance Resampling for Global Illumination. In *Rendering Techniques*, pages 139–146, 2005. (Cited on page 17.)

[235] S. Thakkar and T. Huff. Intel Streaming SIMD Extensions. In *IEEE Computer*, volume 32, pages 26–24, 1999. (Cited on pages 25 and 77.)

[236] D. Theurer. I, Robot, 1983. URL http://www.atari.com. (Cited on page 2.)

[237] J. A. Tsakok. Faster Incoherent Rays: Multi-BVH Ray Stream Tracing. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 151–158, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-603-8. (Cited on pages 28, 79, and 83.)

[238] D. van Antwerpen. Unbiased Physically Based Rendering on the GPU. Master's thesis, Technical University Delft, 2011. (Cited on pages 7 and 97.)

[239] D. van Antwerpen. Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU. In *Proceedings of the Conference on High Performance Graphics 2011*, HPG '11. ACM, 2011. (Cited on page 110.)

[240] M. van der Zwaan, E. Reinhard, and F. W. Jansen. Pyramid Clipping for Efficient Ray Traversal. In *Rendering Techniques'95*, pages 1–10, 1995. (Cited on pages 26 and 78.)

[241] E. Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1997. (Cited on pages 14, 17, 18, 97, and 109.)

[242] C. Wächter and A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques*, pages 139–149. Eurographics Association, 2006. ISBN 3-905673-35-5. (Cited on pages 23 and 31.)

[243] C. Waechter. *Quasi-Monte Carlo Light Transport Simulation by Efficient Ray Tracing*. PhD thesis, University of Ulm, Germany, 2009. (Cited on page 14.)

[244] I. Wald. Realtime Ray Tracing and Interactive Global Illumination. *PhD thesis, Saarland University*, 2004. (Cited on pages 7, 37, 44, and 60.)

[245] I. Wald. On Fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1629-5. (Cited on page 31.)

[246] I. Wald. Active Thread Compaction for GPU Path Tracing. In *Proceedings of the Conference on High Performance Graphics 2011*, HPG '11. ACM, 2011. (Cited on page 97.)

[247] I. Wald and V. Havran. On Building Fast kD-trees for Ray Tracing, and on Doing That in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69, 2006. (Cited on page 22.)

[248] I. Wald and P. Slusallek. State of the Art in Interactive Ray Tracing. In *State of the Art Reports, EUROGRAPHICS 2001*, pages 21–42. EUROGRAPHICS, Manchester, United Kingdom, 2001. (Cited on pages 6 and 122.)

[249] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–165, 2001. (Cited on pages 25, 64, 78, and 80.)

[250] I. Wald, C. Benthin, and P. Slusallek. OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Computer Graphics Group, Saarland University, 2002. (Cited on page 6.)

[251] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive Global Illumination using Fast Ray Tracing. In *Proceedings of the 13th EUROGRAPHICS Workshop on Rendering*. Saarland University, Kaiserslautern University, 2002. avail.at http://www.openrt.de. (Cited on page 60.)

[252] I. Wald, C. Benthin, and P. Slusallek. Interactive Global Illumination in Complex and Highly Occluded Environments. In *Proceedings of the 14th Eurographics workshop on Rendering*, EGRW '03, pages 74–81, Aire-la-Ville, Switzerland, 2003. Eurographics Association. ISBN 3-905673-03-7. (Cited on page 60.)

[253] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, 25(3): 485–493, 2006. ISSN 0730-0301. (Cited on pages 31 and 78.)

[254] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Trans. Graph.*, 26(1), January 2007. ISSN 0730-0301. (Cited on pages 26, 31, 42, and 78.)

[255] I. Wald, C. P. Gribble, S. Boulos, and A. Kensler. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Technical Report UUSCI-2007-012, SCI Institute, university of Utah, 2007. (Cited on page 79.)

[256] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. In Dieter Schmalstieg and Jiří Bittner, editors, *STAR Proceedings of Eurographics 2007*, pages 89–116. The Eurographics Association, September 2007. (Cited on page 31.)

[257] I. Wald, C. Benthin, and S. Boulos. Getting Rid of Packets: Efficient SIMD Single-Ray Traversal using Multi-branching BVHs. In *Symposium on Interactive Ray Tracing 2008*, pages 49–57. IEEE/Eurographics, 2008. (Cited on pages 23, 28, and 80.)

[258] B. Walter, G. Drettakis, and S. Parker. Interactive Rendering using the Render Cache. In D. Lischinski and G.W. Larson, editors, *Rendering techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)*, volume 10, pages 235–246, New York, NY, Jun 1999. Springer-Verlag/Wien. (Cited on pages 6 and 123.)

[259] B. Walter, G. Drettakis, and D. P. Greenberg. Enhancing and Optimizing the Render Cache. In *Proceedings of the 13th Eurographics workshop on Rendering*, EGRW '02, pages 37–42, Aire-la-Ville, Switzerland, 2002. Eurographics Association. ISBN 1-58113-534-3. (Cited on page 6.)

[260] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg. Lightcuts: A Scalable Approach to Illumination. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 1098–1107, New York, NY, USA, 2005. ACM. (Cited on page 60.)

