Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor Thesis

Implementing a Parallel Renderer with Vertex Connection and Merging

submitted by

Pascal Grittmann

on

May 24, 2016

Supervisor

Prof. Dr. Ing. Philipp Slusallek

Advisor

Arsène Pérard-Gayot

Reviewers

Prof. Dr. Ing. Philipp Slusallek
Dr. Karol Myszkowski

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, May 24, 2016 _____

Pascal Grittmann

Acknowledgements

I would like to thank my advisor, Arsène Pérard-Gayot, for his many ideas, fruitful discussions, and useful feedback – and especially for the long debugging sessions. I would also like to thank Prof. Dr. Philipp Slusallek for providing me with the opportunity to work in a field which I love more with every passing day. Furthermore, I would like to thank Javor Kalojanov and Marc Habermann for proofreading (parts of) this thesis.

Abstract

Modern processor architectures rely on parallelism for performance, while realistic rendering using global illumination algorithms is very computationally expensive. Thus, exploiting the hardware capabilities to execute multiple threads in parallel is essential for an efficient renderer. While Path Tracing, for instance, is trivial to parallelize, the more complex algorithms, like Vertex Connection and Merging (VCM), still pose a challenge. In this thesis, we describe a parallel implementation of a renderer using VCM, and discuss the advantages of VCM over Path Tracing. We describe several parallelization strategies and evaluate their performance. In the discussion, we highlight the remaining bottlenecks and propose potential solutions and ideas for future work.

Contents

Chapter 1

Introduction

Photorealistic rendering of three-dimensional (3D) scenes is used in many fields. There is high demand for renderers that achieve short rendering times and interactive frame rates for scenes of various sizes and with various materials. Such renderers often have to run on different types of hardware, from ordinary PCs to large server farms.

In this thesis, we implement a physically based 3D renderer, which achieves interactive frame rates and short rendering times for scenes of various complexity. To accomplish this goal, we combined an already existing fast scene traversal code, written with AnyDSL[1], with a robust and efficient rendering algorithm, namely Vertex Connection and Merging (VCM) [Geo+12]. The renderer was implemented in C++ and runs on the CPU, while the traversal part can run on either the CPU or the GPU. All design decisions in our implementation have been made with the thought in mind, that they might later on be adapted to an implementation on the GPU.

Achieving high performance on modern hardware requires parallelism. The goal of this thesis is to devise a parallelization scheme that works well with complex algorithms, especially VCM. We investigated and compared different approaches to parallelism. A parallel design based on a combination of low-level and high-level parallelism was employed to make maximum use of the fast traversal code, and to reduce shading times as far as possible. With this parallelism, our implementation achieves near-linear scalability for both VCM and Path Tracing. Efficient use of the GPU traversal was possible as well, even though the data transfer between the CPU and the GPU is slow.

VCM is a combination of multiple rendering algorithms and can easily be reduced to any of them. We compared the performance results for a number of these algorithms. Additionally, we implemented an optimized and well tested Path Tracer separately, as a reference. We investigated whether the fewer samples required by a complex technique like VCM actually yield a better convergence over time. In all our scenes, VCM performs either much better or only slightly worse than the other algorithms. The Path Tracer was much simpler to implement and achieves significantly higher frame rates. However, our results show that VCM is often a better choice, and that it is possible to achieve interactive frame rates with VCM as well.

[1] `https://anydsl.github.io/index.html`

Figure 1.1: *The most challenging of our test scenes features complex caustics and consists of almost half a million polygons. We achieved interactive frame rates and short rendering times for a high quality image with this scene.*

1.1 Outline

In Chapter 2, we review the basics of 3D rendering in general and physically based rendering in particular. We briefly review the mathematical foundations and how scenes are represented. The simplest algorithms, Path Tracing and its adjoint, Light Tracing, are described. We discuss the more advanced algorithms, like Bidirectional Path Tracing and VCM in this chapter as well.

Chapter 3 provides the necessary background for efficient hardware implementations on both the CPU and the GPU. After briefly discussing the benefits of the AnyDSL framework, we outline how the traversal library works.

Chapter 4 describes the design of our renderer. We present our parallelization scheme and show how we made as much use as possible out of the fast traversal. Other optimizations, like the light vertex cache, are also treated in this chapter.

Finally, in Chapter 5, we review our implementation in terms of performance, and possible future improvements. We evaluate the performance of the different algorithms, using two complex and five simpler test scenes. The complex scenes serve as real-world examples, whereas the simpler scenes were used to isolate the costs of shading.

Chapter 2

Rendering

Rendering has a solid theoretical foundation, which we will briefly review in this chapter. Most lighting effects that are visible in everyday life can be described using geometric optics[1]. Thus, the natural way to compute light propagation through a scene is by tracing rays between points on the surfaces of the scene – "ray tracing". All algorithms presented in this thesis are based on ray tracing. For a thorough discussion of what kinds of effects are captured by geometric optics and ray tracing, and which require special case handling, see [Vea98].

The renderer described in this thesis is limited to surface interactions. Things like volume rendering and subsurface scattering are left for future work.

2.1 Scene Representation

Rendering a scene requires a description of its geometrical structure and the reflective and transmissive properties of its surfaces. The following sections explain how this information is usually represented in a renderer.

2.1.1 Geometry

There are multiple ways to represent the actual geometry of a 3D scene. The most common approach is by using triangle meshes, as they offer great artistic freedom as well as simplicity in the implementation. Parametric surfaces, constructive solid geometry (CSG), and voxel based representations are examples of other ways to describe the geometry. The traversal library that was used for this thesis supports only triangle meshes.

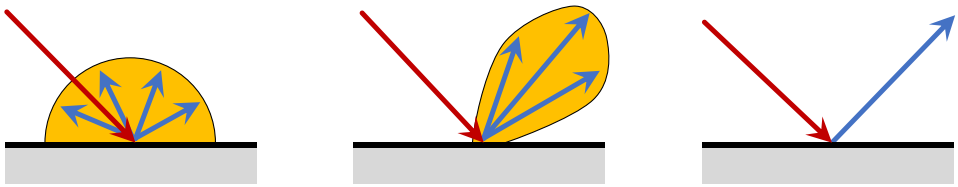
Solving the rendering equation (see below) requires shooting rays and computing their intersections with the scene geometry. This process is often called *scene traversal*. Traversal makes up a huge portion of the rendering time, as it requires calculating thousands of costly ray-triangle intersections for millions of rays. Performance is greatly increased by using acceleration structures, like bounding volume hierarchies or kd-trees, to structure

[1] Geometric optics models the propagation of light using rays. See: `https://en.wikipedia.org/wiki/Geometrical_optics`

the geometry of the scene in a tree. The traversal library, that was used in this thesis, uses a bounding volume hierarchy (BVH) on the GPU, and a multi-branching bounding volume hierarchy (MBVH) [WBB08] on the CPU.

2.1.2 Materials

Apart from describing the geometry of a scene, using triangle meshes, we also need to describe the reflective, transmissive, and emissive properties of the surfaces – their materials. Materials are often classified in diffuse, glossy, and specular materials, depending on how they scatter incoming light. Figure 2.1 illustrates this classification.



