Light Transport Simulation on Special Hardware

Tomáš Davidovič

Thesis for obtaining the title of
Doctor of Engineering Science (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

UNIVERSITÄT DES SAARLANDES

Saarbrücken, Germany, May 2016

Dean of faculty:
Univ.-Prof. Dr. rer. nat. Frank-Olaf Schreyer
Saarland University, Saarbrücken, Germany

Committee chair:
Prof. Dr. Sebastian Hack
Saarland University, Saarbrücken, Germany

Reviewers:
Prof. Dr.-Ing. Philipp Slusallek
Saarland University, Intel VCI, and DFKI, Saarbrücken, Germany

Dr.-Ing. Habil. Karol Myszkowski
MPI Informatik, Saarbrücken, Germany

Doc. Ing. Jaroslav Křivánek, Ph.D.
Charles University, Faculty of Mathematics and Physics
Praha, Czech Republic

Academic assistant:
Dr. Andreas Nonnengart
DFKI, Saarbrücken, Germany

Date of defense:
November 25th, 2016

Tomáš Davidovič

Abstract

It cannot be denied that the developments in computer hardware and in computer algorithms strongly influence each other, with new instructions added to help with video processing, encryption, and in many other areas. At the same time, the current cap on single threaded performance and wide availability of multi-threaded processors has increased the focus on parallel algorithms. Both influences are extremely prominent in computer graphics, where the gaming and movie industries always strive for the best possible performance on the current, as well as future, hardware.

In this thesis we examine the hardware-algorithm synergies in the context of ray tracing and Monte-Carlo algorithms. First, we focus on the very basic element of all such algorithms – the casting of rays through a scene, and propose a dedicated hardware unit to accelerate this common operation. Then, we examine existing and novel implementations of many Monte-Carlo rendering algorithms on massively parallel hardware, as full hardware utilization is essential for peak performance. Lastly, we present an algorithm for tackling complex interreflections of glossy materials, which is designed to utilize both powerful processing units present in almost all current computers: the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU). These three pieces combined show that it is always important to look at hardware-algorithm mapping on all levels of abstraction: instruction, processor, and machine.

Kurzfassung

Zweifelsohne beeinflussen sich Computerhardware und Computeralgorithmen gegenseitig in ihrer Entwicklung: Prozessoren bekommen neue Instruktionen, um zum Beispiel Videoverarbeitung, Verschlüsselung oder andere Anwendungen zu beschleunigen. Gleichzeitig verstärkt sich der Fokus auf parallele Algorithmen, bedingt durch die limitierte Leistung von für einzelne Threads und die inzwischen breite Verfgbarkeit von multi-threaded Prozessoren. Beide Einflüsse sind im Grafikbereich besonders stark , wo es z.B. für die Spiele- und Filmindustrie wichtig ist, die bestmögliche Leistung zu erreichen, sowohl auf derzeitiger und zukünftiger Hardware

In Rahmen dieser Arbeit untersuchen wir die Synergie von Hardware und Algorithmen anhand von Ray-Tracing- und Monte-Carlo-Algorithmen. Zuerst betrachten wir einen grundlegenden Hardware-Bausteins für alle diese Algorithmen, die Strahlenverfolgung in einer Szene, und präsentieren eine spezielle Hardware-Einheit zur deren Beschleunigung. Anschließend untersuchen wir existierende und neue Implementierungen verschiedener Monte-Carlo-Algorithmen auf massiv-paralleler Hardware, wobei die maximale Auslastung der Hardware im Fokus steht. Abschließend stellen wir dann einen Algorithmus zur Berechnung von komplexen Beleuchtungseffekten bei glänzenden Materialien vor, der versucht, die heute fast überall vorhandene Kombination aus Hauptprozessor (CPU) und Grafikprozessor (GPU) optimal auszunutzen. Zusammen zeigen diese drei Aspekte der Arbeit, wie wichtig es ist, Hardware und Algorithmen auf allen Ebenen gleichzeitig zu betrachten: Auf den Ebenen einzelner Instruktionen, eines Prozessors bzw. eines gesamten Systems.

Acknowledgments

No long term project is possible without support and inspiration from many people from all walks of life.

First and foremost, I would like to thank my supervisor Philipp Slusallek for giving me the chance to switch fields from hardware to graphics, for his guidance in the research, and the enthusiasm with which he introduced me to many other greats of the graphics field. I would also like to thank the Saarland University and the Intel Visual Computing Institute for providing funding of my research.

Special thanks also go to my two main collaborators, Jaroslav Křivánek and Miloš Hašan, whose fresh perspective was always there when my inspiration was running dry. I would also like to thank my friends and colleagues at the Saarland University: Iliyan Georgiev, Stefan Popov, Javor Kalojanov, Lukáš Maršálek, Beata Turoňová, Martin Čadík, Vincent Pegoraro, and Ralf Karrenberg for ideas, support, distractions and generally the best atmosphere a man could want in a time.

Sebastian Sylwan and Luca Fascione are solely responsible for fulfilling an impossible childhood dream of mine by giving me the opportunity to do an internship at Weta Digital and, in turn, be an active participant on bringing the works of Tolkien to life. The thanks also belong to the anonymous reviewers, whose comments always helped to produce a better and clearer paper.

And last but not least, I would like to thank all my friends and family, who supported me throughout the years in all the major decisions I had to make.

Thank you all!

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In the past two decades, 3D computer graphics slowly became ubiquitous in our lives. It is an indisputable part of our entertainment as, in the almost twenty years since Toy Story was the first feature length 3D animated movie, 3D animation took over the field almost exclusively. The same is true for live action movies, where the posters moved from boasting that the movie does also contain computer-generated imagery to boasting that the movie does also contain practical effects. And, of course, computer games of just about any kind do not even need a mention.

But it is not just the obvious fields where 3D graphics plays an important role. It is also used in architecture to visualize buildings and offices before they are built, in design to both pitch the initial designs and to reduce the number of model iterations before the final product is approved, in bioinformatics where 3D visualization of proteins with shadows and other lighting effects helps the researchers to better understand the protein structures and their possible effects, and we could go on and on.

Therefore, it is no surprise that there is an ever-present push for a higher realism of graphics, be it more detail, better materials, or more realistic illumination. In car design, faithful 3D visualization allows the car manufacturers to reduce the number of physical prototypes, as well as quickly showing the customers what a car would look like with a different paint or interior. In architecture, the game-like 3D graphics allows us to present the customers with a walk-through of a yet non-existent building. The more realistic illumination also allows evaluating how much light an office desk or a living room will get throughout the year. It could also, with sufficiently accurate models, prevent the effects of London's "Walkie-Talkie" skyscraper, which focuses light on several London streets and melts plastic parts of cars parked there.

The importance of accurately representing the real world in visual effects for action movies is indisputable, as the goal is to present the audience with imagery that is indistinguishable from reality, including – but not limited to – creating whole environments and placing actors' faces on stuntmen bodies. But the realism is also important in animated movies even though the audience knows from the start that the settings are not real. The artists rarely need completely photorealistic images but getting the indirect illumination, the color bleeding, the penumbra shadows, and other such effects right can still greatly enhance the image and leads to convincing realism. And these effects are, indeed, present in all the newer movies by all the big names in the industry such as Disney, DreamWorks, Pixar, and many others.

To achieve this ever-increasing demand for realism, the computer graphics has also evolved. At the very first, only the wireframe models, that is, the edges of the objects, were displayed. This approach gave the user a blueprint-like view of the scene and, for this very reason, it is still used in architecture and similar applications. However, despite the indisputable advantages when it comes to design, the see-through nature is not really appealing for most applications.

The next step was to assign the objects with colors and then, for each pixel, display the color of the closest object seen through the pixel. It is also possible to assign each point of the object its own color to give a perception of surface *texture* such as wood or granite. This gives the users a basic idea about spatial relations of objects in the scene and in this simple form can be still seen today in basic kitchen planers and a similar software.

This approach has further evolved to allow more and more complex algorithms that determine the object's color under a given pixel leading to today's *rasterization* approach with fully programmable pixel shaders. These generally determine the color by evaluating camera position, light positions, and (multiple) textures. It is important to note that all of the computations are strictly local. That is, when computing a pixels color the program does not have any awareness about the rest of the scene and cannot, for example, determine whether it is in shadow in respect to particular light or not. While such limitations can, of course, be worked around, the workarounds always contain some kind of approximation that cannot achieve a completely faithful representation of the scene. For example, shadows can be approximated by storing, for each light and direction from it, the distance to the closest object in an approach called *shadow mapping*. As it obviously is not possible to store the distance for *each direction*, these directions are therefore discretized, leading to jagged shadow edge artifacts, with the artifact severity based on the resolution of the discretization.

To achieve the physical realism, we have to abandon these local illumination methods and, instead, focus on *global illumination*, inspired by the physics of light[1]. In the real world, photons are emitted from a light source

[1]For our purposes, we will consider only the ray optics approximation of the real light behavior as wave optics is neither widely required nor used.

and interact with the environment, reflected around until they are absorbed by a surface. If this surface is our camera's sensor we register the photon and, after many such events, an image is formed. While we can simulate this exact behavior in a computer (the method is called *Light Tracing*), the ratio of all surfaces in the scene that can absorb photons to the surface of the camera sensor is usually so large, that the method as described would be extremely inefficient. In *Path Tracing* we follow a reverse process, starting from the camera and interacting with surface until the path encounters a light source and the final contribution can be computed. Both of these methods, as well as many others that are based on the same principles, are collectively known as the *light transport simulation*, as they simulate how light is transported in a real world and along with physically plausible materials[2] form the basis of the *physically-based rendering*.

In this thesis, we focus on accelerating these two basic approaches and the many algorithms that are based on them. First we look at the most common operation used: Finding the closest surface along a given ray. This is an essential building-block in all algorithms that mimic physics, as we simply need to know the next surface a photon will interact with. Is it also the single most important difference from the rasterization approaches, as it requires access to the whole scene. In Chapter 3 we look at the options of implementing this important operation in a dedicated hardware unit similar to rasterization units used on the modern GPUs.

To achieve a nice noise-free image with a digital camera, we want to capture, on average, somewhere between high tens and low hundreds of thousands of photons per pixel, depending on the sensitivity of the CCD sensor. While the paths used in computer graphics generally carry more information than a single photon, e.g., each represents multiple wavelengths, we still often need tens to hundreds of paths per pixel, leading to tens to hundreds of millions of paths total, for a full HD image. Considering these paths are, at least in the basic approaches, completely independent and can all be computed in parallel, we can see why rendering is often called an *embarrassingly parallel* [Moler 1986] problem. A second important aspect to note about the paths simulated in a computer is that even paths that do not bring any contribution to the camera do come at cost. So, while in nature we can have many billions of photons emitted for each photon that is captured on the camera, our rendering algorithms have to be designed to maximize the number of paths that bring us relevant illumination information. In Chapter 4 we focus on these two objectives and explore mapping of progressively more and more advanced algorithms onto some of most powerful massively parallel hardware currently available.

[2]Such materials are, for example, required to reflect at most as much energy as their receive in order to not violate the law of energy conservation. This somewhat obvious requirement was not common before physically-based rendering.

Unfortunately, even with the most advanced algorithms of today, there are still some scene configurations, such as stainless steel kitchens, that are extremely hard to sample efficiently, see Figure 5.6 for an example. On the other hand, many applications, such as fast design previews do not need exact physical accuracy but, instead, require a solution only accurate enough for the result to look convincing to the human eye. We explore this opportunity in Chapter 5, where we separate the problem into two parts, use different sets of approximation on each of these parts, and map the required steps to both CPU and GPU to utilize the full computing power of the modern computer.

1.1 Our contributions

The content of this thesis builds upon a number of previous work in the field, and its major contributions are based on previously published papers where the author was the main investigator. Below we present a summary of these contributions:

- **A Dedicated Ray Traversal Engine.** (Section 3) Finding the closest surface in a given direction is a fundamental operation in almost all photorealistic rendering algorithms. However, despite its ubiquitous nature and its significant cost that scales with the scene complexity, there still is little hardware acceleration available. Our first contribution is an evaluation of a dedicated Ray Traversal Engine hardware unit with a focus on its connection to a general purpose shading processor rather than just the ray traversal itself. We achieve this by modifying an existing Dynamic RPU design, synthesizing it with a state-of-the-art ASIC technology to obtain its characteristics and building a cycle accurate simulator of the unit. We show that such a unit could trace a significant number of rays for a fairly modest cost in both die area and bandwidth. These contributions are based on [Davidovič et al. 2009] and [Davidovič et al. 2011], where the author developed the VHDL code for hardware synthesis results, the SystemC simulation layer, performed majority of the tests, as well as wrote the main parts of the text.

- **Light Transport Simulation on the GPU.** (Section 4) When looking at a higher level of abstraction, the most important aspect in rendering is having as many efficient paths as possible. This is the goal of our next contribution where we focus on mapping algorithms onto a massively parallel, wide SIMD hardware, specifically the NVIDIA GPU cards using an already existing library for the ray tracing queries. We re-implement many previously proposed solutions for Path Tracing, Bidirectional Path Tracing, and Progressive Photon Mapping algorithms, augment the selection with some novel implementations of our own, and also provide the first GPU implementation of the Vertex Connection and Merging algorithm [Georgiev et al. 2012]. We test each of the algorithms on two generations of NVIDIA GPU and we present an extensive comparison of the various implementations, providing detailed insight into their individual strengths and weaknesses with respect to the properties of the implementation platform. Finally, to evaluate their relative performance in different scenarios we also provide a comparison of the algorithms across multiple scenes. This part of this thesis is based on [Davidovič et al. 2014], where the author wrote the majority of the testing framework, proposed all the novel algorithms, performed the described experiments, and wrote majority of the technical sections.

- **Global and Local VPLs.** (Section 5) For our last major contribution presented in this thesis, we examine the extremely hard problem of interreflections in a highly glossy environment. In such an environment only a fairly narrow bands of paths have a significant contribution to the final image, and finding them by standard sampling techniques is difficult and time consuming. We propose to address this issue by separating the light transport into a global component, providing the overall illumination, and a local component, providing highly localized glossy reflections. This separation allows us to apply different approximations of each component which allows us to significantly increase the total number of contributing paths per pixel. The various steps of the algorithm are then split between the CPU and the GPU and executed in parallel to maximize the utilization of the used computer. This method was originally published in [Davidovič et al. 2010], where the author wrote the majority of the code, proposed several key components on both local and global approximations, performed numerous experiments to evaluate feasibility of each element of the final algorithm, and wrote significant parts of the implementation details and results sections.

- **Other contributions.** Our other contributions, not discussed in detail in this thesis, include investigation of similarities between ray tracing and rasterization [Davidovič et al. 2012a], as well as further advances in many-light methods on the GPU [Davidovič et al. 2012b], and the first publicly available Vertex Connection and Merging implementation *SmallVCM* [Davidovič and Georgiev 2012]. The full list of publications is also a part of this thesis and can be found after the bibliography section (Section My Publications).

Chapter 2

Background

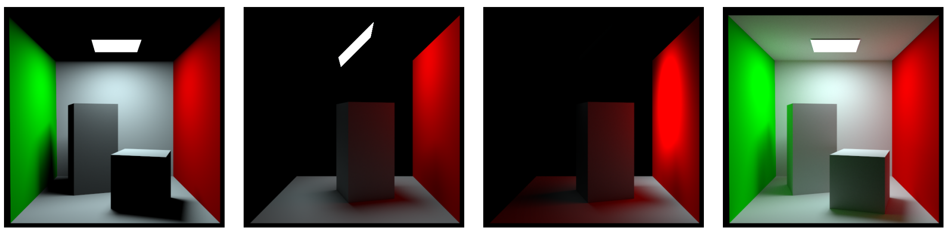

(a) Direct illum. **(b)** Color bleeding **(c)** Ind. shadows **(d)** Full GI

Figure 2.1: *Global Illumination examples.* Direct illumination (2.1a) provides basic intuition about the scene but is far from realistic. Among the missing global illumination effects is color bleeding (2.1b), where light bounces off the red wall and colors the block in the middle. When the red wall receives a stronger illumination, it can also create indirect shadows (2.1c) behind the block. A combination of all the effects gives us full global illumination results (2.1d).

To render realistic global illumination effects, such as seen in Figure 2.1, we first have to understand the basic physical principles of light propagation. In this chapter, we describe the basic concepts of light transport: The basic radiometric quantities used in light transport, the rendering equation that describes light arriving at each point in the scene, the ray tracing operator used to find the closest surface in a given direction, the Bidirectional Scattering Distribution Function (BSDF) used to describe material properties, and the path integral formulation of light transport. We refer to these terms in all the remaining chapters but significantly in Chapter 5, where we separate the rendering equation into two parts and solve them independently, each using a different set of approximations allowed by the separation.

Next, we introduce Monte Carlo integration as a general technique to solve the rendering equation, and describe several rendering techniques used to implement the Monte Carlo approach, such as Path Tracing, Bidirectional Path Tracing, and Virtual Point Lights. Chapter 4 focuses on implementation of these methods on massively parallel machines. Thus, it describes the implementation details many of these methods in much greater detail.

We also give a basic overview of various acceleration structures both for ray tracing and for range queries used in Photon Mapping and its variants. While not the main focus of this thesis, all our work uses these structures in many variants and we refer to them throughout the remaining chapters.

In the last section we focus on hardware acceleration available for rendering. Namely we introduce the concept of Single Instruction Multiple Data (SIMD), used on the CPU, the GPU, as well as many dedicated ray tracing hardware solutions. Then, we introduce the principles and challenges of General-Purpose computing on Graphics Processing Units (GPGPU), which plays a major roles in Chapters 4 and 5 where we focus on various solutions to rendering equation on GPUs. Lastly, we introduce the concept of dedicated hardware ray casting units as an alternative to standard hardware rasterization units and describe the issues encountered when designing such a unit. This is used as a basis for Chapter 3 where we describe integration of one such unit in a larger system.

2.1 Light Transport

In the following section we give a brief overview of light transport theory that describes distribution and interaction of the light with a scene. To reduce the complexity of the problem we use several common simplifications of light transport. Namely we do not consider light polarization, material fluorescence (re-emitting light at a different wavelength), and we assume that basic medium in which light propagates is vacuum, not air. We refer the readers to [Pharr and Humphreys 2004], [Dutré et al. 2006], [Veach 1997], and [Georgiev 2015] for more detailed discussion.

2.1.1 Radiometric Quantities

The basic quantity used in light transport is *Radiant flux* (or *power*), which represents the total amount of energy passing through a region of space per unit time. It is denoted by Φ, measured in Watts.

The area density of flux arriving at a surface is called *irradiance* (E), the area density of flux leaving a surface is called *radiosity* (B), both area measured in $\mathrm{W/m^2}$.

$$E = \frac{\Phi_i}{\boldsymbol{A}} \qquad B = \frac{\Phi_o}{\boldsymbol{A}} \tag{2.1}$$

Lambert's Law states that the amount of light arriving at a surface is proportional to the cosine of the angle θ between the surface normal and the light direction. Figure 2.2 shows two examples using orthogonally emitting light sources $\boldsymbol{A}$. In the first case (left), the arriving light is perpendicular to the surface, the light source area $\boldsymbol{A}$ is equal to the receiving area $\boldsymbol{B}$, and therefore:

$$E_{\boldsymbol{B}} = \frac{\Phi}{\boldsymbol{B}} = \frac{\Phi}{\boldsymbol{A}} = B_{\boldsymbol{A}} \tag{2.2}$$

In the second case (right), the light arrives at an angle such that the receiving area $\boldsymbol{C}$ is larger. Then the following relation holds

$$E_{\boldsymbol{C}} = \frac{\Phi}{\boldsymbol{C}} = \frac{\Phi \cos\theta}{\boldsymbol{A}} = B_{\boldsymbol{A}} \cos\theta \tag{2.3}$$

A more general variant of Equation 2.1, that accounts for non-constant flux arriving at the point$x$, is given as:

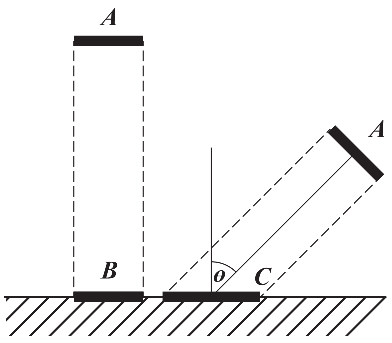


Figure 2.2: *Lambert's Law.* For an area light with the same area A, the illuminated area and, in turn, irradiance depend on the cosine of the angle θ.

$$E(x) = \frac{\mathrm{d}\Phi}{\mathrm{d}\boldsymbol{A}} \tag{2.4}$$

The most important radiometric quantity is *Radiance* (L). It is defined as flux per unit area (perpendicular to the direction of the flux), per unit solid angle:

$$L(x,\omega) = \frac{\mathrm{d}^2\Phi}{\mathrm{d}\omega \mathrm{d}\boldsymbol{A}^{\perp}} = \frac{\mathrm{d}^2\Phi}{\mathrm{d}\omega \mathrm{d}\boldsymbol{A} \cos\theta} \tag{2.5}$$

where $\mathrm{d}\boldsymbol{A}^{\perp}$ is the projected area of $\mathrm{d}\boldsymbol{A}$ onto the direction $\mathrm{d}\omega$, and θ is the angle between ω and the surface normal. Intuitively, it can be seen as the amount of light coming from an infinitely small set of directions centered around ω and arriving at an infinitely small area around a point x. As such, it is the answer to the question "How much light arrives at this point from that direction?".

An important property used in virtually all rendering algorithms is that radiance changes only when there is an interaction with either a surface or with a participating media (e.g., fog or smoke). For a point x in empty space we therefore write:

$$L_o(x,\omega) = L_i(x,-\omega) \tag{2.6}$$

where $L_o(x,\omega)$ is radiance leaving the point x in the direction ω and $L_i(x,-\omega)$ is radiance arriving at the point x from the direction $-\omega$.

2.1.2 The Rendering Equation

The light distribution in a scene is most often modeled using the *rendering equation* by Kajiya [1986]:

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_{\Omega^+} f_r(x,\omega_o,\omega_i) L_i(x,\omega_i) \cos\theta_i \mathrm{d}\omega_i \tag{2.7}$$

It says that the radiance leaving a point x in a direction ω_o, $L_o(x,\omega_o)$, is equal to the radiance emitted from that point in that direction $L_e(x,\omega_o)$ plus the incoming radiance $L_i(x,\omega_i)$ from the whole upper hemisphere Ω^+, multiplied by the *Bidirectional Reflectance Distribution Function* (BRDF) $f_r(x,\omega_o,\omega_i)$ (see Section 2.1.5 for details) and the cosine of the angle θ_i between incoming light direction and surface normal.

In this formulation the equation can describe only light reflection, omitting such important phenomena as refraction on glass. The equation can be extended to integration over the whole sphere Ω around point x [Veach 1997]. The function $f(.)$ then becomes *Bidirectional Scattering Distribution Function* (BSDF).

The rendering equation, as introduced above, assumes light of a single wavelength. Therefore, both L and $f(.)$ are also parametrized by the wavelength λ. Most commonly, rendering systems use only three pseudo-wavelengths: red, green, and blue; which directly map to RGB used in common cameras and displays. However, to achieve effects like dispersion (e.g., rainbow), more wavelengths are needed.

2.1.3 The Ray Tracing Operator

The rendering equation (Equation 2.7) presents light interactions locally with respect to the point x. However, to solve for illumination globally we have to obtain the incoming radiance $L_i(x,\omega_i)$ from the hemisphere.

For the sake of simplicity, let us assume that light interactions can happen only on surfaces, i.e., the scene contains no participating media such as smoke, fog, or, indeed, air. From Equation 2.6 it follows that the $L_i(x,\omega_i)$ will be equal to the reflectance $L_o(x',-\omega_i)$ at a point x', found along the ray starting at the point x in the direction ω_i. This point x' will lie on a surface and, as we do not allow light to pass through surfaces without interaction, it will also lie on the closest such surface.

The ray tracing operator $h(x,\omega)$ provides us with an efficient way to denote such points and lets us rewrite the rendering equation in the following, recursive, way:

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_{\Omega^+} f_r(x,\omega_o,\omega_i) L_o(h(x,\omega_i),-\omega_i)\cos\theta_i \mathrm{d}\omega_i \tag{2.8}$$

that is, the outgoing radiance at the point x is equal to the emitted radiance at that point, plus illumination from surfaces visible from the point x, integrated over directions ω_i.

This leads us to the notion that the illumination could be integrated not only over the hemisphere over the point x, but, instead, over all surfaces visible from the point x. The resulting surface area formulation of the rendering equation reads as follows:

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_{\boldsymbol{A}} f_r(x,\omega_o,\Psi)L_o(y,-\Psi)V(x,y)G(x,y)\mathrm{d}y \tag{2.9}$$

where $\boldsymbol{A}$ are all surfaces of the scene, Ψ is the direction from point x to point y, $V(x,y)$ is a *visibility function*, and $G(x,y)$ is a *geometric term*. Visibility function is a simple binary function that has value of 1 when the path between x and y is not blocked by any objects and 0 otherwise. The geometric term comes from the conversion of differential area to differential solid angle and is given as:

$$G(x,y) = \frac{\cos(N_x,\Psi)\cos(N_y,-\Psi)}{(x-y)^2} \tag{2.10}$$

where N_x and N_y are normals at the points x and y, respectively.

This second formulation is very useful in the case where we know which surfaces are light sources and want to directly integrate the illumination from these light sources onto a given point.

2.1.4 The Path Integral Formulation

While the surface area measure formulation of the rendering equation (Equation 2.9) provides a well-defined formulation of the global illumination problem, its recursive structure poses some limitations on understanding the problem. For example, in Chapter 5 we will be separately solving the direct and indirect illumination, using Virtual Point Lights (see Section 2.3.6) to solve for the indirect illumination. The local understanding of the problem provided by the rendering equation would not be sufficient for that.

The path integral formulation introduced by Veach [Veach 1997], instead, poses the problem as a pure integration problem. In this formulation, the illumination on a pixel j can be computed as an integral over all light paths in the scene passing through the pixel:

$$I_j = \int_{\Omega} f_j(\bar{x})\mathrm{d}\mu(\bar{x}), \tag{2.11}$$

for all light paths $\bar{x}$. A path $\bar{x}$ of a length k is defined as a (k+1)-tuple of vertices:

$$\bar{x} = x_0 x_1 \dots x_k,$$

where each vertex lies on a surface. The length of the path k is given by the number of path segments, each connecting two consecutive vertices, and the path therefore has $k+1$ vertices, where the vertex x_0 is on a camera, and the vertex x_k is on a light source. Please note that this framework considers only the last vertex of the path to be emissive and if a vertex can both emit and reflect light, these effects happen on different paths. However, this is used only for theoretical analysis of the paths and poses no restrictions on the actual implementation, where traced path would generally represent multiple theoretical paths.

The space of all paths length k is Ω_k and has a differential measure:

$$d\mu(\bar{x}) = \mathrm{d}\boldsymbol{A}(x_0)\mathrm{d}\boldsymbol{A}(x_1)\ldots\mathrm{d}\boldsymbol{A}(x_k).$$

The final result of the integral is obtained when integrating over the space of all paths of all lengths $\Omega = \bigcup_{k\geq 1}\Omega_k$, but it can also be separated and have different path lengths solved through different means. For example, we can explicitly sample direct illumination ($k \leq 2$) and approximate the indirect illumination ($k > 2$).

The path contribution function $f_j(\bar{x})$ is a product of the BRDF, the visibility, and the geometry terms on the vertices of the path, finally multiplied by the light emission:

$$f_j(\bar{x}) = \underbrace{\left(\prod_{i=1}^{k-1} f_r(x_{i-1}{\leftarrow}x_i{\leftarrow}x_{i+1})V(x_i{\leftrightarrow}x_{i+1})G(x_i{\leftrightarrow}x_{i+1})\right)}_{\text{path troughput}} L_e(x_k{\rightarrow}x_{k-1}), \tag{2.12}$$

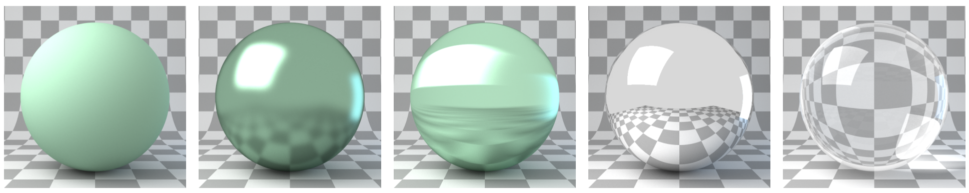
where the $f_r(x_{i-1}{\leftarrow}x_i{\leftarrow}x_{i+1})$ is a formulation of BRDF that uses the previous and the next point along the path x_{i-1}, x_{i+1}, to define the incoming and outgoing directions (ω_i, ω_o) used in Equation 2.13. The visibility function $V(x_i{\leftrightarrow}x_{i+1})$ and the geometric term $G(x_i{\leftrightarrow}x_{i+1})$ are the same as in Equation 2.9, and the emission term $L_e(x_k{\rightarrow}x_{k-1})$ is, again, using a previous point on a path to define the outgoing direction. The product of the BRDF, visibility, and geometric terms is also sometimes called *path throughput*, and represents a fraction of the radiance that is transported from the light source along this path.

The single most useful feature of this formulation is that we can generate the path using any sampling technique as long as we can define a probability of each of the path vertices. That is, unlike the Equations 2.7 and 2.9, we are not bound by following the path from the camera into the scene.

2.1.5 The Bidirectional Scattering Distribution Function

The last element of the rendering equation that needs to be discussed is the *Bidirectional Scattering Distribution Function*, which defines the visual appearance of objects in the scene. It gives an answer to the question: "If we shine light at the surface from direction ω_i, how much light is reflected towards the observer in direction ω_o". Formally we write:

$$f(x, \omega_o, \omega_i) = \frac{\mathrm{d}L_o(x, \omega_o)}{L_i(x, \omega_i)\cos\theta_i \mathrm{d}\omega_i}. \tag{2.13}$$



(a) Diffuse **(b)** Phong **(c)** Ward **(d)** Mirror **(e)** Glass

Figure 2.3: *Five BSDFs used in all the scenes throughout this work.*

When both ω_i and ω_o are in the same hemisphere (with respect to the surface normal) we are talking about bidirectional reflectance distribution function (BRDF). When they are in the opposite hemispheres, we are talking about bidirectional transmittance distribution function (BTDF).

The total energy of reflected and transmitted light cannot exceed the total energy of incident light. All physically based BSDFs therefore have to obey the following energy conservation condition:

$$\forall \omega_o, \int_\Omega f(x, \omega_o, \omega_i) \cos\theta_i \mathrm{d}\omega_i \leq 1. \tag{2.14}$$

This integral represents *albedo*, the total amount of energy that the material can reflect. In practice no materials have albedo equal to 1.

The second important condition for physically-based BRDFs (not BTDFs) is their symmetry (follows from Helmholtz reciprocity principle [Helmholtz 1867, Hapke 2012]). It states that for all pairs of ω_o and ω_i, $f_r(x, \omega_i, \omega_o) = f_r(x, \omega_o, \omega_i)$. That is, the surface will reflect the light in the same way when we exchange the light source and the observer.

Figure 2.3 presents five common BSDFs. *Diffuse* BRDF (2.3a) represents rough surfaces like matted white paint, while *glossy* BRDF (2.3b and 2.3c) are used to represent surfaces with smooth finish. There are many different glossy BRDF models ranging from purely empirical models, such as Phong BRDF [Phong 1975], to models that are derived from the surface microstructures (Cook BRDF [Cook and Torrance 1981], Ward BRDF [Ward 1992]), and models that are used to represent measurements of actual materials (Lafortune BRDF [Lafortune et al. 1997]).

The last two BSDFs on the Figure 2.3 represent the idealized versions of *perfectly specular* materials, where light can contribute to $L_o(x, \omega_o)$ from only a limited set of discrete directions ω_i. *Mirror* BRDF (2.3d) is a mathematically perfect mirror and the BRDF is non-zero only when ω_i is the perfect reflection of ω_o. *Glass* BSDF (2.3e) follows a similar principle with the refraction direction given by *Snell's Law* and the ratio between the reflected and refracted energy given by *Fresnel Equations*. While this behavior is not physically realizable, these idealized models are widely used as many rendering algorithms can utilize these simplifications for substantial performance gains. We refer the readers to [Jenkins and White 1976] for detailed explanation of the underlying physics principles.

In our scenes we use all four introduced types of BSDFs. For glossy we mainly use the common Phong BRDF, as it is the BRDF of choice for many modeling programs from which we obtained our scenes. However for metal surfaces we switch to the Ward BRDF, as it offers a more faithful representation of metals, including support for anisotropy needed to represent, e.g., brushed aluminum.

2.2 Monte Carlo Integration

Neither the recursive integral equation, nor the path integral introduced in the previous section can be solved analytically, except in the most trivial cases. Solving these equations therefore relies on using numerical methods, most commonly Monte Carlo integration. In this section we give a brief overview of random variables, Monte Carlo integration, and its improvements using importance sampling. For more complete introduction into the problematic with respect to graphics we refer to [Pharr and Humphreys 2004, Georgiev 2015].

2.2.1 Random Variables

A *random variable* X is a variable whose value is subject to chance. The values come from some domain, which can be either discrete, e.g., a dice, or continuous, e.g., the probability that a bus arrives at a given time.

In the case of a dice, we have a random variable X that can have values from the discrete domain 1, 2, 3, 4, 5, 6. Assuming the dice is a fair dice, each of these values has the same *probability* $p_i = \frac{1}{6}$. The probability of all values always has to sum up to 1 and, consequently, the maximum probability a single value can have is 1. Such value would be chosen in all cases and the variable would, de facto, cease to be random.

Further, we define a *cumulative distribution function* (CDF) as:

$$P(x) = Pr[X \leq x]. \tag{2.15}$$

It represents the probability of the random variable X achieving value of x or less. In the case of a dice we get: $\frac{1}{6}$, $\frac{1}{3}$, $\frac{1}{2}$, $\frac{2}{3}$, $\frac{5}{6}$, 1. CDFs are often used to map a uniform random number $\xi \in [0, 1)$ (a common output of random generators) to the random variable X.

In the case of continuous random variables, the number of possible values is infinite and the concept of each value having its assigned probability is not applicable. Instead, we introduce the concept of a *probability density function* (PDF), which indicates the density of probability in an area around a given value. For example, a uniform random variable in the range $[a, b]$ the value of the PDF is constant and equal to $\frac{1}{b-a}$.

The PDFs are always positive and always integrate to one. The probability that a value x lies in an interval $[a, b]$ is given as:

$$P(x \in [a, b]) = \int_a^b p(x)\mathrm{d}x \tag{2.16}$$

and he value of the CDF(x) is given as:

$$CDF(x) = P(t \in [-\infty, x]) = \int_{-\infty}^{x} p(t)\mathrm{d}t \tag{2.17}$$

The *expected value* $E[X]$ of a random variable X is, intuitively, the mean value of the X. In the case of a dice, the $E[X] = \frac{1}{6}(1+2+3+4+5+6) = 3.5$. Formally, the expected value is defined as:

$$E[X] = \sum_i x_i p_i \tag{2.18}$$

$$E[f(x)] = \int_D f(x)p(x)\mathrm{d}x \tag{2.19}$$

for the discrete (top) and the continuous (bottom) case, respectively.

Complementary to $E[f(x)]$ is the *variance* $V[f(x)]$, representing how spread out the values of $f(x)$ are from their mean. It is defined as follows:

$$V[f(x)] = E\left[(f(x))^2\right] - E[f(x)]^2 \tag{2.20}$$

Variance is commonly used as a measure of quality of results of Monte Carlo integrator and the goal of many algorithmic improvements is to lower the variance while keeping the costs the same.

2.2.2 Monte Carlo Estimator and Its Error

Informally, the basic Monte Carlo integration works as follows: Given a function $f(x)$ over a domain D, we randomly choose N samples from the domain, evaluate the function $f(x)$ at these samples and average the results.

More formally, for a one-dimensional integral $F = \int_a^b f(x)\mathrm{d}x$ and N random samples $X_i \in [a, b]$, the Monte Carlo estimator is:

$$\hat{F}_N = \frac{1}{N}\sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}. \tag{2.21}$$

Its expected value $E[\hat{F}_N]$ is then equal to the integral $F = \int_a^b f(x)\mathrm{d}x$, when the PDF $p(X_i) > 0$ for all X_i, where the $f(X_i) \neq 0$.

However, unless the $N = \infty$, the actual value of $\hat{F}_N$ can differ from the value of the integral F leading to an error in our estimate.

The *mean square error* (MSE) of an estimator measures its average squared error:

$$\begin{aligned} MSE(\hat{F}_N) &= E[(\hat{F}_N - F)^2] \\ &= E[\hat{F}_N^2] - 2E[\hat{F}_N]F + F^2 \\ &= (E[\hat{F}_N^2] - E[\hat{F}_N]^2) + (E[\hat{F}_N]^2 - 2E[\hat{F}_N]F + F^2) \\ &= V[\hat{F}_N] + (E[\hat{F}_N] - F)^2, \end{aligned} \tag{2.22}$$

where $(E[\hat{F}_N]-F)$ is *bias* and represents the difference between the expected value of the estimator $\hat{F}_N$ and the true value of the integral F. Given that our estimator has $E[\hat{F}_N] = F$, we can say that the mean square error of the estimator is equal to its variance.

The *root mean square error* (RMSE) of an estimate is the square root of MSE:

$$RMSE(\hat{F}_N) = \sqrt{MSE(\hat{F}_N)} = \sqrt{V[\hat{F}_N] + Bias(\hat{F}_N)^2}, \tag{2.23}$$

is expressed in the same units as the estimated integral, and is the most common form of describing image error in rendering.

The number of samples N has an obvious effect on the error of the estimate and for unbiased estimators the relation can be expressed as:

$$\begin{aligned} MSE(\hat{F}_N) &= V\left[\hat{F}_N\right] = V\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] \\ &= \frac{1}{N^2}V\left[\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] = \frac{1}{N}V\left[\frac{f(X)}{p(X)}\right]. \end{aligned} \tag{2.24}$$

Which leads to the commonly quoted intuition that to double the image quality (halve the RMSE), we need to quadruple the number of samples N. The alternative to this bruteforce approach is to lower the actual variance of $\frac{f(X)}{p(X)}$ and this is the focus on the following section.

2.2.3 Efficient Sampling of Monte Carlo Estimator

Intuitively, we want to make sure that the whole domain of the integral gets a roughly similar sample coverage. The naive approach is to choose all X_i in the whole interval with the uniform probability $p(X_i) = \frac{1}{b-a}$. However, placing the samples with a uniform probability does not guarantee uniformly spaced samples, as the samples can arbitrarily clump within the domain.

One possible improvement is to use *stratified sampling* [Mitchell 1996], where the domain is split into N equally sized strata and within each stratum a single uniform sample is selected. Unfortunately, this approach assumed an a priori knowledge of the number of samples N. Another approach is to use *quasi random sequences* [Keller 2013] (such as Halton sequence), which ensure that the samples drawn from the sequence are "nicely" spaced with low discrepancy across the domain. Both approaches are common in many rendering applications.

While the previous sampling approaches strive to achieve a better placement of uniform samples, *importance sampling* is a technique that is trying to improve the sample PDF $p(x)$. The motivation is the following. Let us assume that we have a probability distribution $p(x)$ such that it exactly matches the integrated function $f(x)$, up to a scaling factor c. This factor is necessary to ensure that $p(x)$ integrates to one over the domain and it is immediately obvious that $c = \int_D f(x)$. We can then write:

$$\begin{aligned} \hat{F}_N &= \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)} = \frac{1}{N}\sum_{i=1}^{N}\frac{c \cdot f(X_i)}{f(X_i)} \\ &= \frac{1}{N}\sum_{i=1}^{N} c = c = \int_D f(x). \end{aligned} \tag{2.25}$$

In such an idealized case, where we have so called *perfect importance sampling*, even a single sample will give the exact result of the integral. The drawback is that to obtain such perfect PDF we have to know the target value of $\int_D f(x)$ beforehand.

While the perfect importance sampling is not meaningful, it is still beneficial to try and match the $p(x)$ to the $f(x)$ as closely as possible. For example, in the common case where $f(x)$ is a product of two known functions $g(x)$ and $h(x)$, it is often possible to match the PDF to either $g(x)$ or $h(x)$.

One danger of this approach is that it is as much detrimental when $p(x)$ severely under- or overestimates $f(x)$ as it is beneficial when it matches it. Therefore, when our function of choice (e.g., $g(x)$) underestimates the value of $f(x)$ the final result can be worse than if we used only uniform sampling.

A typical example is when $g(x)$ is the BRDF value, which can often be sampled perfectly, and $h(x)$ is the incoming direct illumination, which can also be sampled almost perfectly (we can sample proportional to light source power, but not the occlusion). However, perfectly sampling their product is, in the general case, impossible or, at least, impractical.

Multiple Importance Sampling, introduced by Veach and Guibas [1995], mitigates this problem. Given n proposal PDFs $p_1(x) \dots p_n(x)$ and a sample x taken from a particular PDF $p_s(x)$, they introduce a weighting heuristic to reduce the impact of $p_s(x)$ being a poor match for $f(x)$ at the given x. Here we show the most important of the introduced heuristics, the *power heuristic*:

$$\omega_s(x) = \frac{p_s(x)^\beta}{\sum_{i=1}^{n} p_i(x)^\beta}. \tag{2.26}$$

The parameter β is commonly set to either 1 (then it becomes the *balance heuristic*) or to 2, a good value empirically determined by the authors. The intuition behind the approach is that when a sample x is chosen from a PDF that underestimates the target function $f(x)$, it will increase the variance and its contribution should be weighted less than when the sample comes from a PDF that matches the function well.

This approach is found in virtually all Path Tracing (Section 2.3.3) implementations, where it is used to combine the aforementioned BRDF sampling technique (path continuation) and direct illumination sampling technique (so called next-event estimation).

2.3 Rendering Techniques

In this section we focus on the most common rendering algorithms. We first briefly describe an algorithm for computing direct illumination and extend it to Whitted-style ray tracing [Whitted 1980] that correctly handles mirror and glass surfaces. We then move onto algorithms for computating the full light transport solution and give an overview of Path Tracing [Kajiya 1986], Bidirectional Path Tracing [Lafortune and Willems 1993, Veach and Guibas 1994], Virtual Point Lights [Keller 1997], and (Progressive) Photon Mapping [Jensen 1996, Hachisuka et al. 2008].

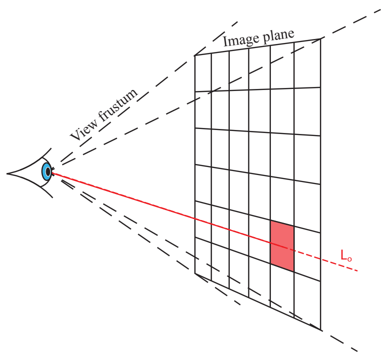


Figure 2.4: *A Pinhole Camera.* All light rays contributing to the image pass through the *Image plane* and through a single common point (pinhole), in our illustration representing the eye's iris. All such rays subtend a volume known as *view frustum*.

In all the algorithms we assume that we have a full scene description available, can directly access all sources of light (be it emissive geometry or dedicated light sources such as point lights), and that we are concerned with only a single camera. For our purposes we will assume a pinhole camera (see Figure 2.4), where all light rays contributing to the image pass through a single point in space. A color of each given pixel is given by radiance L_o of rays that arrive to the camera through this pixel.