[261] B. Walter, A. Arbree, K. Bala, and D. P. Greenberg. Multidimensional Lightcuts. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 1081–1088, New York, NY, USA, 2006. ACM. ISBN 1-59593-364-6. (Cited on page 60.)

[262] B. Walter, K. Bala, M. Kulkarni, and K. Pingali. Fast Agglomerative Clustering for Rendering. In *IEEE Symposium on Interactive Ray Tracing (RT)*, pages 81–86, August 2008. (Cited on pages 23, 31, 41, and 63.)

[263] G. Ward. Adaptive Shadow Testing for Ray Tracing. In *Eurographics Workshop on Rendering*, pages 11–20, May 1991. (Cited on page 60.)

[264] G. J. Ward, F. M. Rubinstein, and R. D. Clear. A Ray Tracing Solution for Diffuse Interreflection. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 85–92, New York, NY, USA, 1988. ACM. ISBN 0-89791-275-6. (Cited on pages 20, 53, and 54.)

[265] T. Whitted. An Improved Illumination Model for Shaded Display. *Commun. ACM*, 23(6):343–349, 1980. (Cited on pages 5, 12, 21, and 37.)

[266] L. Williams. Casting Curved Shadows on Curved Surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '78, pages 270–274, New York, NY, USA, 1978. ACM. (Cited on pages 4 and 54.)

[267] L. B. Wolff and D. J. Kurlander. Ray Tracing with Polarization Parameters. *IEEE Computer Graphics Applications*, 10:44–55, November 1990. ISSN 0272-1716. (Cited on page 11.)

[268] A. Woo, P. Poulin, and A. Fournier. A Survey of Shadow Algorithms. *IEEE Computer Graphics and Applications*, 10(6):13–32, November 1990. ISSN 0272-1716. (Cited on page 4.)

[269] S. Woop, J. Schmittler, and P. Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *Proceedings of ACM SIGGRAPH 2005*, pages 434–444, July 2005. URL http://www.saarcor.de/. (Cited on page 7.)

[270] S. Woop, G. Marmitt, and P. Slusallek. B-KD trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '06, pages 67–77, New York, NY, USA, 2006. ACM. ISBN 3-905673-37-1. (Cited on page 7.)

[271] J. C. Xia and A. Varshney. Dynamic View-dependent Simplification for Polygonal Models. In *Proceedings of the 7th conference on Visualization '96*, VIS '96, pages 327–ff., Los Alamitos, CA, USA, 1996. ISBN 0-89791-864-9. (Cited on page 32.)

[272] K. Yamauchi. Gran Tourismo Series, 1997. (Cited on page 122.)

[273] J.I. Yellot. Science, New Series. 221(4608):382–385, 1983. (Cited on page 56.)

[274] C. Yerli and R. Taylor. Crysis Chat with Cevat Yerli and Roy Taylor, 2007. URL http://www.incrysis.com/index.php?option=com_content&task=view&id=559. (Cited on page 3.)

[275] G. Zachmann. Minimal Hierarchical Collision Detection. In *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST)*, pages 121–128, Hong Kong, China, 2002. (Cited on page 23.)

[276] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time KD-tree Construction on Graphics Hardware. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, pages 1–11, New York, NY, USA, 2008. ACM. (Cited on pages 6, 22, 31, 96, and 142.)

[277] S. Zhukov, A. Inoes, and G. Kronin. An Ambient Light Illumination Model. In George Drettakis and Nelson Max, editors, *Rendering Techniques '98*, Eurographics, pages 45–56. Springer-Verlag Wien New York, 1998. (Cited on pages 55 and 56.)

CURRICULUM VITAE

Jacobus (Jacco) Bikker

Geboren 8 maart 1973 te Barendrecht

Opleiding:

HIO (Hoger Informatica Onderwijs), Hogeschool Utrecht, 1988 - 1993.

H.A.V.O., Gereformeerde Scholengemeenschap "Guido de Brès" te Amersfoort, 1984 - 1988.

Beroepsuitoefening:

2010 - vandaag: NHTV University of Applied Sciences, Breda. Associate Professor Entertainment Technology.

2007 - vandaag: NHTV University of Applied Sciences, Breda. Senior Lecturer and Program Manager.

2006 - 2010: NHTV University of Applied Sciences, Breda. Development of the IGAD Bachelor Program.

2005 - 2006: W!Games, Amsterdam. 3D Engine Specialist.

2002 - 2005: Overloaded PocketMedia, Amsterdam. Senior Game Developer.

1999 - 2002: Davilex Software, Houten. Research & Development of Real-time 3D Technology.

1997 - 1999: Lost Boys Interactive, Amsterdam. Research & Development of Real-time 3D Technology.

1993 - 1997: Unilever / Quest, Naarden. Software Engineer.

COLOPHON

This thesis was typeset with LaTeX 2ε using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used). The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)

The typographic style was inspired by Bringhurst's genius as presented in *The Elements of Typographic Style* [39]. It is available for LaTeX via CTAN as "`classicthesis`".

Final Version as of November 6, 2012 at 16:14.

DECLARATION

This thesis is a presentation of my original research work. Wherever contributions of others are involved, every effort is made to indicate this clearly, with due reference to the literature, and acknowledgement of collaborative research and discussions. The work was done under the guidance of Professor F. W. Jansen.

Delft, September 2012

Jacco Bikker