(a) *Diffuse* **(b)** *Glossy* **(c)** *Specular*

Figure 2.1: *Materials are usually divided in three categories. Diffuse materials (a) scatter light uniformly in all directions, glossy materials (b) scatter light mostly in a small set of directions, called the glossy lobe, and specular materials (c) scatter light from one incoming direction in exactly one outgoing direction.*

Examples for diffuse surfaces include paper and some kinds of plastic. Glossy surfaces include brushed metal and wet floors. Glass, water, and polished metal are examples for specular surfaces. Specular surfaces, and the complex caustics[2] they can create, are very challenging to render, because randomly sampling a pair of directions with a non-zero contribution is impossible with specular surfaces. Improving convergence rates in the presence of specular materials is the motivation behind many advanced rendering algorithms like Vertex Connection and Merging.

The material classification can be used to talk about light carrying paths in terms of the materials at the vertices of the path. A notation similar to regular expressions was introduced by [Hec90]. With this notation, a path that starts at a light source and undergoes multiple diffuse and one specular bounce before hitting the eye is written as “LD*SE”. Veach ([Vea98], pages 231 to 242) extended this notation and used it to discuss what kinds of paths are difficult or impossible to sample.

2.2 Theory

The following sections review the basic theoretical concepts behind rendering with Monte Carlo methods. More in-depth discussions of the topic can be found, for instance, in [PH10], [Vea98], and [Geo15].

[2] Caustics are the complex light patterns that occur if light rays are bundled together by a specular surface like, for instance, glass.

2.2.1 The Light Transport Equation

The light transport equation, or rendering equation, was introduced by [Kaj86]. It describes the distribution of light in equilibrium state. The equation determines the amount of light, in terms of radiance, leaving a point x on a surface in the scene in an outgoing direction ω_o. The outgoing radiance is given by the amount of light emitted in direction ω_o at point x, plus the amount of incident radiance from all directions, that is scattered in direction ω_o.

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f(\omega_i, x, \omega_o) L_i(x, \omega_i) |cos\theta_i| d\omega_i \tag{2.1}$$

Where

- $L_o(x, \omega)$, $L_e(x, \omega)$, and $L_i(x, \omega)$ denote the outgoing, emitted, and incident radiance at point x in/from direction ω, respectively,
- Ω is the set of all directions,
- $f(\omega_i, x, \omega_o)$, the bi-directional scattering distribution function (BSDF), determines the percentage of incident radiance from direction ω_i, arriving at point x, that is reflected or transmitted in direction ω_o,
- θ_i is the angle formed by the surface normal at point x and the incident direction ω_i.

The incident light from all directions can again be determined using the same equation. It holds that $L_i(x, \omega_i) = L_o(y, -\omega_i)$, where y is the first intersection point with the scene of the ray with origin x and direction ω_i.

2.2.2 The Measurement Equation

The goal during rendering usually is to compute an image from a specific point of view, like a camera or the eye of a human observer. With the light transport equation (2.1), it is (theoretically) possible to determine the exact amount of light travelling from any point in the scene in a certain direction. But how can you get the pixel values of the resulting image from these results? The measurement equation expresses this process mathematically. The value, or measurement, $I^{(j)}$ of some pixel j is given by:

$$I^{(j)} = \int_{Sx\Omega} W_e^{(j)}(x, \omega) L_i(x, \omega) |cos\theta| dA(x) d\omega \tag{2.2}$$

Where S denotes the sensor surface and dA the differential area. The sensor responsivity, or importance, $W_e(x, \omega)$ determines how much the incident radiance from direction ω at the point x on the sensor surface contributes to the pixel measurement I. Every pixel measurement has its own responsivity function. Simply put, the responsivity function determines which part of the sensor surface corresponds to the pixel, and from which directions the pixel receives light.

The L_i quantities can be determined with the light transport equation, again using the relation $L_i(x, \omega) = L_o(y, -\omega)$. We can compute the final image, as as perceived by the sensor with surface S, by solving the measurement equation for all pixels.

2.2.3 Monte Carlo Integration

Solving the integrals in the light transport equation and the measurement equation analytically is generally not possible. Due to the high dimension and frequent discontinuities, Monte Carlo integration is the best numeric method for solving those integrals, and other integrals that are common in rendering. Monte Carlo integration is done by drawing random samples from the integral, turning the integral into a discrete sum. The expected value of the Monte Carlo estimator

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \tag{2.3}$$

is the solution of the integral. N is the number of samples, X_i the random variable from which the samples are drawn, $p(X_i)$ its probability density function (pdf), and $f(x)$ the function to be integrated.

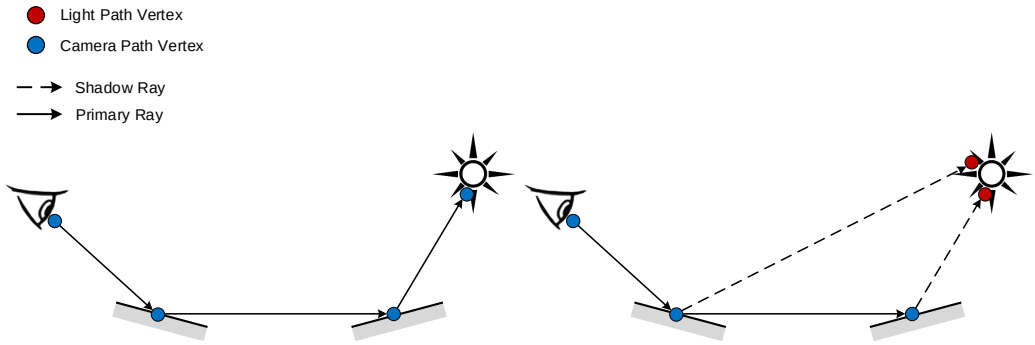
Images rendered using Monte Carlo methods eventually converge to the correct solution, given enough samples. Initially, without enough samples, they suffer from visible noise, due to the variance of the estimator. Reducing the variance, and thus increasing the rate at which the image converges, is the goal behind all algorithmic optimizations for Monte Carlo methods.

2.2.4 Path Tracing

The Path Tracing algorithm was introduced along with the rendering equation in [Kaj86]. It is the straightforward way to apply Monte Carlo integration to the rendering equation. Figure 2.2 illustrates the algorithm. Multiple paths are traced, starting at the camera. If a path happens to hit a light source, the contribution is added. Every path forms a sample of a Monte Carlo estimator, and the expected value of this estimator is the solution to the combination of the measurement equation (2.2) with the light transport equation (2.1).

Next event estimation is a simple optimization to significantly improve the efficiency of Path Tracing. At every intersection point (”vertex”) along the path, a shadow ray is cast towards a light source. If the shadow ray does not intersect anything in front of the light, the illumination is added to the path contribution. If point lights, or other physically incorrect lights, are used, next event estimation is the only way to get any contribution at all, since it is (mathematically) impossible that a randomly sampled ray intersects an infinitesimal point.

Path Tracing is very simple to implement efficiently, because it is trivial to parallize. However, Path Tracing is not efficient at rendering caustics and indirect illumination, because sampling such paths has a low probability but the paths usually have a high contribution. Caustics created by point lights are impossible to sample, because next event estimation



(a) *Path Tracing.* **(b)** *Next Event Estimation.*

Figure 2.2: *Naïve Path Tracing traces paths until they hit a light. Next event estimation reduces variance by connecting every vertex on the camera path to a random vertex on a light source.*

does not work on specular surfaces. These effects are handled much better by Light Tracing, another, equally simple, algorithm. Unfortunately, Light Tracing suffers from other problems.