2.3.1 Direct Illumination

In a direct illumination scenario, only surfaces directly seen by the camera are displayed and their illumination is influenced only by light coming

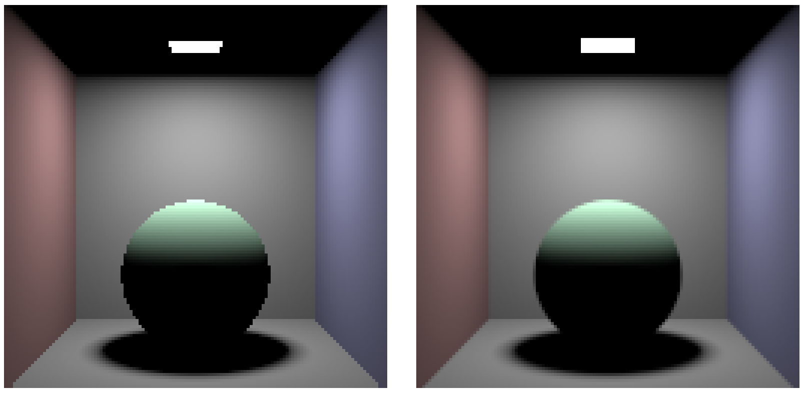


Figure 2.5: *Direct Illumination.* When an image is rendered with only 1 sample per pixel (left), there can be obvious aliasing at the edges. Increasing the number of samples leads to a nicely anti-aliased image (right). Both images rendered with an intentionally small resolution (128×128) to emphasize the aliasing artifacts.

directly from the light sources. Refering to the Equation 2.9 we can write:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\boldsymbol{A}} f_r(x, \omega_o, \Psi) L_e(y, -\Psi) V(x, y) G(x, y) \mathrm{d}y \quad (2.27)$$

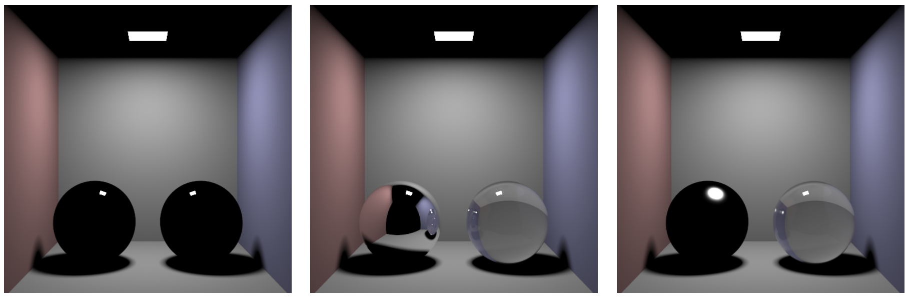
where $L_o(x, \omega_o)$ is determined by the L_e of the directly visible surface, but the integral representing reflected radiance coming from other surface points y takes into account only radiance emitted by those surfaces, not reflected off of them. We can imagine that the camera records only photons that start at a light source and then interact with the scene surfaces at most once.

In the context of ray tracing algorithms, the implementation usually works in the following steps:

1. A ray is shot through the center of a pixel, determining its point x.
2. The integral is evaluated using Monte Carlo integration, randomly positioning samples y on the surfaces proportional to their emitted flux. Ray casting is used to evaluate the visibility function $V(x, y)$.

A drawback of this basic approach is that it assumes a constant L_o for the whole pixel. However, for pixels with a finite extent, this is not true, most notably when the pixel overlaps object boundaries. This results in objectionable aliasing artifacts, such as seen on Figure 2.5a.

To reduce the aliasing, we have to take multiple samples for each pixel. The required number depends on the specific scene configuration, but commonly it is between 4 and 256 samples per pixel. To produce Figure 2.5b we chose a different method and instead of having a fixed number of samples per pixel, we generate a new unique sample x from the pixel whenever we



(a) Direct Illumination (b) Whitted-style (c) Glossy instead of mirror

Figure 2.6: Direct illumination fails to convey many materials, such as glass and mirror (left). Using the Whitted-style ray tracing, the materials of the two balls become immediatelly obvious (middle). Unfortunately, the approach is not directly applicable to glossy materials (right).

generate a new sample y on a light. This way we can continuously improve the quality until we are satisfied with the result.

2.3.2 Whitted-style Ray Tracing

One disadvantage of a pure direct illumination solution is that it cannot really convey highly specular materials, such as mirrors and glass (Figure 2.6a). The reason is that only light from a narrow cone of directions contributes to the integral. In the case of a perfect mirror and glass (as introduced in Section 2.1.5), this narrow cone actually becomes infinitely thin and the probability of sampling a point y on a light such that it would contribute toward L_o is zero.

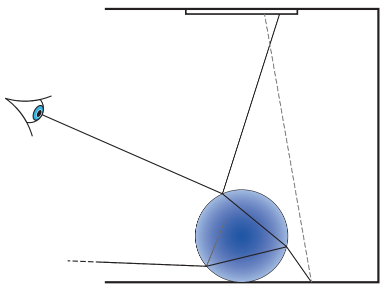


Figure 2.7: *The Whitted-style ray tracing*. Rays that encounter glass bifurcate into the reflection and refraction directions and recurse until they encounter a non-specular surface, or a maximum recursion depth is reached.

Turner Whitted [Whitted 1980] introduced a simple solution for exactly the case of perfect mirror and glass. The idea is that when the BSDF at the point x limits the integral to a discrete set of directions from which it can gather the illumination, these directions are explicitly sampled by further rays. For example, when a ray from camera encounters a glass sphere (Figure 2.7), where only perfect reflection and refraction directions can contribute, the ray bifurcates and continues in both directions. This process is recursively repeated until all the rays reach surfaces that can be directly illuminated. To prevent infinite recursion due to, e.g., perfect internal reflection in glass, a certain maximum recursion depth is usually enforced, selected such that the effect on the result is minimal (we use maximum recursion depth of 20).

Using this technique greatly improves visualization of scenes consisting of a combination of mostly diffuse and perfectly specular materials. However, it cannot handle highly glossy surfaces, where contributing illumination can arrive from a narrow but finite cone of directions, as multiple rays are needed to sample this cone (Figure 2.6c).

2.3.3 Path Tracing

Both previous techniques solved a simplified version of the rendering equation, as they reduce the f_r and L_o terms inside the integral to discrete distributions, respective emitted radiance. On the other hand, Path Tracing, as introduced by Kajiya [Kajiya 1986], is the direct implementation of Equation 2.8, repeated here for convenience:

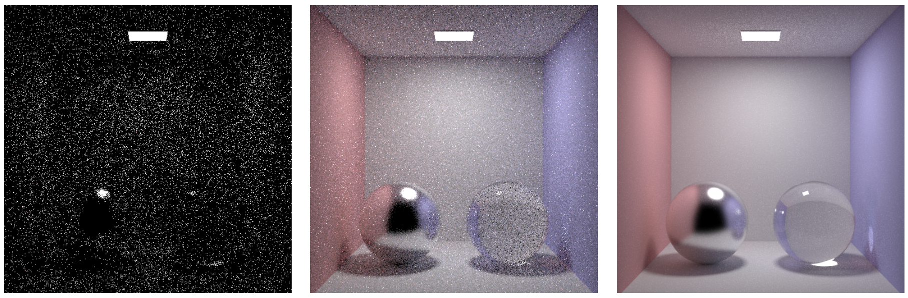
$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega^+} f_r(x, \omega_o, \omega_i) L_o(h(x, \omega_i), -\omega_i) \cos\theta_i \mathrm{d}\omega_i$$

To solve this recursive equation, we start at the camera and trace a ray through one of the image pixels, same as in the previous algorithms. We evaluate L_o at the first intersection point x with the scene by accumulating L_e and sampling the integral. The integral is sampled by choosing a random direction ω_i and tracing a ray in this direction to determine the point $h(x, \omega_i)$ whose L_o will be used in the integral. The process is recursively repeated, forming a *path* of rays, with *vertices* at each interaction with the scene.

It is clear that in a closed scene, this algorithm by itself would lead to infinitely long paths. The naive solution would be to artificially limit the path length to a given number of vertices. However, choosing a maximum path length that will provide good results highly depends on the scene. If the path length is too short, it can cause visible artifacts in scenes, e.g., in a scene with many mirror interreflections. If the path length is too long, many of the paths will perform computations with little to no impact on the final result.

A better approach is to used *Russian Roulette* to probabilistically terminate the paths. It is based on the intuition, that instead of taking 10 random samples we can take only 2, but each of these will have 5$\times$ larger impact. More formally, we can write that a sample x taken from a PDF $p(x)$ with a russian roulette probability p_{rr} evaluates at $\frac{f(x)}{p_{rr} \cdot p(x)}$. The common choice of p_{rr} in the context of Path Tracing is $\int_{\Omega^+} f_r(x, \omega_o, \omega_i) \mathrm{d}\omega_i$, the albedo of the material at a given vertex. The intuition is that paths continuing from dark materials will have overall lower contribution to the final result than paths continuing from light materials.

In practice, these methods are often combined, as maximum path length is useful to prevent extremely long recursions in case of scene with



(a) Naive Path Tracing **(b)** Next-event estimation **(c)** Naive, 4k samples

Figure 2.8: Naive Path Tracing (left) depends on a path randomly hitting light emitting surfaces, leading to high variance (we used 20 samples per pixel). With next-event estimation (middle), the lights are sampled explicitly leading to a significantly better quality for the same number of samples. Path Tracing without the next-event estimation is still noisy after 4000 samples per pixel (right).

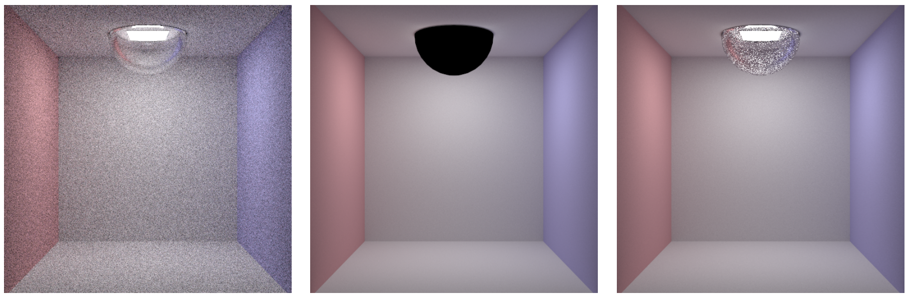
albedos close to, or equal to, one. However, due to the presence of russian roulette, this maximum value does not have a direct impact on performance and can therefore be set sufficiently high (numbers between 20 and 256 are common).

Naive Path Tracing depends only on chance to connect the camera to lights. It is obvious that, as the lights become smaller, the noise increases (Figure 2.8a). To improve this, we can stratify our sampling by identifying all light sources in the scene and making explicit connections to them similar to the direct illumination approach. This technique is called *next-event estimation* and can be formally written as:

$$\begin{aligned} L_s(x,\omega_o) = &\int_{\Omega^+} f_r(x,\omega_o,\omega_i) L_s(h(x,\omega_i),-\omega_i)\cos\theta_i \mathrm{d}\omega_i \\ &+ \int_{\boldsymbol{A}} f_r(x,\omega_o,\Psi) L_e(y,-\Psi) V(x,y) G(x,y) \mathrm{d}y \end{aligned} \tag{2.28}$$

where $L_s(x,\omega)$ is radiance scattered from the point x in the direction ω, and it can be written that $L_o(x,\omega) = L_e(x,\omega) + L_s(x,\omega)$. The first integral of the equation handles light scattered from other surfaces and the second integral light emitted by light sources.

This approach significantly increases the quality of rendering for the same number of camera samples (Figure 2.8b) but still has room for improvement. Veach and Guibas [Veach and Guibas 1995] propose to use Multiple Importance Sampling (see Section 2.2.3) to combine both techniques of connecting to lights. This approach always samples lights through both techniques, i.e., randomly hitting the light and the next-event estimation, and combines them using weights that are based on the probability of a given technique finding the contribution such that the technique resulting in higher variance for each given contribution is assigned lower weight, result-



(a) Path Tracing **(b)** Light Tracing **(c)** Bidirectional Path Tracing

Figure 2.9: Path Tracing with next-event estimation (left) cannot explicitly sample a light behind a fixture. Light Tracing (middle) cannot connect between specular surfaces and pinhole camera, causing the light fixture to appear black. Bidirectional Path Tracing (right) connects camera sub-paths to light sub-paths that already passed through the fixtures, resulting in less noise in the rendering same time (1 second, for all images).

ing in the overall lower variance for the image than if just a single technique was used. We refer the readers to Veach's thesis [Veach 1997] for details.

2.3.4 Bidirectional Path Tracing

In the previous section we have seen the usefulness of sampling lights explicitly. *Bidirectional Path Tracing* extends this idea by sampling a full sub-path starting at the light (*light sub-path*) and connecting the *camera sub-path* to it.

Unfortunately, in many scenes the light source is enclosed in some kind of glass and metal fixture, making direct connection impossible because the light has to get refracted. In this case, the next-event estimation is effectively removed and the algorithm degenerates into Naive Path Tracing (Figure 2.9a). Light Tracing reverses the tracing process, starting at the lights. However, purely specular surfaces appear black (Figure 2.9b) because the probability that the commonly used pinhole camera will lie in the direction of the reflected (or refracted) ray is always zero. Thus, the camera does not receive any radiance from such surfaces.

In Bidirectional Path Tracing both approaches are combined (Figure 2.9c). Furthermore, it allows a direct *vertex connection* not only between a light sub-path and camera and a camera path and lights, but also between the light and camera sub-path vertices. As this presents multiple ways the camera can be connected to the same light (e.g., the same path constructed from camera or from the light), an extra care has to be taken to not account the light's contribution multiple times. The common approach is, again, to use Multiple Importance Sampling [Veach 1997, Georgiev 2012, van Antwerpen 2011a], which automatically assigns higher weight to the more suitable connections.

The common implementation then works in three steps. In the first step, a light sub-path is traced and its vertices are stored. In the second step, a camera sub-path is traced. In the third step, vertices on the light and camera sub-paths are connected, forming several full paths, each of which is properly weighted using Multiple Importance Sampling. A common approach is to not store the first vertex of the light sub-path (i.e., vertex directly on the light) and instead connect each camera sub-path vertex directly to its own, randomly generated, vertex on the light. This removes some correlation, coming from reusing light sub-path vertices, at virtually no additional cost.

While Bidirectional Path Tracing can efficiently capture a large set of paths, it still has problem with specular-diffuse-specular paths, e.g., reflected caustics. The intuition is that connecting through a specular interaction so we arrive at a preselected vertex in the scene (e.g., a light) is generally not possible. To do this we would have to choose only the vertex we want to connect to, but also the exact outgoing direction that will, after reflecting/refracting on the specular objects, arrive at this vertex. As this problem can only be solved at a fairly high cost [Jakob and Marschner 2012, Hanika et al. 2015] it is generally recommended to use rendering approaches based on Photon Mapping which we describe in the next section.

2.3.5 (Progressive) Photon Mapping

While Bidirectional Path Tracing (BPT) has a wider set of efficiently handled light transport paths than Path Tracing, some configurations still present problems. The setting on the left side of Figure 2.11a presents one such case. Both the camera and the light are covered by perfectly specular glass, leaving no option to perform any direct vertex connections between the light and camera sub-paths when evaluating direct illumination. As a result, BPT degenerates into Naive Path Tracing for this type of paths. See Figure 2.10 for the schematic view of such configuration. Approaches based on Photon Mapping relax the way paths are constructed, leading to a better handling of such cases (Figure 2.11b).

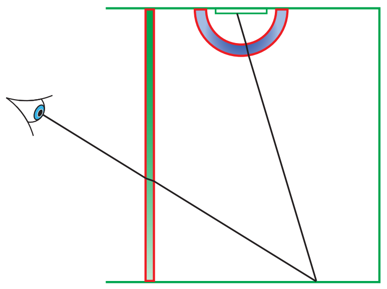
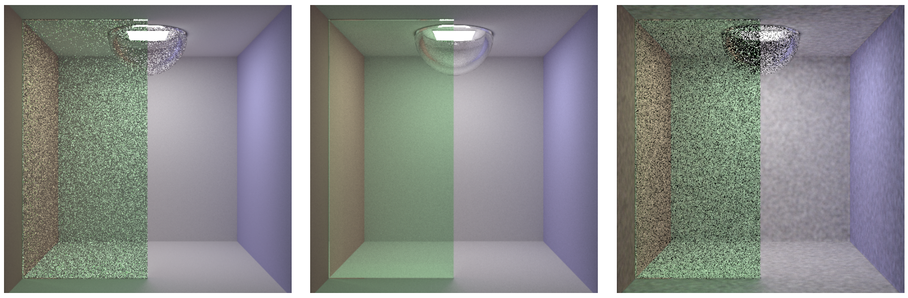


Figure 2.10: In Bidirectional Path Tracing, the path cannot perform any vertex connection without two consecutive non-specular (green) surfaces, as the specular surfaces (red) enforce a single outgoing direction.

The original formulation of Photon Mapping [Jensen 1996] presents a



(a) Bidirectional Path Tracing **(b)** Progressive Photon Mapping **(c)** Progressive Photon Mapping (low quality)

Figure 2.11: A Cornell Box with the left side viewed through a green tinted glass, and the light source inside a glass fixture. Bidirectional Path Tracing (left) degenerates into Naive Path Tracing when surfaces are illuminated through glass as well as seen through glass. Progressive Photon Mapping (middle) can handle such illumination well, at the cost of introducing bias and a lower order of convergence. A low quality Progressive Photon Mapping image (right) shows the splotchy artifacts typical for this rendering technique.

different view on global illumination. It is a two-pass algorithm where, in the first pass of the method photons are emitted from light sources and deposited onto surfaces, storing their position, energy, and incoming direction (for BRDF evaluation). In the second pass, camera sub-paths are traced and the photon density is estimated in the neighborhood of their first non-specular vertex (gather point). The density estimation is usually facilitated by finding the k nearest photons (typically 20-50), using a k-Nearest Neighbors (k-NN) search [Hey and Purgathofer 2002]. The outgoing radiance is computed as:

$$L_o = L_e + \frac{1}{\pi r^2} \sum_{j=1}^{k-1} f_r(x, \omega_o, \omega_{i,p_j}) k_r(d_j) \Delta\Phi_{p_j} \tag{2.29}$$

where x is the position of the gather point, f_r is the BRDF at this position, r is the radius of circle in which our k photons lie, ω_{i,p_j} is the incoming direction of the photon p_j, k_r is a normalized and scaled kernel weight function[1], d_j is the distance of the photon from x, and $\Delta\Phi_{p_j}$ is the flux it carries. The intuitive reason why only $k-1$ photons are used is that the k^{th} photon, at the very border of our search circle, does not contribute all of its energy into our search area. Please see [García et al. 2012] for rigorous derivation.

A key difference from the path-based methods is that the paths are not evaluated exactly. During the density estimation, the end points of light sub-paths (photons) are moved to the gather point position, even in

[1]We use the simplest, cylindrical, kernel, and the k_r is therefore constant inside the radius r and zero otherwise, and could be omitted from the equation.

cases where the nature of light sub-path would not allow this, i.e., if the previous vertex on the sub-path was specular and did not allow other directions (Figure 2.12). While blurring of light transport introduces a new type of error in our L_o estimate, bias, it also allows for a better handling of specular-diffuse-specular type of paths than Bidirectional Path Tracing.

Hachisuka et al. [2008] propose Progressive Photon Mapping, a Photon Mapping based method that, while biased, is consistent, i.e., in the limit it converges to the ground truth solution. In the first pass of the algorithm, camera sub-paths are traced and the gather points are stored. The gather points include all the information required to evaluate the BRDF at the point, as well as the current density estimation radius r, the total accumulated flux and the number of photons accumulated so far. In the subsequent passes, photons are traced from lights. Photons are not stored but, instead, immediately contribute to all gather points in whose radius r they fall. After each such pass, the radius r of each gather point is reduced using the collected statistics in such a way that the radius becomes zero in the limit. This approach has been expanded to Stochastic Progressive Photon Mapping [Hachisuka and Jensen 2009] to allow changing the gather points between photon passes (e.g., for anti-aliasing or depth of field). Knaus and Zwicker [Knaus and Zwicker 2011] change the formulation to remove the requirement for per-pixel statistics. Vorba [Vorba 2011] introduces Progressive Bidirectional Photon Mapping, where a full camera sub-path is traced, density estimation is performed at each non-specular vertex of the path and the results are combined using Multiple Importance Sampling.

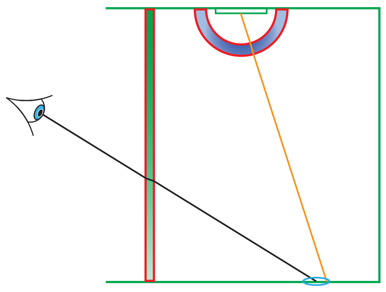
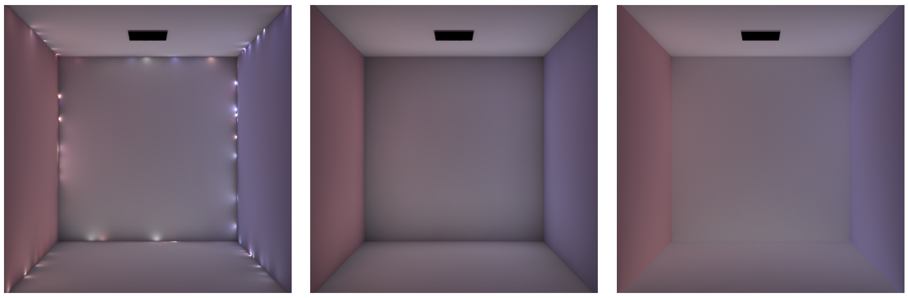


Figure 2.12: Photon Mapping approaches allow "merging" of path vertices on non-specular (green) surfaces, allowing for use of bidirectional methods even where BPT does not.

Georgiev et al. [2012] and Hachisuka et al. [2012] concurrently reformulate the Photon Mapping in the context of path-based methods, allowing combining of Progressive (Bidirectional) Photon Mapping with Bidirectional Path Tracing are using Multiple Importance Sampling (MIS) into a novel algorithm, named *Vertex Connection and Merging* by Georgiev and *Unified Path Sampling* by Hachisuka. This work uses the name Vertex Connection and Merging, as the author participated on the [Georgiev et al. 2012] paper, but the names are completely interchangeable. For derivation of the MIS weights, see [Georgiev 2012].



(a) VPL rendering without clamping **(b)** VPL rendering with clamping **(c)** Reference solution

Figure 2.13: Using Virtual Point Lights leads to objectionable artifacts due to lights' strong contribution on close distance (left). Clamping the contribution removes the objectionable artifacts (middle), but also removes a significant amount of energy, when compared to the reference solution (right). Only indirect illumination is shown.

2.3.6 Virtual Point Lights

The last set of techniques we preset is based on Virtual Point Lights (VPL), first introduced by Keller [Keller 1997] in the context of fast approximation of global illumination. The basic idea was to first render the scene as seen from the lights position and deposit VPLs onto visible surfaces. When rendering the final image, the surfaces visible from the camera are illuminated not only by the main lights (direct illumination), but also by the Virtual Point Lights, for the so-called one-bounce global illumination (the light paths contain one more vertex, or bounce, than direct illumination).

In principle, VPLs are similar to the Bidirectional Path Tracing, introduced Section 2.3.4, with several limitations on the individual path lengths. Light sub-paths are limited to length one, multiple light sub-paths are created at once and each light vertex is turned into a Virtual Point Light. Camera sub-paths are also limited to length one and each light vertex (VPL) from each light sub-path contributes to camera sub-path vertices from all camera sub-paths, without any contribution directly to the camera. It is obvious that the one-bounce restriction of the original method comes from using rasterization as the means of VPL distribution, rather than any inherent property of the method itself. When ray casting is used to distribute VPLs, the restriction on light sub-path length is lifted and VPLs can be used to approximate the full global illumination solution. Throughout this work we only consider this approach and not the one-bounce solution.

Distributing a limited set of light sources to corners and other unexpected places throughout the scene leads to objectionable light splotch artifacts (Figure 2.13a), caused by the presence of the $\frac{1}{\text{distance}^2}$ term in the geometry term in Equation 2.27. The contribution of each VPL to nearby surface can therefore become infinitely large. We can imagine the effect as if we randomly placed invisible torches into a real scene. These torches would create light splotches on the nearby walls and as there would be no obvious sources for these splotches the whole effect would feel unnatural.

The common approach is to clamp each VPL's contribution in such a way that the the artifacts are no longer observable (Figure 2.13b). Unfortunately, this approach can severely decrease the overall illumination in the scene, especially in the corners, limiting the accuracy of the rendered image, especially in the presence of glossy surfaces (see Chapter 5). Another approach is to increase the number of VPLs used in the rendering, as the intensity of each VPL (and thus its splotch) is indirectly proportional to the number of VPLs.

While the original purpose of VPL rendering was a fast approximate global illumination, VPLs based methods, also called many-light methods, are also used in high-fidelity offline rendering. Křivánek et al. [2010] show that to represent materials faithfully a large number of VPLs (millions) is required. While it is possible to simply render the images with millions of VPLs, this is both expensive and unnecessary. Lightcuts [Walter et al. 2005] show that clusters of VPLs that are far away from the illuminated point can be approximated by a single VPL containing energy of the whole cluster[2]. The algorithm hierarchically clusters VPLs based on their position and normal and then, for each point, traverses this tree of VPLs, using a per-material error metric to determine which clusters can be used directly and which have to be further refined. The algorithm has been further developed to allow efficient handling of multiple illuminated points per pixel, such as motion blur or depth of field [Walter et al. 2006], as well as contribution to a full camera sub-path through a pixel [Walter et al. 2012].

A different, global, approach to clustering has been proposed by Hašan et al. [2007]. They formulate the problem as a matrix, where each pixel sample is a row and each VPL is a column. The matrix elements then contain a contribution from a given VPL to a given pixel sample, and the final image is given as a sum of all elements in each row. The authors show that the full matrix has a significant correlation between many of its rows (pixels) and columns (VPLs), and can be approximated by sampling a significantly sparser matrix. This becomes fairly obvious when we consider that a simple image upscaling is a naive utilization of this observation, four pixels in the final image are approximated by a single pixel in the reduced image and the final result is fairly faithful.

The introduced *Matrix Row-Column Sampling* algorithm focuses on reducing the number of required VPLs, instead of pixels used in the example above, by performing a data driven analysis of their importance to the image and approximating many of the VPLs by VPLs with a similar

[2]This approach is somewhat similar to N-body simulation used in gravity, but with the added complication of light's directionality.

contribution. The algorithm evaluates contribution of each VPL to a representative subset of pixel samples, creating a matrix with all the columns, but fewer rows. The columns are then clustered, based on the distance metric between the individual columns, into a predetermined number of clusters. The representative VPLs of these cluster then contribute to all the pixel samples, rendering the final image.

The main advantage of this algorithm over Lightcuts is that, as the algorithm is solely data driven, it does not require any special per-material handling. On the down side, subsampling the image can miss some important features, and the algorithm can become severely memory bound. The idea has been further developed to better handle glossy surfaces [Hašan et al. 2009, Davidovič et al. 2010], lift the restriction of single clustering for the whole image [Ou and Pellacini 2011], and introduce a more advanced method of recovering the full matrix from sparse samples [Huo et al. 2015].

2.4 Acceleration Structures

The ray tracing operator $h(x, \omega)$ as well as the visibility function $V(x, y)$ introduced in Section 2.1.3 require finding an intersection between a ray and the objects in the scene. This process is commonly called ray casting, not to be confused with the rendering technique called Whitted-style ray tracing (Section 2.3.2). While for simple scenes it is possible to simply compute all ray-object intersections for all objects in the scene, it is obvious that this approach would be prohibitively slow for scenes of any practical complexity and, thus, some form of acceleration structure has to be used. In the first part of this section we give an overview of basic data structures used for this task.

Photon Mapping based approaches, introduced in Section 2.3.5, require finding k photons nearest to a given point or, alternatively, all photons within a certain distance from the given point. In the second part of this section we give a brief overview of structures used to accelerate these queries.

2.4.1 Ray Tracing Acceleration

The goal of ray tracing acceleration structures is to reduce the number of ray-objection intersection tests required to determine the first intersection along the ray (ray tracing operator) or any intersection along the ray before a certain distance (visibility function). The approaches can be divided into two broad categories based on whether they divide the scene space (a spatial subdivision) or objects (an object hierarchy).

In the spatial subdivision, each point in space is uniquely identified, but each object may need to be referenced multiple times. An object hierarchy, on the other hand, references each object exactly once, but a point in space may be overlapped by an arbitrary number of nodes in the object hierarchy (including zero). In this section we introduce three major representatives of acceleration structures coming from both categories: grids [Fujimoto et al. 1986] and kd-trees [Clark 1976] are typical spatial subdivision structures, while the bounding volume hierarchy (BVH) [Kaplan 1985] is a typical object hierarchy structure. For a more detailed discussion of acceleration structures we refer the readers to [Havran 2000, Wald et al. 2007]. For the sake of simplicity we assume that the only object that can be contained within the acceleration structure are triangles. We note that this is not necessarily the case as shown in, e.g., [Pharr and Humphreys 2004].

Uniform Grids. Uniform grids [Fujimoto et al. 1986] are the simplest acceleration structure that is commonly used. The space occupied by scene's triangles is divided into uniform cells and each cell references all the triangles that at least partially overlap it. When a ray is traced, it is marched through the cells it intersects and in each cell tested against all triangles contained within the cell. See Figure 2.14 for a 2D example.

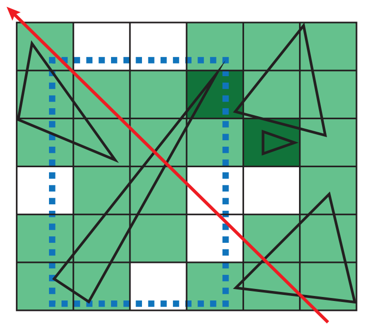


Figure 2.14: *2D Uniform Grid.* Cells contain zero (white), one (light green), or two (dark green) triangles. The traced ray is represented by the red arrow. A blue dashed line represents a bounding box of the largest triangle.

To avoid testing the same triangle multiple times (e.g., the top left triangle in our example), we can cache the intersection results from several last intersections in a scheme known as *mailboxing* [Arnaldi et al. 1987, Amanatides and Woo 1987].

Another important factor is the quality of the grid. An exact box-triangle intersection test used to determine the triangles a cell should reference is both complex and expensive and is often replaced by simpler approaches. The simplest one is to reference a triangle from each cell that intersects the triangle's bounding box. This, while very fast, can severely degrade the quality of the acceleration structure, especially in the presence of long and skinny diagonal triangles. In our example, the blue bounding box intersects twice as many cells as the triangle itself, which would lead to many unnecessary ray-triangle tests. This difference is even more prominent in 3D.

The main disadvantage of uniform grids is their inability to adapt to the scene. The classic example is a *teapot-in-a-stadium* [Haines 1988] type of scene, i.e., a large scene with sparse geometry in most places and highly detailed geometry in the middle. In such a scene, when the grid is coarse (large cells), the detailed geometry in the middle does not benefit from the acceleration structure as all of the triangles will end up in just a few cells. On the other hand, if the grid is fine, to properly accelerate the detailed geometry, the ray has to perform many grid traversal steps even in the sparse geometry areas. As stepping through even an empty cell has a non-zero cost, the efficiency of the acceleration structure drops. An analysis on whether to use a uniform grid for a particular scene or not can be found in [Hapala et al. 2011b].

Multi-level grids [Jevans and Wyvill 1988] are an extension of this approach that allows locally adapting grid resolution to the size and density of the geometry. Kalojanov et al. [2011] show that the performance of even just two-level grids can be competitive to that of the BVH.

Kd-Tree. Kd-trees [Kaplan 1985] are hierarchical spatial subdivision structures that, unlike uniform grids, can adapt to the local scene complexity. The space of the scene is hierarchically subdivided by axis-aligned planes until a termination criteria is met and leafs are created. The build is usually governed by the Surface Area Heuristic (SAH) [MacDonald and Booth 1990]. This heuristic estimates the cost of intersecting a leaf based on the surface area of the leaf, which corresponds to the probability that a random ray will intersect the leaf, and the number of primitives inside the leaf. When a node is split into two child nodes, the split plane is positioned so that the SAH cost of its children is minimized. The SAH can also be used as termination criterion, as it can estimate the relative cost of traversing one level down versus intersecting all triangles referenced by the current node. See Figure 2.15 for a 2D example with termination criterion: Each leaf references at most one triangle.

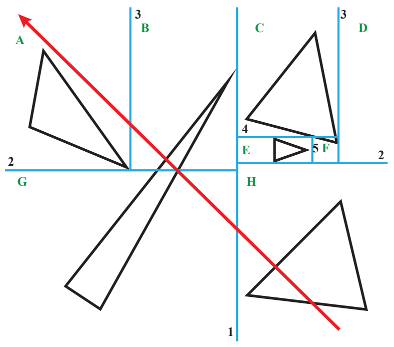


Figure 2.15: *2D kd-tree.* The space is hierarchically (see numbers) divided by axis-aligned lines, until each leaf (marked by letters) contains only one triangle. Note that the triangles can be split and thus referenced from multiple leaves.

When the ray in our example is traced through the acceleration structure, it is first tested against the topmost split plane (line in 2D case), marked with number 1. As the line is intersected, both the left and right child node will be traversed, with the right child being first, as the left child cannot contain a closer hitpoint than the right child. So, hitting in the right child automatically removes the necessity of examining the left child.

Next, we test with the line segment number 2 on the *right* side of the scene. Here the ray does not intersect the segment, so we can discard the entire top subtree containing leaves **C**, **D**, **E**, and **F** with this single test. The ray intersects the triangle in leaf **H** and terminates. If it missed this triangle, the traversal would continue on the *left* side of line number 1. For a complete survey of kd-tree traversal algorithms we refer the reader to [Hapala et al. 2011a].

Unlike the algorithm for uniform grids, the ray traversal algorithm for kd-trees requires a stack of the maximum size equal to the maximum depth of the kd-tree structure. This is not always possible as some special hardware architectures (such as older GPUs) cannot accommodate per-ray stacks. There are several approaches aimed at traversing kd-trees without a stack [Havran et al. 1998, Foley and Sugerman 2005, Popov et al. 2007]. These, however, always incur some performance penalty and should be evaluated for each architecture individually.

Bounding Volume Hierarchy. Bounding Volume Hierarchies (BVHs) [Clark 1976] take a different approach than the two previous structures. Instead of dividing the space and then testing triangles that overlap the given portion of space, the triangles are hierarchically clustered together. The triangles, respectively their clusters, are represented by their bounding volumes, most commonly axis aligned bounding boxes (other options include spheres and ellipsoids). Commonly, each inner node has exactly two child nodes, but, in general, BVHs allow for arbitrary number of child nodes which can be beneficial on certain architectures [Dammertz et al. 2008, Wald et al. 2008, Seiler et al. 2008].

The most common approach to perform the clustering is also based on the SAH. The bounding boxes of all triangles in a node are sorted along one axis and the SAH is evaluated for each possible split of such a list. After all three axis are evaluated, the best split is chosen. Popov et al. [2009] performed an exhaustive search for the best possible clustering and propose to use spatial splits in combination with the classic SAH-based clustering. Bittner et al. [2013] propose an insertion-based optimization that can be applied as a post-process after an SAH-based build to further improve the quality of the structure. In the recent years, BVHs became also popular for dynamic scenes, as there are approaches to build slightly lower quality BVHs extremely fast

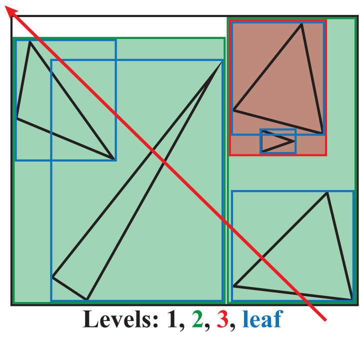


Figure 2.16: *2D Bounding Volume Hierarchy.* The root bounding box of the whole scene (black) is split into two children (green). The right child contains one more inner node (red), before reaching the leaf level, with one triangle per leaf (blue).

[Lauterbach et al. 2009, Pantaleoni and Luebke 2010, Karras et al. 2012, Karras and Aila 2013]. Figure 2.16 shows an example of 2D BVH.

Traversing a Bounding Volume Hierarchy is very similar to traversing a kd-tree – with a few modification. In each node, the ray is tested not against a single plane, but against the bounding boxes of its children. When none of the children are intersected the traversal of this node ends as obviously no object contained within the children can be hit. Note that it is possible that both children are missed even when their parent node had been hit by the ray, e.g., in the case there is a gap between the child bounding boxes, like inside the right green node of our example.

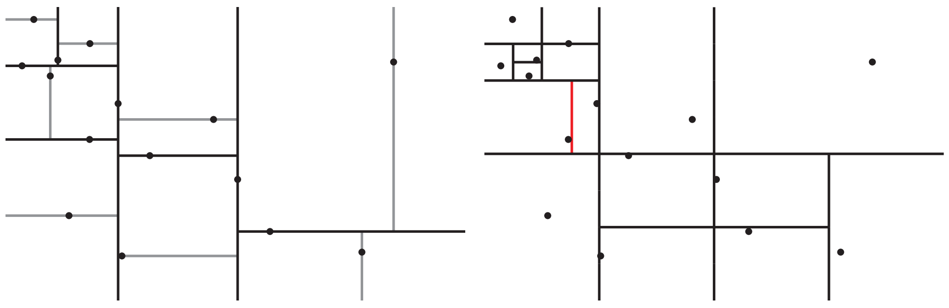
When either one or the other child is intersected, the traversal continues with that child, and when both children are intersected, the traversal continues in the "near" one, pushing the "far" one onto stack. Note that, unlike kd-trees, when an intersection occurs inside the "near" child, we cannot terminate the traversal without testing the content of the "far" child as well, as the nodes may overlap and the "far" child can actually contain a closer triangle. As in the case of kd-trees there has been research in stackless traversal approaches and the author has cooperated on the approach presented in [Hapala et al. 2011a].

They BVHs combine the kd-tree's advantage of adapting to the geometry density with larger nodes and shallower depth, both of which are advantageous on the current memory architectures. This makes BVH the ray casting acceleration structure of choice on both the CPU and the GPU and can it be found in almost all renderers.

2.4.2 Photon Mapping Acceleration

The structures used to accelerate the photon search in Photon Mapping-based approaches have quite a different task. Given a point in space, their main goal for them is to return either the closest k points stored in the structure (k-NN search) or to return all points within a radius r from the query point (range query). While these acceleration structures are used for quite different tasks, the basic structures are similar. Classic Photon Mapping [Jensen 2001] uses a kd-tree to implement its k-NN search, while Progressive Photon Mapping [Hachisuka et al. 2008] typically uses a grid for its range queries with mostly uniform radii.

Kd-Tree. There are two different basic approaches to building a point based kd-tree. Median split (Figure 2.17a) aims at the most shallow structure possible. All photons of a node are sorted along the axis of the node's longest dimension and the node is split through the median photon. These photons are stored with the plane, i.e., it is inner nodes that store photons, leaves of the tree are actually empty (and thus not explicitly stored). In our example, the final level of nodes is marked by gray split lines.



(a) Median split **(b)** Sliding midpoint

Figure 2.17: *2D point kd-tree.* Two different builds of point kd-tree. In the median split approach (top), the split plane always goes through the middle (median) photon of the node, and the photon is stored with the plane. The sliding midpoint approach (bottom) splits nodes in the geometric center. When one of the child nodes would be empty, the split plane slides to put at least one photon into each child.

The goal of sliding midpoint (Figure 2.17b) is to have nodes as cube-like as possible, even at the cost of a deeper acceleration structure. The motivation is that some types of queries (e.g., approximate k-NN queries) tend to perform faster [Maneewongvatana and Mount 1999]. The build algorithm also splits nodes along the longest axis, but in the geometrical center, irrespective of the photon distribution. The only exception is that one of the children would have zero photons, the split plane is slid so it would have at least one photon. This situation is denoted by the red line in our example. Queries into both structures differ in details (e.g., based on where the photons are stored), but the same general principles apply to both.

Executing a range query on either type of structure is straightforward. The structure is traversed from the top, for each child node we determine whether its content can or cannot be within the required range and then either process it or discard it. All photons in the processed nodes are examined and if they are within the query distance they are put into the output set.

The k-NN queries are slightly more complicated. The process works in three stages and uses a priority queue over photons, sorted by their distance from the query point. In the first stage, the query point is traversed down the tree into a leaf that would contain it, if it was a photon. All nodes that were not visited during the traversal, i.e., child nodes that did not contain the query point are put onto a stack. The photon contained in the leaf node is put into the priority queue. In the second stage the previously unvisited nodes from the stack are traversed. When a choice can be made which child to visit, the child closer to the query point is visited. All photons encountered in this stage are also put into the queue, until the queue contains k photons. After that, only nodes that are closer than the farthest photon in the queue are visited, as only these can contain photons that are nearest than our current set. After all valid nodes have been visited, the queue contains the k nearest photons.

Uniform (Hash) Grid. Under certain conditions, uniform grids can efficiently accelerate range queries, with build times that are significantly faster than for the kd-tree. When the photons are stored within a grid with cubic cells of size a^3 and the range query has a diameter $2r \leq a$, the query can be limited to only 8 cells. The 2D example in Figure 2.18 (green query) gives the intuition why. For all queries with radius $r \leq a$, valid photons can obviously lie only in the direct neighborhood of the cell containing the query. If we further limit the radius to $r \leq \frac{a}{2}$, we observe that when the query lies in the left ride of the cell, the radius cannot extend to any of the cells neighboring on the right side and the same rules apply to the other directions. However, if the radius is significantly smaller than $\frac{a}{2}$ (the red query in our example), the efficiency of the acceleration structure drops, as many photons well outside the range have to be examined. Therefore, uniform grids can be a good choice when we have an apriori knowledge of the maximum query radius and do not expect many queries with a radius significantly smaller than that.

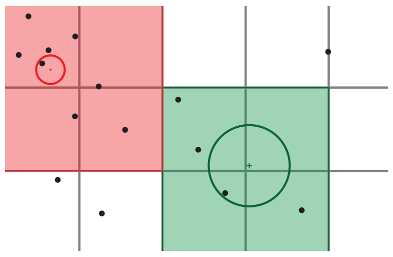


Figure 2.18: *2D point Uniform Grid.* Uniform grids can efficiently execute range queries when the radius is equal or slightly less then half the length of a cell side. Such queries (green) require examining photons in only 8 cells (4 cell in 2D case). Queries with significantly smaller radius (red) still return correct result, but the acceleration is significantly less efficient. If the radius is significantly

The basic build algorithm, i.e., sorting points into the uniform grid, is extremely straightforward and will not be described here. The only challenge is that for the typical relative size of the query radius and the scene, the resulting grid has an enormous number of cells, most of which are actually always empty as they do not contain any surfaces for which photons could even be generated. The solution proposed by Hachisuka et al. [2008] is to use a Hash Grid. In this approach the original, full grid, is never explicitly constructed and each photon is instead stored in a much smaller array at a position determined by a hash of its cell index in the full grid. When a query is made, the required cell indexes are again hashed and all photons stored at the resulting positions are examined. Due to hash collisions it is possible that the query will examine photons from cells that are out of the query's range. While in some pathological cases this make lookup in the grid almost linear with the number of photons, in practice it is not an issue.

2.5 Hardware Acceleration

To render the final image, we need to solve the rendering equation for each pixel, or subpixel, independently. For many rendering algorithms there is literally no communication required between the pixels and, therefore, the problem is widely considered to be *embarassinly parallel* [Moler 1986]. While not always completely true, e.g., generating VPLs or photons requires writing into a shared storage, it is generally true enough to make rendering suitable for massively parallel (i.e., hundreds and more threads) approaches. This is in a stark contrast to many standard applications (e.g., email clients, web browsers, text editors) that require a high single thread performance, but can utilize only a few threads. Therefore, it is only natural that rendering needs its own class of hardware dedicated to accelerating these massively parallel computations.

In this section we first look at the general principles of Single Instruction Multiple Data (SIMD) computing, as it is currently the most common hardware approach to massively parallel systems. Next, we introduce General-Purpose computing on Graphics Processing Units (GPGPU), as the most widely available massively parallel platform. And finally, we look at dedicated ray casting accelerators as a specialized alternative to Graphics Processing Units' (GPUs') dedicated rasterization units.

2.5.1 Basics of Single Instruction Multiple Data

In 1966, Michael Flynn proposed a classification of computer architectures, based on the relation of instructions and the data they process [Flynn 1972]. According to this classification, the computers we generally use are of the Single Instruction Single Data (SISD) type. A typical instruction can look like `fmul R3, R1, R2` and it multiplies registers `R1` and `R2` and stores the result in the register `R3`, each containing a floating point values.

While this is the standard model for most applications, in graphics, as well as physics, video compression, and others, we are dealing with a large amount of data, that is all processed using the same code. One such example is alpha blending of two colors, where the formula $A = \alpha \cdot B + (1 - \alpha) \cdot C$, where A, B, and C are color channels, is applied to many affected pixels. An obvious improvement in this case is to run the computation on multiple pixels in parallel, using the exact same instructions, as each pixel performs the exact same operations.

This observation is the basis of Single Instruction Multiple Data (SIMD) approach. Here each register contains several numbers instead of just one and the operations are applied to each of these numbers. These registers and their associated arithmetic units are also called *vector registers* or *vector units*, as they contain, and operate on, vectors of numbers. The term *SIMD lane* (or simply *lane*) is used to identify elements of the register, such that lane 0 means we are talking about the 0th elements of registers, and the *lane count* refers to the number of lanes in each register. On the CPU, we most commonly encounter vector units with 4 (SSE [Raman et al. 2000, Oberman et al. 1999]) and 8 (AVX [Lomont 2011]) lanes, but other values are also quite common (e.g., SSE has 4 single precision float lanes, but only 2 double precision lanes, while Xeon Phi has 16 single precision lanes).

For brevity, the rest of the explanation will assume 4 lane SIMD. A typical instruction of this class would be `simd.fmul V3, V1, V2`, which would take the four floating point values stored in vector register `V1`, multiple each individually with a corresponding float number from a vector register `V2`, and store the individual results into a vector register `V3`. That is, it performs the operation `for i in 0 to 3:  V3[i] = V1[i] V2[i]`, where `[i]` represents the number in the ith lane of a given register. This is well complemented by the current wide memory buses, where the standard CPU 128-bit buses can deliver four floats at once. On the GPU, the buses are even wider (up to 512 bits or 16 floats) on the Radeon R9 390 cards [AMD 2015]. Therefore, with properly aligned memory, SIMD can use a single instruction to load four floats into a register at no additional cost over loading a single float and achieve a theoretical speed up of 4×.

Unfortunately, while the general code to determine color is the same for each pixel, there can still be differences that prevent a straightforward use of SIMD. The first obvious deviation from the principles mentioned above would be texturing, where the neighboring pixels might need to access texels that are not in a contiguous memory (e.g., the texture is rotated or needs filtering, most often both). In this case, we would still map the neighboring 4 pixels to the 4 lanes of vector registers but the sources for the computation would come from random memory locations. To achieve this, we need to implement a *gather* operation where, given 4 memory addresses, 4 floats are fetched into the 4 lanes, one from each memory location. This can be implemented either in hardware, where the memory controller analyzes the addresses and issues as few memory reads as possible using crossbars and other machinery to land the floats in their respective final lanes. Or it can be implemented in software, where the same process is done on the instruction level. The former approach is common to most GPUs while the latter is used on the standard x86 CPUs.

The basic approach is to perform a 128 bit aligned load of 4 floats from each of the 4 addresses, use bit operations (`AND` and `OR`) to reduce the up to 16 floats to the required 4, and then use swizzle operations (moves floats between lanes) to move them into the target lanes. If two addresses would load the same set of 4 floats the second load is skipped, so loads from consecutive (or identical) addresses can be up to 4× faster when compared to the worst case. An inverse operation, *scatter*, also exists and stores results from the individual lanes onto 4 separate addresses.

The principle that allows us to execute the gather code on only some of the lanes is also used to implement conditional statements, the second deviation from the straightforward application of SIMD. The code that computes the final pixel color will often have conditional statements (e.g., *if*) where individual pixels take different branches. One example would be clamping of opacity, where pixels with opacity greater than 95% would have it snapped to 100%. For each condition, there are generally three possible outcomes: all lanes take the *then* branch; all lanes take the *else* branch; some lanes take *then* and some lanes take *else*. The first two cases are fairly trivial, the condition simply changes the instruction flow and all lanes use the instructions from the given branch. However, in the case where the lanes disagree, the code will take both branches (i.e., first *then* and then *else*), using the previously introduced masking principle to store results of each branch for only the lanes that should be affected.

While it is possible to implement these principles by hand, explicitly taking care of the gather, scatter, and branching, there are also automated tools that make this task easier. In 2007, NVIDIA introduced their Compute Unified Device Architecture (CUDA, [NVIDIA 2015]) and with it a programming model they called Single Instruction Multiple Threads (SIMT). In this model, the programmer writes a standard scalar code that is then automatically mapped onto a SIMD machine. The language provides built-in variables that the program can use to identify in which SIMD lane it is executed, which allows to map computation (e.g., pixel index) to a given lane. The same approach is also used by the cross-platform OpenCL standard [Khronos OpenCL Working Group and Munshi 2015]. Intel's `ispc` compiler [Pharr and Mark 2012] compiles scalar code in slightly extended C language the SSE and AVX SIMD targets, as well as to Intel's Xeon Phi coprocessors [Reinders 2012]. The output of the compiler can be linked into a standard CPU application allowing the users to use only for certain parts of their algorithm. The programming model is the same as the in CUDA and OpenCL, except that Intel calls this approach Single Program Multiple Data (SPMD). A more detailed description of CUDA and its terminology is given in the Section 4.2.1.

2.5.2 General-Purpose computation on Graphics Processing Units

While the history of general video acceleration starts as early as 1970's (the *Gun Fight* arcade used Fujitsu MB14241 video shifter to accelerate 2D sprites), this section will focus on the more recent 3D accelerators, know as Graphics Processing Units (or GPUs). We refer the interested readers to Kumar et al.'s overview of the These, eventually, gave rise to General-Purpose computation on Graphics Processing Units (GPGPU), where the GPU is used as a very wide Single Instruction Multiple Data (SIMD) coprocessor with a specialized memory model.

At first, the computing power and high bandwidth of GPU came from its highly fixed functionality. The main goal was to solve primary visibility, that is, for each pixel determine which triangle in the scene is the closest to camera and display its color. This was done via fixed function rasterization units that, for each triangle of the scene, solved which pixels are covered by the triangle and how far the triangle is from camera. Then, the color of each of these pixel-triangle intersections (fragments) was determined. Often, a fixed lighting model (e.g., Lambert of Phong) would be evaluated at each corner of a triangle and the result interpolated inside the triangle. Lastly, the distance (depth) of the fragment would be compared to the currently closest fragment for each given pixel (using a dedicated depth buffer) and if the new fragment was closer, its color was written to the frame buffer.

While this method of determining which triangles should be displayed was extremely efficient and is still used on the most recent desktop GPUs, the control a programmer had over the triangle's color was severely limited to just a few parameters, e.g., surface color or light position. The fixed function pipeline could not accommodate the ever-increasing demand for higher quality and flexibility, which lead to the GPUs becoming more and more programmable. Initially, the capabilities were extended by simple assembly-like languages, with standards like *OpenGL ARB assembly language* and *DirectX Shader Assembly Language*. Later, these have been superseded by higher level fully featured C-like languages such as HLSL [Gray 2003] and GLSL [Rost et al. 2009] (see [Buck 2010] for the history of programmable shading). The shading hardware of the current generations of GPUs can be considered fully programmable wide SIMD coprocessors and their full functionality can be accessed using modern languages such as OpenCL [Khronos OpenCL Working Group and Munshi 2015] and CUDA [NVIDIA 2015]. The only two components that are left mainly as fixed function are the rasterization and texturing units.

The programming model for both GPU and CPU can be seen as fairly similar, especially if the latter is programmed using the `ispc` or a similar tool to generate SIMD code. The main differences are in the number of threads that can be in flight at the same time and the access to the main memory, but these differences still can be an order of magnitude.

Let us first note that, in this discussion, we will not be using the NVIDIA's definition of a *thread* as a single SIMD lane, but rather the more universal definition of a thread as a scheduling unit. Initially, the common x86 CPUs would have a single hardware thread running on each CPU core. The Operating System would use its scheduling algorithms to assign the many software threads (e.g., an email client, a word processor, a renderer) to these hardware threads. A common practice is to reschedule threads on *page fault*, that is, when the active thread requested a memory page that was not currently in main memory, but on disk. As loading the page from a disk is a relatively long process (in terms of CPU cycles) and the CPU would be idling before it could get the data, it was more efficient to assign it to another software thread.

The principle was later extended to *Hyper Threading*, where multiple hardware threads share the compute resources of a single core. The number of threads per core is often fixed and based on the number of register sets available in hardware, as each thread has its own logical registers. The modern x86 CPU with 6 cores can, therefore, have up to 12 hardware threads running at the same time. This allows for a faster and more fine grained level of switching controlled by the CPU itself and can hide not only page faults, but also cache misses. The principle where another thread is scheduled while the original thread waits for a memory request is also called *latency hiding* as it effectively hides memory latency from the user. Given that most modern applications rely heavily on accessing large amounts of data this can significantly increase the CPU resource utilization. While the theoretical speed up of 2x from doubling the number of hardware threads is rarely achieved, in the context of rendering applications a speed up of 1.5x is a fairly common occurrence when Hyper Threading is enabled.

The same latency hiding approach is used on most of the modern GPUs. Unlike the CPU, the number of threads per core can vary based on the resources required by the threads, e.g., there is a global register pool on the core and can be divided arbitrarily between the threads. The NVIDIA's GeForce GTX 980 currently has a maximum of 64 threads (in the NVIDIA terminology *warps*) per core (Streaming Multiprocessor - SM) and 16 cores, for the total of 1024 hardware threads [NVIDIA 2014]. At the same time, while the CPU SIMD units have 4 to 8 lanes, the GTX 980 has 32 lanes. On the CPU we can therefore schedule $12 \cdot 8 = 96$ lanes of computation simultaneously, while the GPU can arrive at the significantly higher number of $64 \cdot 16 \cdot 32 = 32768$ lanes. In practice, the number of scheduled lanes on the GPU is significantly lower (i.e., the code usually needs more than the minimal amount of resources), but the discrepancy is still large and to fully utilize the GPU resources the actual algorithm has to account for the need of large numbers of concurrently active lanes.

The second major difference is in the memory access itself. Most high performance GPUs, such as the aforementioned GTX 980, are expansion cards connected to the main memory via PCI-Express. And while the bandwidth of a 16 lane 3.0 PCI-Express is impressive 15.75 GB/s, it is still more than an order of magnitude slower than the bandwidth when accessing the GPU's on-card memory (225 GB/s for the GTX 980). Coupled with the higher latency of accessing memory through the PCI-Express bus, it is obvious that high performance applications need to mainly target the on-card memory. On the other hand, the memory itself has a significantly higher bandwidth than even dual channel DDR3-2400 main memory (38.5 GB/s), which helps with memory bound algorithms.

It is also important to note that CPUs are still heavily focused on single-threaded performance and their memory hierarchy is tuned to help with this goal. On the other hand, the memory hierarchy on the GPU is mostly biased towards providing maximum throughput and the memory latency is hidden by other threads (i.e., similar to Hyper Threading), rather than actively reduced by complex caching and out-of-order instruction issue mechanisms. To utilize this throughput, it is important to provide the GPU with comparatively large sets of work, to keep all threads occupied as much as possible. For an extreme example, it is obviously much more effective to have the GPU process a single set of a million rays, rather than a million sets of a single ray, as the latter would leave majority of the GPU unoccupied. Realistic scenarios of work set sizes are examined in Section 4.3.4.

Lastly, we will mention an issue that could be encountered on large production scenes. The on-board memory is fairly small when compared to maximum main memory sizes, as few GPUs have more than 10 GB, while 32 GB or more of main memory is quite common. All the scenes presented in our work fit into the on-board memory of the used hardware but the ever present production drive for higher model detail, more complex scenes, and more detailed textures means that some production scenes would not fit. So, while the topic is beyond the scope of this work, let us briefly describe one of the basic approaches that could be used to address this issue. The idea is to look at the out of core approaches used on the CPU when the scene does not fit into main memory and has to be stored on the disk. In this case, the memory is used as a cache and various tiling and sorting schemes are employed to minimize the number of cache evictions while at the same time servicing the requests. We refer the readers to the papers by Pantaleoni et al. [2010], Eisenacher et al. [2013], and Laine et al. [2013] for inspiration.

2.5.3 Dedicated Ray Casting Units

So far we have only discussed the fully programmable special hardware, used in computer graphics. But the GPU also contains two important fixed function units. The first one is a *texture unit* that, given a properly laid out texture, can do extremely efficient filtered texture lookups into compressed textures. This is a feature that can be utilized by any rendering algorithm and does not require further discussion in the context of this thesis. However, the other fixed function unit, the *rasterization unit*, represents the core of the rasterization rendering algorithm and thus requires more detailed description.

The goal of the rasterization unit is to answer, for each triangle, the following question: “Which pixels are covered by this triangle, and how far is it?” This is almost identical to the question that ray casting acceleration structures help to solve: “Which triangle is the closest, as seen through this pixel?”. Except the question is asked from the position of the triangle, not the pixel.

The main advantage of this approach is that we do not need any acceleration structure and can just stream all the triangles through the rasterization unit to obtain the final answer. The main disadvantage is that this is only directly applicable to perspective and orthographic projections, i.e., all the rays have to share either origin or direction. If we have multiple such sets of rays, we have to stream all the triangles for each of them. So, while this approach is useful for things like primary visibility or shadows from point lights, it cannot answer general visibility queries the way ray casting acceleration structures can. We refer the reader to our *3D Rasterization: A Bridge between Rasterization and Ray Casting* [Davidovič et al. 2012a] for further discussion on the similarities and differences between the two approaches.

It is undeniable that hardware rasterization units are of great benefit to the overall performance of the GPU. The obvious question is whether a similar advantage can be leveraged with a dedicated *ray casting unit* and what are the challenges.

Let us first discuss the simpler case of a completely static scene. While this scenario is not particularly exciting for real-time applications, e.g., games, it still has a lot of merit when high quality is preferred over frame rate, such as in movie production where a single frame can take anywhere from several minutes to several hours to compute.

As we mentioned earlier, the main difference between rasterization and ray casting units is that ray casting can process arbitrary and independent ray queries. To achieve this, we need to have the full scene geometry accessible at all times, either by fitting it all in memory or by utilizing some caching mechanisms. Related to memory is also the problem that a large number arbitrary queries into your data requires a substantial bandwidth. The most straightforward approach, when dealing with bandwidth problems, is to introduce a memory hierarchy in the form of caches. This, in itself, is often sufficient enough to accommodate a moderate number of concurrently traced rays. For example, this approach is used in the FPGA implementation of the RPU and DPU [Woop et al. 2005, Woop 2006], which will be discussed in more detail in Section 3.

Unfortunately, as the complexity of the scene grows, a simple caching scheme might not be enough. Aside from the bruteforce option, i.e., increasing the cache size, it is also possible to increase the cache hit rate by sorting the ray queries by their origins and directions. When a sufficiently large number of ray queries is used, this leads to consecutive rays being fairly coherent, which helps to improve on the temporal locality and thus cache performance. This can be further improved by partitioning the scene and re-sorting rays on the partition boundaries, similar to the CPU out-of-core approaches such as in PantaRay [Pantaleoni et al. 2010]. The drawback of this approach is that it requires a large number of concurrently active rays to find any coherence in the queries and, while suitable for some rendering algorithms, e.g., Forward Path Tracing, it can be problematic for algorithms that store significant state per path, e.g., bidirectional path tracing. Notable hardware designs that adopt this approach are StreamRay [Ramani et al. 2009] and Caustic One [Caustic Graphics, Inc. 2009][3].

The other critical issue is the choice of an acceleration structure. As discussed in Section 2.4.1, there are many valid options and there is no clear candidate for the best one. Based on the research on the GPU (see [Aila and Laine 2009]), it seems that BVH structures are better suited for hardware acceleration than kD-trees, due to their larger nodes and shallower structure. The shallow structure means that the memory system will need fewer data fetches to reach a leaf, while the larger node means that each fetch will bring more relevant data. Ideally, the node would be as wide as the bus width, so each memory transfer would bring exactly one node with no unnecessary data. And, as mentioned earlier, efficient use of memory is of utmost importance for the overall performance. Another good candidate would be uniform or multi-level grids, as they are fast to build (see [Kalojanov and Slusallek 2009, Kalojanov et al. 2011]) and have a predictable traversal order, which allows for efficient pre-fetch of data from memory. Unfortunately, if we leave the actual ray-structure intersection fully programmable, the hardware acceleration would be limited only to ray-triangle intersection and, possibly, a specialized memory hierarchy. Therefore, to fully utilize the potential of hardware acceleration, a choice of acceleration structure has to be made and fixed for the given hardware. Please note that it is still possible to identify operations common to multiple structures and implement hardware acceleration for those, but this is yet another tradeoff between programmability and performance.

The issue becomes even more prominent if we include the common requirement for dynamic scenes in real-time setting, such as when ray tracing would be used to produce effects in games. Here, we have to distinguish two level of dynamics. In the simpler case the scene is only deforming, that is, the number and the topology of triangles is the same across the frames and it is only the triangle vertices that are moving. This allows BVH-based structures to be built once and then refitted as the objects move.

Unfortunately, in the more general case of dynamics, where the number of triangles can change between frames, refitting is not an option and a new acceleration structure needs to be build for every frame. While some of the architectures cites here did include hardware for building acceleration structures, e.g., the DRPU [Woop 2006], it is not the main topic of this thesis and it will not be discussed in detail. For those interested in the topic, a good starting point may be the body of work on building and re-building

[3]The company has been bought by Imagination Technologies and their marketing resources are no longer available.

acceleration structures on the GPU, such as [Kalojanov and Slusallek 2009, Lauterbach et al. 2009, Karras and Aila 2013].

Chapter 3

A Dedicated Ray Traversal Engine

In Section 2.1.3 we have introduced the rendering equation in the ray tracing operator form (Equation 2.8):

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega^+} f_r(x, \omega_o, \omega_i) L_o(h(x, \omega_i), -\omega_i) \cos\theta_i \mathrm{d}\omega_i$$

The purpose of the ray tracing operator $h(x, \omega)$ is to identify the source of radiance arriving at a point x from a direction ω by finding the closest surface point in the scene along a ray defined by this point and direction. Unlike the $L_e(\cdot, \cdot)$ and $f_r(\cdot, \cdot, \cdot)$ terms that are defined locally at a given point, the ray tracing operator has to consider the whole scene and its cost increases with the scene complexity. With the naive approach of testing each ray against all objects in the scene the ray tracing operator would quickly completely dominate the total cost of the equation. Therefore, at least some form of acceleration of these ray casting queries is vital for any practical application.

Three of the most common acceleration structures uniform grids, kd-trees, and bounding volume hierarchies have been introduced in Section 2.4.1. A large body of work on build algorithms for acceleration structures, traversal algorithms, and packet traversal, published in the past fifteen years, caused tremendous speed up of the CPU ray casting performance, e.g., [Havran 2000, Wald et al. 2001, Wald 2004, Reshetov et al. 2005, Günther et al. 2007, Stich et al. 2009, Hapala and Havran 2011]. With the introduction of more programmable GPU we have also seen significant improvements in the performance of ray casting queries on the GPU, with the works by Aila, Laine, and Karras [Aila and Laine 2009, Aila et al. 2012, Karras et al. 2012] achieving the peak ray tracing performance and NVIDIA's OptiX [Parker et al. 2010] providing a simple-to-use ray tracing framework.

However, despite all these advances, real-time applications are still a domain of rasterization. One of the key advantages of rasterization is the extremely efficient dedicated hardware rasterization unit and its tight coupling with the programmable cores used for shading. While, in principle, rasterization also uses the ray tracing operator to find the surfaces closest to the camera, the implementation greatly benefits from all rays having a common origin x and the ray directions ω being parametrized on a 2D pixel grid. Standard ray tracing does not have such limitation on rays, which gives it a huge advantage in the terms of flexibility. However, dealing with arbitrary rays also poses several challenges detailed in Section 2.5.3 and mainly focus on the lack of locality when processing the rays.

Despite these challenges, there have been many different dedicated hardware ray tracing units introduced in both academic and commercial settings. The first implementations focused on testing rays against primitives in efficient fashion [Plunkett and Bailey 1985, Owczarczyk 1988], but they did not utilize any acceleration structures and instead chose the brute-force approach of testing all rays against all primitives. One of the earliest architectures to utilize acceleration structures was the TigerSHARK Architecture by Humpreys and Ananian [1996]. The authors use a wide (16-32) SIMD approach to test a primitive against many rays at once. Each SIMD lane is represented by a ray intersection unit that can intersect a ray against a primitive, with all units testing the same primitive. The authors use a Bounding Volume Hierarchy where a subtree is skipped if all lanes miss its bounding box. The input rays are assumed to have a significant spatial coherence to leverage this principle and the approach would work poorly on randomized rays, due to the "all rays test the same primitive" requirement.

A somewhat similar approach is taken by Ramani et al. [2009] who introduce the StreamRay, a hardware architecture based on stream filtering of rays. Here the rays are organized into streams and, using the acceleration structure tests as filters (e.g., go left/right/both/neither), are gradually filtered into substreams of similar rays. This partitioning of rays into substreams allows for reducing bandwidth towards the acceleration structure and geometry data, as a node or primitive can be fetched once and used for all the rays in a substream. However, to be efficient, the approach requires a large number of active rays and in algorithms that require a substantial path state (e.g., bidirectional path tracing) this can put pressure on memory.

The TRaX architecture by Spjut et al. [2009] explicitly does not utilize SIMD to process multiple rays at once, and instead focuses on accelerating the single ray performance. It takes advantage of the specific memory access patterns, e.g., read-only scene memory, and designs its memory system around them. This architecture is later used as a basis for the works by Spjut et al. [2012] and Kopta et al. [2013, 2015] on energy efficient ray tracing, targeted at the mobile segment.

At Saarland University, Philipp Slusallek's group developed three generations of ray tracing hardware. First, in 2002, Schmittler et al. introduced the SaarCOR hardware design [Schmittler et al. 2002]. This design consists of fixed function traversal and ray-triangle intersection units, using 32 packets of 4 rays each to hide memory latency. Unlike most of the previous architectures, this design also incorporated shading, albeit only with a fixed function unit.

To address the fixed function limitations, Woop et al. presented the RPU [Woop et al. 2005] architecture. The RPU provides programmable SIMD shader units coupled to fixed function, dedicated traversal units, using a kd-tree acceleration structure. The last generation of the design, also developed by Woop, is the Dynamic RPU, or DRPU [Woop 2006]. It adds support for dynamic scenes by using refittable B-KD trees [Woop et al. 2006b] instead of the previously used kd-trees.

As can be expected, all of the introduced publications focus mainly on the acceleration of the ray tracing operator itself. However, due to the generally recursive nature of ray tracing, the coupling between shading and ray tracing is of even greater importance than in rasterization. Rasterization is, generally, a streaming process. A triangle is rasterized into fragments, those are sent for shading, and then are either stored in the framebuffer or not. The ray tracing shaders, on the other hand, generally produce other rays, e.g., shadow rays or reflection rays leading to a feedback loop and two-way communication between the units. Even though some of the architectures do have their own programmable shading units, these units were not the main focus of the work and their capabilities are, obviously, fairly limited.

In this chapter we focus on this missing piece and present a solution that proposes to place a small dedicated hardware ray traversal engine (RTE) directly on the die of a modern processor that can be used to execute general purpose shading. For our evaluation we chose the Cell Broadband Engine™ (Cell/B.E.), due to its unique modular architecture, but the results do not, in any way, depend on this choice and can be transferred also to other architectures.

3.1 Ray traversal engine

Our ray traversal engine (RTE) implementation is based on the well-documented FPGA implementation of the DRPU by Woop [Woop 2006]. We extend the previous investigation into the ASIC implementation of this design [Woop et al. 2006a] by considering options given by a fully custom 90 nm process by IBM. We will first give a brief overview of the basic blocks used, then discuss frequency, area scaling, and the connected design decisions.

3.1.1 Design blocks

The RTE (Figure 3.1) is a heavily pipelined unit with finegrained multi-threading. It uses B-KD trees [Woop et al. 2006b] for its acceleration structure. Our B-KD tree implementation offers a two-level hierarchy, where each leaf of the top tree can contain either a triangle or a transformation matrix and pointer to a subtree.

The basic unit of computation is a thread, which processes four rays at once. Each pipeline stage can accommodate a different thread and there is no overhead in switching threads. The whole RTE can process up to 64 threads (256 rays) at once. While there is no strict requirement for coherence of rays within the same thread, it can bring a performance boost. Also, coherence between all currently processed rays improves cache hit rates.

The IO unit strongly depends on the particular implementation details of integrating RTE with the rest of the system. We will therefore not elaborate on its implementation, but we can imagine it as a kind of memory controller that handles ray and result queues in main memory, akin to what is assumed in [Aila and Karras 2010]. The basic functionality is to fetch rays for idle threads and store results from finished ones.

The actual RTE core consists of two units: traversal and geometry. Each is connected to main memory via a 4-way associative 32 kB read-only cache. A single larger cache has been considered, but there would be either high contention for its single port, or it would have to be two port cache,

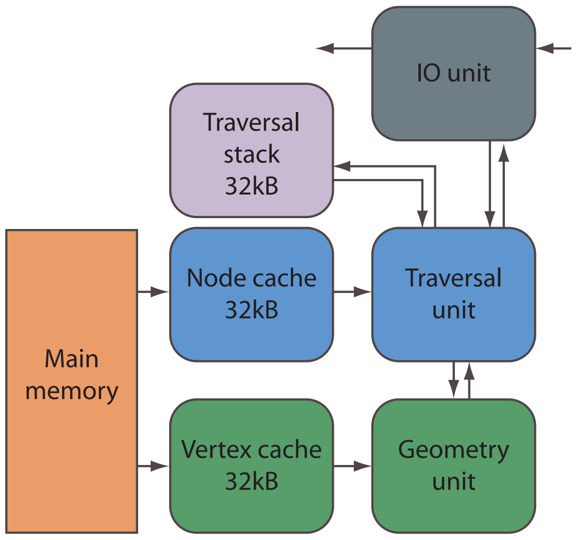


Figure 3.1: *RTE block diagram.* The RTE core consists of traversal and geometry units. Each unit is connected to main memory via 32 kB cache. Traversal unit has an additional 32 kB memory for traversal stack and connection to the IO interface to return results and receive new queries.

resulting in higher complexity. Also, as each unit requires different data, there is no redundancy in fetches from main memory that would lower the efficiency.

Another two parts, not shown in Figure 3.1 are a per-thread stack memory and a ray state buffer. The per-thread stack is accessed with up to 1 read and 1 write each cycle, where the read occurs when a thread is submitted to traversal unit and the write occurs when the result of traversal stage is that both children should be intersected. Each item in the stack consists of a node address (4 B) and the near and far values for each ray (32 B), a total of 36 B per item. For convenience we consider a maximum scene depth of 32. Making the internal stacks shorter and spilling overflow into main memory, as described in [Aila and Karras 2010], would also be an option.

The ray state buffer is a per-ray storage that stores the current closest hit distance, the ID of the closest triangle, and the barycentric coordinates of the hitpoint.

For each thread, the traversal unit fetches the required node. If the node is a leaf it is sent to the geometry unit. For inner nodes it intersects the child nodes and based on the result either pops a new node from stack, traverses the single hit child, or traverses the closer child while pushing the farther child onto the stack. If the stack is empty it informs the IO unit that traversal has completed.

The geometry unit computes both the intersection with triangles and the transformation of rays for traversing the bottom tree when a two-level hierarchy is used. This is possible because both operations require very similar functional units with a minimal reconfiguration. When this unit receives a leaf node, it first fetches either three vertices (for intersection) or matrix rows (for transformation). The thread waits until all three memory fetches are finished before further execution.

For some of the measurements, We additionally equip the geometry unit with a 1 MB 4-way associative L2 read-only cache. For this cache we assume a latency of 100 cycles, and a 512 bit wide access to memory (fetches 64 B at once). When the L2 cache is not present, the L1 caches are connected directly to memory and assume a 128 bit wide access. The main memory is assumed to have a latency of 600 cycles.

3.1.2 RTE synthesis

We first synthesized all the major arithmetic sections of both units as well as the cache logic using a fully custom proprietary IBM 90 nm process [Davidovič et al. 2009]. All synthesized parts were able to run at frequencies of over 2 GHz. This was done without any special fusion of dependent arithmetic units. Also, considering the already quite high latency of both traversal (14 cycles) and geometry (36 cycles) units, more stages could be added to further stabilize the frequency without a significant impact on the overall performance. While both optimizations can lead to higher frequency than reported, we opted against their implementation as the preliminary results are more than satisfactory.

We have not explicitly synthesizes all the required memories, instead we borrowed statistics from similar memories from the Cell/B.E. processor [Flachs et al. 2005]. This processor has a 32 kB L1 cache running at 3.2 GHz as well as 1-read, 1-write port local store memory running at the same frequency. As these two types of memory fulfill all requirements we have on our memory blocks we conclude that memory frequency will not be an issue.

The area of the RTE is not of the prime concern for our results, so we present only a rough upper bound. We used a range of area estimation techniques to confirm that RTE comfortably fits into the area of less than 15 mm^2 which is the area of each of the Synergistic Processor Unit (SPU) coprocessors of Cell/B.E.. Further reduction of the size is possible by the aforementioned fusion of arithmetic units.

3.2 Simulator architecture

We have performed two distinct simulations. The first simulation was on the actual low-level VHDL code, that is used as a base for our synthesis results. While this confirms that the design is correct and the synthesized frequencies valid, the simulation itself is very slow.

To solve this we also designed a cycle-accurate SystemC [Open SystemC Initiative 2006] model of the RTE. SystemC is a C++ library that allows for using many high-level language constructs that are not available in VHDL, while at the same time allowing cycle accurate simulations. We integrated this RTE model as part of our ray tracing framework, essentially replacing its standard traversal and intersection routine.

Normally our software framework uses the following workflow: First, a primary ray is generated from the camera. It is then intersected with the scene and the closest geometry is found. If there is any hit, an integrator is invoked that queries material for its BRDF, retrieves lights from the scene description, performs shading, and possibly generates other rays. Once the final color of the primary ray is computed, another primary ray is generated, until the whole image has been rendered.

While this sequential process works very well for CPU rendering with low amounts of parallelism (4-16 threads at once), it is ill-suited for massively parallel hardware acceleration. We have therefore modified the algorithm as follows. First, all primary rays are generated and put into the input queue of the RTE. The RTE emulation engine is then started and after each cycle its output queue is checked. If there is a ray in the output queue, it is passed to the integrator that processes the ray. This can, and generally does, generate other rays, that are in turn fed back into the input queue ahead of any stored primary rays. When there are no more rays in the input queue and no active rays are in the RTE, the frame has finished. Should generating all primary rays occupy too much memory at once, it is also possible to split the image into several blocks or batches, based on the memory configuration.

3.2.1 Shader models

Unfortunately, this system does not lend itself easily to the standard recursive shaders, nor to any kind of shader where the integrator needs result of a traced ray. There are two possible solutions to the problem. Firstly, we can use tail-recursion, for example adopted by [Caustic Graphics, Inc. 2009]. Here the integrator generates all required rays in a fire-and-forget manner, attaching to them all the necessary information. Shadow rays would, for example, carry the radiance that should be added if they are not occluded. When a shadow ray hits the light it was generated for, the contribution is added to the pixel value. If the shadow ray hits any other object the light is occluded and no contribution is accumulated.

This approach has several drawback. It lacks adaptivity, i.e., it is not possible to evaluate next event estimation samples and decides that it would be optimal to shoot more such samples. It spawns large and poorly controllable number of rays. Multiple rays independently writing to the same pixel require atomicity and can lead to poor memory locality.

The other approach is to use continuations. Here the integrator is effectively split at each call for *traceRay* and its state is stored. Once the tracing of a ray has been finished, the previous shader state is fetched from a global storage and the integrator continues. This can be easily implemented by a finite state machine, but requires a stack in the case ray bifurcation is allowed. One of the advantages is that all the computations concerning one pixel are self-contained in the shader which precludes the atomicity and locality requirements of tail recursion. It also allows examining the partial results and based on them determine the actual number of samples to take. The disadvantages include larger per-ray storage and higher sensitivity to latency.

3.2.2 Acceleration structure partitioning

To increase cache coherence, Aila and Kerras [Aila and Karras 2010] introduce the concept of partitioning the acceleration structure into multiple treelets (subtrees), each of which fits into the L1 cache. As we consider this approach highly relevant for our results, we adapted it for B-KD-trees and included it in our measurements.

The basic principle is that the whole structure is split into many small subtrees, called treelets. Each of the treelets has its own ray queue and when a ray crosses a treelet boundary, its traversal is stopped and it is put into the corresponding treelet's input queue. While the original paper introduced several methods for scheduling treelets to ray traversal engines, we use only the simplest one called *lazy scheduler*. It simply takes the treelet with the largest queue and processes it until the queue is empty. Then it switches to another treelet, with the currently largest queue. The more complex methods are meant to balance situation where there are multiple ray traversal engines, which is something we do not currently consider for our scenario.

3.3 Results

We tested the design on three scenes of moderate complexity, the **Conference** (283k triangles), **Fairy forest** (174k triangles), and **Kitchen** (253k triangles). We test both shader approaches using 16 rays per pixel ambient occlusion. We focus on the cache hit rates, throughput (rays per second), ray latency (important for continuation shaders), and the bandwidth requirements of our unit, measured both with and without an L2 cache.

The reason why we provide not only cache hit rates, but also their bandwidth is that one should not automatically assume that higher hit rates are more desirable. The perfect ray tracing algorithm would need to access each node and triangle at most once and the caches would have a hit rate 0% but the cache bandwidth would significantly lower. We also cannot consider bandwidth in isolation as the simplest way to achieve the lowest possible bandwidth is to not trace any rays. Therefore, for evaluation we need to consider the ray throughput, the bandwidth it requires, and the cache hit rate that helps to lower the bandwidth together and draw conclusions only from the combination of the three, as we will see below.

In Section 3.3.1 we provide results for RTE without the use of treelets, we follow up with Section 3.3.2 commenting on the results using treelets and close with Section 3.3.3 on using BVH instead of B-KD-tree.

3.3.1 Standard implementation

Table 3.1, providing results for the standard RTE implementation, offers several insights. First and most important is, that the L2 cache helps in all the scenes, giving us a speed up by 1.3-2.3×. The reason for this is somewhat obvious, the L2 cache provides large portion of the scene with lower latency (100 cycles) than main memory (600 cycles). The L2 cache also exhibits relatively high hit rate, above 85 % for nodes and over 70 % for vertices. It is important to note that while the L2 hit rate is higher for continuation shaders, this is actually caused by the lower L1 cache hit rate, which causes more requests to L2 cache, increasing the bandwidth as noted at the beginning.

Conference
Fairy forest
Kitchen

Continuations shaders						
L2	off	on	off	on	off	on
L1 hit rate [%]	68/53	69/55	55/44	56/45	83/77	89/85
L1 bandwidth [$\mathrm{GB\,s^{-1}}$]	6.7/4.7	15.7/11.0	5.0/3.4	11.7/7.9	11.2/7.6	16.7/11.3
L2 hit rate [%]	-	95/84	-	88/73	-	90/80
L2 bandwidth [$\mathrm{GB\,s^{-1}}$]	-	4.8/4.9	-	5.2/4.3	-	1.8/1.7
Ray bandwidth [$\mathrm{GB\,s^{-1}}$]	0.14	0.14	0.14	0.14	0.14	0.14
Mem bandwidth [$\mathrm{GB\,s^{-1}}$]	4.4	4.2	4.3	7.5	3.7	2.1
Latency [cycles]	6.0 M	2.7 M	12.0 M	5.4 M	3.3 M	2.3 M
Throughput [MRays/s]	47.8	112.4	20.8	48.1	66.6	**100**
Tail-recursive shaders						
L2	off	on	off	on	off	on
L1 hit rate [%]	76/64	78/67	63/53	64/55	87/82	85/80
L1 bandwidth [$\mathrm{GB\,s^{-1}}$]	8.3/5.8	16.3/11.2	6.0/4.1	12.5/8.4	12.6/8.5	16.8/11.4
L2 hit rate [%]	-	93/79	-	86/71	-	93/86
L2 bandwidth [$\mathrm{GB\,s^{-1}}$]	-	3.6/3.7	-	4.5/3.8	-	2.6/2.3
Ray bandwidth [$\mathrm{GB\,s^{-1}}$]	0.14	0.14	0.14	0.14	0.14	0.14
Mem bandwidth [$\mathrm{GB\,s^{-1}}$]	4.4	4.1	4.1	6.2	3.3	2.1
Latency [cycles]	4.9 M	2.6 M	10.0 M	5.0 M	3.0 M	2.3 M
Throughput [MRays/s]	61.1	**117.4**	25.0	**52.3**	75.4	99.2

Table 3.1: *Results without treelets.* We measure all three scenes with both tail-recursive and continuation shaders and both with and without 1 MB L2 cache. The reported results are L1 cache hit rate, required L1 bandwidth in $\mathrm{GB\,s^{-1}}$, the same for L2 cache (if applicable), ray traffic bandwidth in $\mathrm{GB\,s^{-1}}$, total required memory bandwidth, both from cache and rays, latency in cycles (Latency) and throughput in million rays per second. The L1 and L2 cache results are given in format vertices/nodes.

The ray bandwidth, created by reading and writing rays from input and into output queues, is an order of magnitude lower in all the measurements than the total bandwidth to main memory and thus relatively insignificant. We can also see that with the total memory traffic between 2 and 6 $\mathrm{GB\,s^{-1}}$, we are well beneath the peak performance of current GPU memory systems.

Another very important aspect is the ray latency (noted in Table 3.1 simply as *Latency*). This represents the average number of cycles between receiving a ray into the input queue and writing the result into the output queue. The latency ranges from 2 million to 12 million cycles (1-6 ms at 2 GHz) for both tail-recursive and continuation shaders. This effectively prohibits any kind of active or passive waiting on the shading side. By active waiting we mean an actual spin loop that checks ray status. By passive waiting we mean not scheduling the thread, akin to when threads are waiting for global memory access on NVIDIA GPUs. We would therefore keep a work queue of rays to be processed and whenever tracing a ray is required, we would store the whole shader state, submit the ray query, and fetch a different ray from the work queue. Considering that with ray bifurcation the shader state actually contains a ray stack, the whole process becomes significantly more involved than the tail-recursive shaders.

Also, looking at the ray throughput, we can see that the tail-recursive shaders lead to almost universally better results than continuation shaders, and never actually perform significantly worse. The improvement always corresponds to increase in L1 cache hit rate and bandwidth, suggesting that tail recursion gives us noticeably more coherent rays.

So far we have concluded that the overall best choice would be using tail recursion with an L2 cache, giving us 50-100 million rays per second. However, assuming that the L2 cache is occupied roughly equally by both nodes and vertices, it can hold up to 25 % of the whole scene, for each of our scenes. The cache therefore effectively lowers the demand on ray coherence. But considering larger scenes this effect would become less pronounced. We would therefore prefer to increase the coherence itself rather than mitigate the impact of incoherence.

3.3.2 Treelet implementation

Towards this goal we implemented the treelet approach introduced by Aila and Kerras [Aila and Karras 2010], as described in Section 3.2.2. We use treelets of approximately 32 kB size, so each treelet can fit into the L1 Node cache. The results are summed up in Table 3.2.

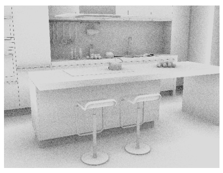
We present the same statistics as in Table 3.1 and additionally provide two statistics that are specific to the treelet mechanism. The first is the number of input queue switches. It represents the how many times was the RTE switched from working on one treelet to another. Obviously, the lower the number the better, as processing a treelet typically results in loading most of its nodes into the L1 cache. As the size of each treelet matches the size of our L1 cache, this effectively means invalidating all the previous cache records. Corresponding to that is the average queue size, measured when the queue's treelet was switched to active. It represents the number of rays that are processed between the cache invalidations. Here, the larger the number the better.

Looking at the L1 hit rate, bandwidth, and the overall performance, we can conclude that the treelets do provide overall improvement over the standard implementation. Because the rays are moved to and from the RTE on each treelet boundary crossing we can see an order of magnitude increase in ray traffic bandwidth. This is offset by the fact that due to the increased

Conference

Fairy forest

Kitchen



	Conference		Fairy forest		Kitchen	
Continuations shaders						
L2	off	on	off	on	off	on
L1 hit rate [%]	83/70	86/73	79/65	81/66	89/82	91/85
L1 bandwidth [$\mathrm{GB\,s^{-1}}$]	9.3/6.5	14.0/9.8	8.0/5.4	12.1/8.1	11.7/7.9	15.3/10.3
L2 hit rate [%]	-	79/62	-	72/57	-	76/66
L2 bandwidth [$\mathrm{GB\,s^{-1}}$]	-	2.0/2.7	-	2.3/2.8	-	1.4/1.6
Ray bandwidth [$\mathrm{GB\,s^{-1}}$]	1.3	1.3	2.2	2.2	1.4	1.4
Mem bandwidth [$\mathrm{GB\,s^{-1}}$]	5.5	7.1	5.8	9.5	4.2	4.8
Latency [cycles]	2.8 M	2.2 M	5.8 M	4.6 M	3.6 M	3.1 M
Queue switches [-]	104718	103067	196828	199421	130285	129507
Avg. queue size [-]	87	88	86	85	79	79
Throughput [MRays/s]	67.2	101.2	33.0	49.3	69.1	90.4
Tail-recursive shaders						
L2	off	on	off	on	off	on
L1 hit rate [%]	96.2/92.3	97/93	95/89	96/89	97/95	98/95
L1 bandwidth [$\mathrm{GB\,s^{-1}}$]	14.0/9.8	15.4/10.7	13.9/9.3	15.5/10.3	15.0/10.2	16.1/10.9
L2 hit rate [%]	-	73/61	-	70/64	-	70/62
L2 bandwidth [$\mathrm{GB\,s^{-1}}$]	-	0.4/0.7	-	0.6/1.1	-	0.4/0.5
Ray bandwidth [$\mathrm{GB\,s^{-1}}$]	1.2	1.2	2.0	2.0	1.3	1.3
Mem bandwidth [$\mathrm{GB\,s^{-1}}$]	2.5	2.8	3.7	4.4	2.3	2.5
Latency [cycles]	15.8 M	15.4 M	30.0 M	28.7 M	24.3 M	23.7 M
Queue switches [-]	13220	13176	19118	19252	15092	15255
Avg. queue size [-]	659	661	829	823	653	647
Throughput [MRays/s]	103.6	**112.8**	57.8	**64.4**	90.2	**96.7**

Table 3.2: *Results with treelets.* We again measure all three scenes with both tail-recursive and continuation shaders and both with and without 1MB L2 cache. The reported results are L1 cache hit rate, required L1 bandwidth in $\mathrm{GB\,s^{-1}}$, the same for L2 cache (if applicable), ray traffic bandwidth in $\mathrm{GB\,s^{-1}}$, total required memory bandwidth, both from cache and rays, latency in cycles (Latency) and throughput in million rays per second. Two treelet specific statistics are the number of queue switches and the average size of queue that has been scheduled for processing.

coherence between rays, the total bandwidth to memory (including the ray bandwidth) is actually lower than in the implementation without treelets.

We can see a drop of about 10 % in the L2 hit rate, combined with a significant drop in the L2 bandwidth. The lower L2 hit rate is caused by all rays sharing the same treelet which completely fits into L1, so the requests to L2 occur mostly when the active treelet changes. The treelets are processed with priority given by the number of queued rays, which results is a somewhat round-robin order of processing, where a just processed treelet will not be scheduled until it accumulates enough rays. This violates the common cache assumption that recently processed data will be needed again soon and results in the lower hit rate on the L2 cache. This is also manifested by much closer ray throughput between the versions with and without L2 cache.

The tail-recursive shaders clearly and consistently provide better results than continuation shaders, mainly due to the fact that they have much more rays in flight that can be sorted into treelet queues. As result, there are about 10× less queue switches with queues being on average 10× larger than when using continuations.

The only drawback is significant increase in ray latency (to 15-30 million cycles, i.e. 7.5-15 ms), which, however, is not so important when tail recursion is used.

3.3.3 Using BVH

While using B-KD-tree treelets significantly improved the L2-less performance for both **Conference** and **Kitchen** scenes, the **Fairy forest** scene showed unsatisfactory results. The treelets did indeed balance the performance between versions with and without L2 cache, but the absolute performance was still only slightly above half of what we could achieve in the other two scenes.

We suspected that the B-KD-tree might be a poor fit for the scene and modified our RTE to handle BVHs instead. We modified only the simulation engine itself, without any considerations for the changes in area or frequency.

We implemented two different approaches to the BVH. The first approach we call *Node BVH*, where each node contains its own bounding box and only indices to the children. The traversal then checks whether ray hits a node, and if so always proceeds to both children, determining the first one based on the node split plane and ray direction. The second approach we call *Child BVH*, where each node contains bounding boxes of both its children and we only descend to the children the ray actually intersects.

While there is no principal difference between the approaches, two aspects have to be considered. First, the *Child BVH* needs to perform 12 ray-plane intersections, while the *Node BVH* needs to perform only 6. This essentially means a factor of 2× in terms of area requirements for ray traversal unit. The other aspect to consider is treelet implementation. Should we choose to use *Node BVH*, a ray can descend to a child that resides in another treelet only to discover it does not intersect the child, thus generating two unnecessary treelet transitions.

The Table 3.3 shows that the *Node BVH* drawback of unnecessary treelet transitions outweighs any performance gain by using the BVH acceleration structure. The *Child BVH* approach offers approximately 12 % speed up, both with and without L2 cache, but further tests showed that a similar speed up is achieved in the other two scenes as well.