2.2.5 The Importance Transport Equation

Light Tracing belongs to the group of so-called particle tracing algorithms. These algorithms trace the paths of photons emitted from the lights. Particle tracing can be formulated mathematically, using the importance transport equation. By treating the importance $W_e(x, \omega)$ as an emitted quantity, similar to how light sources emit light, the same transport rules that can be applied to light also apply to importance. The resulting equation is very similar to the light transport equation, except that importance, instead of radiance, is traced through the scene.

$$W_o(x, \omega_o) = W_e(x, \omega_o) + \int_\Omega f(\omega_o, x, \omega_i) W_i(x, \omega_i) |cos\theta_i| d\omega_i \tag{2.4}$$

Here, W_o, W_e, and W_i denote the outgoing, emitted, and incident importance, respectively. Note that the direction arguments of the BSDF have been swapped. The BSDF still describes the scattering of light, rather than importance, and light travels in exactly the opposite direction. Special care has to be taken if the BSDF is not symmetric, for instance due to refraction or the use of shading normals[3]. Veach discussed the implications of non-symmetric BSDFs in detail [Vea98]. We used his proposed solutions regarding shading normals and refractions in our implementation.

2.2.6 Light Tracing

Light tracing [DLW93] is the adjoint of Path Tracing. Instead of tracing paths from the camera to the lights, light tracing traces paths from the lights to the camera. Conversely, it solves the importance transport equation, instead of the light transport equation. Next

[3] Shading normals are normals that are not perpendicular to the underlying geometry. They are used, for instance, to achieve smooth looking or highly detailed surfaces with few primitives.

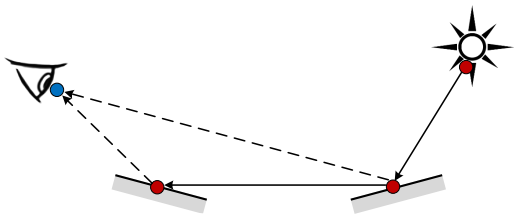


Figure 2.3: *Light Tracing traces paths starting at the light source, connecting every vertex to the camera.*

event estimation also works with light tracing, by connecting every vertex to the camera and adding the resulting contributions to the corresponding pixels. As with point lights in Path Tracing, this is the only way to get any contribution at all, if the camera is modeled as an infinitesimal pinhole camera, which is often the case. Figure 2.3 illustrates the paths generated during light tracing.

Light tracing is very efficient at handling caustics and indirect illumination. The probability to sample these paths is proportional to their contribution. However, the inefficiencies from which Path Tracing suffers in the presence of point lights or small area lights also occur in Light Tracing for pinhole cameras and small sensors. Specular reflections directly seen by a (pinhole) camera are difficult or impossible to sample. Also, the probability to sample a path that actually contributes to the rendered image is often very low. Thus, direct illumination is very noisy. Figure 2.4 shows the differences between Light Tracing and Path Tracing after the same rendering time.

2.3 Efficient Algorithms

In this thesis, we focus on three important aspects of a rendering algorithm. An algorithm should be consistent, robust, and efficient.

A **consistent** algorithm converges to the correct result, given enough time.

A **robust** algorithm can handle many, ideally all, kinds of scenes well.

An **efficient** algorithm requires as few samples as possible to obtain a high quality result.

VCM is one of the few algorithms to fulfill all three criteria. Before discussing what makes VCM consistent, robust, and efficient, we review the techniques and algorithms on which VCM is based.

2.3.1 Multiple Importance Sampling

Importance sampling, see for instance [PH10] pages 688 to 690, is crucial for Monte Carlo methods. Variance, and thus the required number of samples, is greatly reduced when sampling from a distribution that is proportional, or at least similar, to the integrand. Unfortunately, finding such a distribution is not always possible, because the shape of the integrand is often unknown or too complex. In many cases, it is possible to find multiple distributions that each correspond to different parts of the integrand (see Figure 2.5).

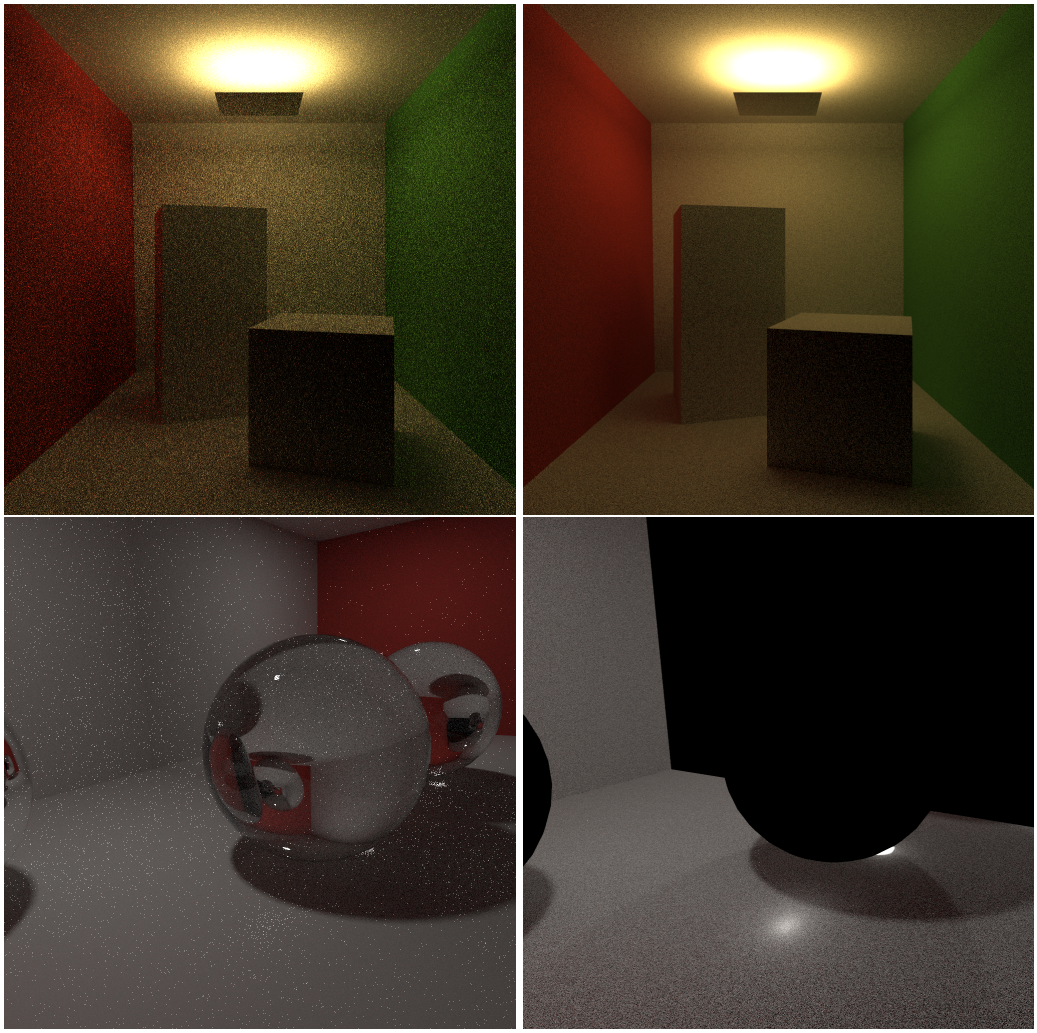


Figure 2.4: *Comparison of the images created by Path Tracing (left) and Light Tracing (right) within five seconds. With Light Tracing, indirect illumination is much smoother and caustics are already very clear. Path Tracing suffers from a lot of noise but it can capture directly visible reflections and refractions. Note that the contribution from direct illumination in the lower images is much less noisy with Path Tracing.*