Given the fact that going from the very lightweight B-KD-tree nodes

L2	off	on
B-KD-tree [MRays/s]	57.7	64.4
Node BVH [MRays/s]	50.0	54.9
Child BVH [MRays/s]	65.0	73.5

Table 3.3: *Acceleration structures.* We show performance in million rays per second on the Fairy forest scene, using tail-recursive shaders, treelets and three different acceleration structures.

to BVH nodes required for *Child BVH* approach would introduce significant changes in the whole design, we did not pursue this design any further.

We conclude that the lower performance in **Fairy forest** is not due to the acceleration structure itself, but rather due to the complex traversal paths of the rays we generated. This is also supported by the highest number of queue switches as well as the longest average ray latency among the three scenes.

3.4 Conclusion

In this chapter we introduced a hardware implementation of a ray traversal engine (RTE) that could act as a fragment generation unit on the GPU in lieu of current rasterization engines. Using a 90 nm process, the RTE has been confirmed to run at frequencies above 2 GHz, can fit into area less than 15 mm^2, and achieves performance of over 100 million rays per second while keeping bandwidth to main memory below 5 GB s^{-1}. The unit was tested in two hardware variants, with and without 1 MB L2 cache. The L2 showed great benefit in mitigating impact of ray incoherence, even when combined with treelet-based B-KD tree which, in itself, reduces the incoherence.

We also tested two BVH based alternatives to the B-KD tree acceleration structure. The *Node BVH*, where each node knows only its own bounding box, proved to be slower due to frequent transitions into treelets that were actually completely missed by the ray. This problem is not present in *Child BVH*, where each node knows its children bounding boxes, but the significantly larger required footprint (12 plane tests instead of 2) outweighs the modest 12% speed advantage over B-KD tree. Two competing shader styles are also compared, tail-recursive shaders and continuations shaders. Despite the tail-recursive shaders having less control over the number of rays spawned and their tendency to swarm the RTE with trace requests, we found them to be almost universally better. Even with 16 rays per pixel the memory consumption and bandwidth are very reasonable, while at the same time it provides a large pool of rays that nicely complements the treelet approach.

In conclusion, we propose using tail-recursive shading wherever possible, as it does fairly well match the feed forward scheme used in the current GPUs. But even for algorithms that would be hard pressed to use tail-recursion (e.g., Bidirectional Path Tracing or Metropolis Light Transport), a dedicated ray traversal engine using B-KD tree with treelets would be an interesting, useful, and reasonably small addition to the current GPUs.

Chapter 4

Light Transport Simulation on the GPU

In the previous chapter we have discussed a hardware approach to lowering the cost of the ray tracing operator, one of the most expensive parts of the rendering equation. And while it is an efficient way of reducing the total cost of a light transport path, the most effective way would be to not even require the computation of this path. Which is precisely the goal of the various existing global illumination algorithms: Reducing the overall number of paths required to deliver an image at a given quality by making sure that each path brings as much relevant information as possible. In the context of Monte Carlo integration, we view this as the difference between reducing a cost of a Monte Carlo sample and reducing the number of samples required to obtain the result with a given precision. Therefore, in this chapter, we abstract from the ray tracing operator problem by using a well tuned software ray tracing library [Karras et al. 2012] and instead focus on the global illumination algorithms and their mapping to wide-SIMD hardware.

Global illumination research has recently focused on progressive global illumination algorithms: Algorithms that converge to the correct solution of the rendering equation (under some assumptions) given enough time and provide meaningful partial results during convergence. This category consists of well-known approaches, such as Path Tracing and Bidirectional Path Tracing, as well as more recent ones: Progressive Photon Mapping (including a bidirectional version) and Vertex Connection and Merging.

At the same time, Graphics Processing Units (GPUs) became more flexible and many global illumination algorithms were ported to them. These ports have mostly focused on proof-of-concept implementations that show a performance improvement over CPU rendering.

We take it as established that progressive global illumination algorithms are useful, that they can be ported to the GPU, and that they achieve significant speedups. But are these implementations optimal? And which progressive algorithms perform the best on GPUs? These questions are far from answered: There has been a complete lack of such a rigorous and systematic study of the implementation and performance of different progressive global illumination algorithms on GPUs.

Our work fills this gap with a first comprehensive and in-depth investigation of the problem. We have surveyed and reimplemented the best published techniques for GPU-based Path Tracing, Bidirectional Path Tracing, and Progressive Photon Mapping, so we can analyze the impact of both high-level and low-level optimizations on their performance. We also take advantage of the lessons learned from these investigations to develop the first GPU implementation of the recent Vertex Connection and Merging algorithm [Georgiev et al. 2012] (the algorithm has been independently developed by Hachisuka et al. [2012] as Unified Path Space, we use the former name), showing that even relatively complex light transport algorithms can be efficiently mapped to the GPU. In addition, we present new techniques that outperform the existing ones in most cases. For example, our Light Vertex Cache, a new approach to mapping connections of sub-path vertices in Bidirectional Path Tracing to the GPU, outperforms the state-of-the-art implementations by 30-60%. With the implementation of the aforementioned algorithms in a single system, we provide a detailed comparison of their performance on scenes with various characteristics and provide a comprehensive in-depth analysis of the findings.

Our work can have broad benefits in both research and industry, ranging from low-level implementation insights for the practitioner to high-level implications for driving research towards finding algorithms and implementation techniques that improve upon the approaches used in our results.

4.1 Related Work

Our work focuses on progressive Monte Carlo methods that converge to the solution of the Rendering Equation [Kajiya 1986]. An overview of these methods and the challenges associated with their GPU implementation is given in Section 4.2. In the rest of this section, we discuss related work on other GPU-based global illumination approaches, as well as on ray shooting.

Realtime Global Illumination. A large volume of work has focused on realtime global illumination on the GPU using rasterzation. To achieve this performance, the methods generally offer only an approximate light transport solution. Because our focus is on algorithms that converge to the exact solution, we refer to [Ritschel et al. 2012] for an overview of the realtime methods.

Many-light methods are an important class of real-time global illumination algorithms based on Instant Radiosity [Keller 1997]. These methods first distribute a number of virtual lights into the scene, approximating the global illumination, and then each pixel is illuminated by a selected subset of the lights. Please refer to Section 2.3.6 for details.

Ray Tracing. As was discussed in the previous chapter, a ray tracing query is at the heart of virtually every light transport algorithm, both in the form of finding the closest intersection along a given ray, and testing visibility between two points. The research in this area focuses on three main issues: selection of a suitable acceleration structure, algorithms for their effective construction, and algorithms for their effective traversal.

The initial research on GPU ray shooting focused on overcoming the hardware limitations of then current GPUs by using specially adapted data structures and traversal algorithms [Carr et al. 2002, Foley and Sugerman 2005, Popov et al. 2007]. More recently, Aila and Laine [2009, 2012] showed how to approach the theoretical peak ray casting performance using the Spatial Bounding Volume Hierarchies (SBVH) [Stich et al. 2009], which became the structure of choice for GPU. A large body of work focuses on the fast construction of acceleration structures directly on the GPU to enable interactive rendering of dynamic geometry. Most of this research focuses on BVHs [Lauterbach et al. 2009, Pantaleoni and Luebke 2010, Hou et al. 2011, Karras and Aila 2013], uniform and multi-level grids [Kalojanov and Slusallek 2009, Kalojanov et al. 2011] and kd-trees [Zhou et al. 2008]. We refer the reader to Section 2.4.1 for an overview of acceleration structures.

Well-established high-performance ray shooting solutions are publicly available. These libraries include Intel's Embree [Woop et al. 2013] (aimed at more CPU SIMD architectures) and for the GPU NVIDIA's OptiX [Parker et al. 2010] and the software framework of Karras et al. [2012], which is the basis of our implementation.

4.2 Overview

Tracing full light transport paths is the most important building block for all the investigated light transport algorithms. In Section 4.3, dedicated to *Path Tracing* [Kajiya 1986], we focus on finding the best implementation of this building block. In 2002, Purcell et al. [2002] mapped Path Tracing to the then current GPUs, dealing mostly with their limited programmability. However, with the modern GPUs being fully programmable, the focus of the more recent work has shifted to fully utilizing their compute capabilities. The main challenge of efficient Path Tracing implementation is the reduction of thread divergence within warps, caused by paths of different lengths. A few threads are still processing the long paths, while other threads are idle already. Novák et al. [2010] propose to use path regeneration, where persistent threads whose path has already terminated are assigned a new path from a larger pool of paths. Van Antwerpen [2011b] improves on this approach by compacting the paths so all regenerated paths are processed in a contiguous block, further increasing thread coherence. We examine their approaches, determine their best kernel configurations for the current GPU architectures, and test them against simple, yet previously unpublished, single-kernel implementations.

Section 4.4 focuses on the main challenge in implementing *Bidirectional Path Tracing* [Lafortune and Willems 1993, Veach and Guibas 1994, 1995], that is, efficient evaluation of camera and light sub-path vertex connections. The key GPU challenge is storing and connecting sub-paths of varying lengths. Novák et al. [2010] address this issue by modifying the basic algorithm to limit the light sub-paths to a maximum length of 5. Van Antwerpen [2011c] also modifies the algorithm and avoids storing sub-paths by retracing a full light sub-path for each segment of the camera sub-path, so only one segment has to be stored at a given time. His other approach [van Antwerpen 2011b] uses a user-defined maximum path length to conservatively pre-allocate memory for the sub-paths. In this section, we evaluate these modifications, as well as propose a new algorithm, based on the Light Vertex Cache, which allows for a simple implementation and outperforms the current state-of-the-art by 30-60%.

In Section 4.5, we investigate methods based on *Photon Mapping*. Spatial data structures used to accelerate photon map queries are the main challenge and the focus of our investigation. To avoid the prohibitive cost of transfering all the photons to the CPU, it is necessary to build the acceleration structure directly on the GPU. Zhou et al. [2008] show that it is possible to build kd-trees at interactive rates. However, this approach exhibits random memory access patterns that are suboptimal for the GPU. Alternatively, grids can be used both for k-NN and range queries [Purcell et al. 2003, Hachisuka et al. 2008]. Hachisuka and Jensen [Hachisuka and Jensen 2010] also propose the Stochastic Hash Grid, which avoids an explicit construction phase at the cost of storing only a subset of the generated photons. We examine and evaluate these acceleration structures and provide some optimizations to further accelerate the queries.

Using the experience gathered above allows us to present in Section 4.6 the first GPU implementation of the recently introduced *Vertex Connection and Merging* algorithm [Georgiev et al. 2012]. It combines both Bidirectional Path Tracing and Progressive Photon Mapping in a common framework and allows for efficient handling of a wide range of lighting effects.

Having implemented many of the state-of-the-art light transport simulation algorithms on a GPU, we have a unique opportunity to compare them with each other. In Section 4.7 we compare the convergence graphs of the best implementation of each of the introduced algorithms on a set of six diverse scenes, explain the behavior of each algorithm with respect to the scene characteristics, and draw conclusions as to which algorithm is best suited for which type of scene.

Summary of Challenges Associated with GPUs. There are three main challenges for efficiently implementing light transport algorithms on the GPU.

- First is code divergence within a warp, which can greatly reduce the GPU utilization due to the SIMD nature of execution units.
- Second, the memory management possibilities from within the GPU code are limited at best and GPU algorithms have to have their memory requirements known prior to kernel execution. Efficient GPU utilization requires a large number (several thousand) threads executed in parallel, which prevents generous allocation of per-thread memory as a solution to the memory management challenge.
- Third, to utilize the full memory bandwidth of the GPU, the accesses from a single SIMD execution unit should be coalesced, that is, access neighboring addresses.

This has wide implications across all algorithms. E.g., Path Tracing performance benefits from work compaction, which results in coherent primary rays being executed together in a compact block of warps.

4.2.1 Terminology

In Sections 2.5.1 and 2.5.2 we have introduced basic terminology of SIMD and GPU programming. As this chapter discusses algorithm efficiencies that depend on hardware specifics, let us briefly introduce some concepts we will be referring to. We will introduce these concepts for NVIDIA's CUDA platform, however many are universal for any architecture using a wide SIMD model (e.g., Xeon Phi or AMD Radeon).

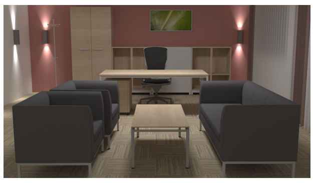
The basic execution unit of CUDA is called a *thread*, which executes scalar code called a *kernel*. This is a major deviation from the standard terminology as these threads are more similar to SIMD lanes, rather than CPU threads which are more similar to CUDA *warps*.

Each thread is allocated a certain number of *registers* and a certain amount of *local memory*. Local memory is used when the executed code requires more registers than are available, and is allocated only for the currently executed threads. Threads are grouped into *warps* of 32 threads (typically implemented as 4× 8-way SIMD). All threads in each warp execute the same instruction in each clock cycle. When threads in a warp need to execute different branches of code, all threads have to execute all the code (see Section 2.5.1) and masks are used to make sure the results are used only for threads that actually should have executed the code. This is called code divergence, and has negative impact on efficiency proportional to the size of the code in the different branches. *Global memory* is memory that can be seen by both the CPU and GPU. It contains all inputs and outputs of a

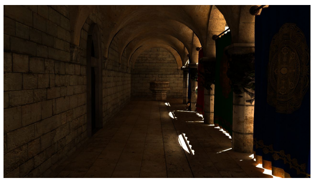
(a) CoronaRoom (680k triangles): A living room. The illumination comes mostly from sun and sky, the lamp on the left provides little illumination besides the spots on the wall. The windows do not contain glass. Scene courtesy of Ludvík Koutný (http://raw.bluefile.cz/).

(b) CoronaWatch (918k triangles): The watch has a bezel made of highly glossy metal, glass with specular reflections and refractions, and a black dial with diffuse numbers. Illumination is provided by several large area lights. Scene courtesy of Jerome White.

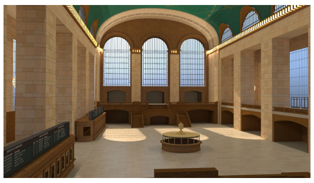
(c) LivingRoom (783k triangles): A room seen in a mirror. The objects on the table are illuminated by two small lamps next to the mirror, more lights are at the other side of the room. The major feature of the scene is the caustics on the table, reflected in the mirror. Scene courtesy of Iliyan Georgiev.



(d) BiolitFull (166k triangles): An office scene illuminated solely by area lights enclosed in diffuse tube-like fixtures. Only the spots directly beneath and above the fixtures are directly illuminated, and they act as secondary light sources. Scene courtesy of Jiří "Biolit" Friml (http://biolit.wordpress.com/).



(e) CrytekSponza (262k triangles): A modified version of the classic Sponza. The camera is in one of the arcades on the ground floor, the only illumination is coming around the drapes from a strongly illuminated atrium. Scene courtesy of CryTEK (http://www.crytek.com/cryengine/cryengine3/downloads).



(f) GrandCentral (1527k triangles): A large open hall illuminated by an environment map and over 900 point lights. Each of the 200 alcoves near the ceiling contains one point light. The remaining point lights are on the chandeliers in the side halls. Scene courtesy of Cornell University Program of Computer Graphics (http://www.graphics.cornell.edu/).

Figure 4.1: Our test scenes.

kernel call, including all intermediate data between consecutive kernel calls. The GPU has an *atomic counter* primitive, which we use for dealing with variable-sized inputs and outputs (e.g. queues) and for compaction. In our implementation we have not observed any contention problems due to many threads simultaneously incrementing the same counter.

All of the presented algorithms are progressive in nature. For clarity, we distinguish between a *frame*, the result of a single progressive iteration, and an *image*, the final result obtained by averaging multiple frames. Raw low-level performance is compared on a single frame as a basic workload unit, while higher level comparisons between different Monte Carlo estimators or completely different algorithms measure error between images and a reference solution.

4.2.2 Testing Setup

We have implemented our algorithms on top of a publicly available implementation of GPU ray casting [Karras et al. 2012]. We use their acceleration structure as well as ray casting core and claim no contributions in these areas. We support environment illumination, point lights, directional lights, and area lights; our BSDFs include reflection and refraction, diffuse textures, glossy lobes from Kelemen, Ward, Ashikhmin-Shirley, and Phong BRDF models, and Fresnel-weighted combinations of diffuse and glossy components. We use the Tiny Encryption Algorithm to generate random numbers [Zafar et al. 2010]. While not used in this survey, our design also supports using low-discepancy sequences.

For our tests, we chose six scenes (Figure 4.1), representing various configurations found in practical applications. All are rendered in 720p resolution (1280x720), i.e., roughly 1 megapixel.

All our tests have been performed on a computer with Intel i7-3770K @ 3.50GHz and 16GB RAM. We tested on two different NVIDIA GPUs of two architecture generations: Gainward Phantom GeForce GTX 580 3GB (Fermi architecture [NVIDIA 2011]), and Gainward GTX 680 4GB (Kepler architecture [NVIDIA 2012b]). We note that while the GTX 680 is a newer card and has higher theoretical FLOPS, the architecture is significantly different from GTX 580 and some of these differences are adversarial to our algorithms. For example, the global memory access is no longer cached by L1 cache but only in L2, and the clock rate has been decreased (from 1566 MHz to 1072 MHz for our cards).

4.3 Path Tracing

We first focus on Path Tracing [Kajiya 1986], one of the simplest and most well-understood light transport simulation algorithms. All implementations discussed represent different mappings of the same algorithm onto the GPU. As tracing paths is an essential building block for all algorithms introduced in the later sections, good understanding and optimization of Path Tracing has significant impact on their performance.

4.3.1 Algorithm Overview

Path Tracing generates path samples by simulating a random walk through a scene. A path starts with a primary ray at the camera. It is traced into the scene and on each surface hit the path is extended into a random direction. To increase efficiency and prevent infinitely long paths, Russian roulette at each path vertex randomly determines whether the path will be extended or terminated. The survival probability is commonly based on surface albedo (in the range 0-1).

In the basic algorithm a path contributes to the frame only when it eventually hits a light. However, this is inefficient and the algorithm is almost always used with next event estimation (direct illumination) [Kajiya 1986]. At each hitpoint, in addition to path extension, a random light is

```
(a) kernel path in all paths:                                   // Generate
        path.ray = setup primary ray for path

    while Any path active:
(b)     kernel path in all paths:                               // Extend path
            if path.termianted: return
            trace path.ray
            if no hit:
                accumulate background color
                path.terminated = true
            else:
                accumulate surface emission
                compute contribution of a random light
                path.directIllum = (shadowRay, contrib)
                if terminated with russian roulette:
                    path.teminated = true
                else: path.ray = sample BSDF

(c)     kernel path in all paths:                               // Shadow test
            if path.directIllum.contrib ≠ 0:
                if path.directIllum.shadowRay not blocked:
                    accumulate path.directIllum.contrib
```

Algorithm NaivePTmk: *Naive Path Tracing (multiple kernels)*: Naive GPU implementation of Path Tracing. All paths are processed in parallel, each path is assigned to one thread. Kernel (a) generates a primary ray for each path, and kernels (b) and (c) perform path extension and shadow test, respectively, until all paths have terminated.

sampled and, if visible from the hitpoint, its contribution is accumulated. With this strategy, lights of finite extent can be sampled in two ways: Direct connection or random hit. These two strategies are combined using Multiple Importance Sampling (MIS) [Veach and Guibas 1995].

4.3.2 Survey of Existing GPU Implementations

All implementations introduced in this section use multiple kernels. For clarity, we make this information part of the implementation name.

Naive Path Tracing (multiple kernels). To motivate the discussion on previous work, let us first consider the GPU implementation of *Naive Path Tracing (multiple kernels)* (Algorithm NaivePTmk). All paths used to obtain a frame (usually one path per pixel) are processed in parallel. The implementation uses one thread for each path, each path keeps its current state in global memory. In kernel (a), all paths are initialized and

```
    // All threads marked as idle
    // All paths in pathQueue
    while pathQueue not empty and any thread not idle:
(a)     kernel thread in all threads:                    // Regenerate
            if thread.state == idle:
                thread.path = next path in queue
                thread.ray = setup primary ray for thread.path
                thread.state = active

(b)     kernel thread in all threads:                    // Extend path
            if thread.state == idle: return
            trace thread.ray
            if no hit:
                accumulate background color
                thread.state = idle
            else:
                accumulate surface emission
                compute contribution of a random light
                thread.directIllum = (shadowRay, contrib)
                if terminated with russian roulette:
                    thread.state = idle
                else: thread.ray = sample BSDF

(c)     kernel thread in all threads:                    // Shadow test
            if thread.directIllum.contrib ≠ 0:
                if thread.directIllum.shadowRay not blocked:
                    accumulate thread.directIllum.contrib
```

Algorithm RegenerationPTmk: *Path Tracing with Regeneration (multiple kernels)*: This algorithm is almost identical to Algorithm NaivePTmk, but it decouples threads from paths. Kernel (a) now resides within the main while loop, and initializes new path from the path queue for any thread that is idle. This reduces the number of idle threads in each loop and increases GPU utilization. Kernels (b) and (c) are almost identical, with the difference that intermediate data are now stored with the thread rather than with the path.

their primary rays are generated. While there is any active path, all active paths are extended (kernel (b)) and their next event estimation is evaluated (kernel (c)).

It is important to note that because of SIMD, even inactive threads (i.e. terminated paths) have to be executed as long as there is at least one active thread in their warp. If Russian roulette terminates 50% of active paths after each path extension, the utilization of GPU will be 100% during the first extension, then drop to an average of 50% for the second extension, 25% for the third, and so on, which is clearly detrimental to the overall performance.

Path Tracing with Regeneration (multiple kernels). To address this issue, Novák et al. [2010] propose *Path Tracing with Regeneration (multiple kernels)* (Algorithm RegenerationPTmk). This implementation decouples threads from paths. It uses a fixed pool of threads. Each thread processes one path at a time. When a thread has no assigned path or its path has terminated, we say it is idle, otherwise it is active. All threads are idle at the start.

While there are any active threads and the path queue is not empty, all idle threads are assigned a new path from the path queue (kernel (a)), all paths are extended (kernel (b)), and their shadow ray is cast (kernel (c)). This way, all threads on the GPU are active until the queue becomes empty. Another advantage is that the path state is kept only for currently processed paths, so the required memory depends only on the number of threads, not the number of paths.

Stream Path Tracing with Regeneration (multiple kernels). Van Antwerpen [2011b] introduces *Stream Path Tracing with Regeneration (multiple kernels)* (Algorithm StreamingPTmk) to improve upon the previous approach. The inefficiency comes from code divergence in kernel (a) of the Algorithm RegenerationPTmk. When at least one thread in a warp needs to regenerate its path, all threads in the warp have to execute the kernel, even though the other threads do not need regeneration.

Van Antwerpen proposes to use stream compaction [Sengupta et al. 2007] to separate the threads into active and idle threads (Figure 4.2). This way, at most one warp can have both active and idle threads, essentially removing the code divergence. Another advantage is that the coherent primary rays of the new paths will be assigned to consecutive threads that will be executed together, and it has been shown that ray coherence (within warps) has positive effect on ray casting performance [Wald et al. 2001]. Similarly, the shadow rays for next event estimation are compacted to reduce code divergence in kernel (c).

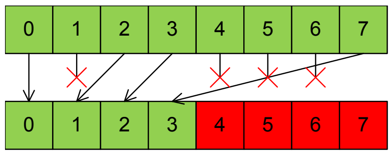


Figure 4.2: *Compaction*: Active paths are compacted (green), while terminated paths are discarded (red).

The compaction is a part of kernel (b), avoiding separate compaction kernel calls. It uses two arrays of threads, one as input and the other as output, and a global atomic counter, initially set to zero. For each active thread in the input set, the counter is increased by one and its old value is used as the thread's target position in the output set.

```
    // Two thread pools threadsIn, threadsOut
    // All threadsOut marked as idle
    // All paths in pathQueue
    // atomics:  pathCount = 0, directCount = 0
    while pathQueue not empty and any path active:
(a)     kernel thread in all threadsOut:                // Regenerate
            if thread index $\geq$ pathCount:
                thread.path = next path in queue
                thread.ray = setup primary ray for thread.path
                thread.state = active

        // Swap threadsIn $\leftrightarrow$ threadsOut
        // pathCount = 0, directCount = 0
(b)     kernel thread in all threadsIn:                 // Extend path
            if thread.state == idle: return
            trace thread.ray
            if no hit:
                accumulate background color
                thread.state = idle
            else:
                accumulate surface emission
                compute contribution of a random light
                if contrib $\neq$ 0:
                    index = directCount++
                    threadsOut[index].directIllum =
                        (thread.pixel, shadowRay, contrib)
                if not terminated with russian roulette:
                    thread.ray = sample BSDF        // atomic
                    index = pathCount++
                    threadsOut[index] = thread

(c)     kernel thread in all threadsOut:                // Shadow test
            if thread index < directCount:
                if thread.directIllum.shadowRay not blocked:
                    accumulate thread.directIllum.contrib to
                        thread.directIllum.pixel
```

Algorithm StreamingPTmk: *Streaming Path Tracing with Regeneration*: Similar to Algorithm RegenerationPTmk, but threads do not “own” their path for its entire lifetime. Instead, paths that are still active are compacted to threads with low index. The atomic counter pathCount contains the current number of active paths. Two arrays of threads, threadsIn and threadsOut are used for compaction. In kernel (a), first “pathCount” threads in the threadsOut set contain active paths, and paths are regenerated for all the remaining threads. The two sets are swapped, the “pathCount” counter is reset, and kernel (b) processes all threads from threadsIn. Paths that are not terminated are compacted to the threadsOut set. Direct illumination with non-zero contribution is handled in the same way. Note that a thread can now handle path extension and shadow test for different pixels.

In practice, incrementing this counter is a two-stage process where all threads within a warp first determine how many threads are still active and then the warp makes a single atomic add. This way we limit the memory traffic to the atomic counter as well as preserve thread coherence where possible. This approach can be further extended to accumulate across warps within a block and do a single atomic add per block, but it is quite a bit more complicated and we did not observe any benefit in any of our tests.

Wavefront Path Tracing. Laine et al. [2013] analyze Path Tracing in the cases when BSDFs are expensive to evaluate (e.g., surface characteristics described by complex noise functions). Such situations can lead to extreme code divergence. Their solution separates BSDF evaluation (for both next event estimation and continuation sampling) into a separate kernel call, sorts paths based on their BSDF and executes the BSDF kernels in a coherent fashion. However, this technique is only effective for these expensive BSDFs. For simpler BSDFs, such as those used in our test scenes, the overhead of extra kernel calls and sorting greatly outweights any gains from increased execution coherence and they recommend executing all such BSDFs in a single kernel call.

4.3.3 Proposed Alternative Implementations

All of the presented implementations launch multiple kernels, at least one per path extension. This approach has several potential bottlenecks: kernel launch overhead, path state stores and loads, and the fact that the number of active paths has to be communicated to the CPU. We investigate confining the whole algorithm into a single kernel launch, which naturally removes all three potential bottlenecks simultaneously.

Naive Path Tracing (single kernel). We propose *Naive Path Tracing (single kernel)* (Algorithm NaivePTsk) as a simpler alternative to NaivePTmk. All three kernels and the while loop are combined into a single large kernel, giving us code that is essentially the same as a standard CPU path tracer. While, in theory, all paths are still processed in parallel, the execution specifics of CUDA impose some degree of serialization. Let us assume that the number of paths to be traced is significantly larger than the number of threads that can be processed by the GPU at once. In such a case, the GPU schedules threads up to its capacity, these threads process their paths, and when all threads within a scheduling unit (*block* in CUDA) have terminated, new threads are scheduled, resulting in path regeneration on a coarser level than individual threads.

This approach has several advantages. Path state does not have to be explicitly stored and loaded, as it is kept in thread local memory all the time (which benefits from L1 cache even on the Kepler architecture of the GTX 680 GPU [NVIDIA 2012a]). This also means that there is no need to allocate any per-path memory on the GPU. The memory footprint is

```
    kernel path in all paths:
(a)     path.ray = setup primary ray for path                  // Generate
        while path.terminated == false:
            trace path.ray
            if no hit:
                accumulate background color
                path.terminated = true
            else:
                accumulate surface emission
                compute unoccluded contribution of a random light
(c)             if contribution ≠ 0:                            // Shadow test
                    if shadowRay not blocked:
                        accumulate contribution
(b)             if terminated with russian roulette:           // Ext. path
                    path.teminated = true
                else: path.ray = sample BSDF
```

Algorithm NaivePTsk: *Naive Path Tracing (single kernel)*: This is a single-kernel version of Algorithm NaivePTmk. All path states are kept in local memory and only for threads currently executed on the GPU, reducing the required memory footprint. The code is greatly simplified and essentially identical to a standard CPU implementation.

governed solely by the number of concurrently executing threads. Only one kernel is launched, effectively removing any impact that kernel execution overhead has on the overall performance and when this kernel terminates we know that all paths have terminated.

Path Tracing with Regeneration (single kernel). To explore per-thread regeneration in the single-kernel setting, we introduce *Path Tracing with Regeneration (single kernel)* (Algorithm RegenerationPTsk). We use persistent threads where the number of threads is set to the GPU capacity for concurrent threads. Path (re)generation is again moved into the while loop and threads that do not have a path are assigned one from a path queue.

While code divergence in the regeneration step is still an issue, compaction is no longer an option. The use of two sets of threads would require a global barrier to swap the sets. However, such barriers are currently not supported, so we did not explore this option any further.

Using a single-kernel implementation has a potential drawback. Suppose that the separate kernels in a multi-kernel implementation have significantly different register requirements. The number of registers influences the number of threads a GPU can process concurrently, which in turn influences the overall performance. When such kernels are combined into a

```
    // All threads marked as idle
    // All paths in pathQueue
    kernel thread in all threads:
        while pathQueue not empty:
(a)         if thread is idle:                              // Regenerate
                thread.path = next path in queue
                thread.ray = setup primary ray for thread.path
                thread.state = active
            trace path.ray
            if no hit:
                accumulate background color
                thread.state = idle
            else:
                accumulate surface emission
                compute contribution of a random light
(c)             if contribution $\neq$ 0:                     // Shadow test
                    if shadowRay not blocked:
                        accumulate contribution
(b)             if terminated with russian roulette:        // Ext. path
                    thread.state = idle
                else: thread.ray = sample BSDF
```

Algorithm RegenerationPTsk: *Path Tracing with Regeneration (single kernel)*: This is a single-kernel version of Algorithm RegenerationPTmk, utilizing persistent threads. When a thread has no path assigned, it is given a new path from the queue, and processes the path until its termination.

single kernel, the GPU cannot use more threads for the more lightweight steps of the code and is potentially underutilized in those steps.

We acknowledge that both are straightforward implementations of Path Tracing, and none, in itself, is a major contribution. However, comparing these simpler implementations with the new ones introduced in previous work is beneficial and will lend us useful insights.

4.3.4 Results and Discussion

To configure the implementations, i.e., to set the required number of registers and the size of the thread pool used by the implementations with regeneration, we have measured all possible configurations and used the one that resulted in the highest performance. The optimal number of registers greatly varies between both the individual implementations and the GPU architectures with the difference between the best and the worst in the the

Algorithm	RegenerationPTmk						StreamingPTmk					
GPU	GTX 580			GTX 680			GTX 580			GTX 680		
No. kernels	#1	#2	#3	#1	#2	#3	#1	#2	#3	#1	#2	#3
CoronaRoom	23.3	36.4	**42.7**	20.8	**31.3**	29.5	50.0	**55.9**	52.4	48.9	**51.4**	40.6
CoronaWatch	52.3	54.8	**55.9**	**35.8**	34.5	34.0	71.9	**78.1**	68.3	**51.6**	49.5	42.6
LivingRoom	30.4	39.1	**43.9**	27.6	**34.1**	33.5	55.0	**59.6**	55.7	56.0	**57.1**	49.0
BiolitFull	27.9	41.0	**49.5**	24.5	**35.0**	33.5	60.5	**68.9**	66.0	61.0	**65.0**	53.0
CrytekSponza	36.8	58.8	**73.4**	31.9	**48.3**	44.6	95.1	**95.4**	88.2	74.5	**74.8**	52.8
GrandCentral	20.1	33.0	**37.9**	19.2	**29.3**	28.1	43.4	**49.1**	46.4	44.0	**47.3**	38.6
Units	millions of rays per second (more is better)											

Table 4.1: Performance, in millions of rays per second, of RegenerationPTmk, i.e. Path Tracing with Regeneration (multiple kernels), and StreamingPTmk, i.e. Streaming Path Tracing with Regeneration (multiple kernels), for different kernel configurations (see Section 4.3.5).

tested range (32 – 63 registers) being up to 2×. As a result, we cannot give a summary advice and only recommend always conducting performance tests for each reimplementation or new hardware.

Memory requirements. All implementations require only a few megabytes of local memory for the active thread variables that do not fit into registers. Multi-kernel implementations require additional global memory to store the input of individual kernels. This translates to less than 100 MB per frame in all methods.

Kernel configurations. For Algorithms RegenerationPTmk and StreamingPTmk, we tried several kernel configurations aside from the three-kernel variant presented in the pseudocode. The results are summed up in Table 4.1 and reference the total number of kernels in each configuration, with the performance of the default three-kernel configuration given in columns #3.

First, we tried to separate the ray casting part of kernel (b) into a separate kernel, to better utilize the dedicated ray casting kernels from [Aila et al. 2012]. However, this four-kernel variant dropped the performance to less than 40% on both GTX 580 and GTX 680 compared to the three-kernel variant. The bottleneck was in the increased loading and storing of path data between kernel runs, suggesting that further increases in the number of kernels would not bring any benefits.

Going the opposite way, we reduced the number of kernels to two (Table 4.1, columns #2), by making the Regeneration kernel (a) part of the Extend path kernel (b). This saves one set of loads and stores, at the cost of lower GPU utilization in the regeneration step. Importantly, the effectivity of compaction is preserved.

We also tried reducing to just a single kernel (Table 4.1, columns #1) by folding in the Shadow test kernel (c). Here we save another load and store, this time at the cost of losing the benefits of compaction. Note, that this is still different from RegenerationPTsk, which runs only one kernel for each frame, while RegenerationPTmk with one kernel runs this kernel for each path vertex.

Table 4.1 shows the performance, in millions of rays per second, of RegenerationPTmk and StreamingPTmk with one, two, and three kernels on both GTX 580 and GTX 680. Starting with StreamingPTmk, it is quite clear that the two-kernel configuration is the best, having superior performance in almost all scenes. The three-kernel configuration on GTX 680 performs significantly worse than the one- and two-kernel configurations, mainly due to changes in memory system and caching of global memory accesses.

The situation with RegenerationPTmk is less clear. On GTX 580, the three-kernel variant clearly outperforms the other two configurations. However, this changes for GTX 680, where the lower number of loads and stores into global memory results in a slight advantage of the two-kernel configuration. One of the reasons for the difference between two and three kernels in StreamingPTmk and RegenerationPTmk is that StreamingPTmk performs non-coalesced accesses in the compaction phase, making it more sensitive to the missing L1 cache. For the following performance analysis, we use the optimal RegenerationPTmk and StreamingPTmk kernel configuration for their respective GPU (that is, we use three-kernel RegenerationPTmk for GTX 580, two-kernel RegenerationPTmk for GTX 680, and we use the two-kernel StreamingPTmk for both).

Performance tests. For each of these implementations we measured performance (in rays per second) for different numbers of paths per frame. We tested from 330 thousand to 100 million paths per frame, which covers a wide range of desired applications, from 1 path per pixel at resolution of 640×480 to 100 paths per pixel at resolution of 1280×720. The individual per-scene measurements, to be found in the supplemental material in Appendix A, have been aggregated in Figure 4.3.

Let us first look at Figure 4.3a (GTX 580). Here, StreamingPTmk, is the clear winner across all path counts. Its base performance at 10^6 paths per frame is increased by another 15% for 10^7 paths, making it almost 50% faster than RegenerationPTsk and RegenerationPTmk, which compete for being the second fastest. The comparison of RegenerationPTsk and RegenerationPTmk, respectively, shows that our RegenerationPTsk is less sensitive to the size of workload, keeping stable performance from approximately 10^6 paths per frame up. RegenerationPTmk can still outperform the single-kernel implementation, but requires 10^7 paths or more, and even then the difference is marginal. Both naive implementations, NaivePTmk and NaivePTsk, exhibit low performance. As NaivePTmk requires allocated memory for each path of the frame, it could not be tested for the full range. We note that for $2 \cdot 10^6$ paths per frame NaivePTsk is faster than RegenerationPTmk, witnessing of the overhead of global memory stores and loads.

For GTX 680 (Figure 4.3b) the story changes. Both StreamingPTmk

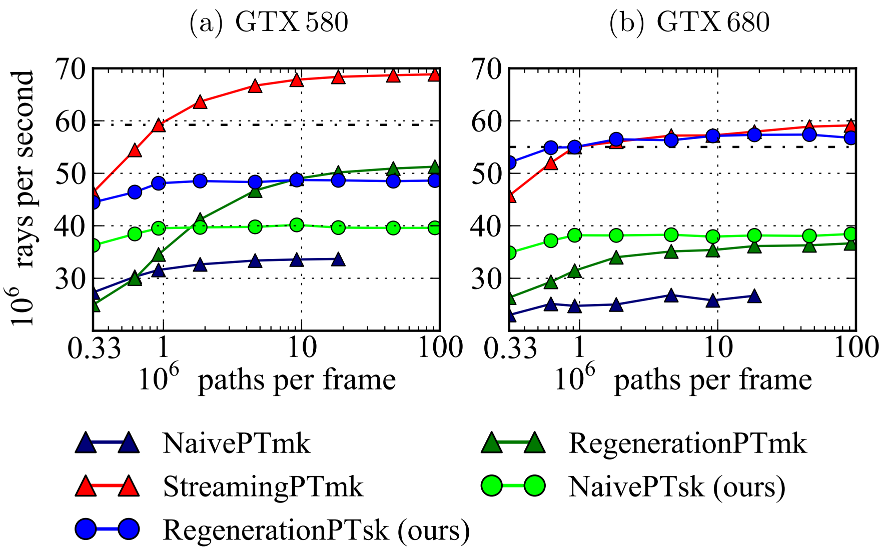


Figure 4.3: *Path tracing performance.* Performance in rays per second with increasing number of paths per frame, averaged across all six test scenes.

and our RegenerationPTsk perform roughly the same, with RegenerationPTsk having more stable and StreamingPTmk slightly higher peak performance. This is caused by the difference in the memory subsystem on the Kepler architecture of the GTX 680 GPU. While StreamingPTmk benefits from compaction and slightly better GPU utilization, our RegenerationPTsk has the benefit of storing intermediate data in the L1-cached local memory instead of global memory. For similar reasons NaivePTsk is faster than RegenerationPTmk in all cases.

4.3.5 Conclusion

Lower numbers of larger kernels benefit both Fermi (GTX 580) and Kepler (GTX 680) architectures. The disadvantage of more loads and stores outweighs gains from the optimal number of concurrently executed threads for a given step. The relative performance of the measured kernel configurations of RegenerationPTmk differs between the architectures, with three-kernel configuration being the fastest on GTX 580, and two-kernel (joined Regeneration and Extend path kernels) on GTX 680. For StreamingPTmk, the two-kernel configuration is the fastest on both architectures.

With regard to the performance on GTX 680, our Path Tracing with Regeneration (single kernel) (RegenerationPTsk) and Streaming Path Tracing with Regeneration (multiple kernels) (StreamingPTmk) have similar performance, but our implementation is faster for low number of paths per frame, as well as simpler to implement. This changes on the older GTX 580, where StreamingPTmk is the optimal implementation for all but the lowest number of paths per frame with its peak performance being over 50% faster than our RegenerationPTsk, which competes with its multi-kernel variant for the second place.

4.4 Bidirectional Path Tracing

While Path Tracing is sufficient for simpler open scenes, scenes with more complex indirect illumination (e.g., BiolitFull) greatly benefit from more advanced Bidirectional Path Tracing (BPT) [Lafortune and Willems 1993, Veach and Guibas 1994]. The algorithm itself is more complicated than Path Tracing and as such opens opportunity for a different type of optimization. In Path Tracing we focused only on the very low-level mapping of a single algorithm onto the GPU. Here, on the other hand, we examine options of modifying the underlying Monte Carlo estimator (and thus the algorithm itself) in order to better adapt to a GPU implementation.

4.4.1 Algorithm Overview

Bidirectional Path Tracing, as originally described by Lafortune and Willems [1993] and Veach and Guibas [1994], generates, for each image sample, two separate sub-paths: one starting at the camera and one at a light (Figure 4.4). The first vertex of each sub-path is located directly on the camera or on a light, respectively. The sub-paths are extended, by adding one vertex at a time, in the same way as in Path Tracing. After the two sub-paths have been generated, each vertex of the camera sub-path is connected to each vertex of the light sub-path, forming full paths (connecting camera to light). We can view this as a generalization of Path Tracing with next event estimation, in which the light sub-path had always just a single vertex directly on the light source. As there are multiple ways to construct each full path from light to camera, these options of generating paths are combined using Multiple Importance Sampling (MIS) [Veach and Guibas 1995].

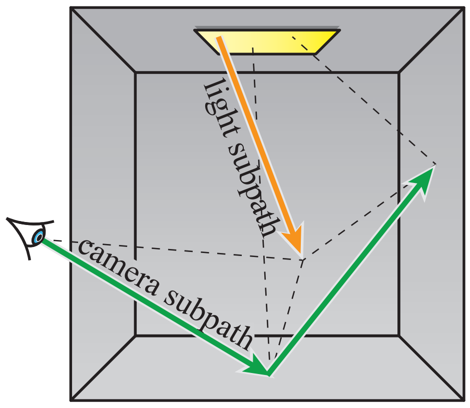


Figure 4.4: The standard Bidirectional Path Tracing sample consists of a camera sub-path (green) and a light sub-path (orange), where each vertex on the camera sub-path is connected to each vertex on the light sub-path (dashed).

The original formulation of MIS by Veach and Guibas [1995] assumes that when two vertices are connected, all vertices on both sub-paths preceding the connected vertices have to be accessed to gather the required data to compute the appropriate MIS weight. Recursive Multiple Importance Sampling (MIS), introduced in [van Antwerpen 2011c, Georgiev 2012], removes this requirement and allows computing the MIS weight from information stored only in the vertices being connected. This is especially important for GPU implementation, where random memory accesses should be limited. All implementations presented here use this method.

Connecting each camera sub-path vertex to all the vertices on the light sub-path introduces two new GPU implementation issues that have to be addressed. First, where PT has a fixed memory footprint per path, the memory requirements in BPT depend on the length of the light sub-path, as the whole light sub-path has to be traced and stored before the camera sub-path can be started. While the average length of a path is not high, this storage has to be multiplied by the number of parallel threads. Second, unlike PT, the work required per camera sub-path vertex depends on the light sub-path length and can be vastly different for different camera sub-paths, which complicats an efficient mapping to GPU.

4.4.2 Survey of Existing GPU Implementations

Bidirectional Path Tracing with Regeneration (RegenerationBPT). The first fully GPU-based implementation was introduced in [Novák et al. 2010]. It uses two separate passes. In the first pass, all light sub-paths are generated and stored in the GPU memory. In the second pass, camera sub-paths are created and traced as in Path Tracing with Regeneration, except that each vertex is also connected to all vertices of a randomly chosen light sub-path. To address the memory issue, the authors limit the length of light sub-path to five vertices. Limiting the maximum light sub-path length also requires a more complex logic for computing MIS weights and the authors therefore did not use MIS, which has a negative impact on the final image quality.

Multi-path Bidirectional Path Tracing (MultiBPT). Both the memory consumption and the workload issues are solved by the algorithm introduced by van Antwerpen [2011c], originally under the name *Streaming Bidirectional Path Tracing*, which we changed to avoid confusion with a later introduced algorithm of the same name by the same author.

Instead of storing the whole light sub-path, the algorithm traces one complete camera sub-path for each light sub-path vertex, which requires storing only one light and one camera sub-path vertex. This naturally solves both the storage and the uneven load problem, at the cost of more camera paths; the algorithm essentially spends more time on “camera-side” effects (e.g., anti-aliasing) than on “light-side” effects (e.g., caustics). As tracing of

```
while pathQueue not empty and any path active:
    foreach thread in all threads:
        if thread is idle:
            thread.camera = setup camera path
            thread.light = setup light path
    CompletedPaths = 0
    while CompletedPaths < 60%:
        foreach thread in all threads:
            if thread.light not terminated:
                Extend thread.light
                thread.lightVertices += light vertex
            if thread.camera not terminated:
                Extend thread.camera
                thread.cameraVertices += camera vertex
            if thread.light and thread.camera terminated:
                CompletedPaths++
    foreach thread in all terminated threads:
        Generate all lightVertices and cameraVertices pairs
        foreach vertex pair in thread:
            shadowRay = pair.light to pair.camera
            if shadowRay not occluded:
                accumulate contribution
```

Algorithm StreamingBPT: *Streaming Bidirectional Path Tracing with Regeneration*: All threads are initialized with a camera and light sub-path. Then a two stage algorithm is executed, where all sub-paths are extended in a similar way to Algorithm StreamingPTmk (details left out for brevity). When more than 60% threads have both sub-paths terminated, all pairs of vertices for each thread are generated (implicitly), and all such pairs have their visibility evaluated and contributions accumulated. All terminated threads are then regenerated, until there are no paths left in the queue.

both sub-paths is interleaved, an efficient implementation requires reusing the same code for both camera and light sub-paths, making the implementation quite involved.

Combinatorial Bidirectional Path-Tracing. Pajot et al. [2011] present a hybrid two-stage implementation of BPT. All sub-paths are generated on the CPU, and the GPU performs only connections between all camera and all light sub-path vertices. Unfortunately, this way some paths (e.g., caustic paths) can only be handled by the CPU. As this is not a pure

GPU implementation, we are including it only for completeness and it does not appear in our comparison.

Streaming Bidirectional Path Tracing with Regeneration (StreamingBPT). In contrast to his [2011c] algorithm, van Antwerpen [2011b] presents a more traditional BPT (Algorithm StreamingBPT). The approach is a two stage algorithm, using a pool of threads, with one thread processing one camera and light sub-path pair. Initially, all threads have their sub-paths generated. Both sub-paths are then extended, with their vertices stored with the thread. When more than 60% of the threads have both their sub-paths terminated, the algorithm enters the second stage, in which all pairs of light and camera sub-path vertices within each thread are tested for visibility and their contributions are accumulated to the image. The pairs are formed implicitly and tested with one thread for each pair, which solves the issue with uneven work per vertex. After that, all terminated threads have their sub-paths regenerated and the whole algorithm is repeated until there are no more paths to be traced. The required storage size is determined by the size of the thread pool and the user-defined maximum path length.

4.4.3 Proposed Alternative: Light Vertex Cache BPT

To remove the requirement for user-defined maximum path length while keeping the implementation as simple as possible, we introduce the Light Vertex Cache Bidirectional Path Tracing algorithm (Algorithm LVC-BPT). The key idea is that instead of connecting each camera sub-path vertex to all vertices from a given light sub-path, the vertex is connected to a given number of uniformly and randomly chosen vertices across all light sub-paths. It can also be seen as first choosing a random light sub-path (similar to RegenerationBPT) with probability proportional to its number of vertices and then choosing a uniformly random vertex on the path, which arrives at the same uniform probability for all light sub-path vertices.

This, along with the recursive MIS weight computation, enables us to store all vertices in a single global *Light Vertex Cache (LVC)*, without storing any information regarding the light sub-path they originate from. As all vertices are stored in a common cache, we do not need to know the maximum path length. Instead, we only need the average path length, to allocate a large enough cache. We estimate this by tracing a small number (ten thousand) of light sub-paths, only counting the number of vertices they would store. This kernel takes less than 1 ms on both tested GPUs and has to be performed only once for each scene. Using the average path length, we compute the expected number of light sub-path vertices (adding a 10% safety margin) and reserve the required memory for the cache. In theory, it is possible that the algorithm will generate more light vertices than the LVC

```
vertexCount = 0                                   // Preparation phase
foreach path in 10k light paths:
    while path not terminated:
        trace path.ray if no hit: return
        vertexCount += 1
        path = extend path
averageLength = vertexCount/10 k
LVCache = reserve |light paths| · averageLength · 1.1
connections = max(1, ⌈averageLength⌉)
vertexCount = 0                                   // Light trace
foreach path in light paths:
    while path not terminated:
        trace path.ray
        if no hit: return
        LVCache[vertexCount++] = path.vertex
        path = extend path
foreach path in camera paths:                     // Camera trace
    while path not terminated:
        trace path.ray
        if no hit: return
        repeat connections times:
            path connects to LVCache[random]
        path = extend path
```

Algorithm LVC-BPT: *Light Vertex Cache BPT* (proposed algorithm): In LVC-BPT we first, once for each scene, estimate average light path length (Preparation phase), reserve room in a light vertex cache for the estimated total number of light sub-path vertices including a 10% safety margin and estimate the number of connections for each camera sub-path vertex. We then execute two main stages of the algorithm. First, we trace all light sub-paths, storing the light sub-path vertices in the cache. Second we trace all camera sub-paths, connecting the camera vertices to the required number of random light vertices in the cache.

capacity, in which case we would discard the extra vertices (causing bias). However, this has not happened in any of our experiments.

The implementation of LVC-BPT is fairly straightforward and can be based on any of the algorithms introduced in Section 4.3. We present results based on Path Tracing with Regeneration (single kernel) (as LVC-BPTsk) and on Streaming Path Tracing with Regeneration (multiple kernels) (as LVC-BPTmk).

As the second pass of LVC-BPT accesses the cache in a random pat-

Algorithm	Vertex Storage	Path extension	Shadow test	Other	On our configuration
StreamingBPT	$2 \cdot S \cdot L \cdot$ B	$4 \cdot S \cdot 64$ B	$S \cdot 44$ B	$S \cdot 100$ B	1.1 GB
MultiBPT	$S \cdot 100$ B	$2 \cdot S \cdot 64$ B	$S \cdot 44$ B	—	108MB
NaiveBPT	$L \cdot E \cdot 100$ B	—	—	—	8–10 MB
LVC-BPTsk (ours)	$P \cdot AL \cdot 100$ B	—	—	—	16–164 MB
LVC-BPTmk (ours)	$P \cdot AL \cdot 100$ B	$2 \cdot S \cdot 64$ B	$S \cdot (V+1) \cdot 44$ B	—	148–316 MB
S – thread pool size; E – number of concurrently executed threads L – maximum path length; P – paths per frame; AL – average light sub-path length					

Table 4.2: *Summary of BPT memory requirements*: We present the memory requirement of each component of each algorithm as a function of several parameters. We also give the total amount of memory the algorithm used on our configuration.

tern, we load the vertices through the texture units in an Array of Structures (AoS) layout for optimal performance.

4.4.4 Results and discussion

Within our frame work we have implemented and tested the following five algorithms:

- StreamingBPT [van Antwerpen 2011b] represents the current state-of-the-art algorithm. To confirm that our performance is on par with the paper, we measured the number of samples (i.e., camera and light sub-path pairs) on the same scene and GPU as in the original paper and our implementation (8.66 million samples per second) was roughly twice as fast as reported in the original paper (3.64 million samples per second).

- MultiBPT [van Antwerpen 2011c]. We use a straightforward extension of the approach by complementing it with the dual algorithm of tracing one light sub-path for each camera sub-path vertex. During the progressive rendering of the image, we alternate between the two algorithms, balancing the number of camera and light sub-paths.

- NaiveBPT is a straightforward port of CPU code to GPU to compare the relative gain of the more advanced implementations. The implementation consists of two while loops of NaivePTsk (Alg. NaivePTsk) within a single kernel. Persistent threads are used for better control of memory requirements (see below).

- LVC-BPT is our new algorithm. Its two versions use either Path Tracing with Regeneration (for LVC-BPTsk) or Streaming Path Tracing with Regeneration (for LVC-BPTmk) as the basic algorithm for tracing camera and light sub-paths.

Memory requirements. Table 4.2 gives a summary of memory used by each of the algorithms as a function of several parameters. The state-of-the-art StreamingBPT uses the most memory. It uses two sets of threads and requires large storage for all light sub-path vertices using up to 1.1 GB of memory. MultiBPT stores only one light sub-path vertex per thread and does not use two sets of threads for each sub-path, lowering the total memory requirements to 108 MB. NaiveBPT performs all its computation within a single kernel launch, so it does not require any extra memory for path extension and shadow test kernels, lowering memory requirements to only 8–10 MB. LVC-BPTsk also stores only light vertices but the memory is given by the average light sub-path length and the total number of light sub-paths per frame. LVC-BPTmk again adds requirement for the path extension and the shadow test kernel stores.

Performance tests. In Bidirectional Path Tracing, the number of rays per second does not provide a good comparison between the algorithms. Instead, we measure performance as the time required to achieve a given image quality in terms of Root Mean Square Error (RMSE) with the respect to reference solution (computed by NaiveBPT in 10 hours).

We performed measurements on both GTX 580 and GTX 680 with 10^6 samples per frame and chose our target quality as the RMSE achieved by the state-of-the-art StreamingBPT in 10 minutes on a GTX 580. Table 4.3 shows the relative speedup against StreamingBPT on GTX 580. The average result is a simple average of the speedups for each given algorithm. Note, that we do not use *LivingRoom* in this comparison, as the RMSE is dominated by the missing reflected caustics that none of the BPT methods can reasonably capture (Figure 4.5).

When we look at the results on GTX 580, we notice a surprisingly high performance of NaiveBPT. On the *GrandCentral* scene it outperforms the StreamingBPT, and on *CoronaWatch* it is even tied for the fastest algorithm with our LVC-BPTsk. In these scenes, the work for each sample is highly uniform, which mitigates the inefficiencies of the naive approach. However, on average, the naive approach is about 15% slower than StreamingBPT. MultiBPT is the slowest of the algorithms, mainly due to its more complex implementation and imperfect interleaving of camera and light sub-path tracing. Both LVC implementations are faster than StreamingBPT on all scenes, with an average speedup of 33%. A major factor is that, un-

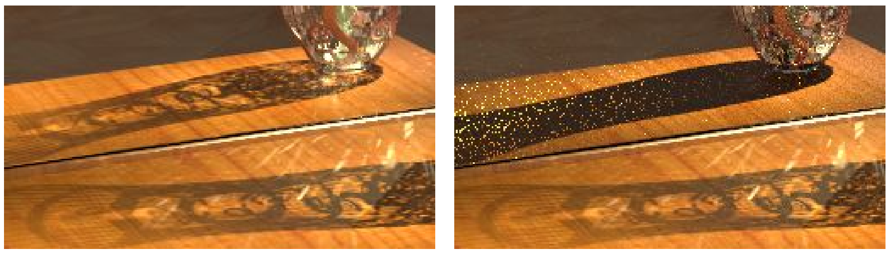


Figure 4.5: Reflected caustic dominates the RMSE of LivingRoom scene. **Left**: Inlay from reference image. **Right**: Inlay from LVC-BPTsk after 15 min.

	GeForce GTX 580				
	StreamingBPT	NaiveBPT	MultiBPT	**LVC-BPTsk**	**LVC-BPTmk**
CoronaRoom	1.00×	0.91×	0.81×	1.15×	**1.21**×
CoronaWatch	1.00×	**1.41**×	0.93×	**1.41**×	1.15×
BiolitFull	1.00×	0.52×	0.57×	1.53×	**1.81**×
CrytekSponza	1.00×	1.07×	0.93×	**1.35**×	1.29×
GrandCentral	1.00×	0.70×	0.70×	1.29×	**1.34**×
Average	1.00×	0.85×	0.77×	**1.33**×	**1.33**×
	GeForce GTX 680				
	StreamingBPT	NaiveBPT	MultiBPT	**LVC-BPTsk**	**LVC-BPTmk**
CoronaRoom	0.86×	0.72×	0.69×	**1.24**×	1.19×
CoronaWatch	0.74×	1.17×	0.80×	**1.32**×	1.07×
BiolitFull	0.89×	0.42×	0.58×	1.73×	**1.92**×
CrytekSponza	0.84×	0.92×	0.92×	**1.55**×	1.12×
GrandCentral	0.97×	0.55×	0.68×	1.39×	**1.49**×
Average	0.85×	0.68×	0.71×	**1.38**×	1.27×

Table 4.3: *Relative BPT speed up*: Speed up of different BPT algorithms, in terms of time to a given quality, relative to StreamingBPT on GTX 580. The target quality is chosen as the RMSE achieved by StreamingBPT in 10 minutes on GTX 580.

like StreamingBPT, LVC-BPT stores only light vertices, which comprise less than 40% of all vertices stored by StreamingBPT. The algorithm also benefits from a more straightforward control flow.

The results on GTX 680 are consistent with our findings from Section 4.3. We again see a drop in the absolute performance of multi-kernel implementations, significantly influenced by the lack of L1 cache for global memory accesses. Our LVC-BPTsk is the only algorithm that actually shows increase in performance on GTX 680. All other algorithms, including our LVC-BPTmk, show a decrease in performance.

Increasing the sample rate to 10^7 samples per frame, the findings are again consistent with the findings from Section 4.3, and multi-kernel implementations (StreamingBPT, MultiBPT, and LVC-BPTmk) benefit from the increased number of samples more than single-kernel variants (NaiveBPT and LVC-BPTsk). The full results can be found in the supplemental material in Appendix A.

4.4.5 Conclusions

In this section we surveyed several Bidirectional Path Tracing algorithms. NaiveBPT and StreamingBPT implement the standard BPT Monte Carlo estimator by only limiting the maximum path length, while other approaches modify the estimator to achieve better GPU mapping. Our proposed LVC-BPT significantly simplifies the implementation by decoupling light and camera sub-paths.

We compared the state-of-the-art StreamingBPT with NaiveBPT, MultiBPT, and our LVC-BPT. As LVC-BPT can utilize almost any Path Tracing implementation, we measured two versions: LVC-BPTsk, based on Path Tracing with Regeneration (single kernel), and LVC-BPTmk, based on Stream Path Tracing with Regeneration (multiple kernels). We conducted performance tests, measuring the time required to achieve a given image quality. The difference between LVC-BPTsk and LVC-BPTmk closely follows the differences between their respective Path Tracing algorithms. The single-kernel implementation is simpler and more suited for GTX 680 and low numbers of samples per frame, while the multi-kernel implementation is slightly more involved and is more suited for GTX 580 and larger numbers of samples per frame. The simplicity of the LVC-BPT implementation allows it to outperform the other algorithms by 30-60% on all tested configurations.

4.5 Photon Mapping-Based Approaches

While Path Tracing and Bidirectional Path Tracing are an excellent choice for a wide range of scenes, some effects, e.g., the reflected caustics in Figure 4.5, remain notoriously hard to capture. In this section, we focus on a family of methods based on Photon Mapping (PM) [Jensen 2001] that can handle such effects.

Photon Mapping based approaches are similar to BPT in that they require both light and camera sub-paths. However, unlike BPT, camera sub-path vertices connect to all light sub-path vertices within a certain radius, an operation that requires a suitable acceleration structure. These acceleration structures have to be rebuilt for every frame and are the focus of this section.

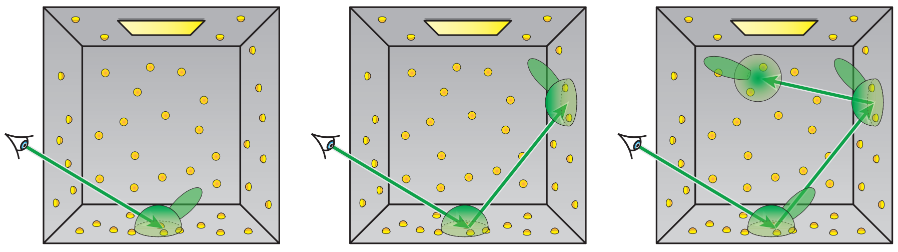

(a) Progressive PM **(b)** Stochastic PPM **(c)** PBPM

Figure 4.6: The original Progressive Photon Mapping (left) performs density estimation on the first camera sub-path vertex. Stochastic Progressive Photon Mapping (center) extends camera sub-path using glossy BSDF components and performs density estimation on both vertices, using only diffuse BSDF components on the first one. Progressive Bidirectional Photon mapping (right) traces full camera sub-path, performs density estimation on all its vertices, and weights them using Multiple Importance Sampling.

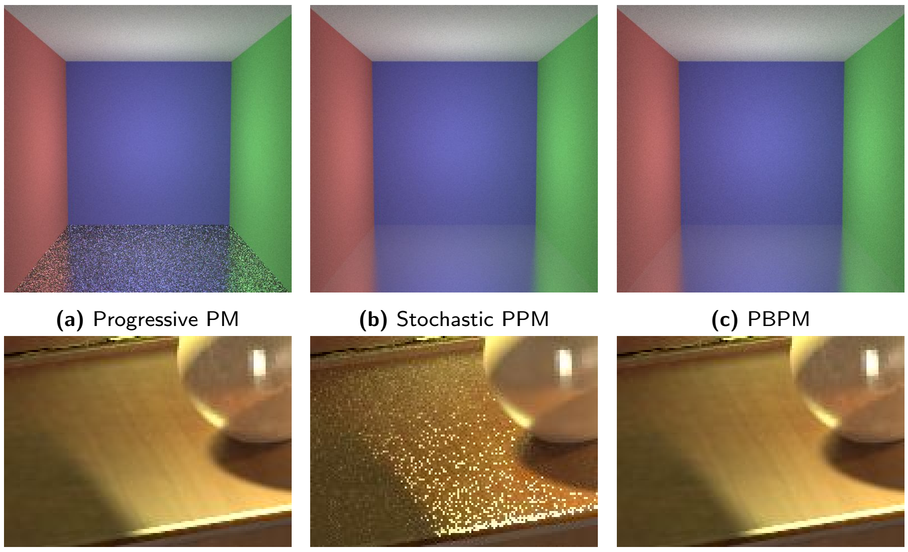
Unlike in previous sections, where the differences between the compared variants consisted in mapping to the GPU or a slight algorithmic modification, in this section we examine acceleration structures with very different asymptotic complexities for both construction and querying. To compare the structures, we implemented three different algorithms based on Progressive Photon Mapping and we compare the results achieved with different data structures in all three of them.

All Photon Mapping based approaches share the following two-pass algorithm. In the first pass, light sub-paths are traced into the scene, on each interaction with the scene a photon is stored, and the sub-paths are extended in the same way as in BPT. The photons store only their position, incoming direction, and energy; they do not require the BSDF. In the second pass, camera sub-paths are traced and density estimation is performed at their vertices. Each photon within some radius r of the hitpoint is treated as if it arrived exactly at the hitpoint, that is, incoming direction and energy of the photon is used to evaluate the BSDF at the hitpoint. Contributions from all such photons are accumulated and divided by πr^2. The choice of the radius depends on the specific algorithm, with the common choices being a fixed radius (range query) and a radius such that k nearest photons contribute (k-nearest neighbor, k-NN, query). Which vertices of the camera sub-paths perform the density estimation also depends on the specific algorithm.

Progressive Photon Mapping (PPM). Progressive Photon Mapping by Hachisuka and Jensen [2008] uses range queries to perform density estimation on the first non-specular vertex of each camera sub-path (see Figure 4.6a). Using per-vertex statistics such as number of accumulated photons, they reduce the query radius in such a way that the whole algorithm is consistent.

Stochastic Progressive Photon Mapping (SPPM). In their follow up paper Hachisuka et al. [2009] show that the per-vertex statistics can be reused for all vertices originating from the same pixel. This can be used to improve performance on glossy surfaces, as standard density estimation on glossy surfaces produces noisy results (Figure 4.7a). SPPM instead uses only the diffuse component of the BSDF on the first camera sub-path vertex, extends the sub-path using glossy components and performs another density estimation on the second vertex (see Figure 4.6b). This often leads to less noisy results (Figure 4.7b).

Progressive Bidirectional Photon Mapping (PBPM). While highly glossy surfaces greatly benefit from SPPM (Figure 4.7, top row), always extending the camera sub-path can be adversarial when the glossy lobe is wide (Figure 4.7, bottom row). In that case, PPM is actually better. To address this issue, Vorba [2011] introduces Progressive Bidirectional Photon Mapping (PBPM), where the camera sub-path is extended in the same way as in Path Tracing and density estimation is performed on each of its vertices. Multiple Importance Sampling is then used to properly weight the


(a) Progressive PM **(b)** Stochastic PPM **(c)** PBPM

(d) Progressive PM **(e)** Stochastic PPM **(f)** PBPM

Figure 4.7: *PPM, SPPM, and PBPM on glossy surfaces.* Given a Cornell Box with a highly glossy floor (top row), Progressive Photon Mapping (a) produces a noisy image, because only very few photons on the floor produce a significant contribution. Stochastic Progressive Photon Mapping (b) instead extends the camera sub-path in the direction of the glossy lobe and performs density estimation on the diffuse wall, giving a much smoother result. However, for only slightly glossy surfaces (bottom row) it is not beneficial to follow the glossy lobe and PPM (d) produces less noisy result than SPPM (e). Progressive Bidirectional Photon Mapping (c, f) uses multiple importance sampling to weight both techniques to produce a noise-free image in both cases.

individual contributions, leading to a smooth result on both high and low gloss surfaces (Figure 4.7c and 4.7f).

Knaus and Zwicker [2011] show that the per-vertex (or per-pixel) statistics are not required and the radius can be reduced using a global scaling factor. In all our implementations we use this approach rather than the original reduction scheme.

All three algorithms share common elements, many of which we have already addressed. The sole new challenge is an efficient implementation of density estimation using a range query, accelerated through the use of a spatial data structure. Since both photon generation and queries are done on the GPU, it is essential that the data structure construction is also handled by the GPU. In the next part we focus on this aspect.

4.5.1 Survey of Existing GPU Implementations of Photon Map Search Structures

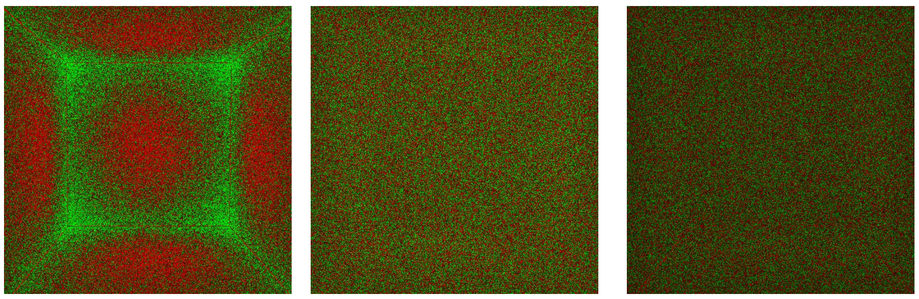
kD-tree. Zhou et al. [2008] describe an algorithm for GPU construction of kD-trees, the acceleration structure used in the original Photon Mapping

[Jensen 1996]. The algorithm first sorts photons by their coordinates and then builds the kD-tree incrementally, by levels. For each level of the kD-tree three prefix sums and three scatter/gather operations are executed. This build process is significantly more involved than the build process of Hash Grids, introduced below and our experiments show that even the range queries are slower (Table 4.6).

(Full) Hash Grid. While kD-trees excel at queries with an unknown or highly varying radius, their build as well as traversal algorithms are quite costly. The original Progressive Photon Mapping [Hachisuka et al. 2008] implementation instead uses Hash Grids. We use the name Full Hash Grid to distinguish it from Stochastic Hash Grid introduced later. Here, the whole scene is partitioned into a grid with cell sizes roughly equal to the diameter of the expected queries and the photons are stored in these cells. As representing each cell in memory is unnecessary, a 1D array of cells is used instead, typically equal in size to the number of light sub-paths. A photon's position in this array is given by a hash of its coordinates in the full grid. A good hash function should be used (we use Jenkins' hash [Jenkins 1997]). The construction of the structure is simple and easy to parallelize (see Algorithm 1). When querying the grid for photons within radius r, we iterate through all cells that are within this radius, collect all the photons, and discard those that are farther than r. When the radius is smaller than half of the cell edge length, only 8 cells have to be searched.

```
// Each cell has 1 atomic counter
// storage - array of photon indices
foreach cell:
    cell.counter = 0
foreach photon:
    cell[hash(photon)].counter += 1
Exclusive prefix sum over cell.counter
foreach photon:
    position = cell[hash(photon)].counter++
    storage[position] = photon index
```

Algorithm 1: *Building hash grid*: Each cell has a single atomic counter, that is initially set to 0. Each photon increments this atomic counter, to determine how many photons belong to each cell. Exclusive prefix sum is performed over these counters, giving a start index on which photon indices belonging to each cell should be stored. In the final pass each photon increments the counter and fetches its old value. The photon's index is stored at the position given by this value in the storage array. The range of photons in the storage array that belong to a cell with index *cidx* is given by cell[*cidx*-1].counter (inclusive) to cell[*cidx*].counter (exclusive), with cell[-1].counter = 0.



(a) Stochastic Hash Grid **(b)** Rectified Stochastic Hash Grid **(c)** Full Hash Grid

Figure 4.8: The positive (green)/Negative (red) difference of PPM using the original Stochastic Hash Grid (left) and a Path Traced reference using the Cornell Box with walls that have an albedo of 0.99. The corners of the Cornell Box are visibly lighter than they should be. Our Rectified Stochastic Hash Grid (center) matches the results given by the Full Hash Grid (right).

This approach has two drawbacks. When the radius is significantly smaller than the size of a cell, the cell can contain many photons that will be outside the query radius and are discarded, causing overhead. The second problem stems from hash collisions when multiple full cells are mapped into a single hash cell. As a result, the cell can again contain many photons that will be outside the query range.

Stochastic Hash Grid. Hachisuka and Jensen [2010] identify two GPU-specific problems with the Full Hash Grid approach and propose the Stochastic Hash Grid to address them. First, for efficient access all photons in the same cell have to be contiguous in memory, e.g., sorted by counting sort using atomic counters. The second problem stems from the uneven number of photons in each cell – for example, surfaces close to lights can have a significantly higher density of photons. This means that the number of photons processed in each query can be significantly different between different camera vertices in a warp, lowering the GPU efficiency. Instead of storing all photons, they propose to store only one photon for each cell, uniformly and randomly chosen from all photons that belong to the cell with its energy increased accordingly. In their implementation each cell has an atomic counter and whenever a photon should be stored in a cell, it is simply written there and the counter is increased. For rendering, the energy of the photon is multiplied by the value of the counter.

4.5.2 Rectified Stochastic Hash Grid

The Stochastic Hash Grid is based on the assumption that independent threads tracing the photons lead to equal probability for each photon to be the last one written to a cell. Unfortunately, this is not the case, as photons with longer paths have a higher probability of being the last. This is demonstrated by Figure 4.8a, depicting the positive (green) and negative (red) luminance difference between SPPM and a reference rendering of a Cornell Box (walls with albedo 0.99). When compared to the results of Full Hash Grid (Figure 4.8c), it is obvious that the original method is biased towards longer paths.

Our Rectified Stochastic Hash Grid (Figure 4.8b) selects photons using reservoir sampling (see Algorithm R in [Vitter 1985]), where the n^{th} photon replaces the stored photon with a probability of $p = \frac{1}{n}$. This gives each photon an equal probability to be selected for the cell, irrespective of the order they arrive in.

Our second modification solves a possible race condition when writing the photon into a cell. As the write of a larger structure is not atomic, it is possible to have a result that is combined from photons from multiple threads. To prevent this, we store each photon into its globally unique memory location and write only the index of the photon.

4.5.3 Implementation Detail: Improved Hash Grid Query

Algorithm 2 shows two different approaches to performing a range query in a Hash Grid. The standard *NaiveQuery* processes all cells that are within range in a serial manner. On a GPU, all threads wait until each thread has processed all photons in its current cell before processing the next cell. This means that if the total number of photons is the same across the threads but differs between the cells, some threads might be idle, while others are still processing their photons from a given cell. Our *WhileQuery* removes this issue in a manner similar to the "while-while" loop used in [Aila and Laine 2009]. First all threads find their next photon to process from all the cells in range before the photons are processed. This way the query execution is driven only by the number of photons for each thread and not by their distribution within the grid cells.

We measured the performance of Progressive Photon Mapping (PPM), Stochastic Progressive Photon Mapping (SPPM), and Progressive Bidirectional Photon Mapping (PBPM) using both query algorithms on Full Hash Grid. Table 4.4 represents the results as a speedup of the whole algorithm when using *WhileQuery*. We see that in many scenes the difference for both PPM and SPPM is negligible. However, in *BiolitFull* and *CrytekSponza* the speedup is 10-43%, as both scenes have greatly varying photon density. The effect on PBPM is significantly smaller, possibly due to the overall complexity of the algorithm, meaning the density estimation itself represents only a smaller fraction of the total time. The effect is more pronounced on GTX 680 than on GTX 580.

```
def NaiveQuery:
    activeCell = cellsInRange.nextCell
    while activeCell ≠ None:
        activePhoton = activeCell.nextPhoton
        while activePhoton ≠ None:
            if activePhoton is in range:
                Process activePhoton
            activePhoton = activeCell.nextPhoton
        activeCell = cellsInRange.nextCell
def WhileQuery:
    activeCell = cellsInRange.nextCell
    while True:
        repeat
            activePhoton = activeCell.nextPhoton
            if activePhoton = None:
                activeCell = cellsInRange.nextCell
                activePhoton = activeCell.nextPhoton
                if activeCell = None:
                    return
            if activePhoton not in range:
                activePhoton = None
        until activePhoton ≠ None
        Process activePhoton
```

Algorithm 2: *Hash Grid Query*: The *NaiveQuery* processes each cell in range serially, introducing possible inefficiencies when cells examined by threads in a warp contain different numbers of photons. The *WhileQuery* essentially concatenates all photons from all cells in range and processes this list, limiting the inefficiency only to cases when the total number of photons in range differs between threads.

4.5.4 Results and Dicussion

In this section we discuss memory requirements and performance of these structures. All our implementations are single-kernel (the performance reasons are identical to LVC-BPT) and we therefore omit the *sk* suffix from the acronyms.

Memory requirements. Both Full Hash Grid and Rectified Stochastic Hash Grid require only 4 B per cell. In our setup the Hash Grids occupy only 3.5 MB. kD-tree memory requirements depend on the specific flavor used, but in our tests the size of the tree was always below 20 MB. In all cases, the required storage is dominated by the photons not the data structure.

	GeForce GTX 580			GeForce GTX 680		
	PPM	SPPM	PBPM	PPM	SPPM	PBPM
CoronaRoom	0.99	0.99	0.99	0.98	1.03	1.03
CoronaWatch	1.01	1.01	1.01	1.02	0.99	1.01
LivingRoom	1.03	1.03	1.00	1.01	1.03	1.01
BiolitFull	1.33	1.31	1.03	1.43	1.34	1.04
CrytekSponza	1.09	1.08	1.05	1.11	1.13	1.08
GrandCentral	1.03	1.03	1.01	0.98	0.98	1.02
Average	1.09	1.09	1.01	1.10	1.10	1.03

Table 4.4: *WhileQuery speedup*: Speedup of *WhileQuery* over *NaiveQuery* tested on Progressive Photon Mapping (PPM), Stochastic Progressive Photon Mapping (SPPM), and Progressive Bidirectional Photon Mapping (PBPM).

	GeForce GTX 580			GeForce GTX 680		
	Full	Stoch.	kD	Full	Stoch.	kD
CoronaRoom	**1.00**	0.87	0.43	**0.91**	0.78	0.46
CoronaWatch	**1.00**	0.54	0.44	**0.91**	0.48	0.43
LivingRoom	**1.00**	0.51	0.15	**1.06**	0.59	0.13
BiolitFull	**1.00**	0.45	0.11	**1.04**	0.50	0.10
CrytekSponza	**1.00**	0.64	0.25	**1.21**	0.80	0.22
GrandCentral	**1.00**	0.53	0.21	**1.10**	0.61	0.18
Average	**1.00**	0.55	0.22	**1.06**	0.61	0.20

Table 4.5: *Acceleration Structure comparison*: Speedup, in terms of time to a given quality, of Full Hash Grid (Full), Rectified Stochastic Hash Grid (Stoch.), and kD-tree (kD), relative to Full Hash Grid on GTX 580.

Performance. Table 4.5 shows the relative speedup of the three acceleration structures as tested on PPM: Full Hash Grid (Full), Stochastic Hash Grid (Stoch.), and kD-tree (kD). To compare them we use the same time to the same quality (RMSE) method introduced in Section 4.4.4. Our baseline is RMSE achieved by Full Hash Grid on GTX 580 in 10 minutes.

We can see that the overall performance of kD-trees is, at best, 2× slower than the Full Hash Grid: not only is the build time of the kD-tree larger than for Full Hash Grid (up to 75×), but the queries themselves also take slightly more time, as the traversal of the tree is more costly than simply gathering photons from 8 cells.

We note that Hash Grid, unlike kD-tree, is susceptible to hash collisions, where the 8 examined cells will include photons from different parts of the scene. This effect is responsible for the longer query times in the *LivingRoom*, where the highly concentrated caustic photons show up and

					GeForce GTX 580				
			Full		Rectified Stochastic			kD-tree	
Algorithm	#photons	Light	Construct.	Camera	Light	Camera	Light	Construct.	Camera
CoronaRoom	147k	16.81 ms	1.56 ms	10.82 ms	16.72 ms	10.59 ms	18.50 ms	34.45 ms	15.37 ms
CoronaWatch	175k	15.18 ms	1.68 ms	14.77 ms	15.21 ms	12.97 ms	14.69 ms	41.76 ms	20.75 ms
LivingRoom	1788k	49.93 ms	9.49 ms	27.29 ms	51.52 ms	21.16 ms	47.85 ms	506.27 ms	31.56 ms
BiolitFull	1493k	36.73 ms	6.69 ms	17.78 ms	37.95 ms	12.32 ms	35.55 ms	504.84 ms	25.05 ms
CrytekSponza	571k	42.73 ms	3.50 ms	14.26 ms	43.65 ms	12.14 ms	36.77 ms	191.52 ms	13.51 ms
GrandCentral	746k	44.30 ms	4.54 ms	18.76 ms	44.77 ms	16.02 ms	42.00 ms	254.72 ms	20.18 ms
					GeForce GTX 680				
CoronaRoom	147k	16.09 ms	2.05 ms	14.65 ms	16.55 ms	14.53 ms	16.20 ms	32.79 ms	16.46 ms
CoronaWatch	175k	15.48 ms	2.27 ms	17.20 ms	15.50 ms	16.55 ms	14.97 ms	44.36 ms	21.69 ms
LivingRoom	1788k	38.59 ms	17.36 ms	27.39 ms	41.61 ms	20.24 ms	36.67 ms	603.88 ms	30.10 ms
BiolitFull	1493k	27.70 ms	12.49 ms	19.39 ms	30.96 ms	14.73 ms	25.42 ms	610.04 ms	24.20 ms
CrytekSponza	571k	27.45 ms	5.83 ms	17.16 ms	28.35 ms	15.84 ms	24.16 ms	232.81 ms	15.95 ms
GrandCentral	746k	31.12 ms	7.64 ms	23.49 ms	34.56 ms	19.79 ms	35.97 ms	311.52 ms	18.92 ms

Table 4.6: Time per frame for Full Hash Grid (Full), Rectified Stochastic Hash Grid (Rectified Stochastic), and kD-tree, broken down to the separate passes of PPM: photon tracing (Light), acceleration structure construction (Construct.), and camera sub-path tracing and density estimation (Camera). The times have been averaged over 15 minute runs. Note that the Stochastic Hash Grid does not have a separate construction phase, as photons are inserted during photon tracing. The column #photons shows the average number of photons per frame.

have to be evaluated in unrelated cells across the scene. Even so, the overall performance of the Full Hash Grid on this scene is better than the kD-tree mainly due to the shorter build time.

While the Rectified Stochastic Hash Grid is about 25% faster, per frame, the lower number of photons used in density estimation results in an overall lower performance than the Full Hash Grid. We conclude that the cost of building a Full Hash Grid is negligible compared to the benefits and use it in all our tests.

4.5.5 Conclusions

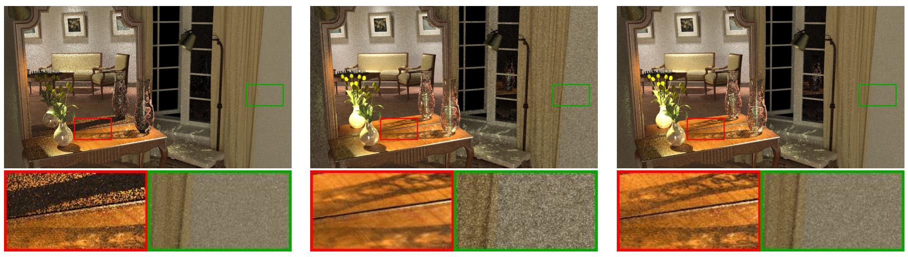
In this section, we investigated three progressive algorithms based on Photon Mapping, namely Progressive Photon Mapping (PPM), Stochastic Progressive Photon Mapping (SPPM), and Progressive Bidirectional Photon Mapping (PBPM). The common element of all the algorithms is density estimation, based on gathering photons in a certain radius from a query point. We examined three data structures designed to accellerate this process: kD-tree, Full Hash Grid, and Rectified Stochastic Hash Grid. Note that we did not use the original Stochastic Hash Grid [Hachisuka and Jensen 2010], as its incorrect convergence prevents using our quality metric.

As an implementation improvement of Hash Grid queries, we proposed the *WhileQuery* that processes photons from all cells as a single group, reducing thread divergence during evaluation. This speeds up range queries on Full Hash Grid by up to 43% and is used in all our tests.

While the Rectified Stochastic Hash Grid is faster per frame than the Full Hash Grid, due to the fact that it has no construction phase, this does not make up for the lower number of stored photons, resulting in 13 to 55% slower performance in all scenes except the *CrytekSponza*. Both Hash Grids significantly outperform the kD-tree, mainly due to its substantial construction cost. As result, we recommend using the Full Hash Grid with our *WhileQuery* for GPU implementation of Photon Mapping methods.

4.6 Vertex Connection and Merging

In the following we combine the experience gathered from the previous sections to introduce the first GPU implementation of the recent Vertex Connection and Merging (VCM) algorithm [Georgiev et al. 2012, Hachisuka et al. 2012]. While Bidirectional Path Tracing fails to capture reflected caustics and methods based on Photon Mapping have difficulties producing noise-free diffuse surfaces under illumination from distant light sources, Vertex Connection and Merging combines the best of both algorithms by using Multiple Importance Sampling to give a high weight to the best strategy for each situation (see Figure 4.9).



(a) Bidirectional Path Tracing **(b)** Progressive Bidirectional PM **(c)** Vertex Connection and Merging

Figure 4.9: The reflected caustics (red) are extremely difficult for Bidirectional Path Tracing (left). Progressive Bidirectional Photon Mapping (middle), on the other hand, results in noticeable noise on the diffuse walls (green). In the same rendering time (10s), Vertex Connection and Merging (right) handles both effects well.

4.6.1 Algorithm Overview

In the original paper the authors use a standard approach to Bidirectional Path Tracing, tracing an equal number of light and camera sub-paths, forming predetermined path pairs. First, all light sub-paths are traced and their vertices are stored. Then camera sub-paths are traced, each camera sub-path vertex is connected to all vertices on the corresponding light sub-path, as well as merged with vertices (i.e., 'photons'), from all light sub-paths, that are within a given range. Vertex merging is a name used for an operation virtually identical to the range query in Progressive Bidirectional Photon Mapping, the only difference being the different calculation of the Multiple Importance Sampling weights.

4.6.2 Proposed GPU Implementation

Given our Light Vertex Cache Bidirectional Path Tracing (LVC-BPT) introduced in Section 4.4.3 and our GPU implementation of Bidirectional Photon Mapping (Section 4.5) a GPU implementation of Vertex Connection Merging is easy. Our implementation first traces all light sub-paths. Then a Full Hash Grid is built over these vertices to accelerate range queries (we use our *WhileQuery*) as in PBPM. Finally, a camera pass is performed where each camera sub-path vertex is connected to a predetermined number of light sub-path vertices (identical to LVC-BPT) and also merged with light sub-path vertices using a range query (identical to PBPM).

4.6.3 Results and Discussion

Memory requirements. Memory requirements are almost identical to the requirements of LVC-BPT. Using the formulas introduced in Table 4.2 we arrive at 17 to 171 MB for VCMsk, with VCMmk adding another 137 to

	GeForce GTX 580		GeForce GTX 680	
	VCMsk	VCMmk	VCMsk	VCMmk
CoronaRoom	**1.00**×	0.98×	**0.93**×	0.85×
CoronaWatch	**1.00**×	0.79×	**0.87**×	0.71×
LivingRoom	1.00×	**1.04**×	**0.92**×	0.88×
BiolitFull	1.00×	**1.03**×	**0.92**×	0.86×
CrytekSponza	**1.00**×	0.91×	**1.00**×	0.85×
GrandCentral	**1.00**×	0.97×	**0.92**×	0.87×
Average	**1.00**×	0.95×	**0.93**×	0.83×

Table 4.7: Relative performance of VCM implemented with a single kernel (VCMsk) and using two kernels (VCMmk).

158 MB. We also need memory for the Full Hash Grid used to accelerate vertex merging range queries, adding a small memory footprint of 3.5 MB.

Kernel configurations. Similar to Light Vertex Cache BPT, we tested both single-kernel (VCMsk) and multi-kernel (VCMmk) implementation of VCM. Table 4.7 shows the performance relative to VCMsk on the GTX 580. We can see that in almost all cases, VCMmk is inferior to VCMsk. On GTX 580, VCMmk outperforms VCMsk in only two scenes by 3 and 4% respectively, but in general is approximately 5% slower. This difference is more pronounced on the GTX 680. Because multi-kernel VCM uses a larger light sub-path vertex structure as well as the whole merging stage, it has greater pressure on the memory system, leading to a decrease in performance with respect to the single-kernel implementation. We conclude that the single-kernel VCM is the better choice for both GPUs.

To conclude, our Vertex Connection and Merging implementation draws heavily on the experiences from both Bidirectional Path Tracing and Progressive Bidirectional Photon Mapping implementations. The main approach is almost identical to our LVC-BPT, using a single-kernel implementation. Compared to the CPU implementation used in [Georgiev et al. 2012], our GPU implementation achieves a 6 to 10× speedup on the scenes tested here.

4.7 Algorithm Comparison

Up until now we have focused on optimizing the individual algorithms. Now, with state-of-the-art GPU implementations of a number of light transport simulation algorithms within a single framework at our disposal we have a unique opportunity to compare the algorithms against each other. Our comparison is "unbiased" in the sense that we did not introduce any of the algorithms in this chapter and so have no motivation to selectively prefer

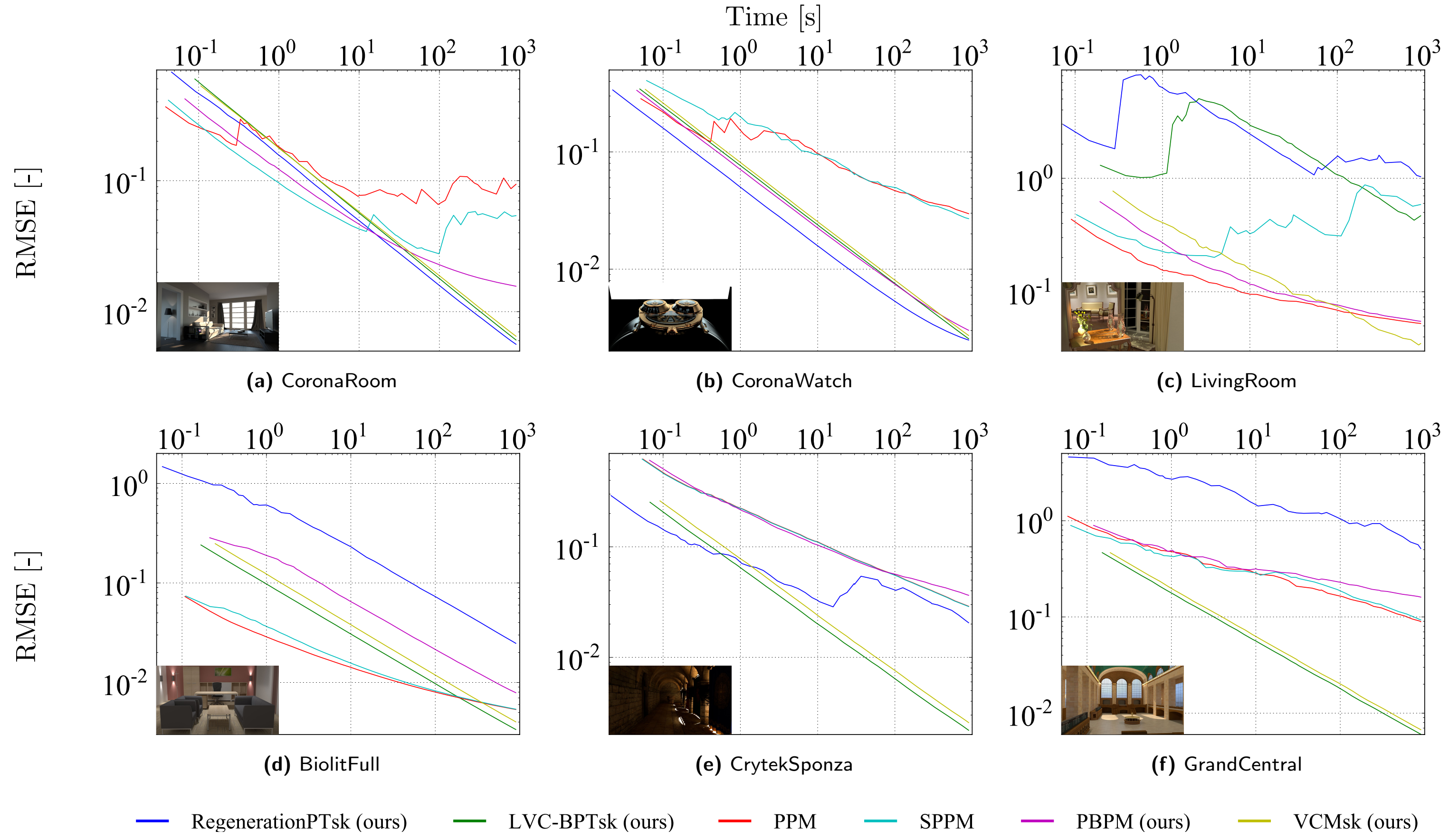


Figure 4.10: The log-log plot of RMSE-vs-time convergence of the six tested methods on each of the test scenes.