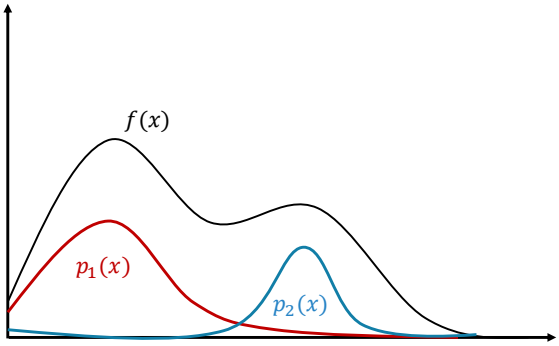


Figure 2.5: *Often, different distributions work well for different parts of an integrand. Multiple Importance sampling offers a way to combine samples from multiple such distributions in a way that produces good results overall.*

Multiple Importance Sampling (MIS) [VG95] combines samples from different distributions in a way that minimizes variance. Multiple sampling techniques, and thus distributions, are used to generate samples. The samples are then weighted, taking the pdf values of all the techniques into account, that could create this sample as well. The estimator for multiple importance sampling is given by

$$F_N = \sum_{i=1}^{N} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(X_{i,j}) \frac{f(X_{i,j})}{p_i(X_{i,j})} \tag{2.5}$$

where N is the number of sampling techniques, n_i the number of samples from technique i and w_i the weighting function. The weights have to sum up to one.

Weighting is usually done using a heuristic function. It can be proven that the power heuristic yields good results. It is given by

$$w_i(X_{i,j}) = \frac{n_i^\beta p_i^\beta(X_{i,j})}{\sum_{k=1}^{N} n_k^\beta p_k^\beta(X_{i,j})} \tag{2.6}$$

Were n_i is the number of samples from technique i, and p_i is the pdf of the technique. Setting $\beta = 2$ is a particularly good choice for the power heuristic. The balance heuristic, another heuristic that works well in practice, can be obtained by setting $\beta = 1$.

MIS is used in Bidirectional Path Tracing and in VCM. It is the reason why these algorithms are efficient.

2.3.2 Bidirectional Path Tracing

Bidirectional Path Tracing (BPT) [VG94] [LW93] combines Path Tracing and Light Tracing, using MIS. A path is traced starting at the light source, and another path starting at the camera. The vertices of the camera path are connected to the vertices of the light path, by tracing a shadow ray between them. Every such connection forms a (potential) path from the camera to the light source. The idea is illustrated in Figure 2.8(a). Additionally,

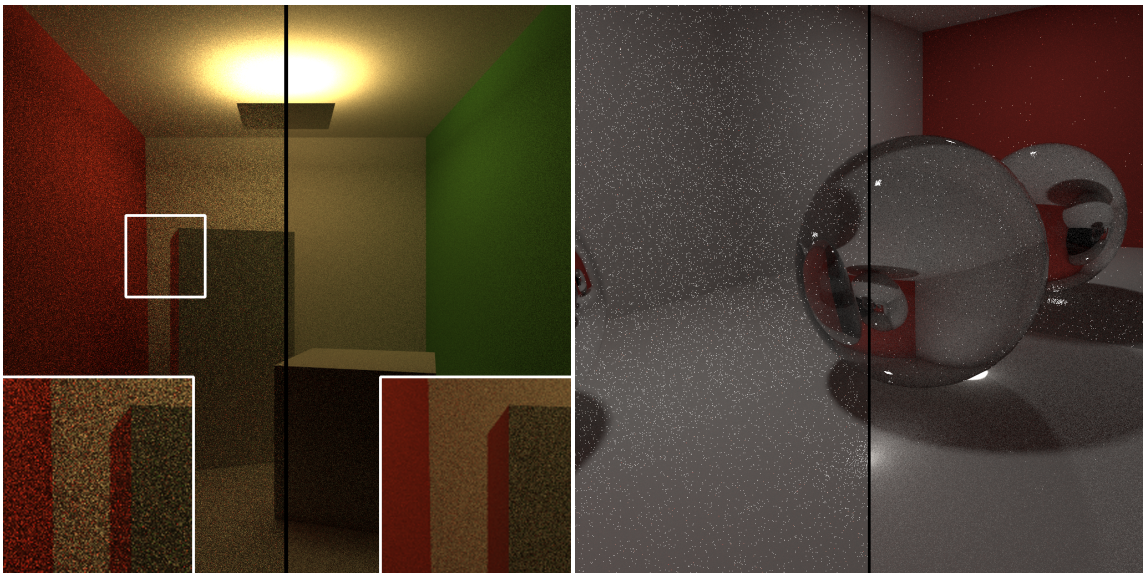


Figure 2.6: *Bidirectional Path Tracing (right parts) can handle indirect illumination and caustics as well as Light Tracing and direct illumination as well as Path Tracing (left parts).*

it is possible, and beneficial, to also use next event estimation with BPT. In that case, BPT combines the contributions from four sampling techniques:

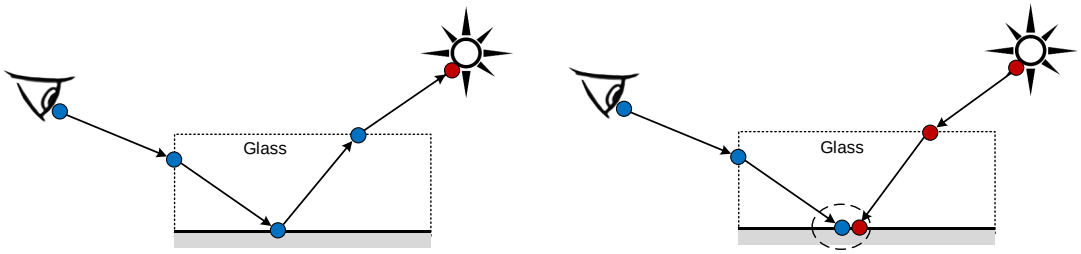
1. connecting light vertices to the camera,
2. connecting camera vertices to a point on a light,
3. connecting camera vertices to light vertices,
4. randomly hitting a light during the Path Tracing step.

MIS can – and should – be used to combine the contributions from those techniques. Figure 2.6 shows how BPT combines the benefits from Path Tracing and Light Tracing, without suffering from the drawbacks of either.

There is one specific kind of paths, namely specular-diffuse-specular (SDS) paths, that pose a challenge for BPT and many other algorithms, because sampling these paths is very difficult. SDS paths represent reflections and refractions of caustics. One example can be seen on the mirror wall close to the right border of Figure 2.6. Figure 2.7 illustrates why these paths are difficult to sample. They require randomly sampling a direction that hits the specular surface in a way such that the refracted, or reflected, ray hits the light source (or the camera). SDS paths can only be sampled with BPT, if lights or cameras with a non-zero area are used. Furthermore, SDS paths have a low probability but usually a high contribution. Thus, the variance for those paths is high.