.

any of them. Our results are not strictly GPU-specific; we are not aware of a similar unbiased comparison for CPU implementations either.

Figure 4.10 shows results for GTX 680. Progressive Photon Mapping (PPM), Stochastic Progressive Photon Mapping (SPPM), and Bidirectional Photon Mapping (PBPM) use single-kernel implementations and all algorithms that perform density estimation use the Full Hash Grid and our *WhileQuery*. The results for GTX 580 using multi-kernel versions of RegenerationPTmk and LVC-BPTmk closely follow the results of GTX 680. The graphs are given in the supplemental material in Appendix A. Our references are computed by NaiveBPT in 10 hours except for the *LivingRoom* scene where the reference is computed by VCMsk, as BPT cannot resolve the reflected caustic even after 10 hours.

4.7.1 Path Tracing

As expected, Path Tracing excels in scenes with a simple illumination. From our test scenes, it achieves the best results on *CoronaRoom*, a mostly diffuse scene where the majority of the illumination comes from an environment lighting behind the glass-less window.

Good results are also achieved on *CoronaWatch*, which is dominated by direct illumination. However, on Figure 4.10b we can see that the convergence of PT starts to level off after approximately 100 s, due to inappropriately sampled gloss-to-gloss transport.

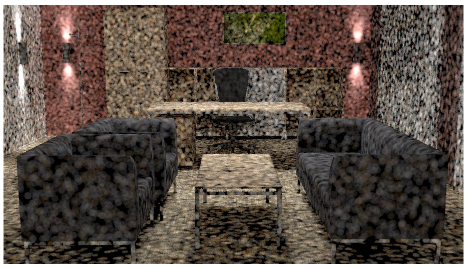
The somewhat surprising poor performance on *GrandCentral* is caused by the many individual point lights in the niches beneath the ceiling. While the overall illumination of the scene is smooth, these niches are each illuminated by essentially a single point light, which poses a great challenge for next event estimation. This causes the majority of the variance we see in the graph.

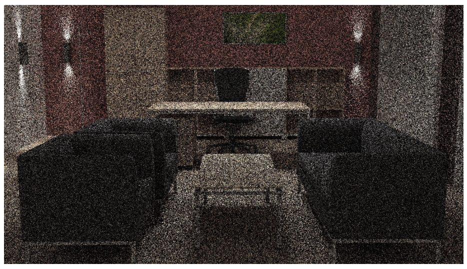
The other three scenes are strongly illuminated by indirect light sources, which renders next event estimation essentially useless in these cases and the overall convergence of PT suffers.

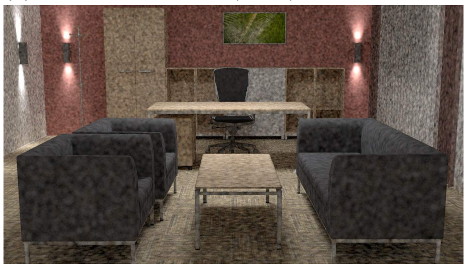
4.7.2 Bidirectional Path Tracing

Bidirectional Path Tracing performs well on all the scenes except *LivingRoom*, a scene tailored to showcase Vertex Connection and Merging, where BPT does not have any technique suitable for efficiently capturing reflected the caustics.

On the two scenes dominated by direct illumination, i.e., *CoronaRoom* and *CoronaWatch*, the algorithm is slower than Path Tracing, as the extended set of techniques offered by BPT is not really useful. In *GrandCentral*, the niches are illuminated by paths traced from the point lights,


(a) PPM - 1st iteration (0.11s)


(b) BPT - 1st iteration (0.17s)


(c) PPM - 10th iteration (0.86s)

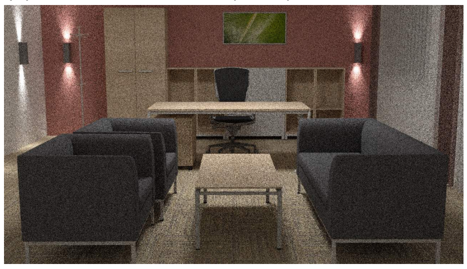

(d) BPT - 10th iteration (1.63s)

Figure 4.11: After the first iteration (top) the PPM (a) with a large radius gives a better initial intuition about the lighting than BPT (b). After the tenth iteration (bottom) the asymptotically faster convergence of BPT has already removed this early advantage.

greatly reducing noise when compared to PT. The algorithm naturally handles well both scenes that are dominated by indirect illumination (*BiolitFull* and *CrytekSponza*).

4.7.3 Photon Mapping-based Methods

The Photon Mapping-based methods are most beneficial on *LivingRoom*, where none of the path-based algorithms can efficiently capture the reflected caustics. Somewhat surprising is the good behavior of Progressive Bidirectional Photon Mapping on both *CoronaRoom* and *CoronaWatch*, when compared to Progressive Photon Mapping and Stochastic Progressive Photon Mapping. The key insight here is that PBPM has Path Tracing without next event estimation amongst its techniques and both scenes, with their large area lights, represent a very good case for this technique. Even up to the point that the convergence on *CoronaWatch* is actually dominated by it, matching the convergence rate of path based techniques.

In the case of *BiolitFull*, PPM and SPPM give very good results in a short time. This is due to mostly diffuse nature of the scene, where each photon contributes to several pixels, giving a good, albeit blurry, initial estimate (Figure 4.11).

However, in the *CrytekSponza*, which is purely diffuse and also indirectly illuminated, the results are quite different. Unlike *BiolitFull*, where all the lights are within a single room and roughly half the scene and a third of the lightsources are visible to the camera, in *CrytekSponza* we see only a fraction of the scene and all the lights are completely outside the view. As result, a significantly lower fraction of all photons contributes to the frames (0.11% vs 6.3%), giving much more noisy results.

4.7.4 Vertex Connection and Merging

VCM excels in *LivingRoom*, which has been tailored to showcase the algorithm. It resolves the reflected caustics using techniques from PBPM, while resolving the diffuse light transport with BPT techniques. In the other scenes however, VCM simply mirrors the performance of BPT, in general being slightly slower, as the vertex merging techniques have a non-negligible cost.

4.8 Conclusion

In this chapter we presented an extensive study of GPU-based implementations of several progressive light transport simulation algorithms. For each algorithm, we evaluated existing and new approaches on GPUs of two different NVIDIA architectures, the older Fermi (GTX 580) and the newer Kepler (GTX 680) architecture. To accelerate the ray tracing operator we use a highly optimized GPU library by NVIDIA as the best currently available substitute for a more ideal hardware solution, such as discussed in Chapter 3.

In the Path Tracing section we examined the low-level details of mapping the basic building block of the more complex algorithms path sampling onto the GPU. We conclude that for optimal performance it is beneficial to use a low number of separate kernels, as the lower number of loads and stores outweighs the gains from improved GPU occupancy. Notably, on Kepler, the speed gained by using Path Tracing with Regeneration with only a single kernel actually matches the speed gained by stream compaction used in state-of-the-art Streaming Path Tracing with Regeneration, using 2 kernels.

In the Bidirectional Path Tracing section, we show that maximal simplification of the algorithm structure leads to the best performance. We proposed our Light Vertex Cache BPT, storing only light path vertices without the notion of sub-paths. Doing so increases the performance by 30-60% when compared to the stateof-the-art, while at the same time removing the necessity of a maximum path length.

In the Photon Mapping section we show that a simpler but asymptotically slower algorithm, in our case the Hash Grid, can outperform a more complex asymptotically faster algorithm, in our case the kD-tree. Another important low-level optimization is our *WhileQuery*, used to gather photons from a Hash Grid. By removing thread synchronization after gathering photons from a single cell, we reduce the thread divergence of the gather process, which can increase the performance by up to 40%. All of the findings are combined in Vertex Connection and Merging, showing the first GPU implementation of the algorithm.

Our algorithm comparison shows that the raw performance of Path Tracing makes it ideal for scenes with a low lighting complexity, but that the more sophisticated sampling strategies of Bidirectional Path Tracing are useful in scenes with more complex lighting. In most scenes, the performance of Vertex Connection and Merging follows that of BPT, but due to the overhead of merging (which has only a marginal impact on the final image) it is about 15% slower to achieve the same image quality. Of course, in scenes with a strong reflected caustic component, VCM outperforms BPT since the merging is essential to capturing these light paths. The Photon Mapping based algorithms do not present a significant advantage over any of the other algorithms.

Chapter 5

Global and Local VPLs

Despite the advances in rendering algorithms described in the previous chapter, accurate rendering of scene with multiple glossy materials remains a major challenge. Scenes where the illumination is strongly influenced by glossy interreflections (such as Figure 5.1) are fairly prominent in design applications. There, providing effective previews is an important part of the design process and tradeoff of small inaccuracies for significant performance gains can be acceptable. All of the algorithms introduced in the previous section will arrive at the perfectly accurate solution, but as none of them has a good sampling strategy for this type of paths it would require a prohibitively large number of samples and none of them is therefore well suited for this task.

On the other side of the spectrum of available algorithms are interactive techniques, which trade off accuracy for performance. These approaches achieve performance by limiting their support for materials by assuming diffuse or low-frequency material representations, sometimes extensible to perfectly specular [Wang et al. 2009] and/or requiring significant precomputation as in PRT-based algorithms [Cheslack-Postava et al. 2008]. Many-

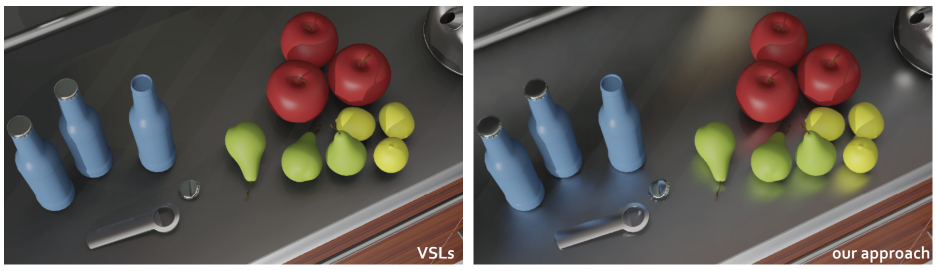


Figure 5.1: *Comparison of our approach with Virtual Spherical Lights (VSLs).* Left: VSLs fail to capture small local glossy reflections (time: 6 m 26 s). Right: Our method (time: 4 m 59 s) computes these reflections more efficiently and accurately by using visibility approximations for the low-rank global light transport and using local lights for the high-rank localized light transport.

light methods are gaining popularity because of their simplicity and their ability to effectively leverage GPU performance. These formulations convert global illumination into the problem of rendering with many virtual point lights (VPLs) (see Section 2.3.6). These approaches assume that any individual VPL does not significantly affect lighting. Thus, "spiky" lights (either because of BRDF or proximity) are eliminated through clamping their contribution to some user-specified maximum value.

However, clamping can negatively impact accurate material perception [Křivánek et al. 2010], and thus curb the use of these algorithms in design applications. Recently, virtual spherical lights [Hašan et al. 2009] were introduced to avoid the illumination loss from clamping, but at the expense of blurring some of the sharp reflections that provide important material cues. In this chapter, we introduce an alternative approach that alleviates these limitations and offers a better approximation of global illumination rendering with sharp glossy reflections.

Our approach is to split the light transport into two components: A dense global component and a sparse localized component. The global component of the light transport matrix corresponds to standard clamped VPLs and can be approximated well by techniques like matrix row-column sampling [Hašan et al. 2007]. In the context of the previous chapter, the global component would correspond to mostly diffuse illumination cast over long (with respect to the scene size) distances. We further observe that visibility can be separately (and often more coarsely) approximated compared to shading, particularly in scenes with glossy materials. We leverage this observation to construct a novel visibility clustering algorithm that requires fewer shadow maps than standard matrix row-column sampling.

To retrieve the clamped illumination, Kollig and Keller introduced a clamping compensation algorithm [2004]. This approach uses Path Tracing on all paths that would have been clamped in the VPL solution and weights their contribution to obtain the correct unbiased result. It largely bases its effectiveness on the assumption that most paths are not clamped and therefore will not require the path traced compensation. It also assumes that all surfaces are diffuse which allows limiting the maximum ray lengths in path tracing, further improving the performance. Unfortunately, in scenes with a significant amount of glossy surfaces neither assumption holds and essentially the whole scene has to be path traced, degenerating the overall algorithm to the performance of Path Tracing. Using more advanced global illumination algorithms instead of Path Tracing is theoretically possible, even though we are not aware of any such approach being used in practice, but would not solve the principal issue: If a large portion of the illumination is clamped, the performance of the algorithm is dominated by the performance of the chosen compensation approach.

In contrast, our approach approximates the local, high-rank component of the light transport, using another many-light strategy with a different set of approximations. We introduce a scheme to trace *local* virtual lights and have these local lights correct the global solution by compensating for the energy lost through clamping. As each local light contributes only the clamped portion of the energy, its effect is both short distance and focused: The more specular the material, the more focused is the contribution of the light. Based on this observation, our local lights illuminate small tiles of 32×32 pixels, and we are able to approximate their visibility, noting that most shadowing effects are already correctly handled in the global many-light component of the solution. The overall time required to render the local lights is on the same order as for the global lights. Thus, by coupling visibility approximation for global lights and compensation through local lights we can shade highly glossy materials efficiently.

Our global and local VPL technique with visibility approximations can achieve efficient rendering of scenes with glossy light transport in minutes for scenes with complex materials and lighting. The proposed solution could benefit applications where previewing of product appearance is critical and can have significant economic impact, for example in industrial design and architecture.

5.1 Related Work

Many previous global illumination methods have been designed without focus on glossy interreflection and thus have problems when applied to glossy scenes. In many cases, the problem is that illumination is gathered from a discrete set of samples that has not been adapted to the requirements of the illumination receivers. In other techniques the adaptivity comes at a significant cost.

Non-adaptive methods. Rendering algorithms based on the instant radiosity formulation [Keller 1997] generate a number of virtual point lights (VPLs) to approximate indirect illumination and shade visible surfaces from these lights. Matrix row-column sampling [Hašan et al. 2007] samples a small number of rows and columns from the large matrix of light-surface connections. Lightcuts [Walter et al. 2006] use a hierarchy on the set of lights and/or surface samples to improve the scalability of connection computations. Ritschel et al. [2008] use approximate shadow maps to accelerate rendering. Laine et al. [2007] propose incremental instant radiosity for animated sequences, where only a subset of shadow maps is recomputed in each frame.

In theory, many-light methods provide a correct, unbiased solution to the rendering equation; however, "spikes" (singularities) can be produced in parts of the image that strongly depend on a single VPL, since the BRDF and the geometry term can have a very large value compared to the light density. This is normally handled by *clamping*, which often removes interesting glossy illumination effects. The virtual spherical light (VSL) approach [Hašan et al. 2009] addresses the singularities and clamping requirement, but in some cases the density of the non-adaptively distributed VSLs is insufficient to reproduce sharp details of glossy reflection. This is also true for lightcuts variants that do adaptively choose a subset of lights for shading, but the original discrete set itself is usually not sufficient for highly glossy effects.

The micro-rendering framework of Ritschel et al. [2009] provides interactive solutions and does not lead to singularities, but also has no mechanism to increase the density of samples in areas of spatial proximity or in glossy BRDF lobes, and cannot be easily extended to multiple bounces. Laurijssen et al. [2010] replaces sharp BRDFs on a cluster of camera path vertices by a smoother distribution, thereby reducing noise in indirect highlights.

Adaptive methods. Path Tracing, Bidirectional Path Tracing, and Metropolis Light Transport [Veach 1997] use a number of strategies based on BRDF importance sampling to preferentially find paths with strong contributions to image pixels. These are unbiased approaches that can deliver highest quality; however, they are wasteful in the sense that expensively constructed paths are usually not reused for many pixels. Kollig and Keller [2004] compensate for the clamped illumination in instant radiosity approaches by a recursive path tracing approach. However, in glossy scenes the compensation can be as slow as pure path tracing. Our local light approach is very similar in how it computes the compensation for clamping, but it is capable of doing so in time comparable to the clamped many-light rendering.

Segovia et al. [2006] introduced a many-light method that traces VPLs from the camera; this is related to our approach, but their VPLs are used globally and only applied to diffuse scenes. Path reuse [Bekaert et al. 2002] uses a similar concept to our local lights in a path-tracing context, but full visibility is checked for the connections used for path reuse, which limits the possible speed-up. In contrast, our technique can afford to approximate the visibility for local lights, since it is computing only the compensation to a global pass that already handles most shadowing effects.

Photon Mapping with final gathering [Jensen 2001] can utilize adaptive BRDF importance sampling from the camera, but the photon distribution itself is not adaptive, so large numbers of photons and nearest neighbors are required in glossy scenes. Progressive Photon Mapping [Hachisuka et al. 2008] variants addresses memory requirements of photon mapping by multipass processing, but does not fundamentally improve convergence. Please see Chapter 4 for more details on these methods.

Other related work. Our visibility clustering solution is similar to [Dong et al. 2009], which uses k-means on light positions and normals, while we use a data-driven approach based on a sparse sampling of visibility and shading. Arikan et al. [2005] decompose irradiance into near and far components, handling them by different approaches. This is related to our clamping-compensation decomposition, but does not consider glossy interreflection. Cheslack-Postava et al. [2008] introduce a precomputed method based on visibility approximation. Their data-driven light tree construction is related to our visibility clustering algorithm.

5.2 Overview

For this analysis, we will use the path integral formulation of global illumination, introduced by Veach [Veach 1997] and described in greater detail in Section 2.1.4. In this formulation the illumination on a pixel j can be computed as an integral over all light paths in the scene passing through the pixel:

$$I_j = \int_\Omega f_j(\bar{x}) d\mu(\bar{x}),$$

where μ is a measure on the path space $\Omega = \bigcup_{k\geq 1} \Omega_k$, and Ω_k is the space of paths with k segments, $\bar{x} = x_0 x_1 \ldots x_k$, such that x_0 is the camera position, x_1 is the surface point directly visible through the pixel, x_k is on a light source and $x_2, \ldots, x_{k-1}$ are any light bounce points in the scene. The path contribution $f_j(\bar{x})$ is a product of BRDF, visibility, and geometry terms on the vertices of the path, finally multiplied by light emission:

$$f_j(\bar{x}) = \left(\prod_{i=1}^{k-1} f_r(x_{i-1} {\leftarrow} x_i {\leftarrow} x_{i+1}) V(x_i {\leftrightarrow} x_{i+1}) G(x_i {\leftrightarrow} x_{i+1}) \right) L_e(x_k {\rightarrow} x_{k-1}),$$

where $f_r(x_{i-1} {\leftarrow} x_i {\leftarrow} x_{i+1})$ is a BSDF function, $V(x_i {\leftrightarrow} x_{i+1})$ is the visibility term, and $G(x_i {\leftrightarrow} x_{i+1})$ is the geometry term, and $L_e(x_k {\rightarrow} x_{k-1})$ is the emission of the path's endpoint.

We will assume the scene is lit by a set of direct point light sources $\mathcal{L}_d$; area lights or environment maps can be handled by discretization to a large number of point lights. Under this assumption, paths of length 1 and 2 (emission and direct illumination) can be easily handled by summation over point lights, so the problem is reduced to computing the indirect component over paths of length 3 or more. Let $\Omega_{ind} = \bigcup_{k\geq 3} \Omega_k$. The indirect component I_j^{ind} can be computed by Monte Carlo integration, sampling N random paths and summing their contributions:

$$I_j^{ind} = \int_{\Omega_{ind}} f_j(\bar{x}) d\mu(\bar{x}) \approx \sum_{i=1}^{N} \frac{f_j(\bar{x}_i)}{\rho(\bar{x}_i)}. \tag{5.1}$$

Here $\rho(\bar{x})$ is the path density (the number of paths per unit measure), and has to be positive for all paths with non-zero contributions. If all samples are independent and identically distributed with probability density $p(\bar{x})$, then we get the familiar $\rho(\bar{x}) = Np(\bar{x})$, though this does not have to be the case; for example, stratification can be applied.

Many-light methods. Algorithms based on the many-light approximation [Keller 1997] provide a sampling strategy for the above integral, by tracing sub-paths from direct light sources, treating these sub-paths as virtual point lights (VPLs), and connecting them to the visible surface samples (i.e. camera sub-paths of the form $x_0 x_1$). This can conveniently be expressed as a lighting matrix $\mathbf{A}$ of light-sample contributions (i.e., the element $\mathbf{A}_{ij}$ will be the contribution of light j to sample i). In theory, this provides an unbiased solution to the global illumination problem. When compared to the more traditional Monte Carlo-based methods, such as introduced in the previous chapters, the main advantages of the many-light algorithms are: Low number of samples required for a visually smooth image (i.e., low rather than high frequency noise), the simplicity of the algorithm, and the possibility of many efficient implementations on various hardware platforms. All these reasons lead to the significant popularity of many-light methods in recent research.

Unfortunately, this sampling strategy is not always ideal, since the VPL density can be insufficient in corners and within glossy lobes. More precisely, the problem is that the path contribution $f_j(\bar{x})$ contains the following terms that have not been importance-sampled [Hašan et al. 2009]:

$$f_r(x_0 \leftarrow x_1 \leftarrow x_2)\, G(x_1 \leftrightarrow x_2)\, f_r(x_1 \leftarrow x_2 \leftarrow x_3) \tag{5.2}$$

If any of these terms is large (which often happens in corners and within glossy lobes), then a naive application of the algorithm can cause disturbing artifacts. This is usually handled by clamping the terms in (Eq. 5.2) to a user-specified constant c, and also by replacing the second BRDF term by a diffuse approximation (which we do not do). The clamping has an additional benefit of lowering the rank of $\mathbf{A}$, which improves the convergence of methods like row-column sampling and lightcuts, but the major drawback is that much glossy interreflection is lost. Virtual spherical lights [Hašan et al. 2009] use blurring rather than clamping to lower the rank of $\mathbf{A}$, which preserves illumination energy, but loses the clarity of glossy reflections.

Compensation. Compensating for the clamped illumination was proposed by Kollig and Keller [2004], but their solution is most efficient for diffuse scenes; in the presence of glossy BRDFs it is only marginally more efficient than pure path tracing. Instead, we notice that the low-rank, dense, *global* portion of the transport is handled well by the primary many-light technique, while the missing illumination tends to be sparse or *localized* in position-direction space. We propose to use a second many-light technique based on *local lights* to compensate. Clamping only the geometry term is not sufficient in our case, as highly specular surface interactions will have objectionable artifacts over longer distances than diffuse surfaces. Therefore,

```
def render(scene, opts):
    img = zero_img(opts.img_size)
    dfb = create_deep_framebuffer(scene, opts.img_size)
    global_lights = scene.direct_lights() + trace_indirect_lights(scene)
    render_global_component(scene, opts, dfb, global_lights, img)
    render_local_component(scene, opts, dfb, global_lights, img)
    return img

def render_global_component(scene, opts, dfb, global_lights, img):
    # create reduced shading and visibility matrices
    S = zeros(opts.num_rows, len(global_lights))
    V = zeros(opts.num_rows, len(global_lights))
    row_pixels = choose_random_pixels(dfb, opts.num_rows)
    foreach pixel in row_pixels:
        shading, visibility = render_row_on_gpu(pixel, global_lights)
        S(pixel.index, :) = shading
        V(pixel.index, :) = visibility

    # visibility clustering
    clusters = [initial_cluster(global_lights, S, V)]
    while len(clusters) <opts.num_clusters:
        c = extract_highest_cost_cluster(clusters)
        c1, c2 = split(c)  # see Section 5.3
        clusters += [c1, c2]

    # render clusters
    foreach c in clusters:
        vis_mask = render_visibility_on_gpu(dfb, c.rep)
        foreach light in c.items:
            shading = render_shading_on_gpu(dfb, light, opts.clamp)
            img += vis_mask * shading

def render_local_component(scene, opts, dfb, global_lights, img):
    foreach pixel in dfb.pixels:
        for i in range(0, opts.num_local_lights_per_pixel):
            # create local lights by BRDF sampling
            direction = pixel.sample_brdf()
            light = trace(pixel.position, direction)
            if light == None: continue

            # connect to global light to get intensity
            # and $\omega_i$ of the local light
            intensity, incoming = connect(light, global_lights)

            # local light rejection
            clamped_term = fr(camera <- pixel <- light) * G(pixel, light) *
                  fr(pixel <- light <- incoming)
            if clamped_term <opts.clamp / 2: continue

            # choose a tile that contains this pixel
            tile = choose_random_offset_tile(pixel)

            # find the density of this lights, accounts for rejection
            rho = 0
            foreach p in tile.pixels: rho += eval_area_pdf_on_gpu(p, light)

            light.incoming = incoming
            light.intensity = intensity / rho
            foreach p in tile.pixels: img[p] += shade_w2_on_gpu(p, light)

def connect(local_light, global_lights):
    # for simplicity we show a single powe-sampled connection:  other schemes
    # are possible (we use Kollig-Keller compensation)
    global_light, prob = sample_power(global_lights)
    intensity = shade(local_light, global_light) / prob
    return intensity, normalize(global_light.pos - local_light.pos)
```

Algorithm GLL: *Global and Local VPLs*:Pseudo-code of our algorithm.

it is necessary to clamp on the product of the geometry term and the BRDFs. We define the clamping weights w_1 and w_2:

$$w_1(\bar{x}) = \min\left(1, \frac{c}{f_r(x_0 \leftarrow x_1 \leftarrow x_2)\, G(x_1 \leftrightarrow x_2)\, f_r(x_1 \leftarrow x_2 \leftarrow x_3)}\right)$$

and $w_2(\bar{x}) = 1 - w_1(\bar{x})$.

To solve the final indirect illumination we can therefore split the integral into the clamped part, I_j^1, and the compensation part, I_j^2:

$$I_j^1 = \int_{\Omega_{ind}} w_1(\bar{x}) f_j(\bar{x}) d\mu(\bar{x}) \quad I_j^2 = \int_{\Omega_{ind}} w_2(\bar{x}) f_j(\bar{x}) d\mu(\bar{x}) \tag{5.3}$$

We can compute I_j^1 by converting it to a dense, low-rank global light matrix $\mathbf{A}_g$ and applying any suitable many-light algorithm. For our implementation, we use the method of *visibility clustering* (Section 5.3). To handle the compensation I_j^2, Section 5.4 introduces novel local lights to convert the problem into a sparse, high-rank local matrix $\mathbf{A}_l$. Figure 5.2 illustrates our system and Algorithm GLL gives a pseudo-code.

5.3 Visibility Clustering for Global Lights

In this section we introduce a data-driven visibility clustering algorithm that improves upon the matrix row-column sampling technique of Hašan et al. [2007], especially in cases where many column samples (and shadow map computations) are desired. Shadow map computations tend to be more expensive than shading, but shading is what needs to be sampled more densely for highly glossy materials. Therefore, we propose to partition the global lights into c clusters, render a single representative shadow map to approximate the *visibility* in each cluster, and combine that with the *shading* from *all* lights in the cluster.

Similar to [Hašan et al. 2007], we first sample a small subset of the rows of $\mathbf{A}_g$, resulting in the reduced matrix $\mathbf{R}$. We then optimize the clustering on $\mathbf{R}$ and use the clustering for the full columns of $\mathbf{A}_g$. Due to the low-rank nature of $\mathbf{A}_g$, this produces good results.

Clustering objective. Each element of $\mathbf{R}$ expresses the contribution of a global light to a selected pixel and each such contribution is the product of visibility and shading. Therefore, we can decouple the matrix into the visibility and shading components, $\mathbf{R} = \mathbf{V} \odot \mathbf{S}$, where $\odot$ denotes element-wise matrix multiplication. We will denote the columns of $\mathbf{V}$ and $\mathbf{S}$ as v_i and s_i. The key step in our algorithm is finding a clustering of the columns of $\mathbf{R}$ together with associated representatives from $\mathbf{V}$, such that the error of approximating the visibility within each cluster by the representative visibility is minimized.

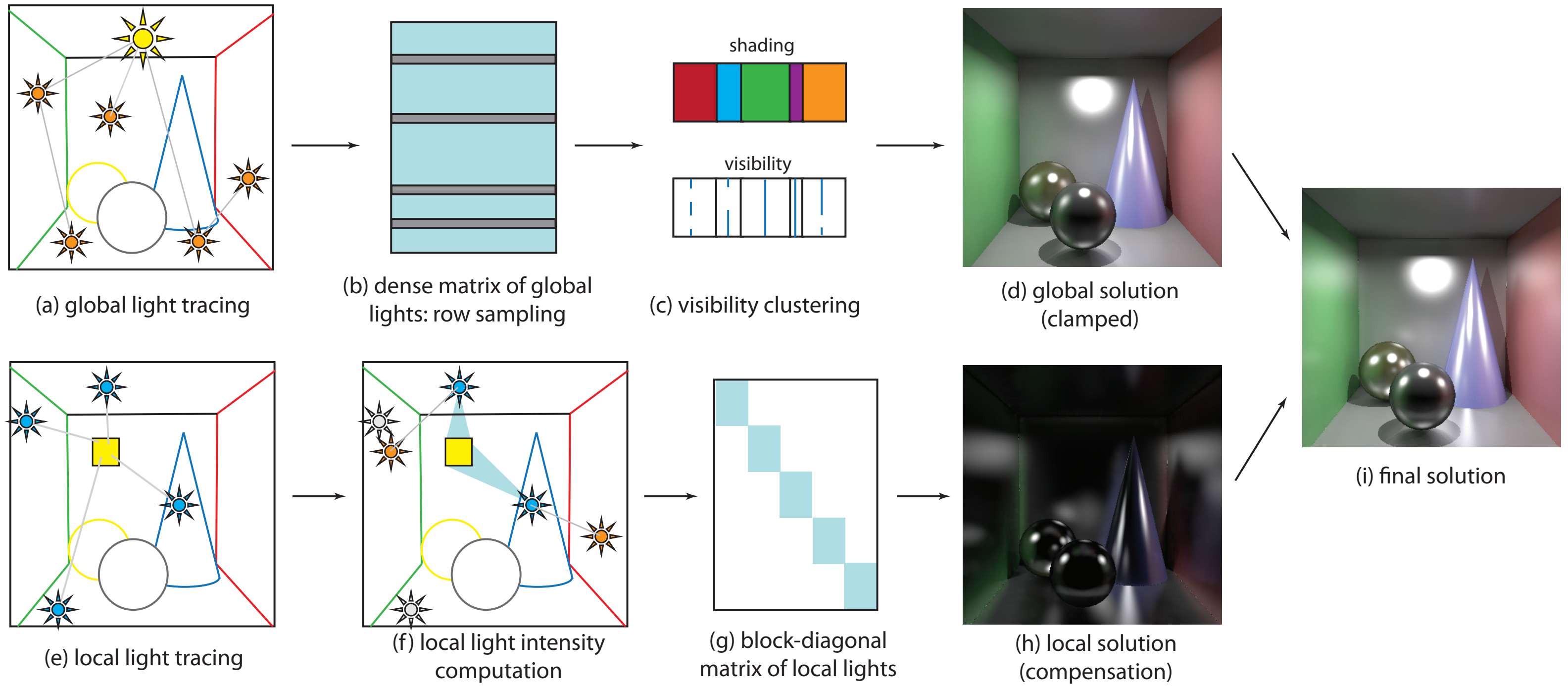

shading
visibility
(a) global light tracing
(b) dense matrix of global lights: row sampling
(c) visibility clustering
(d) global solution (clamped)
(e) local light tracing
(f) local light intensity computation
(g) block-diagonal matrix of local lights
(h) local solution (compensation)
(i) final solution

Figure 5.2: *Conceptual overview of our algorithm.* Standard (global) virtual point lights are created by particle tracing (a) and a dense, clamped lighting matrix is assembled and row-sampled as in Hašan et al. [2007] (b). The reduced row matrix is separated into shading and visibility matrices; a clustering is found, with a binary visibility representative for every cluster (c). The shading is accumulated using the visibility representatives (d). Local lights are traced from image tiles (e) (here shown for one tile), and their intensities are computed by connection to global lights and probability density summation (f). This defines a sparse matrix of local light contributions (g) and is used to compute the clamping compensation (h). Adding (d) and (h) produces the final result (i).

Assume that a clustering $\mathcal{C} = C_1, \ldots, C_c$ is given, together with representative indices $r_1, \ldots, r_c$ in each cluster. We define the cost of a cluster as the error (in L_2-norm) incurred with the *optimal* representative. A light's *distance to the representative* can be defined as the error incurred by using the representative's visibility rather than the lights's own, i.e., $d_{i,r_p} = \|s_i \odot (v_i - v_{r_p})\|$; note that such a distance measure is non-symmetric. To find the optimal clustering, we want to minimize the sum of squared distances from each light to its representative. More formally, the cost of a clustering can be expressed as:

$$cost(\mathcal{C}) = \sum_{p=1}^{c} cost(C_p) = \sum_{p=1}^{c} \sum_{i \in C_p} \|s_i \odot (v_i - v_{r_p})\|^2. \tag{5.4}$$

This clustering problem is related to the k-means and k-median problems, but with a different, non-symmetric distance measure.

Clustering algorithm. We use a hierarchical partitioning approach: starting with all lights in a single cluster, we keep splitting the cluster with the currently largest cost until the desired number of clusters is reached. To split a cluster into two, we choose the column that is farthest from the current representative as a second representative and iterate the following two steps, each of which is guaranteed not to increase the objective function:
1. Create two partitions by assigning each column to the closer of the two representatives based on the non-symmetric distance d_{i,r_p}
2. Pick an optimal representative for the two partitions. (A representative of a cluster is optimal if it minimizes the cluster cost.)

We could iterate until convergence, but we found that 1-2 iterations are sufficient to find a good cluster split. To pick the optimal representative efficiently, we leverage the fact that visibility is a binary function. For each row of the cluster submatrix, we compute two numbers: The errors incurred by approximating the visibility in the row by 0 and by 1; this can be done in a linear pass. In a second linear pass, we evaluate the error of any representative by picking and summing the appropriate error values for each row.

Due to the representative refinement step, the hierarchical partitioning strategy is slowest in the beginning, when spliting very large clusters. We improve performance by using the position of the global lights for the first 16 splits, roughly doubling the clustering speed with no negative impact. We also parallelize the hierarchical splitting algorithm, by keeping a thread-safe priority queue of the current clusters and using multiple worker threads for splitting.

Final rendering. Once the clustering has been found, we render the global lights by iterating over the visibility clusters. For each cluster, we compute the shadow map for the cluster's representative, query it to produce a *shadow mask* that specifies a visibility value for each pixel, and accumulate the shading for each visible pixel from all lights in the cluster (unlike row-column sampling, which uses both visibility and shading only from the representative).

5.4 Local Lights

Recall that we want to find a technique to compute the component I_j^2 (defined in Equation 5.3) missing (clamped away) from the global solution. Intuitively, as the global solution leads to a dense, low-rank matrix, one would expect that the compensation problem can be formulated as a high-rank, sparse matrix and that an algorithm should exist that takes advantage of this particular structure. Note that this algorithm has to be a complete global illumination solution (we could set $w_1 = 0$ and $w_2 = 1$, leaving all work to the compensation), but ideally we would like it to be well adapted to handling exactly the localized illumination effects that remain in I_j^2. We have found such a technique, based on the concept of *local lights*.

Derivation of local lights. Consider a simple gathering algorithm that would compute the compensation I_j^2 by tracing r rays using BRDF importance sampling at point x_1, connecting each hitpoint x_2 to a single global light g by importance sampling according to power, and scaling the contribution by the clamping compensation weight $w_2(\bar{x})$. The global lights already handle multiple indirect bounces; therefore, so does this gathering algorithm. This is a variation of the Kollig-Keller approach [2004] and also bears similarity to our Light Vertex Cache BPT, introduced in Section 4.4.3.

Note that the ray hitpoints x_2 could be thought of as "local lights" that contribute their illumination to only a single pixel: The one they were sampled from. This thought experiment suggests the wastefulness of the technique; would it not be better to contribute the illumination to neighboring pixels as well, thus amortizing the effort in creating the local light?

We consider a small image tile of size $t \times t$, where $t = 32$ is a typical value, and we let all the local lights that originate from this tile contribute to all pixels in the tile. The key challenge is to define the contribution of such a local light at position x_2 to any tile pixel x_1. Using Equation (5.1), all we need is to define the contribution and density of the imaginary path $\bar{x} = x_0 x_1 x_2 x_g$, where x_g is the position of the global light g. We find that this contribution will be:

$$\frac{w_2(\bar{x}) f_r(x_0 \leftarrow x_1 \leftarrow x_2) G(x_1 \leftrightarrow x_2) f_r(x_1 \leftarrow x_2 \leftarrow x_g) E_g(x_2)}{\rho(x_2) p_g},$$

where $E_g(x_2)$ is the irradiance due to the global light g at point x_2, p_g is the discrete probability of choosing g out of all global lights and $\rho(x_2)$ is the density of local lights generated from the current tile, computed as the aggregate density of generating a local light at x_2 by BRDF sampling from all pixels in the tile:

$$\rho(x_2) = \sum_{x_1 \in \text{tile}} \rho(x_0 \to x_1 \to x_2).$$

The density $\rho(x_0 \to x_1 \to x_2)$ depends on the exact distribution used for BRDF importance sampling (which may or may not precisely match the BRDF) and on the number of samples taken.

Visibility. We ignore visibility computation in the above equations, both in the geometry term $G(x_1 \leftrightarrow x_2)$ and in the density term $\rho(x_0 \to x_1 \to x_2)$. Note that x_2 is necessarily visible from the point x_1 where the local light originated, and other points x'_1 in the tile will usually also be visible from x_2, especially over short distances (and over long distances we often have $w_2 = 0$ anyway). This assumption causes very few problems and allows for an efficient computation of local lights on the GPU. Note that a local light can sometimes contribute to a part of the tile that is quite distant from the pixel where it originated, though in practice the compensation weight w_2 will often be very low or zero in such situations, and we have not observed problems caused by this.

Rejection. Local lights often end up in areas that have not been clamped (where $w_1 = 1$), and will have $w_2 = 0$ and therefore zero contribution. However, checking if w_2 is indeed zero over the whole tile would be expensive; we instead use the following heuristic: If w_2 would have been zero for the generating pixel even if the clamping constant c was halved, the light is rejected. Note that this does not introduce bias, but care must be taken to correctly define $\rho(x_0 \to x_1 \to x_2)$; it is the density of lights created at x_2 by sampling from x_1 that would *not* have been rejected.

Reusing samples traced from the camera has been explored in [Segovia et al. 2006] and [Bekaert et al. 2002]; our local lights are distinguished by several features, including their use only for the compensation I_j^2, the possibility of rejection in areas where compensation is not necessary, and their locality and visibility approximation that allow for efficient GPU implementation.

5.5 Implementation Details

This section lists some implementation improvements to the performance and image quality of the algorithm.

Local light clamping. In scenes with glossy surfaces very close to each other, the occasional "spikes" (singularities) can still appear even with local lights. To prevent the resulting artifacts, a small amount of clamping can be applied to the contribution of a local light to pixels and/or the connection to the global light. Connection clamping can be compensated

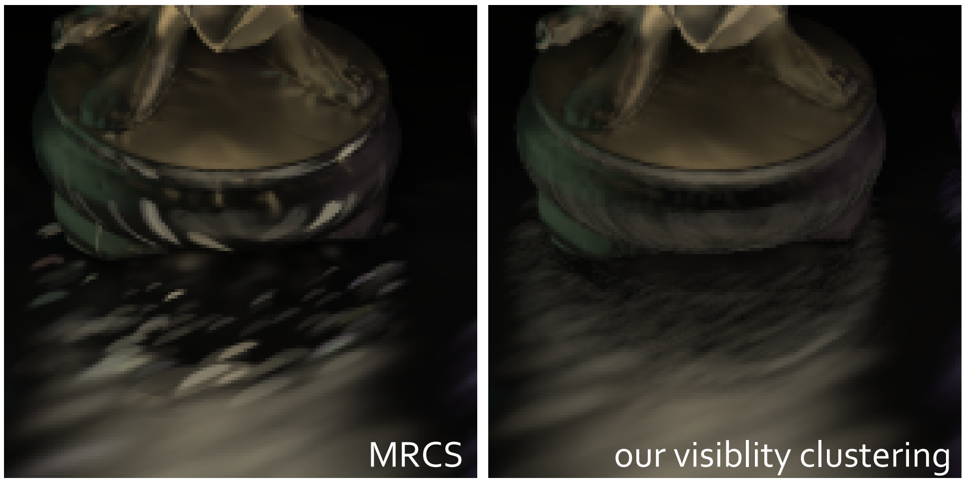


Figure 5.3: *Global light methods.* Comparison of matrix-row column sampling (left: 202 seconds) and visibility clustering (right: 213 seconds). Both methods use 200,000 VPLs and compute only indirect illumination. MRCS selects 10k lights for shadow map evaluation and shading, while visibility clustering uses only 5k shadow maps and shades from all VPLs, yielding less splotchy images.

by the Kollig-Keller technique [2004], applied to the local-global connection. We use this technique in the Tableau, Disney, and Kitchen 2 scenes, using 3 – 10 compensation rays per local light. This removes the bias introduced by connection clamping at the expense of additional ray tracing.

Jittered tiling. Subdividing the image into fixed tiles has the disadvantage that tile boundaries can be perceptible. We overcome this problem by randomly choosing, for each local light, a corresponding tile that contains the pixel where the local light originated. Furthermore, we can also weight the contributions to tile pixels by any kernel that preserves energy; we use a linear ramp on the tile edges to render them even less perceptible. These techniques improve the rendering quality while not introducing bias into the result. The only bias in the local light method comes from ignoring visibility and their optional clamping.

Antialiasing. For the local lights, as well as for larger visibility clusters, we employ interleaved sampling antialiasing, where each light contributes to only a single subpixel within a pixel.

5.6 Results

In this section we first provide insight into our methods for each transport matrix component individually. Next, we show how the strengths of the two methods complement each other, resulting in fast generation of high quality images. We then show results for four scenes, each with its glossy-glossy interaction challenges, and compare them with path tracing and the virtual spherical light (VSL) technique from [Hašan et al. 2009]. The image sizes are 800×600 and use 3×3 anti-aliasing with interleaved sampling. All our measurements are done on a system with two Intel Xeon X5560 processors (with 4 cores each), 8 GB RAM, and an NVIDIA GTX 480 graphics card. Path traced images were generated on a cluster of 16 nodes, where each node was a dual Xeon 2.83 GHz (4-core) machine. Reported times for path traced images are the time using only one node of the cluster (computed by adding the times on all cluster nodes together).

Figure 5.3 shows a detail of our **Tableau** scene (for the full image see Figure 5.7) rendered with matrix row column sampling (MRCS) and our method, both rendered in approximately the same time. It clearly demonstrates the extra image quality our method is able to achieve in scenes with curved glossy surfaces because we are able to use more shading samples, even though we use fewer shadow maps for visibility. However, we note that visibility clustering and MRCS can be used together, if the difference between the ranks of the visibility and shading matrices is low. In such a setup the visibility clusters are used as an initial clustering for MRCS, which can then further refine them into the desired number of shading clusters. However, we do not use this approach, because it brings almost negligible speedup when the method is paired with local light rendering, since local light generation and rendering accounts for a large (approximately 50%) fraction of total rendering time.

The sparse high-rank component of the transport matrix is handled by the newly proposed local light algorithm. While this algorithm excels in capturing local glossy interactions, Figure 5.4 shows the limitations of the method as a stand-alone solution. Due to the local visibility approximation, all shadows smaller than the tile size are effectively smeared away (compare with Figure 5.5).