2.3.3 Photon Mapping and Progressive Photon Mapping

Another kind of algorithm, Photon Mapping [Jen96], is actually very efficient at capturing SDS paths. Photon Mapping is done in two steps, or passes. The first pass traces the photon paths, starting on the light sources. The vertices of these paths (that is the photons) are stored. The second pass traces paths starting at the camera. A nearest neighbor search is performed at every camera vertex, to find all photons within a fixed radius. The camera



(a) *BPT* **(b)** *PM*

Figure 2.7: *In BPT (left), SDS paths can only be captured by unidirectional techniques, that is Path Tracing or Light Tracing. Sampling one of the few outgoing directions at the diffuse surface, that will actually intersect the light, is very unlikely for small area lights and impossible for point lights. Photon Mapping (right) reuses light paths that happen to hit the diffuse surface.*

vertices and the photons within the radius are treated as if they lay at the same point, they are "merged". The algorithm is illustrated in 2.8(b). Figure 2.7(b) illustrates how photon mapping can handle SDS paths efficiently, by reusing paths. Even if only a few photons hit a diffuse surface, their contributions can be used for many camera paths.

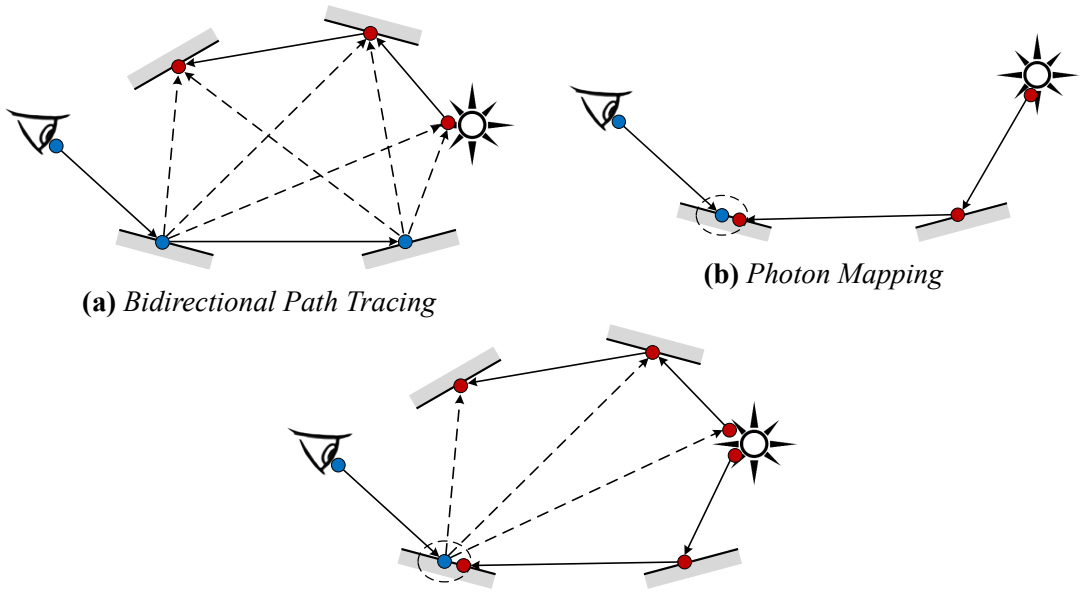
Merging vertices within a non-zero radius introduces bias. Progressive Photon Mapping (PPM) [HOJ08] makes Photon Mapping consistent by reducing the radius over time, hence progressively reducing the bias as well.

Unfortunately, (Progressive) Photon Mapping is inefficient at handling diffuse and glossy surfaces, especially if illuminated by far away light sources. The photon density on these surfaces is usually not high enough. Even after five minutes, diffuse and glossy surfaces are still very noisy, as can be seen in Figure 2.9. Multiple approaches exist to alleviate this problem, for instance Stochastic Progressive Photon Mapping [HJ09] and Bidirectional Photon Mapping [Vor11], but none of those can handle all types of paths well, especially not if glossy materials are involved. It would be desirable to find a more robust algorithm, that combines the benefits of BPT and PPM, whithout suffering from any of the drawbacks.

2.3.4 Vertex Connection and Merging

Vertex Connection and Merging (VCM) [Geo+12] uses multiple importance sampling to combine Bidirectional Path Tracing and Progressive Photon Mapping. Hence the name Vertex Connection (BPT) and Merging (PPM). The same algorithm, with a slightly different derivation, was also proposed by [HPJ12] under the name Unified Path Sampling.

Figure 2.8(c) illustrates the path sampling during VCM. The camera vertices are connected to the vertices of one light path, and merged with the vertices of all light paths within the radius. Figure 2.10 shows that VCM can handle SDS paths as well as Progressive Photon Mapping, while also producing less noise on diffuse and glossy surfaces. Because Progressive Photon Mapping is biased but consistent, VCM is also biased but consistent.



(a) *Bidirectional Path Tracing*

(b) *Photon Mapping*

(c) *Vertex Connection and Merging*

Figure 2.8: *Illustrations of paths that would be sampled by BPT, (P)PM, and VCM.*

2.3.5 Partial MIS Weight Evaluation

A major problem of BPT and VCM is that evaluating the MIS weights requires knowledge of how the path was sampled at every vertex. In other implementations, that was often solved by iterating over the path. However, that is not very efficient, especially not in a parallel implementation.

[Geo12] showed that it is possible to compute the MIS weights based on only the information that is stored in the last vertex of a path. Three quantities are stored in every vertex of both the camera and the light paths, and they are updated after every scattering step. Out of those quantities, the total MIS weight can be computed. Computing the MIS weights this way executes the same instructions on every vertex, which makes GPU implementations more efficient and also allows to compute the weights using SIMD on the CPU.

With this partial evaluation scheme it is unnecessary to store any information regarding the path structure. Only the vertices themselves are needed, which enables some additional optimizations in the implementation.

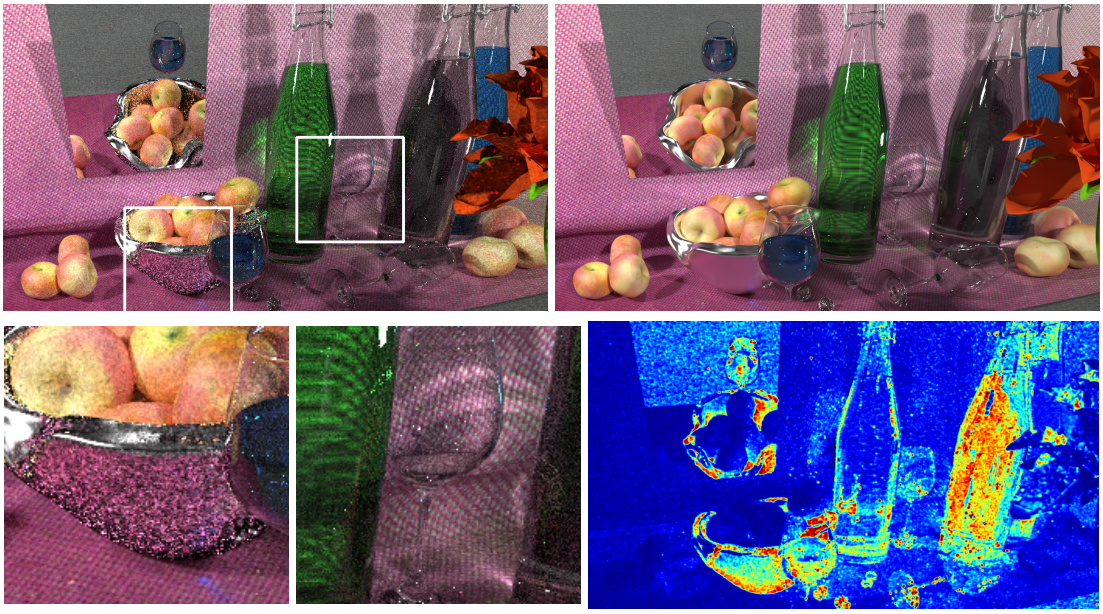


Figure 2.9: *The image rendered with PPM after five minutes (left) compared to the reference (top right). The structural similarity difference image (bottom right) highlights the problems of PPM. Although the caustics seen through the glass are very converged, the glossy and diffuse surfaces are still very noisy, as are their reflections.*

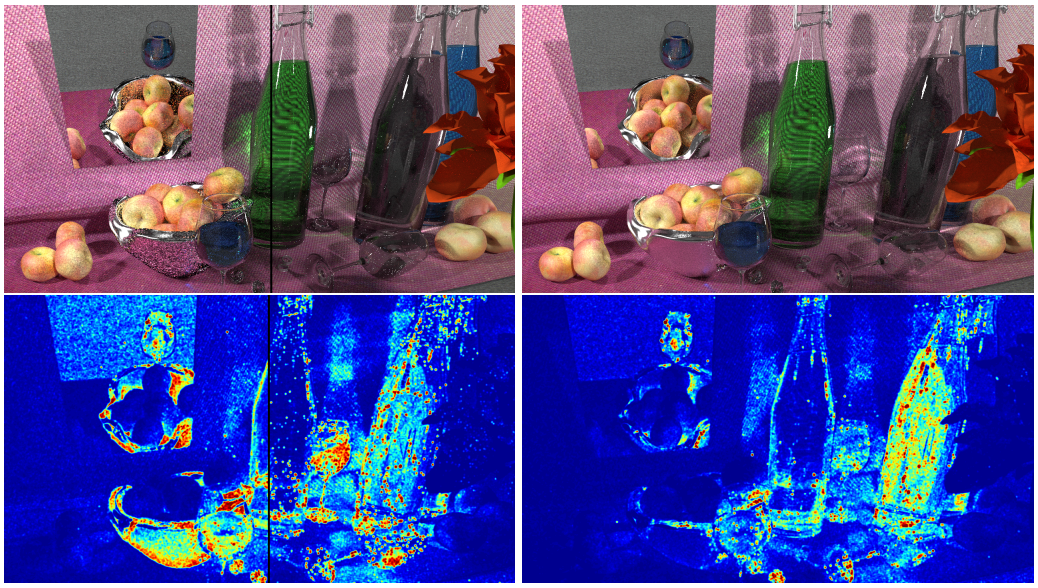


Figure 2.10: *VCM (right) combines the benefits of BPT and PPM (left), without suffering from any of the drawbacks, except for the bias from the Photon Mapping. The SSIM images at the bottom highlight the parts where the individual algorithms are inefficient.*

Chapter 3

Performance

There are two common approaches to achieving fast rendering times. Finding better algorithms that require fewer samples, as described in the previous chapter; and implementing an algorithm in a way that reduces the time per sample. In this thesis, we used both approaches. We implemented an efficient algorithm, Vertex Connection and Merging, in a parallel renderer, using a fast scene traversal library. The following sections briefly summarize some background information and related work on high performance implementations of those algorithms.

3.1 Efficient Implementations

At first glance, ray tracing appears to be trivial to parallelize. The abundance of rays that can be generated and processed independently from each other lends to this belief. Unfortunately, things are not that simple. Today's hardware relies a lot on data parallelism, that is, applying the same operation on multiple values at the same time. This is called single instruction, multiple data (SIMD) on the CPU, and single instruction, multiple thread (SIMT) on the GPU. Thus, efficient ray tracing on both the CPU and the GPU requires tracing many coherent rays at the same time. In this way, the same operations can be applied to those rays at the same time. While on the CPU this can be achieved by tracing hundreds of rays in parallel, fully utilizing the GPU requires millions of rays.

3.1.1 Taking Advantage of the CPU

Achieving interactive frame rates with ray tracing on the CPU was first done by tracing coherent packets of rays and using SIMD to process the rays inside those packets in parallel [Wal+01]. This approach does not work very well for Path Tracing and other global illumination techniques. The coherence of the rays inside a packet decreases with every bounce. To account for this problem, bounding volume hierarchies with a branching factor of more than two were used. With those, a single ray can be intersected with multiple bounding boxes in parallel [WBB08]. The current state of the art in terms of traversal performance on the CPU, Embree [Wal+14], uses both approaches.

3.1.2 Taking Advantage of the GPU

The GPU, with its SIMT approach to parallelism, relies heavily on processing a huge amount of coherent data. The smaller cache size, larger bandwidth, and reliance on latency hiding methods make the GPU quite different from the CPU. An implementation that works well on the CPU does not necessarily work well on the GPU, and vice versa. [LKA13] summarized those differences and showed how Path Tracing can be done efficiently on the GPU, even in the presence of costly materials. Currently, OptiX [Par+10] is considered the fastest GPU ray tracing framework. [Dav+14] compares the performance of a number of GPU implementations of Path Tracing, Bidirectional Path Tracing, Photon Mapping, and VCM.

3.1.3 The AnyDSL Framework

It is often desirable to implement a renderer, or another high performance application, on different kinds of hardware, for instance on the GPU and the CPU. Mapping the same application to different hardware requires special case code. Abstracting this mapping from the rest of the renderer is very desirable. In most programming languages, expressing such abstractions is very difficult and creates code that is very hard to understand and maintain. For instance, in C++ template metaprogramming can be used for this purpose [GS08], but the code tends to become very long and very messy. AnyDSL [Any] has the goal to allow to efficiently and cleanly express such abstractions. It also promises many additional benefits, like vectorization and partial evaluation. AnyDSL is based on its intermediate representation, Thorin [LKH15], and offers a programming language called Impala, a dialect of Rust.

Currently, the AnyDSL framework is not yet mature enough to implement a complex renderer. It lacks essential things like polymorphism. Compilation times for large code are very long, and debugging is difficult. However, an efficient scene traversal library was implemented in Impala. The traversal library runs on both the CPU and the GPU. It can compete with the state of the art on both platforms, although the development time was quite short and the code is comparatively simple and clean.

In this thesis, we use the Impala traversal library to implement a parallel renderer, that may also serve as a prototype and reference for a potential future implementation of an entire renderer in Impala.

3.2 The Traversal Library

The traversal library processes lots of rays in parallel. It expects an array of rays as input and returns an array with the corresponding hit points. For efficient ray intersection tests, a bounding volume hierarchy (BVH) is used. The BVH is passed to the traversal, along with some other scene information.

On the CPU, the traversal uses a multi-branching bounding volume hierarchy (MBVH) [WBB08]. A MBVH is a BVH with a branching factor of more than two. Currently, a branching factor of four is used. Groups of eight rays are intersected with all four children

of a node or with the contained triangles, using SIMD. Additionally, multiple threads are processing groups of rays in parallel.