While global methods lack local glossy effects and local lights lack fine shadows, combining both we achieve the desired high quality images.

	Kitchen 1	Tableau	Disney	Kitchen 2
# global lights	300k	200k	200k	100k
# vis. clusters	10k	5k	15k	10k
# local lights	17.1M	55.6M	13.5M	25.1M
Row render	18.6 s	22.6 s	11.3 s	15.4 s
Vis. clustering	19.0 s	13.0 s	28.6 s	12.5 s
Global render	249.6 s	145.6 s	90.8 s	170.5 s
Local render	40.6 s	162.1 s	33.0 s	57.5 s
Total	**327.8 s**	**343.3 s**	**163.7 s**	**255.9 s**

Table 5.1: *Timings and statistics.* Number of lights and visibility clusters (top) and breakdown of the time spent on different parts of the algorithm (bottom).

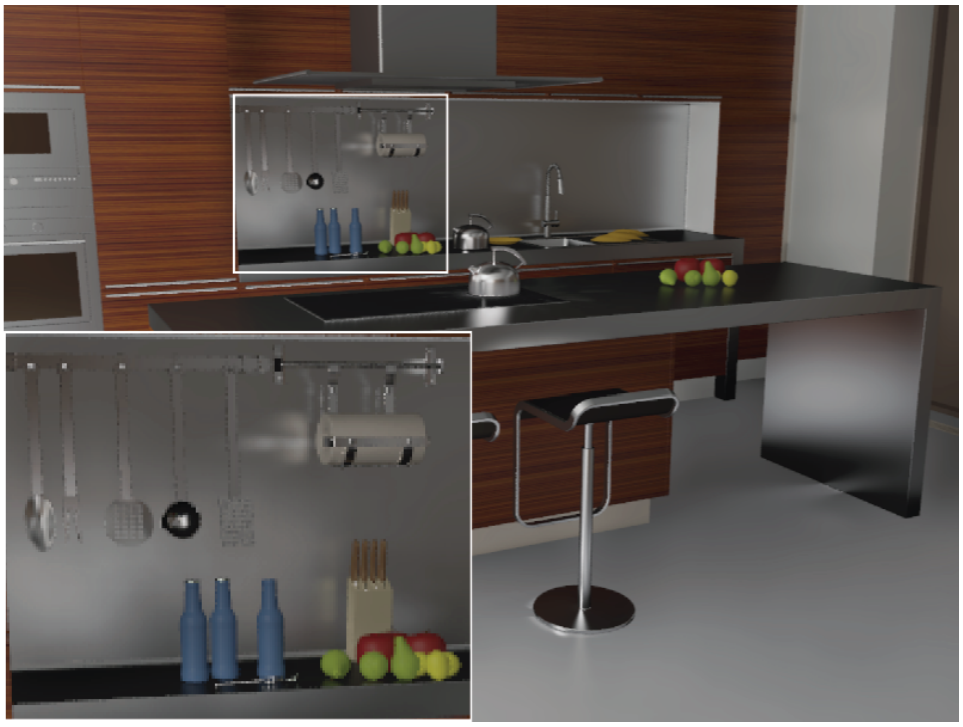


Figure 5.4: *Local lights only.* We observe that when only local lights are used, fine shadows may disappear (e.g., there are no shadows behind the bottles).

Figure 5.5 shows a typical glossy scene (**Kitchen 1**) separated into its global and local components. Global lights provide the basic illumination and shadows, but the far wall appears to be diffuse. Local lights correct this incorrect perception of material properties by providing glossy reflections of the paper towels and faucet, and of course all the other missing local interactions. These two components are combined for the final result image. Note that all three images show indirect illumination only.

Figure 5.6 shows the **Kitchen 1** scene (253,433 triangles) rendered using our method, path tracing, and VSLs for comparison. We also show insets of areas where our technique is able to accurately compute reflections that contribute to material perception (where VSLs miss those features). In fact, VSLs give results similar to the solution using only global lights in Figure 5.5. The VSL approach misses reflections of the towel rack, faucet, and the yellow plates on the back wall, the bar stool rods on the base of the stools, and other reflections on the counter. The two color-coded error images show that the overall light distribution of our method matches the reference more closely. However, we can notice that the illumination at the right side of the glossy wall is not perfectly even. This is result of slight noise in the connection strategy for local lights.

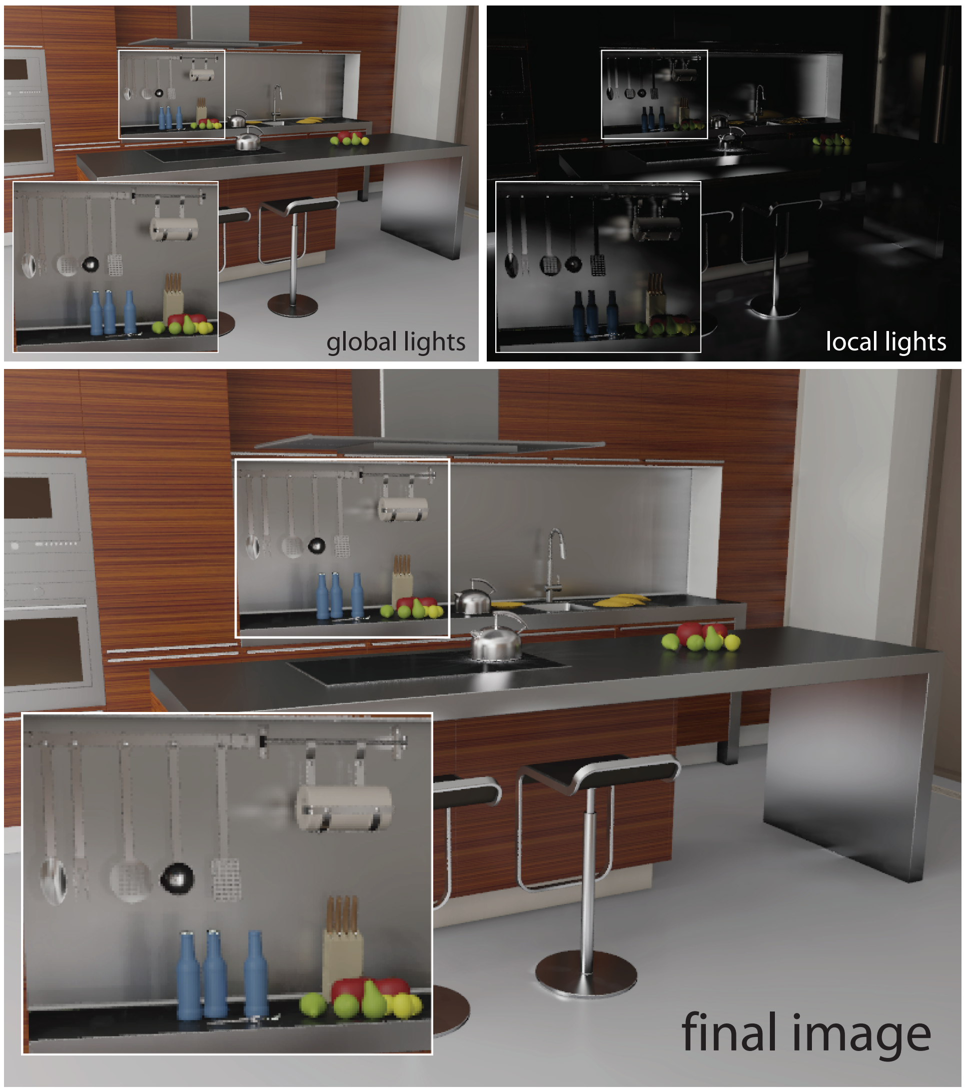


Figure 5.5: *Component separation.* Left: the long distance effects captured by global lights: Notice the shadows behind the bottles on the counter. Center: We show the local glossy-glossy interaction, mainly between the back wall and the towel rack and faucet just in front of it. Right: combination of both components, resulting in an image with both local and global effects. (The images show indirect illumination only.)

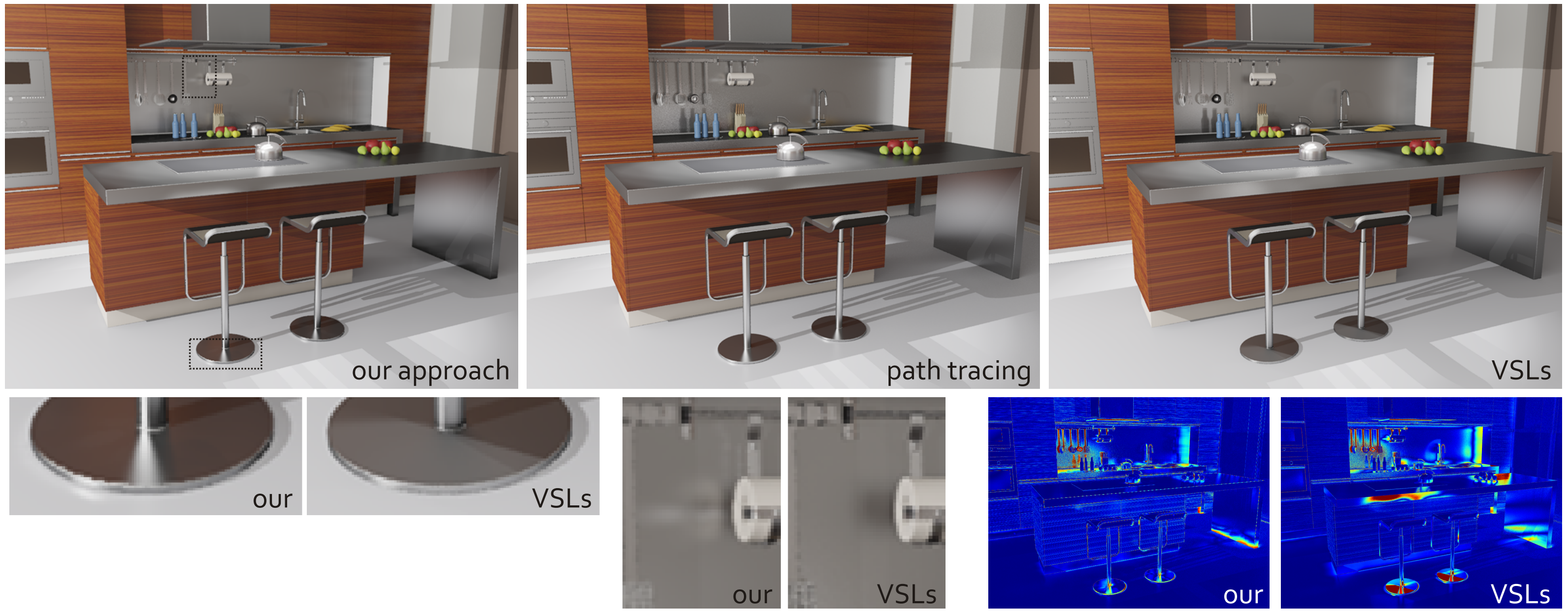


Figure 5.6: *Kitchen 1.* Top-right: Our approach (5 min 28 sec). Top-middle: Reference path traced solution (still noisy, 106 hours). Top-right: Virtual Spherical Lights (VSLs), after 6 min 25 sec. Insets in the bottom row show some of the glossy interactions captured by our method that are missing from the VSL rendering. Bottom-right: Color-coded relative error images of our method and VSL against the reference solution.

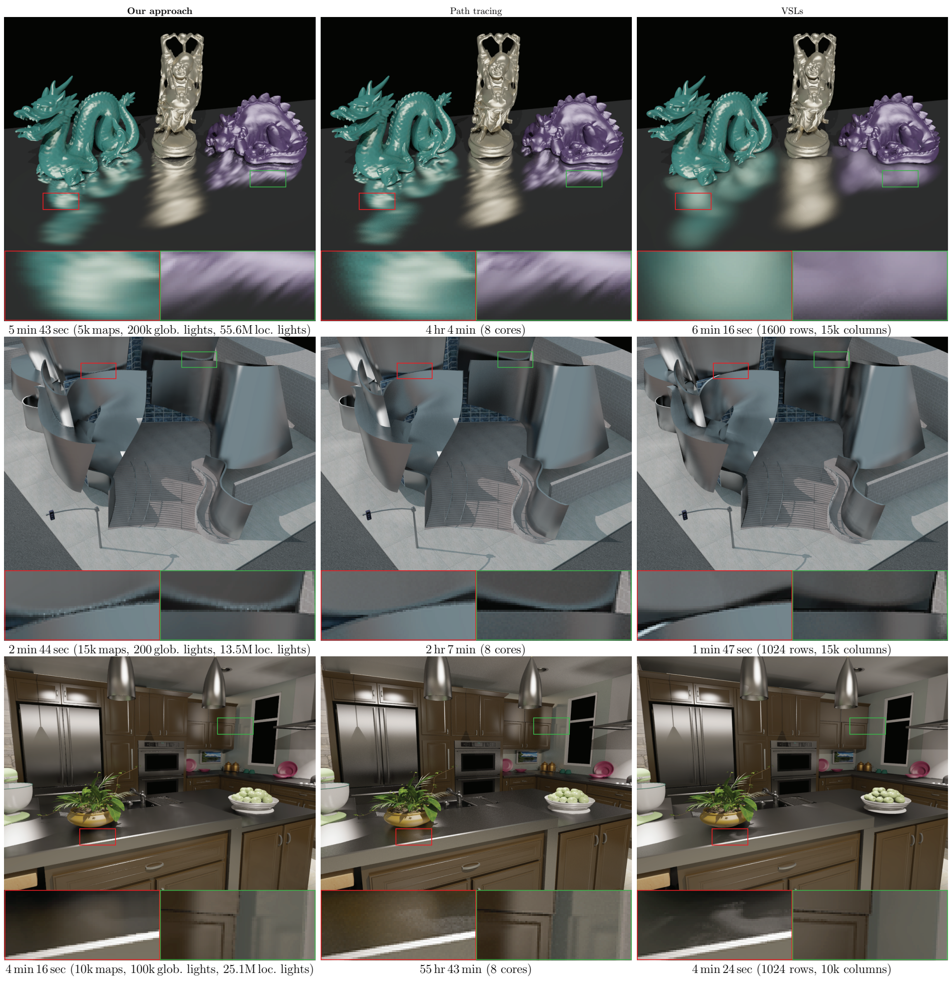

Our approach
Path tracing
VSLs
5 min 43 sec (5k maps, 200k glob. lights, 55.6M loc. lights)
4 hr 4 min (8 cores)
6 min 16 sec (1600 rows, 15k columns)
2 min 44 sec (15k maps, 200 glob. lights, 13.5M loc. lights)
2 hr 7 min (8 cores)
1 min 47 sec (1024 rows, 15k columns)
4 min 16 sec (10k maps, 100k glob. lights, 25.1M loc. lights)
55 hr 43 min (8 cores)
4 min 24 sec (1024 rows, 10k columns)

Figure 5.7: *Results: Tableau*, *Disney*, *Kitchen 2* from Hašan et al. [2009] and comparisons with path tracing and VSLs. Note that the Tableau scene has higher gloss than in the original paper. Corresponding times, shadow map numbers, and number of global and local lights are also reported. Insets show where our technique captures lighting features that match the path tracer, but are missing in the VSL solutions.

Figure 5.7 shows the **Tableau**, **Disney**, **Kitchen 2** scenes from [Hašan et al. 2009]. Note that we have increased the gloss of the floor in **Tableau** (compared to the original paper) to demonstrate our ability to render high gloss accurately. We also provide comparison with Progressive Photon Mapping [Hachisuka et al. 2008] and Stochastic Progressive Photon Mapping [Hachisuka and Jensen 2009] as supplemental material in Appendix B.

In **Tableau**, our technique is able to capture both the long-range and local reflections when compared to VSLs. In fact, the VSL reflections are quite blurry, and can cause incorrect material perception [Ramanarayanan et al. 2007, Křivánek et al. 2010]. In comparison with the path tracer, our solution in this difficult scene is quite accurate. In **Disney** our algorithm is able to capture the two blue caustics (see insets) and the shape of highlights accurately when compared with the VSL approach. In **Kitchen 2** our technique accurately computes reflections of the cupboards in the back. However, while it captures reflection of the pot in the front, this reflection is darker. Similar darkening is also perceivable in **Disney** on the left wall. Both are caused by application of slight clamping on local light contribution.

Table 5.1 gives a detailed breakdown of individual stages of our algorithm. To fully utilize the computing power, we run the local light generation on the CPU concurrently with the global and local light rendering on the GPU. The local rendering time therefore also includes the time required to finish local light generation.

Antialiasing. Our interleaved antialiasing scheme enables the efficient computation of antialiased images with less than a 2$\times$ cost in performance. For example, for **Kitchen 1**, the regular image takes approximately 130 seconds, while the 3$\times$3 anti-aliased image takes approximately 330 seconds to compute. All results presented are for antialiased images.

5.7 Conclusion

In this chapter we present a component-based rendering algorithm for rendering of highly glossy materials with global illumination. Our approach is to split the light transport into two parts: A global, low-rank component and a sparse, localized, high-rank component. We approximate the low-rank component using a novel visibility clustering algorithm and the high-rank component by using local virtual lights to compensate for lost energy due to clamping. Both CPU and GPU are used in solving both local and global component, coupling novel approximations of light transport components with highly efficient utilization of all hardware resources at our disposal. Our solution is suitable for previewing in industrial design applications where materials like metals and plastics are common and should be reproduced with high fidelity.

While this approach expands the range of materials that can be rendered accurately, it has a few limitations. Local VPLs still need clamping when there are highly glossy or close-range interactions. Fine shadows can become dull due to the visibility approximation in the local lights component. The number of local VPLs required is highly scene dependent. A progressive algorithm could be designed for the local component. Future work could explore automated selection of parameters and clamping.

Chapter 6

Conclusion

It is the author's strong belief that the growing demand for physically-based graphics that has emerged in the past decades will not cease in the foreseeable future. This can be witnessed in the movie industry where, despite the immense advances in rendering algorithms and hardware capabilities, the average frame render times have stayed the same, if not increased. Given that the single-threaded performance of the modern CPUs became more or less stagnant, it is imperative that graphics research examines not only new algorithms, but also keeps evaluating a wide variety of more specialized hardware options, be it full dedicated hardware accelerators or just instruction set augmentations.

In this thesis we focused on these hardware options on three levels of granularity, evaluating a dedicated hardware accelerator of ray casting as the basic component of all physically based algorithms, investigating GPU mappings of many such algorithms onto state-of-the-art massively parallel architectures, and proposing a novel algorithm for handling glossy interreflections that utilizes both CPU and GPU to achieve full potential of available hardware. In this final chapter, we provide a summary of the major contributions, discuss possible future works, and present closing thoughts.

Ray Traversal Engine

Despite ray casting being the basic element of almost all physically-based rendering algorithms used in practice, its hardware acceleration is only now starting to appear in consumer products, for example, in Imagination Technologies' PowerVR processors. This puts these algorithms at a distinct disadvantage when compared to rasterization, which can rely on extremely efficient rasterization units tightly coupled with general purpose shading processors of the modern GPU.

To address the issue, Chapter 3 introduced a hardware implementation of ray traversal engine (RTE) that could be used in the place of traditional rasterization units. Using a 90 nm process we could synthesize the RTE to fit into 15 mm^2 area (the area of Cell's SPU unit), run at more than 2 GHz, delivering 100 million rays per second while keeping the required bandwidth below 5 $\mathrm{GB\,s^{-1}}$. We believe that using a more advanced, smaller, process would increase the performance even further, without the bandwidth requirements going above the capabilities of modern memory systems.

To achieve this performance, we evaluated several configurations of the RTE, different acceleration structures, and different shader models. We conclude that best performance is achieved when the acceleration structure uses treelets to sort ray access to the nodes of the structure, paired with tail recursive shaders that are suited to providing a large number of rays that can be sorted, and with a large L2 cache to mitigate memory access incoherence that the ray sorting did not prevent.

Global Illumination on GPU

Having a hardware ray tracing acceleration on a GPU would be of little use, if we could not also provide efficient algorithms that utilize it. Chapter 4 presents several such light transport algorithms and their mapping onto GPU. It evaluates both existing and novel approaches of mapping these algorithms onto two generations of the NVIDIA GPUs and draws conclusions on the best ways of implementing each of the algorithms.

The described algorithms follow in logical order, each building on the conclusions of the previous algorithms. The first introduced algorithm is Path Tracing, as tracing a single camera (or light) path is the basic prerequisite of all the other algorithms. This is followed by Bidirectional Path Tracing, which brings the added complexity of storing the paths instead of just tracing them and storing the results. Here we introduced our Light Vertex Cache, where we took inspiration from implementation of Virtual Point Lights and applied it to the Bidirectional Path Tracing to introduce the fastest and simplest implementation of the algorithm.

Next we looked at photon lookup acceleration structures, as approaches built on Photon Mapping require not only storing the light path vertices (photons), but also efficient range queries into them. With solutions for all of these algorithms, we could also provide the first GPU implementation of the recently introduced Vertex Connection and Merging algorithm that combines Bidirectional Path Tracing and Progressive Photon Mapping to efficiently render a wider variety of scenes than either of the algorithms on its own.

The chapter concludes with a comparison of all of the introduced algorithms on a set of scenes to help us determine which algorithms are most suitable for which scenes. Here we showed that while Bidirectional Path Tracing and Vertex Connection and Merging are invaluable when rendering complex scenes, simple scenes can still be best served by simple Path Tracing.

Global Local Lights

The previous chapter concluded with an overview of many advanced algorithms over a wide spectrum of scenes. Unfortunately, there are still scenes, such as those containing glossy interreflections, where even the most advanced light transport algorithms struggle to provide the exact solution in a practical timeframe.

The algorithm introduced in Chapter 5 focuses on a different approach to solving this problem, divides the light transport in the scene into a local and global component and uses different approximations for each of the components to provide high quality results in a reasonable time. Both components are solved using Virtual Point Lights (VPLs), but each component uses different method to generate and render its VPLs.

On the global level, the algorithm captures longer distance light transport effects. Solving this components requires several hundred thousand the required number of VPLs and while this is orders of magnitude lower than the number used for the local component, it is still impractical to use all of them. Based on the observation that in the glossy setup the shading component of the illumination changes significantly faster than the visibility component, the algorithm utilizes visibility clustering to allow VPLs to share their visibility using shadow maps. This allowed us to evaluate BRDFs for all VPLs, but visibility for only several thousand of them, resulting in significant speed up over the naive solution at virtually no cost to the image quality.

For the local component, the algorithm generates the VPLs from camera to obtain VPLs relevant to the final image. These VPLs are responsible only for the localized effects not handled by the global component and, as such, their contribution can be limited to only a small screen tile around the pixel they were generated for. This, along with assuming that the visibility does not change inside this tile, allows us to process tens of millions of such VPLs, giving the scene the VPL density required to obtain high quality result.

To achieve the best possible performance, the algorithm leverages both the CPU (VPL tracing, visibility clustering) and the GPU (shadow maps, local VPL contributions) of the machine using each for the task it is best suited for. This allowed us to achieve greater performance than would be possible should we focus solely on either of the two main processing units available to us.

Final Thoughts

In this thesis we presented the idea that while development of new rendering algorithms is important, we should always keep in mind their possible implementation on both the current and future hardware. While previously there was a strong incentive to move from specialized hardware to general purpose processing, this has been slowly reversed in the recent years and many modern processors now include dedicated hardware units for video decoding, encryption, and other specialized tasks. The author believes that ray tracing is another area that would greatly benefit from such a unit. We can already see basic versions of such units emerging in the mobile market, where the specialized hardware brings not only performance but also high energy efficiency.

The main open question is whether the unit should be only for static scenes or also for real-time rendering. This, in turn, opens a questions of the best acceleration structure and of hardware accelerated building of such acceleration structures. The author expects that, despite showing the advantages of hardware ray tracing engine, full adoption is not feasible until these questions are sufficiently answered. He would encourage researchers to investigate in this direction.

However, it is not just a mobile market that is influenced by power consumption considerations. Virtually all modern computers have to consider power consumption and, more importantly, power dissipation in their basic design, as power dissipation is the chief reason why the CPU frequency stopped at around 4 GHz and has not moved for a decade. With the instruction level parallelism almost exhausted, we can no longer depend on single threads going faster but, instead, have to utilize the fact that the processors are going wider. More cores in a package, more threads on a core, two, four, sixteen, or thirty-two identical arithmetic operations driven by a single instruction. The most powerful current processors even cut down on single threaded performance, by removing features like out-of-order instruction issue, and tradeoff programmer's convenience for higher peak performance.

It is therefore essential to keep the hardware architecture in mind when evaluating rendering algorithms, old and new. Even the best single threaded algorithm will eventually become worse than simpler algorithms that can utilize the full power of this hardware evolution. The author believes that rendering on modern hardware architectures is a problem that can never be really solved, as long as new hardware keeps being developed. The author cannot presume in what direction will this lead next, but he strongly believes that when new types of computers, quantum or otherwise, become a common computing platform, rendering algorithms should be, and will be, among the first to explore their capabilities.

Appendix A

Supplemental Material for Light Transport Simulation on the GPU

A.1 Path Tracing

Figure A.1 shows the performance, in rays per second, of our Path Tracing algorithms for each tested scene and both GPUs. The results for GTX 580 show that StreamingPTmk outperforms all algorithms irrespective of the scene. The only scene with behavior not matching the average is CoronaWatch. There our NaivePTsk and RegenerationPTsk clearly outperform RegenerationPTmk at all measured paths per frame, and outperform even StreamingPTmk up until 10^6 paths per frame. This is due to the fact that many paths have length of only one segment (directly hit the area light, or miss all geometry) or two (reflect off the bezel), which greatly increases coherence of the traced rays. On the GTX 680 we, again, see results matching our previous assessments, with the only outlier being, again, the CoronaWatch scene, for the very same reasons as on the GTX 580.

A.2 Bidirectional Path Tracing

Table A.1 shows performance of our Bidirectional Path Tracing (BPT) algorithms not only at 10^6 samples per frame, as in the paper, but also adds 10^7 samples per frame. As in Path Tracing, we can see the increase in performance with more paths per frame. The single-kernel algorithms, i.e., NaiveBPT and LVC-BPTsk, exhibit very little performance gain when given more samples per frame, while the multi-kernel ones gain a boost of up to 39%. Overall we can see that on GTX 580, at 10^7 samples per frame, our LVC-BPTmk has the highest performance on all tested scenes, while on GTX 680, LVC-BPTsk and LVC-BPTmk have a roughly similar performance.

A.3 Algorithm Comparison

Figure A.2 shows the RMSE-vs-time convergence on both tested GPUs. On GTX 580 (top) we used different algorithms for Path Tracing (StreamingPTmk instead of our RegenerationPTsk) and Bidirectional Path Tracing (LVC-BPTmk instead of LVC-BPTsk), but the convergence curves still closely match GTX 680 curves reported in the paper. Figures A.3-A.8 show the final images (after 15 minutes) rendered by each of the methods on both GPUs.

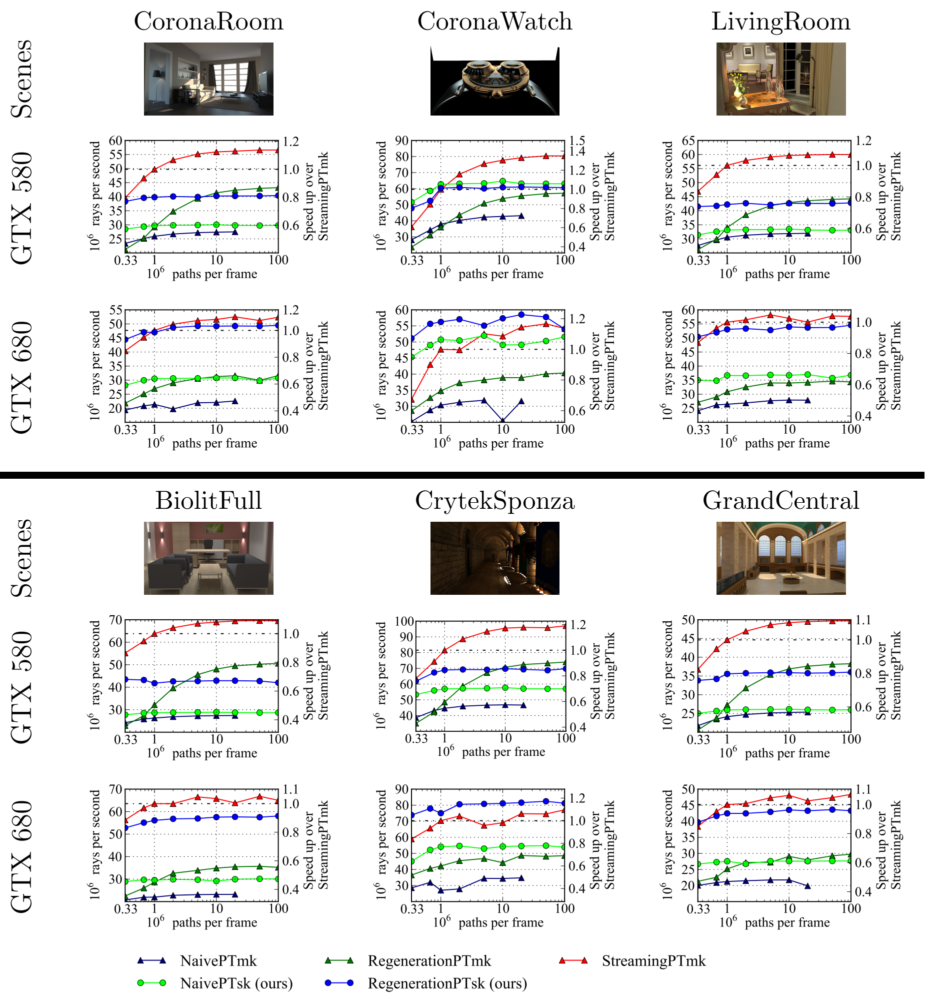


Figure A.1: *Path tracing performance.* Performance in rays per second with increasing number of paths per frame, given for each scene and both GPUs. The right axis shows performance relative to StreamingPTmk at 10^6 paths per frame.

10^6 samples per pass on GeForce GTX 580

	StreamingBPT	NaiveBPT	MultiBPT	**LVC-BPTsk**	**LVC-BPTmk**
CoronaRoom	1.00×	0.91×	0.81×	1.15×	**1.21**×
CoronaWatch	1.00×	**1.41**×	0.93×	**1.41**×	1.15×
BiolitFull	1.00×	0.52×	0.57×	1.53×	**1.81**×
CrytekSponza	1.00×	1.07×	0.93×	**1.35**×	1.29×
GrandCentral	1.00×	0.70×	0.70×	1.29×	**1.34**×
Average	1.00×	0.85×	0.77×	**1.33**×	**1.33**×

10^7 samples per pass on GeForce GTX 580

	StreamingBPT	NaiveBPT	MultiBPT	**LVC-BPTsk**	**LVC-BPTmk**
CoronaRoom	1.18×	0.92×	1.05×	1.18×	**1.55**×
CoronaWatch	1.13×	1.15×	1.00×	1.18×	**1.42**×
BiolitFull	1.02×	0.52×	0.95×	1.53×	**2.07**×
CrytekSponza	1.07×	1.00×	1.05×	1.29×	**1.44**×
GrandCentral	1.10×	0.71×	0.96×	1.28×	**1.50**×
Average	1.17×	0.86×	1.07×	1.36×	**1.66**×

10^6 samples per pass on GeForce GTX 680

	StreamingBPT	NaiveBPT	MultiBPT	**LVC-BPTsk**	**LVC-BPTmk**
CoronaRoom	0.86×	0.72×	0.69×	**1.24**×	1.19×
CoronaWatch	0.74×	1.17×	0.80×	**1.32**×	1.07×
BiolitFull	0.89×	0.42×	0.58×	1.73×	**1.92**×
CrytekSponza	0.84×	0.92×	0.92×	**1.55**×	1.12×
GrandCentral	0.97×	0.55×	0.68×	1.39×	**1.49**×
Average	0.85×	0.68×	0.71×	**1.38**×	1.27×

10^7 samples per pass on GeForce GTX 680

	StreamingBPT	NaiveBPT	MultiBPT	**LVC-BPTsk**	**LVC-BPTmk**
CoronaRoom	0.99×	0.73×	0.87×	1.26×	**1.37**×
CoronaWatch	0.93×	1.18×	1.01×	**1.40**×	1.21×
BiolitFull	0.94×	0.42×	0.96×	1.76×	**2.13**×
CrytekSponza	0.92×	0.92×	1.11×	**1.57**×	1.22×
GrandCentral	1.03×	0.55×	0.91×	1.41×	**1.57**×
Average	0.95×	0.68×	0.96×	**1.42**×	1.41×

Table A.1: *Relative BPT speed up*: Speed up, in the terms of time to a given RMSE, of different BPT algorithms, relative to StreamingBPT with 10^6 samples per pass on GTX 580.

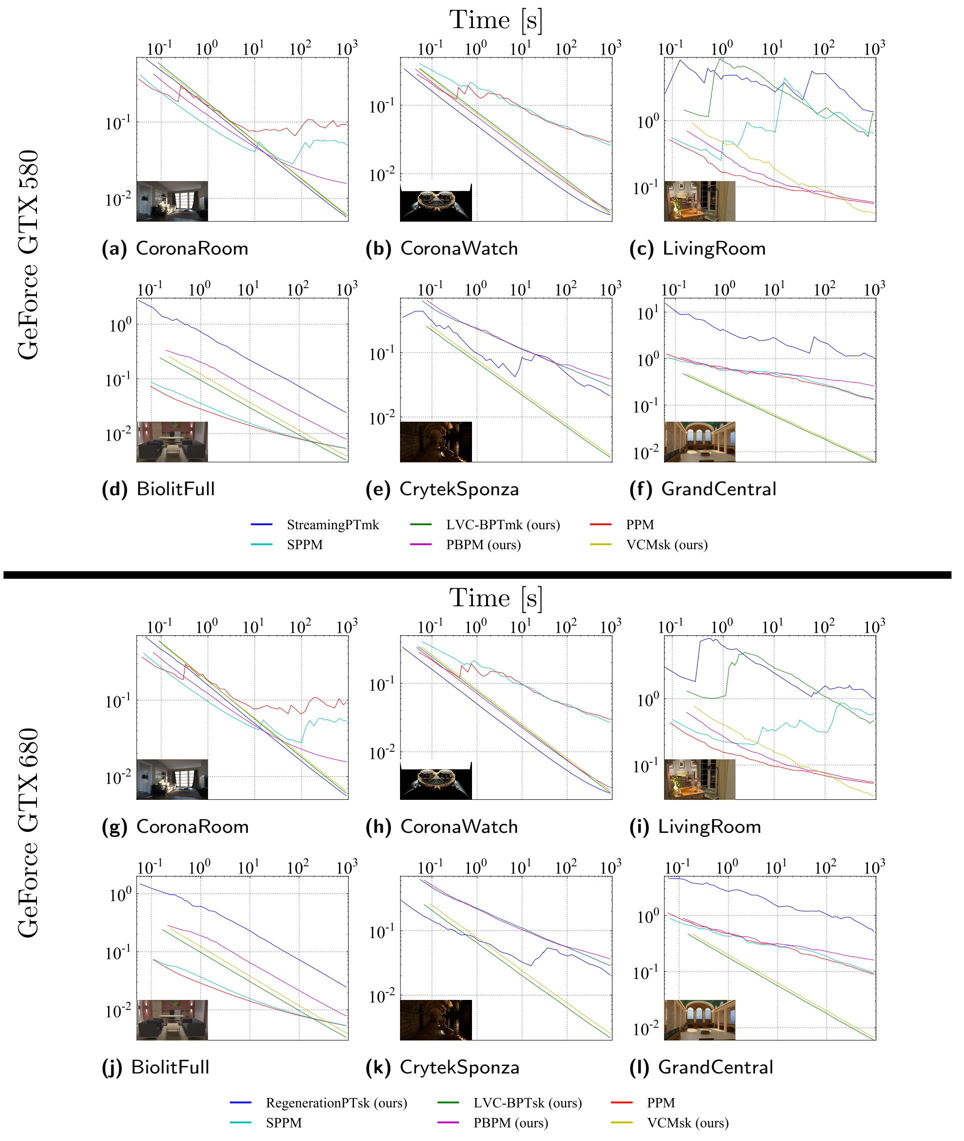
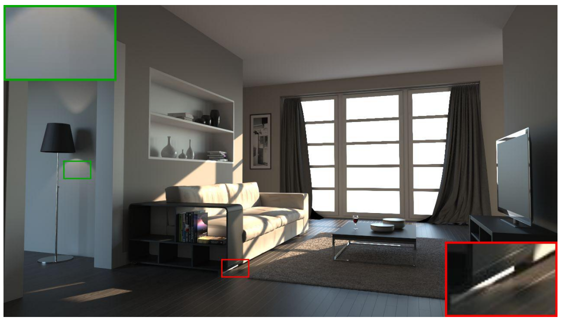


Figure A.2: The log-log plot of the RMSE-vs-time convergence of the six tested methods on each of the test scenes. Top graphs are results from GTX 580, the bottom graphs from GTX 680.



GeForce GTX 580

(a) StreamingPTmk

(b) LVC-BPTmk

(c) PPM

(d) SPPM

(e) PBPM

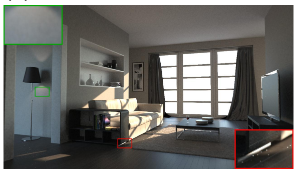
(f) VCMsk

GeForce GTX 680

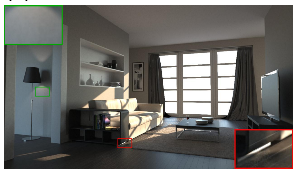
(g) StreamingPTmk

(h) LVC-BPTmk

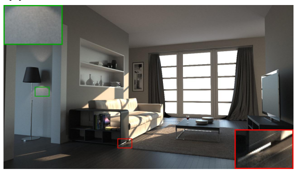
(i) PPM



(j) SPPM



(k) PBPM



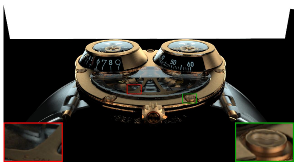
(l) VCMsk

Figure A.3: *CoronaRoom*. All image results of Algorithm Comparison.

GeForce GTX 580

(a) StreamingPTmk

(b) LVC-BPTmk


(c) PPM

(d) SPPM

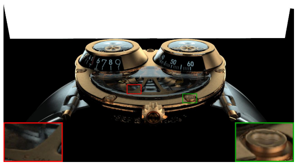
(e) PBPM

(f) VCMsk

GeForce GTX 680

(g) StreamingPTmk

(h) LVC-BPTmk


(i) PPM

(j) SPPM

(k) PBPM

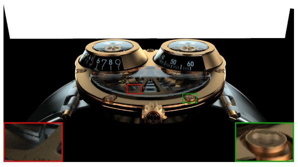

(l) VCMsk

Figure A.4: *CoronaWatch*. All image results of Algorithm Comparison.

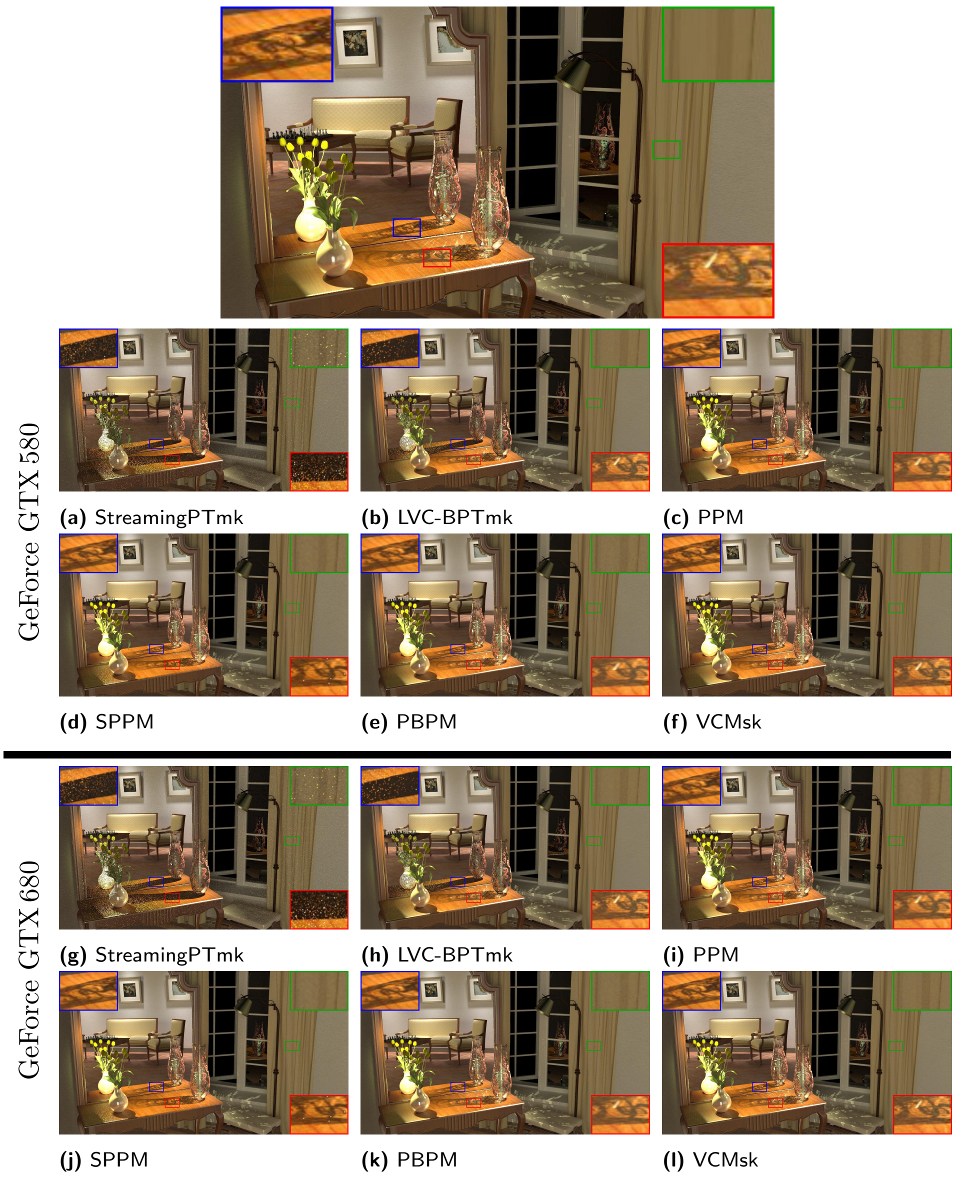


Figure A.5: *LivingRoom*. All image results of Algorithm Comparison.