The GPU traversal uses a standard BVH. Blocks of 64 rays are intersected with the nodes and triangles of this BVH. The rays within such a block are processed in parallel and multiple blocks may also be processed in parallel.

The (M)BVHs were built using spatial splits [SFD09]. We used settings favoring runtime performance over building times. Furthermore, we stored precomputed (M)BVHs for our scenes, to simplify testing.

The traversal library offers two different traversal functions. One for traversing primary rays, and another for traversing shadow rays. When traversing primary rays on the one hand, it is necessary to find the closest intersection. For shadow rays on the other hand, it suffices to determine if there is any intersection at all.

Chapter 4

Implementation

The renderer was implemented in C++, using the Impala traversal code as a library. The focus of the implementation was on finding a promising and scalable parallel design. We made sure that most parts of this design can also be implemented on the GPU in an efficient way. Thus, the renderer can also serve as a starting point and reference for a future implementation in Impala.

Vertex Connection and Merging (VCM) was chosen as the rendering algorithm for two reasons. Because it is a combination of the most common rendering algorithms, and because it is robust. From an implementation point of view, VCM is attractive because insights gained from a high performance implementation of VCM often also apply to the sub-algorithms. Apart from that, a renderer using a robust algorithm can handle a large variety of scenes, which is a desirable property.

In order to make efficient use of the traversal, thousands of rays have to be processed at the same time. Thus, the traditional approach of tracing one complete path at a time cannot be used. Instead, thousands of paths have to be traced at the same time. Every traversal step extends all those paths by another ray. This process is often referred to as wavefront Path Tracing.

The implementation combines two sources of parallelism. The traversal processes lots of rays in parallel, and the shading processes the resulting hit points in parallel as well. We refer to this as the low-level parallelism, because the individual tasks that can be parallelized are many and small.

Even on the CPU, the low-level parallelism alone is not enough to fully utilize all cores all the time, because the traversal and the shading depend on each other's output. The second source of parallelism, the high-level parallelism, further increases the CPU utilization. The high-level parallelism is based on processing sets of rays in parallel. When using the CPU traversal, the high-level parallelism improves load balancing. When using the GPU traversal, the high-level parallelism helps to keep the CPU busy, while the GPU is traversing rays. Two different approaches were experimented with. They are described in Section 4.3.3.

4.1 Overview

Figure 4.1 illustrates the structure of the renderer. The renderer is split into three separate libraries. The frontend, the traversal, and the main renderer form individual libraries. The frontend manages the user interface, loads the scene, and initializes the integrator for rendering. It also builds the acceleration structure (BVH or MBVH) or loads it from a file and, if using the GPU traversal, transfers the scene data to the GPU. The main renderer consists of the ray generation, the integrators, the material system, the ray queues, and the schedulers.

The integrators implement the rendering algorithm. Two integrators are provided: one for Path Tracing and one for VCM. The VCM integrator can be restricted to any of its sub-algorithms. Path tracing was implemented separately to serve as a reference.

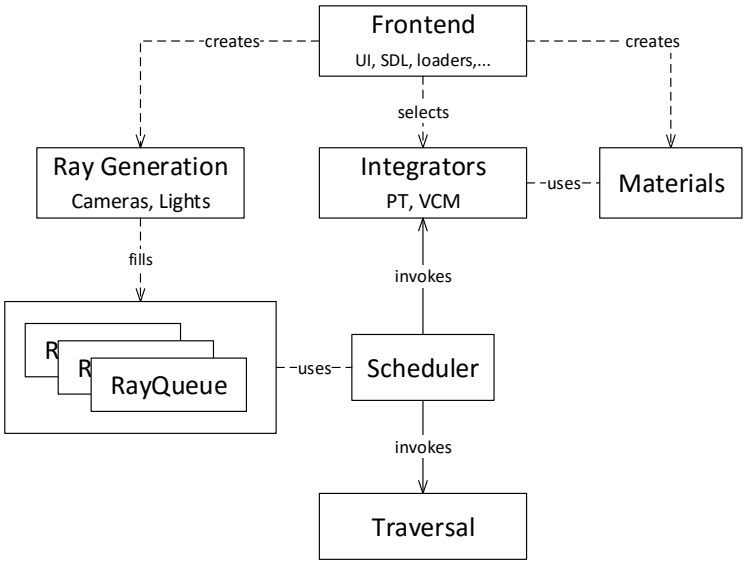


Figure 4.1: *The architecture of the renderer. Interacting with the user, loading the scene, and building the acceleration structure happens within the frontend. The integrators implement the actual rendering algorithms. The scheduler implements the high-level parallelism and provides the queues of rays and hit points for the integrator and the traversal.*

Paths are traced in a wavefront fashion. The process is illustrated in Figure 4.2. Initially, the camera or the light sources are sampled to obtain a set of primary rays. Those rays are then traversed, generating a set of hit points. Shading the hit points might create continuation rays. The continuation rays will again be traversed, and their hit points will be shaded, and so on. The process is repeated until there are no more rays left, that is, until all paths are terminated. Shading may also generate shadow rays. Shadow rays do not create more rays, and traversing shadow rays is faster, because the exact hit point location does not have to be determined. Therefore, our implementation separates shadow rays from primary rays.

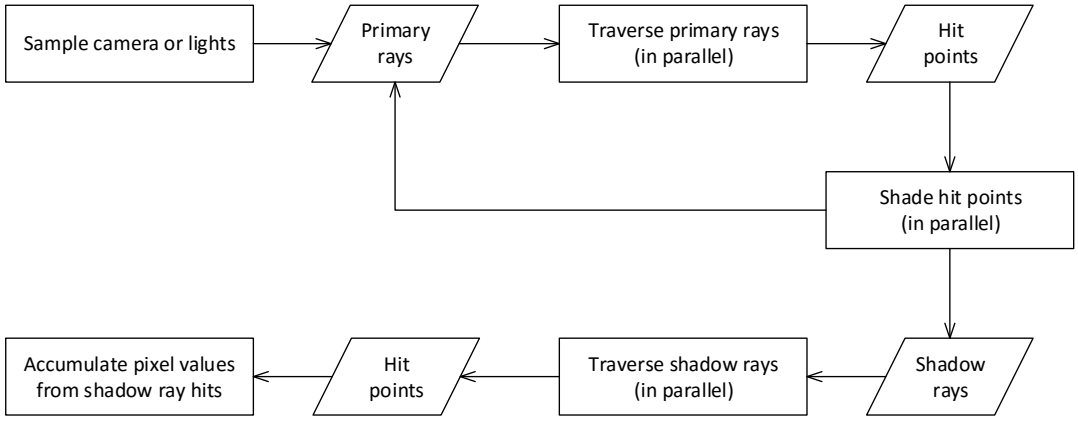


Figure 4.2: *The rendering process. Queues of rays are filled and traversed in parallel. The resulting hit points are also shaded in parallel. Shading generates continuation rays and shadow rays. Shadow rays are traversed and shaded separately for efficiency.*

4.1.1 The Material System

The material system is used to evaluate and sample the bidirectional scattering distribution function (BSDF). The design of the material system is based on a simplified and improved version of the system in PBRT ([PH10], pages 423 to 499). The material at a specific hit point is represented by a BSDF object, which consists of one BRDF object and one BTDF object. These three objects together store all relevant data at the hit point (for example the texture color, the index of refraction, and the Fresnel term). Storing all necessary information in the BSDF objects allows precomputing expensive operations, like texture lookups. Hence, evaluating the same BSDF multiple times is more efficient. A thread-local memory arena is used to store the BSDFs in an allocation free manner, allowing BSDFs of arbitrary sizes. Figure 4.3 illustrates the design using a simple glass material as an example. The actual material object, assigned to a primitive in the scene, stores the textures and other general information. The material object is used to create the BSDF objects for one specific hit point.