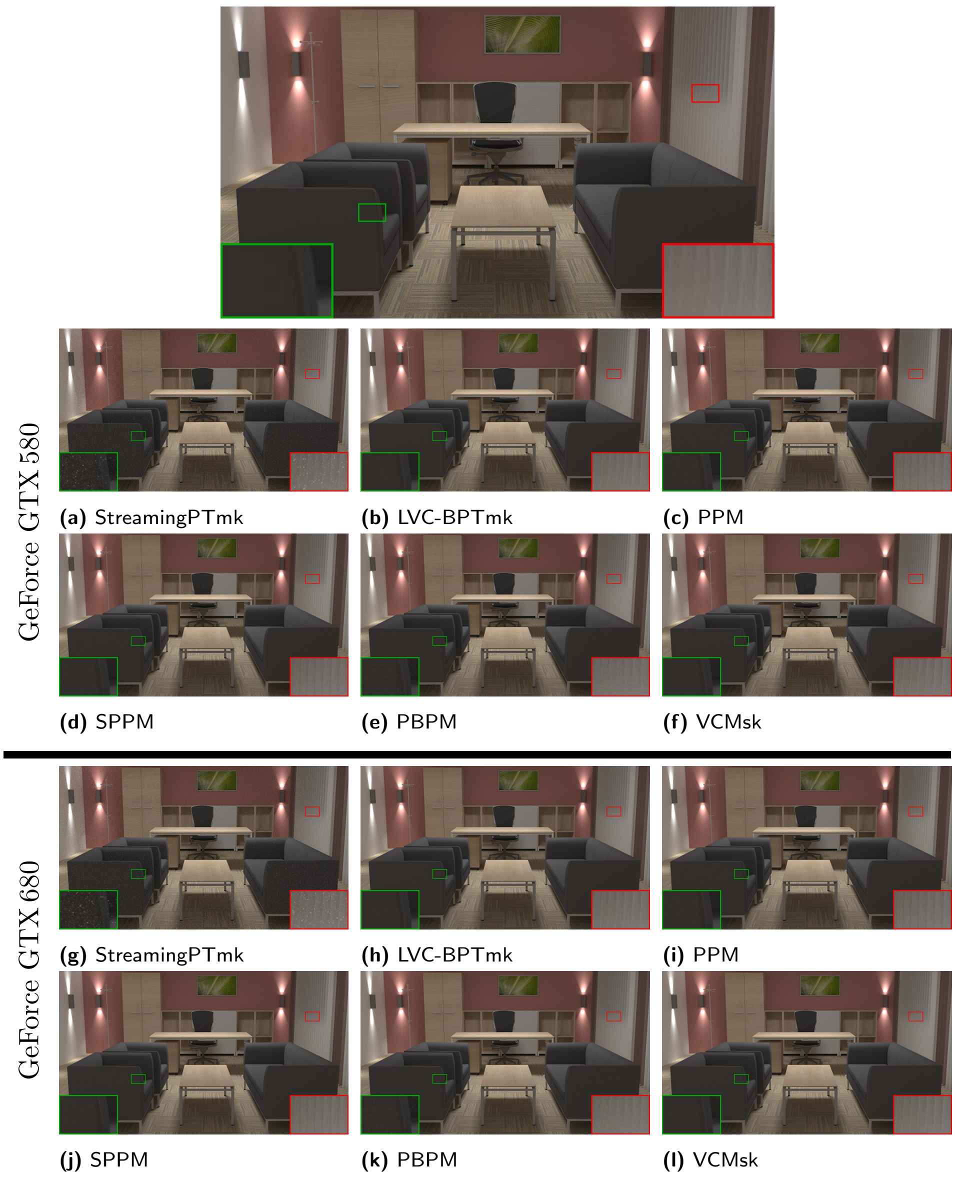
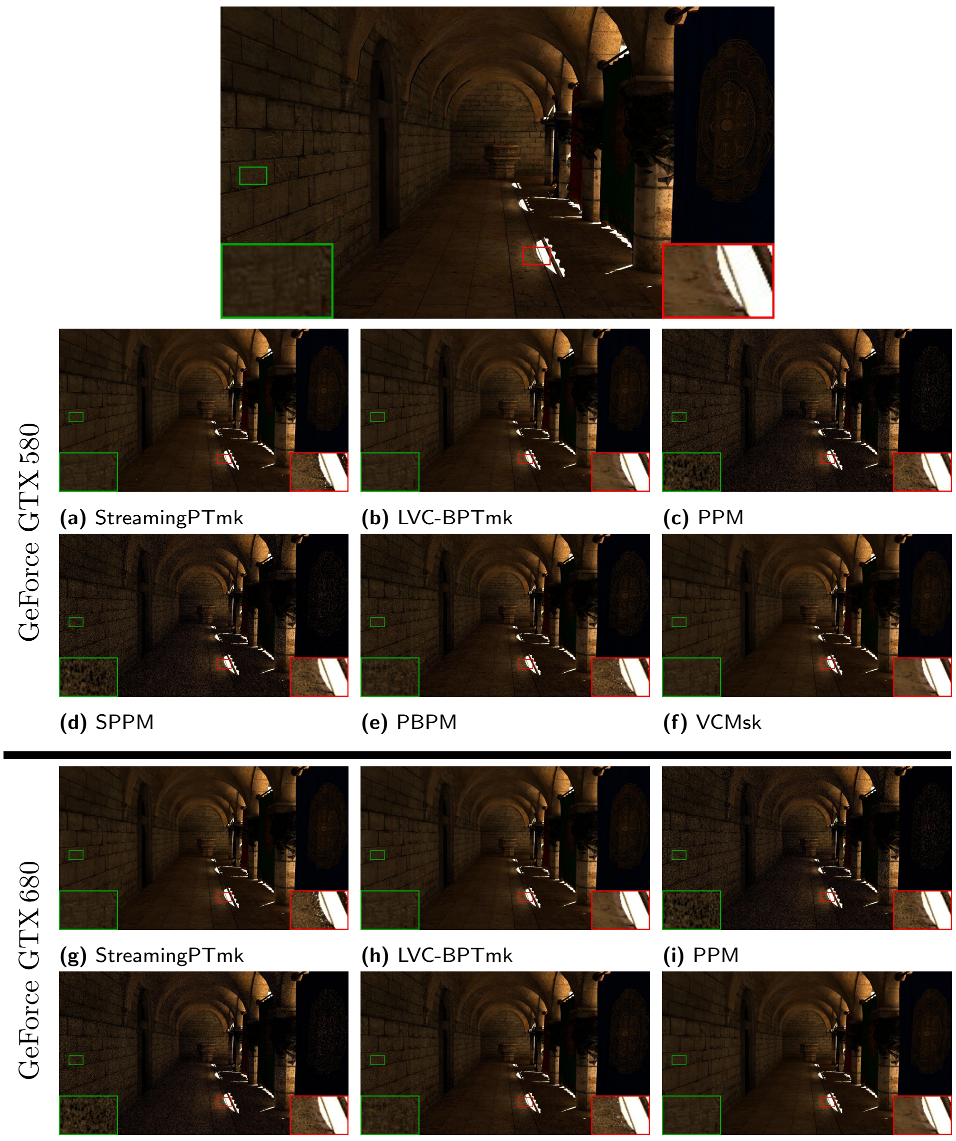


Figure A.6: *BiolitFull*. All image results of Algorithm Comparison.



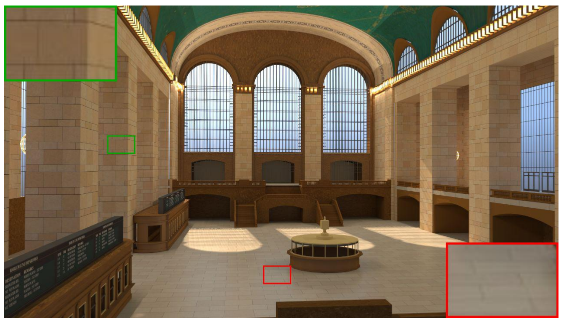
(a) StreamingPTmk **(b)** LVC-BPTmk **(c)** PPM

(d) SPPM **(e)** PBPM **(f)** VCMsk

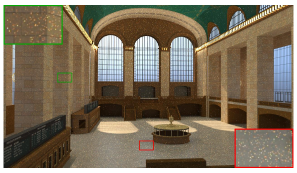
(g) StreamingPTmk **(h)** LVC-BPTmk **(i)** PPM

(j) SPPM **(k)** PBPM **(l)** VCMsk

Figure A.7: *CrytekSponza*. All image results of Algorithm Comparison.

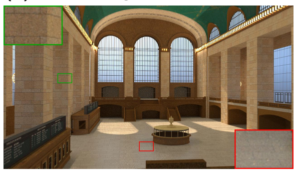


GeForce GTX 580

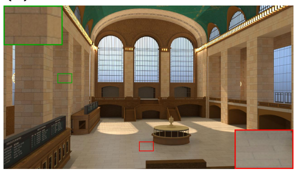

(a) StreamingPTmk

(b) LVC-BPTmk

(c) PPM

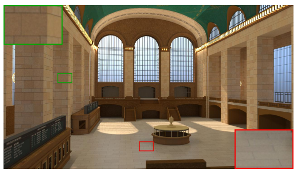

(d) SPPM

(e) PBPM

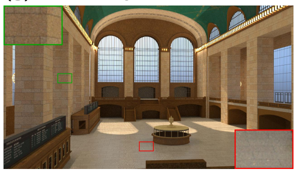

(f) VCMsk

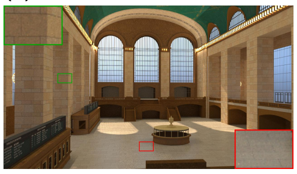
GeForce GTX 680

(g) StreamingPTmk


(h) LVC-BPTmk

(i) PPM


(j) SPPM


(k) PBPM

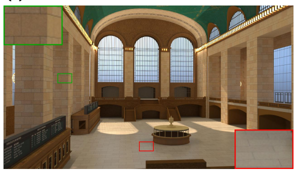

(l) VCMsk

Figure A.8: *GrandCentral*. All image results of Algorithm Comparison.

Appendix B

Supplemental Material for Global and Local VPLs

B.1 Stochastic Progressive Photon Mapping

In Figures B.1 and B.2 we provide comparison of our method with CPU implementation of Progressive Photon Mapping [Hachisuka et al. 2008] and Stochastic Progressive Photon Mapping [Hachisuka and Jensen 2009]. The timings are machine with Intel Xeon X5560 processors (with 4 physical, 8 logical cores each). Both algorithms are implemented as CPU only. To compensated for this, we the reported results are for roughly 5× longer runtime for each method, when compared to ours. We let the algorithms run even after this and confirmed that no significant improvement could be achieved by providing slightly longer time.

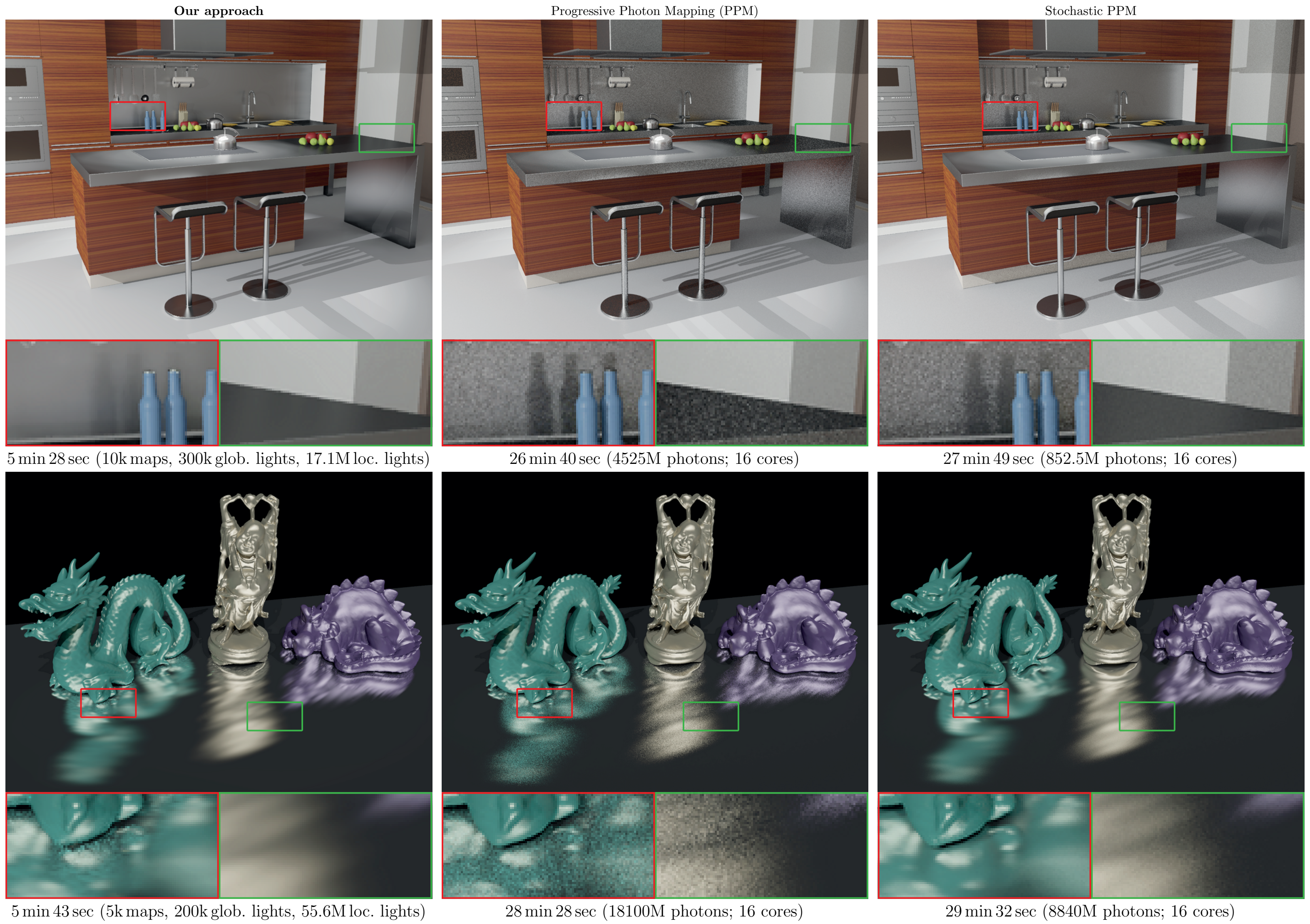

Our approach
Progressive Photon Mapping (PPM)
Stochastic PPM
5 min 28 sec (10k maps, 300k glob. lights, 17.1M loc. lights)
26 min 40 sec (4525M photons; 16 cores)
27 min 49 sec (852.5M photons; 16 cores)
5 min 43 sec (5k maps, 200k glob. lights, 55.6M loc. lights)
28 min 28 sec (18100M photons; 16 cores)
29 min 32 sec (8840M photons; 16 cores)

Figure B.1: *Results:* Comparison of our method with Progressive Photon Mapping and Stochastic Progressive Photon Mapping

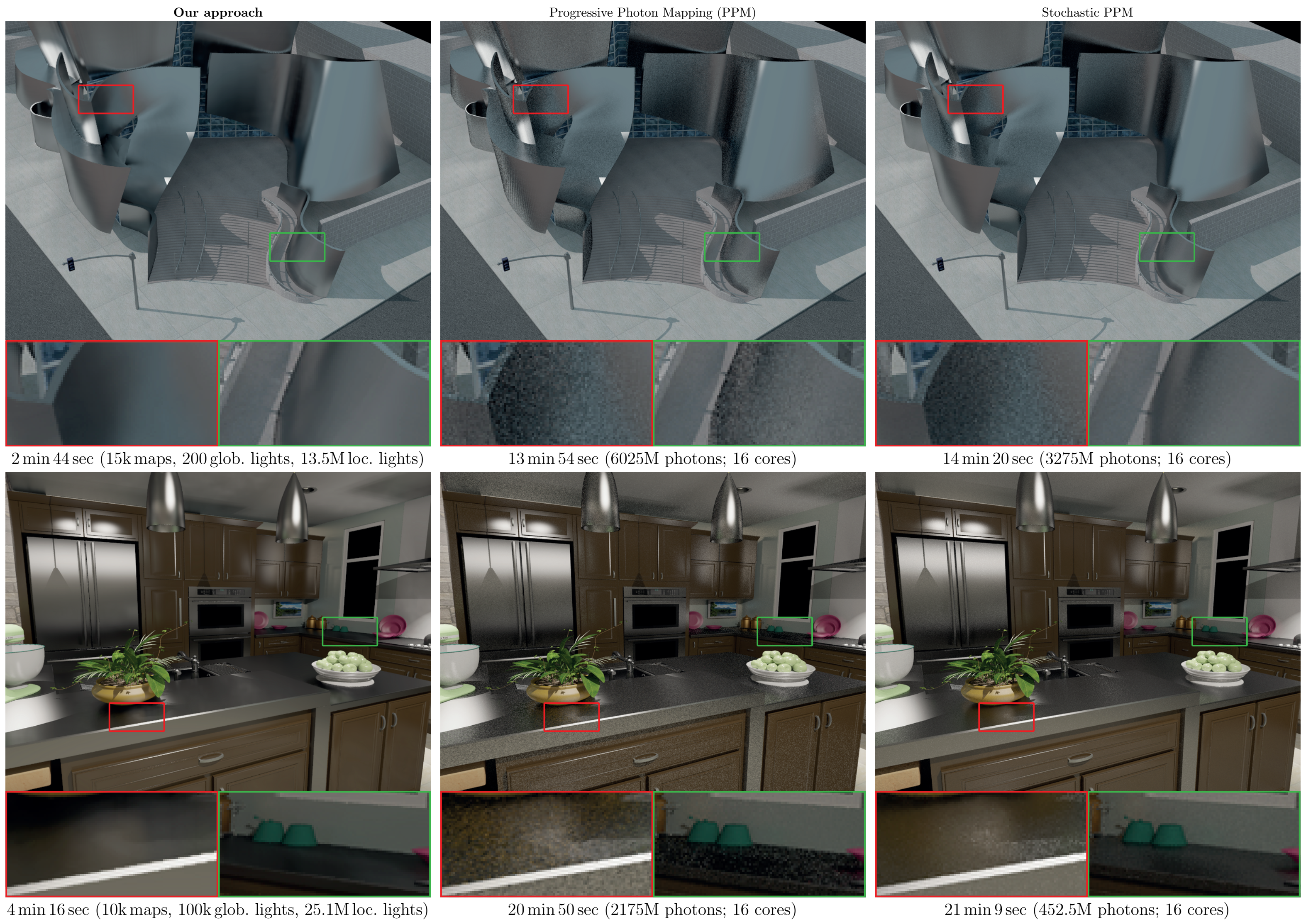

Our approach
Progressive Photon Mapping (PPM)
Stochastic PPM
2 min 44 sec (15k maps, 200 glob. lights, 13.5M loc. lights)
13 min 54 sec (6025M photons; 16 cores)
14 min 20 sec (3275M photons; 16 cores)
4 min 16 sec (10k maps, 100k glob. lights, 25.1M loc. lights)
20 min 50 sec (2175M photons; 16 cores)
21 min 9 sec (452.5M photons; 16 cores)

Figure B.2: *Results:* Comparison of our method with Progressive Photon Mapping and Stochastic Progressive Photon Mapping

Bibliography

Timo Aila and Tero Karras. Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics*, pages 113–122. Eurographics Association, 2010. URL http://dl.acm.org/citation.cfm?id=1921497.

Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics 2009*, pages 145–149, 2009.

Timo Aila, Samuli Laine, and Tero Karras. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, 2012.

John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, page 10, 1987.

AMD. AMD Radeon R9 390 specification. http://www.amd.com/en-us/products/graphics/desktop/r9, July 2015. URL http://www.amd.com/en-us/products/graphics/desktop/r9.

Okan Arikan, David A Forsyth, and James F O'Brien. Fast and detailed approximate global illumination by irradiance decomposition. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 1108–1114. ACM, 2005. URL http://dl.acm.org/citation.cfm?id=1073319.

Bruno Arnaldi, Thierry Priol, and Kadi Bouatouch. A new space subdivision method for ray tracing CSG modelled scenes. *The Visual Computer*, 3 (2):98–108, 1987.

Philippe Bekaert, Mateu Sbert, and John Halton. Accelerating path tracing by re-using paths. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 125–134. Eurographics Association, 2002. URL http://dl.acm.org/citation.cfm?id=581914.

Jiří Bittner, Michal Hapala, and Vlastimil Havran. Fast insertion-based optimization of bounding volume hierarchies. In *Computer Graphics Forum*. Wiley Online Library, 2013. URL http://onlinelibrary.wiley.com/doi/10.1111/cgf.12000/full.

Ian Buck. The evolution of GPUs for general purpose computing. In *Proceedings of the GPU Technology Conference 2010*, 2010.

Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proc. Graphics Hardware 2002*, pages 37–46, 2002. ISBN 1-58113-580-7. URL http://dl.acm.org/citation.cfm?id=569046.569052.

Caustic Graphics, Inc. CausticRT platform, 2009. http://www.caustic.com/.

Ewen Cheslack-Postava, Rui Wang, Oskar Akerlund, and Fabio Pellacini. Fast, realistic lighting and material design using nonlinear cut approximation. In *ACM Transactions on Graphics (TOG)*, volume 27, page 128. ACM, 2008. URL http://dl.acm.org/citation.cfm?id=1409081.

James H Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. In *Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, SIGGRAPH ’81, pages 307–316, New York, NY, USA, 1981. ACM. ISBN 0-89791-045-1. doi: 10.1145/800224.806819.

Holger Dammertz, Johannes Hanika, and Alexander Keller. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In *Computer Graphics Forum (Proc. 19th Eurographics Symposium on Rendering)*, pages 1225–1234, 2008.

Tomáš Davidovič and Iliyan Georgiev. SmallVCM renderer. http://www.smallvcm.com, 2012.

Tomáš Davidovič, Lukáš Maršálek, Nicolas Maeding, Markus Kaltenbach, Peter-Hans Roth, and Philipp Slusallek. Ray tracing element for Cell/B.E., 2009. URL http://graphics.cg.uni-saarland.de/index.php?id=452.

Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, Philipp Slusallek, and Kavita Bala. Combining global and local virtual lights for detailed glossy illumination. *ACM Trans. Graph.*, 29:143:1–143:8, 2010. doi: 10.1145/1882261.1866169. URL http://graphics.cg.uni-saarland.de/index.php?id=davidovicsa2010.

Tomáš Davidovič, Lukáš Maršálek, and Philipp Slusallek. Performance considerations when using a dedicated ray traversal engine. In *19th International Conference on Computer Graphics, Visualization and Computer Vision 2011 (WSCG 2011) Pilsen*, pages 65–72, February 2011. ISBN 978-80-86943-83-1. URL http://graphics.cg.uni-saarland.de/?id=davidovicwscg2011.

Tomáš Davidovič, Thomas Engelhardt, Iliyan Georgiev, Philipp Slusallek, and Carsten Dachsbacher. 3D rasterization: A bridge between rasterization and ray casting. In *Proceedings of Graphics Interface 2012*, pages 201–208, Toronto, Ont., Canada, Canada, 2012a. Canadian Information Processing Society. ISBN 978-1-4503-1420-6. URL `https://graphics.cg.uni-saarland.de/2012/3d-rasterization/`.

Tomáš Davidovič, Iliyan Georgiev, and Philipp Slusallek. Progressive lightcuts for GPU. *ACM SIGGRAPH 2012 Talks*, 2012b. doi: 10.1145/2343045.2343047. URL `https://graphics.cg.uni-saarland.de/2012/progressive-lightcuts-for-gpu/`.

Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, and Philipp Slusallek. Progressive light transport simulation on the GPU: Survey and improvements. *ACM Trans. Graph.*, 2014. ISSN 0730-0301.

Z. Dong, T. Grosch, T. Ritschel, J. Kautz, and H.-P. Seidel. Real-time indirect illumination with clustered visibility. In *Vision, Modeling, and Visualization Workshop 2009*, 2009.

Phil Dutré, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination, 2nd Edition*. A K Peters, Natick, MA, 2006.

Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. Sorted deferred shading for production path tracing. In *Computer Graphics Forum*, volume 32, pages 125–132. Wiley Online Library, 2013.

B. Flachs, S. Asano, S. H. Dhong, P. Hotstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processing unit for a CELL processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 134–135, 2005. doi: 10.1109/ISSCC.2005.1493905.

Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a GPU raytracer. In *Proc. of Graphics Hardware 2005*, pages 15–22, 2005. ISBN 1-59593-086-8. doi: 10.1145/1071866.1071869.

Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. Arts: Accelerated ray-tracing system. *Computer Graphics and Applications, IEEE*, 6(4):16–26, 1986.

R García, Carlos Ureña, and Mateu Sbert. Description and solution of an unreported intrinsic bias in photon mapping density estimation with constant kernel. In *Computer Graphics Forum*, volume 31, pages 33–41. Wiley Online Library, 2012. URL `http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2011.02081.x/full`.

Iliyan Georgiev. Implementing vertex connection and merging. Technical Report Nov. 12, Saarland University, 2012. URL `http://www.iliyan.com/publications/ImplementingVCM`.

Iliyan Georgiev. *Path Sampling Techniques for Efficient Light Transport Simulation*. PhD thesis, Saarland University, Saarbrücken, Germany, 2015.

Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič, and Philipp Slusallek. Light transport simulation with vertex connection and merging. *ACM Trans. Graph.*, 31(6):192:1–192:10, November 2012. ISSN 0730-0301. doi: 10.1145/2366145.2366211.

Kris Gray. *Microsoft DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing*, 2007.

Toshiya Hachisuka and Henrik Wann Jensen. Stochastic progressive photon mapping. *ACM Trans. Graph.*, 28(5):141:1–141:8, December 2009. ISSN 0730-0301. doi: 10.1145/1618452.1618487.

Toshiya Hachisuka and Henrik Wann Jensen. Parallel progressive photon mapping on GPUs. In *ACM SIGGRAPH ASIA 2010 Sketches*, pages 54:1–54:1, 2010. ISBN 978-1-4503-0523-5. doi: 10.1145/1899950.1900004.

Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *ACM Trans. Graph.*, 27(5):130:1–130:8, December 2008. ISSN 0730-0301. doi: 10.1145/1409060.1409083.

Toshiya Hachisuka, Jacopo Pantaleoni, and Henrik Wann Jensen. A path space extension for robust light transport simulation. *ACM Trans. Graph.*, 31(6):191:1–191:10, November 2012. ISSN 0730-0301. doi: 10.1145/2366145.2366210.

Eric Haines. Spline surface rendering, and whats wrong with octrees. *Ray Tracing News*, 1(2), 1988.

Johannes Hanika, Marc Droske, and Luca Fascione. Manifold next event estimation. *Computer Graphics Forum (Proc. of Eurographics Symposium on Rendering)*, 34(4):87–97, June 2015.

Michal Hapala and Vlastimil Havran. Review: Kd-tree traversal algorithms for ray tracing. In *Computer Graphics Forum*, volume 30, pages 199–213. Wiley Online Library, 2011.

Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. Efficient stack-less BVH traversal for ray tracing. In *Proceedings 27th Spring Conference on Computer Graphics (SCCG) 2011*, pages 29–34, 2011a. URL `http://graphics.cg.uni-saarland.de/?id=hapalasccg2011`.

Michal Hapala, Ondrej Karlík, and Vlastimil Havran. When it makes sense to use uniform grids for ray tracing. In *Proceedings of WSCG*, pages 193–200, 2011b.

Bruce Hapke. *Theory of Reflectance and Emittance Spectroscopy, 2nd edition.* Cambridge University Press, 2012. ISBN 0521883490.

Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

Vlastimil Havran, Jiří Bittner, and Jiří Žára. Ray tracing with rope trees. In *14th Spring Conference on Computer Graphics*, pages 130–140, 1998.

Miloš Hašan, Fabio Pellacini, and Kavita Bala. Matrix row-column sampling for the many-light problem. *ACM Trans. Graph.*, 26(3), July 2007. ISSN 0730-0301. doi: 10.1145/1276377.1276410.

Miloš Hašan, Jaroslav Křivánek, Bruce Walter, and Kavita Bala. Virtual spherical lights for many-light rendering of glossy scenes. *ACM Trans. Graph.*, 28(5):143:1–143:6, 2009.

H. Von Helmholtz. *Handbuch der Physiologischen Optik.* 1867.

Heinrich Hey and Werner Purgathofer. Advanced radiance estimation for photon map global illumination. In *Computer Graphics Forum*, volume 21, pages 541–545. Wiley Online Library, 2002.

Qiming Hou, Xin Sun, Kun Zhou, Christian Lauterbach, and Dinesh Manocha. Memory-scalable GPU spatial hierarchy construction. *Visualization and Computer Graphics, IEEE Transactions on*, 17(4):466–474, 2011. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5648735`.

Greg Humphreys and C Scott Ananian. Tigershark: A hardware accelerated ray-tracing engine. *Senior independent work, Princeton University*, 1996.

Yuchi Huo, Rui Wang, Shihao Jin, Xinguo Liu, and Hujun Bao. A matrix sampling-and-recovery approach for many-lights rendering. *ACM Trans. Graph.*, 34(6):210, 2015.

Wenzel Jakob and Steve Marschner. Manifold exploration: A Markov chain Monte Carlo technique for rendering scenes with difficult specular transport. *ACM Transactions on Graphics (TOG)*, 31(4):58, 2012. URL `http://dl.acm.org/citation.cfm?id=2185554`.

Bob Jenkins. Hash functions. *Dr Dobbs Journal*, 22(9), 1997.

Francis A Jenkins and Harvey E White. Fundamentals of optics 4th edition. *Fundamentals of Optics 4th edition by Francis A. Jenkins, Harvey E. White New York, NY: McGraw-Hill Book Company, 1976*, 1, 1976.

Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag. ISBN 3-211-82883-4. URL `http://dl.acm.org/citation.cfm?id=275458.275461`.

Henrik Wann Jensen. *Realistic Image Synthesis using Photon Mapping*. A.K. Peters, 2001.

David Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. 1988.

James T. Kajiya. The rendering equation. In *Computer Graphics (Proc. of SIGGRAPH)*, pages 143–150, 1986. ISBN 0-89791-196-2. doi: 10.1145/15922.15902.

Javor Kalojanov and Philipp Slusallek. A parallel algorithm for construction of uniform grids. In *Proc. of High-Performance Graphics 2009*, pages 23–28, 2009. URL `http://dl.acm.org/citation.cfm?id=1572773`.

Javor Kalojanov, Markus Billeter, and Philipp Slusallek. Two-level grids for ray tracing on GPUs. In Oliver Deussen Min Chen, editor, *EG 2011 - Full Papers*, pages 307–314, Llandudno, UK, 2011. Eurographics Association. doi: 10.1111/j.1467-8659.2011.01862.x. URL `http://diglib.eg.org/EG/CGF/volume30/issue2/v30i2pp307-314.pdf`.

Michael R Kaplan. The use of spatial coherence in ray tracing. *ACM SIGGRAPH Course Notes 11*, 1985.

Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. *Proc. of High-Performance Graphics 2013*, pages 89–99, 2013.

Tero Karras, Timo Aila, and Samuli Laine. Understanding the efficiency of ray traversal on GPUs framework. `http://code.google.com/p/understanding-the-efficiency-of-ray-traversal-on-gpus/`, 2012.

Alexander Keller. Instant radiosity. In *Computer Graphics (Proc. of SIGGRAPH)*, pages 49–56, 1997. ISBN 0-89791-896-7. doi: 10.1145/258734.258769. URL `http://dx.doi.org/10.1145/258734.258769`.

Alexander Keller. Quasi-monte carlo image synthesis in a nutshell. In *Monte Carlo and Quasi-Monte Carlo Methods 2012*, pages 213–249. Springer, 2013.

Khronos OpenCL Working Group and Aaftab Munshi. The OpenCL specification, Version: 2.1, 2015.

Claude Knaus and Matthias Zwicker. Progressive photon mapping: A probabilistic approach. *ACM Trans. Graph.*, 30(3):25:1–25:13, May 2011. ISSN 0730-0301. doi: 10.1145/1966394.1966404.

Thomas Kollig and Alexander Keller. Illumination in the presence of weak singularities. In *Monte Carlo and Quasi-Monte Carlo Methods*, pages 245–257, 2004.

D Kopta, K Shkurko, J Spjut, E Brunvand, and A Davis. Memory considerations for low energy ray tracing. In *Computer Graphics Forum*, volume 34, pages 47–59. Wiley Online Library, 2015.

Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis. An energy and bandwidth efficient ray tracing architecture. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 121–128. ACM, 2013.

Jaroslav Křivánek, James A Ferwerda, and Kavita Bala. Effects of global illumination approximations on material appearance. *ACM Transactions on Graphics (TOG)*, 29(4):112, 2010. URL `http://dl.acm.org/citation.cfm?id=1778849`.

Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proc. of CompuGraphics '93*, 1993.

Eric P. Lafortune, Sing-Choong Foo, and Kenneth E. Torrance andDonald P. Greenberg. Non-linear approximation of reflectance functions. In *Proc. SIGGRAPH '97*, volume 31, pages 117–126, 1997. doi: 10.1145/258734.258801.

Samuli Laine, Hannu Saransaari, Janne Kontkanen, Jaakko Lehtinen, and Timo Aila. Incremental instant radiosity for real-time indirect illumination. In *Eurographics Symposium on Rendering*, pages 277–286, 2007.

Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: Wavefront path tracing on GPUs. *Proc of High-Performance Graphics 2013*, pages 137–143, 2013.

Jurgen Laurijssen, Rui Wang, Ph Dutré, and Benedict J Brown. Fast estimation and rendering of indirect highlights. In *Computer Graphics Forum*, volume 29, pages 1305–1313. Wiley Online Library, 2010. URL `http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2010.01726.x/full`.

Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast BVH construction on GPUs. In *Computer Graphics Forum*, volume 28, pages 375–384, 2009. URL `http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2009.01377.x/full`.

Chris Lomont. Introduction to Intel Advanced Vector Extensions. *Intel White Paper*, 2011.

J David MacDonald and Kellogg S Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990. URL `http://link.springer.com/article/10.1007/BF01911006`.

Songrit Maneewongvatana and David M Mount. Its okay to be skinny, if your friends are fat. In *Center for Geometric Computing 4th Annual Workshop on Computational Geometry*, volume 2, 1999.

Don P Mitchell. Consequences of stratified sampling in graphics. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 277–280. ACM, 1996.

Cleve Moler. Matrix computation on distributed memory multiprocessors. *Hypercube Multiprocessors*, 86:181–195, 1986. This is for Embarrassingly Parallel.

Jan Novák, Vlastimil Havran, and Carsten Daschbacher. Path regeneration for interactive path tracing. In *EUROGRAPHICS 2010, short papers*, pages 61–64, 2010.

NVIDIA. *Fermi Compute Architecture Whitepaper*, 2011.

NVIDIA. *CUDA C Programming Guide 5.0*. NVIDIA, 2012a.

NVIDIA. *NVIDIA GeForce GTX 680 Whitepaper*, 2012b.

NVIDIA. *NVIDIA GeForce GTX 980 Whitepaper*, 2014.

NVIDIA. *CUDA C Programming Guide 7.0*. NVIDIA, 2015.

Stuart Oberman, Greg Favor, and Fred Weber. AMD 3DNow! technology: Architecture and implementations. *Micro, IEEE*, 19(2):37–48, 1999.

Open SystemC Initiative. IEEE standard SystemC language reference manual. *IEEE Computer Society*, 2006.

Jiawei Ou and Fabio Pellacini. Lightslice: Matrix slice sampling for the many-lights problem. *ACM Trans. Graph.*, 30(6):179, 2011.

J. Owczarczyk. Ray tracing: A challenge for parallel processing. *Proc Parallel Processing for Computer Vision and Display, Leeds*, 1988.

Anthony Pajot, Loïc Barthe, Mathias Paulin, and Pierre Poulin. Combinatorial bidirectional path-tracing for efficient hybrid CPU/GPU rendering. In *Computer Graphics Forum*, volume 30, pages 315–324, 2011.

Jacopo Pantaleoni and David Luebke. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proc. of High-Performance Graphics 2010*, pages 87–95, 2010. URL `http://dl.acm.org/citation.cfm?id=1921493`.

Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. PantaRay: Fast ray-traced occlusion caching of massive scenes. In *ACM Transactions on Graphics (TOG)*, volume 29, page 37. ACM, 2010. URL `http://dl.acm.org/citation.cfm?id=1778774`.

Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. OptiX: A general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, July 2010. URL `http://dl.acm.org/citation.cfm?id=1778803`.

Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Elsevier Science & Technology Books, 2004. URL `http://www.pbrt.org`.

Matt Pharr and William R Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13. IEEE, 2012.

Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975. ISSN 0001-0782. doi: 10.1145/360825.360839.

David Plunkett and Michael Bailey. The vectorization of a ray-tracing algorithm for improved execution speed. *IEEE Computer Graphics and Applications*, 8(5):52–60, 1985.

Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless KD-tree traversal for high performance GPU ray tracing. In *Computer Graphics Forum*, volume 26, pages 415–424, 2007.

Stefan Popov, Iliyan Georgiev, Rossen Dimov, and Philipp Slusallek. Object partitioning considered harmful: Space subdivision for BVHs. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 15–22. ACM, 2009. URL `http://dl.acm.org/citation.cfm?id=1572772`.

Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *ACM Trans. Graph.*, volume 21, pages 703–712, July 2002.

Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proc. Graphics Hardware 2003*, pages 41–50, 2003.

Srinivas K Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE micro*, (4):47–57, 2000.

Ganesh Ramanarayanan, James Ferwerda, Bruce Walter, and Kavita Bala. Visual equivalence: Towards a new standard for image fidelity. *ACM Trans. Graph.*, 26(3):76:1–76:11, 2007.

Karthik Ramani, Christiaan P Gribble, and Al Davis. StreamRay: A stream filtering architecture for coherent ray tracing. In *ACM Sigplan Notices*, volume 44, pages 325–336. ACM, 2009. URL `http://dl.acm.org/citation.cfm?id=1508282`.

James Reinders. An overview of programming for Intel Xeon processors and Intel Xeon Phi coprocessors. *Intel Corporation: Santa Clara*, 2012.

Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1176–1185, New York, NY, USA, 2005. ACM Press. doi: 10.1145/1186822.1073329.

Tobias Ritschel, Thorsten Grosch, Min H. Kim, Hans-Peter Seidel, Carsten Dachsbacher, and Jan Kautz. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph.*, 27(5):129, 2008.

Tobias Ritschel, Thomas Engelhardt, Thorsten Grosch, H-P Seidel, Jan Kautz, and Carsten Dachsbacher. Micro-rendering for scalable, parallel final gathering. In *ACM Transactions on Graphics (TOG)*, volume 28, page 132. ACM, 2009. URL `http://dl.acm.org/citation.cfm?id=1618478`.

Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz. The state of the art in interactive global illumination. In *Computer Graphics Forum (STAR)*, volume 31, pages 160–188, 2012. URL `http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2012.02093.x/full`.

Randi J Rost, Bill Licea-Kane, Dan Ginsburg, John M Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL shading language*. Pearson Education, 2009.

Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR: A hardware architecture for ray tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 27–36. Eurographics Association, 2002. URL `http://dl.acm.org/citation.cfm?id=569051`.

Benjamin Segovia, Jean-Claude Iehl, Richard Mitanchey, and Bernard Péroche. Bidirectional instant radiosity. In *Proceedings of the 17th Eurographics Workshop on Rendering, to appear*, 2006.

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008*, 2008.

Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proc. Graphics Hardware 2007*, pages 97–106, 2007. ISBN 978-1-59593-625-7. URL `http://dl.acm.org/citation.cfm?id=1280094.1280110`.

Josef Spjut, Andrew Kensler, Daniel Kopta, and Erik Brunvand. Trax: A multicore hardware architecture for real-time ray tracing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(12):1802–1815, 2009. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5324031`.

Josef Spjut, Daniel Kopta, Erik Brunvand, and Al Davis. A mobile accelerator architecture for ray tracing. In *Proceed-ings of 3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-3)*, 2012.

Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proc. of High-Performance Graphics 2009*, pages 7–13, 2009. URL `http://dl.acm.org/citation.cfm?id=1572771`.

Dietger van Antwerpen. A survey of importance sampling applications in unbiased physically based rendering. Technical report, 2011a.

Dietger van Antwerpen. Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU. In *Proc. of High-Performance Graphics 2011*, pages 41–50, 2011b. doi: 10.1145/2018323.2018330.

Dietger van Antwerpen. Unbiased physically based rendering on the GPU. Master's thesis, Delft University of Technology, the Netherlands, 2011c.

Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, 1997.

Eric Veach and Leonidas Guibas. Bidirectional estimators for light transport. In *Proc. of Eurographics Rendering Workshop*, pages 147–162, 1994.

Eric Veach and Leonidas J. Guibas. Optimally combining sampling techniques for Monte Carlo rendering. In *Computer Graphics (Proc. of SIGGRAPH)*, pages 419–428, 1995. ISBN 0-89791-701-4. doi: 10.1145/218380.218498. URL `http://doi.acm.org/10.1145/218380.218498`.

Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, March 1985. ISSN 0098-3500. doi: 10.1145/3147.3165.

Jiří Vorba. Optimal strategy for connecting light paths in bidirectional methods for global illumination computation. Master's thesis, Charles University in Prague, 2011.

Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, 2004.

Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, volume 20, pages 153–165, 2001.

Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics 2007*, pages 89–116, 2007.

Ingo Wald, Carsten Benthin, and Solomon Boulos. Getting rid of packets: Efficient SIMD single-ray traversal using multi-branching BVHs. August 2008.

Bruce Walter, Sebastian Fernandez, Adam Arbee, Kavita Bala, Michael Donikian, and Donald Greenberg. Lightcuts: A scalable approach to illumination. *ACM SIGGRAPH Conference Proceedings*, 2005.

Bruce Walter, Adam Arbree, Kavita Bala, and Donald P. Greenberg. Multidimensional lightcuts. *ACM Trans. Graph.*, 25(3):1081–1088, 2006.

Bruce Walter, Pramook Khungurn, and Kavita Bala. Bidirectional lightcuts. *ACM Transactions on Graphics (TOG)*, 31(4):59, 2012. URL http://dl.acm.org/citation.cfm?id=2185555.

Rui Wang, Rui Wang, Kun Zhou, Minghao Pan, and Hujun Bao. An efficient GPU-based approach for interactive global illumination. *ACM Trans. Graph.*, 28(3):91, 2009.

Gregory J. Ward. Measuring and modeling anisotropic reflection. In *Proc. SIGGRAPH 92*, pages 265–272, 1992.

Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980. ISSN 0001-0782. doi: 10.1145/358876.358882.

Sven Woop. *DRPU: A Programmable Hardware Architecture for Real-time Ray Tracing of Coherent Dynamic Scenes*. PhD thesis, Saarland University, 2006.

Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: A programmable ray processing unit for realtime ray tracing. In *ACM Trans. on Graph. (Proceedings of SIGGRAPH 2005)*, volume 24, pages 434–444, 2005.

Sven Woop, Erik Brunvand, and Philipp Slusallek. Estimating performance of a ray-tracing ASIC design. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 7–14. IEEE, 2006a. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4061540.

Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD trees for hardware accelerated ray tracing of dynamic scenes. In *SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware*, volume 3, pages 67–77, 2006b.

Sven Woop, Louis Feng, Ingo Wald, and Carsten Benthin. Embree: Ray tracing kernels for CPUs and the Xeon Phi architecture. In *ACM SIGGRAPH 2013 Talks*, pages 44:1–44:1, 2013. URL http://dl.acm.org/citation.cfm?id=2504515.

Fahad Zafar, Marc Olano, and Aaron Curtis. GPU random numbers via the tiny encryption algorithm. In *Proc. of High-Performance Graphics 2010*, pages 133–141, 2010. URL http://dl.acm.org/citation.cfm?id=1921500.

Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):126:1–126:11, December 2008. ISSN 0730-0301. doi: 10.1145/1409060.1409079. URL http://doi.acm.org/10.1145/1409060.1409079.

My Publications

Tomáš Davidovič and Iliyan Georgiev. Smallvcm renderer. http://www.smallvcm.com, 2012.

Tomáš Davidovič, Lukáš Maršálek, Nicolas Maeding, Markus Kaltenbach, Peter-Hans Roth, and Philipp Slusallek. Ray tracing element for cell/b.e., 2009. URL http://graphics.cg.uni-saarland.de/index.php?id=452.

Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, Philipp Slusallek, and Kavita Bala. Combining global and local virtual lights for detailed glossy illumination. *ACM Trans. Graph.*, 29:143:1–143:8, 2010. doi: 10.1145/1882261.1866169. URL http://graphics.cg.uni-saarland.de/index.php?id=davidovicsa2010.

Tomáš Davidovič, Lukáš Maršálek, and Philipp Slusallek. Performance considerations when using a dedicated ray traversal engine. In *19th International Conference on Computer Graphics, Visualization and Computer Vision 2011 (WSCG 2011) Pilsen*, pages 65–72, February 2011. ISBN 978-80-86943-83-1. URL http://graphics.cg.uni-saarland.de/?id=davidovicwscg2011.

Tomáš Davidovič, Iliyan Georgiev, and Philipp Slusallek. Progressive lightcuts for GPU. *ACM SIGGRAPH 2012 Talks*, 2012. doi: 10.1145/2343045.2343047. URL https://graphics.cg.uni-saarland.de/2012/progressive-lightcuts-for-gpu/.

Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, and Philipp Slusallek. Progressive light transport simulation on the gpu: Survey and improvements. *ACM Trans. Graph.*, 2014. ISSN 0730-0301.

Tomáš Davidovič, Thomas Engelhardt, Iliyan Georgiev, Philipp Slusallek, and Carsten Dachsbacher. 3D rasterization: A bridge between rasterization and ray casting. In *Proceedings of Graphics Interface 2012*, pages 201–208, Toronto, Ont., Canada, Canada, 2012. Canadian Information Processing Society. ISBN 978-1-4503-1420-6. URL https://graphics.cg.uni-saarland.de/2012/3d-rasterization/.

Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič, and Philipp Slusallek. Light transport simulation with vertex connection and merging. *ACM Trans. Graph.*, 31(6):192:1–192:10, November 2012. ISSN 0730-0301. doi: 10.1145/2366145.2366211.

Michal Hapala, Tomas Davidovic, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. Efficient stack-less bvh traversal for ray tracing. In *Proceedings 27th Spring Conference on Computer Graphics (SCCG) 2011*, pages 29–34, 2011. URL `http://graphics.cg.uni-saarland.de/?id=hapalasccg2011`.

Beata Turoňová, Lukas Marsalek, Tomáš Davidovič, and Philipp Slusallek. Progressive stochastic reconstruction technique (psrt) for cryo electron tomography. *Journal of structural biology*, 189(3):195–206, 2015.