The evaluation method of the BSDF object selects the corresponding method in the BRDF or BTDF object, depending on whether the incoming and outgoing directions are on the same side of the surface. When sampling a direction from the BSDF, importance sampling is used to determine whether to sample the BRDF or the BTDF. The probability for importance sampling is determined by the BTDF. This kind of importance sampling was much simpler to implement by restricting our system to only one BRDF and one BTDF per BSDF. The system is still as expressive as the one in PBRT, because we instead combine multiple BRDFs or BTDFs into a single BRDF or BTDF.

4.2 The VCM Integrator

The structure of the renderer is best understood by looking at the implementation of an integrator. In this section, we discuss the most complex one, the VCM integrator. VCM is implemented in two passes. Each pass follows the structure outlined in Figure 4.2.

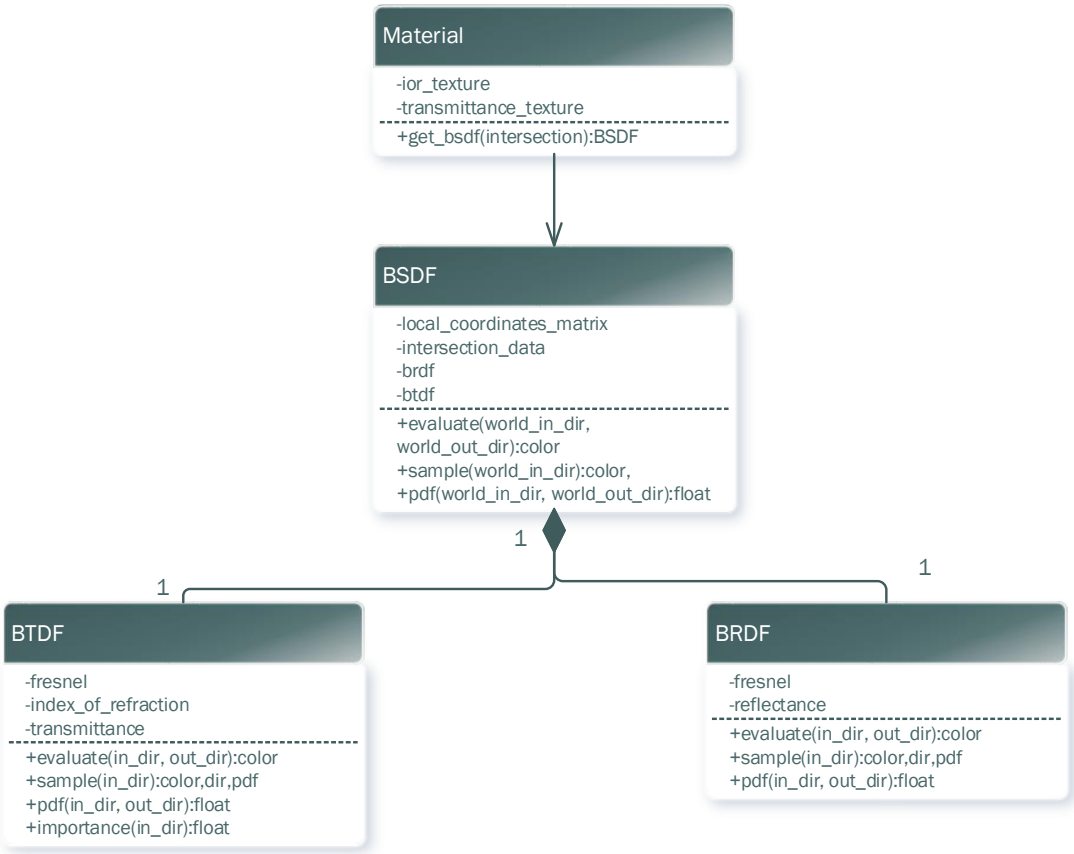


Figure 4.3: *Components of the material system for the example of a simple glass material. Materials store the textures, constant colors, and other general data. BSDF objects contain the texture values and other information specific to one hit point. Storing the data in that way reduces the number of texture lookups and other costly (memory) operations.*

The two passes differ only in what happens during shading, and what kinds of rays are generated. The first pass traces the light paths, and the second pass traces the camera paths. Because we are processing thousands of paths at the same time, rather than one path at a time, we need to somehow associate some state information to those paths. Figure 4.4 shows the data that is stored in the state of a path from the VCM integrator. The data is the same for both camera and light paths. The states of all paths are stored along with the last ray of that path inside the ray queue.

```
1  stuct VCMState {
2    // Identifies the path via the pixel
3    // and pixel sample that it belongs too.
4    int pixel_id;
5    int sample_id;
6
7    // Every path has its own random number generator state.
8    RNG rng;
9
10   float4 throughput;
11   int path_length : 7;
12
13   // Indicates whether the path originated on a directional light source
14   // Only used during light tracing.
15   bool started_on_dir_light : 1;
16
17   // Russian roulette probability for continuing this path.
18   float continue_prob;
19
20   // The partial weights for MIS.
21   float dVC;
22   float dVCM;
23   float dVM;
24 };
```

Figure 4.4: *The state data that is stored along with every ray from VCM.*

The first pass of a VCM frame traces the light paths. Light sources are sampled for as many rays as there will be camera rays. Hence, for each camera path there is a light path to connect to. The rays sampled from the lights are traversed and shaded until all paths are terminated, as shown in Figure 4.2. Every hit point (vertex) is stored, except for hit points on specular surfaces. Connecting to, or merging with, a vertex on a specular surface makes no mathematical sense, as a randomly chosen pair of directions has zero probability to have a non-zero contribution. Apart from storing the vertices, the partial MIS weights are updated, and shadow rays towards the camera are computed. The contribution of non-occluded shadow rays is stored in the throughput of its state. Light paths are terminated if their throughput reaches zero, or with Russian Roulette. If the path is not terminated, a continuation ray is sampled. The continuation rays are traversed and processed the same way as the primary rays. Thus, they might also create shadow rays and continuation rays, until all paths are terminated.

Once all light paths have been traced and their vertices stored, the range search structure

for photon mapping is built. We used an adapted version of the hash grid from [Dav]. Optimizing the photon mapping performance was not part of this thesis, although a limited number of obvious optimizations were made. Building the photon map is not parallelized – at least not on a per photon basis, that is, not with low-level parallelism – but multiple search queries are executed in parallel.

The second pass traces the camera paths. The process is the same as with the light paths. Only instead of connecting every hit point to the camera, they are connected to a point on a light source and the vertices on a light path, and then merged with all light vertices within the radius.

4.3 Improvements

Thanks to the fast traversal, even the first version of our implementation performed quite well. The improvements described in this section increase the performance even further. All improvements combined achieved a speed-up of up to a factor of four in our tests. The following sections describe the most interesting improvements.

The light vertex cache, described in section 4.3.1, simplifies storing the light path vertices and thus improves the performance of the photon mapping implementation. It also allows to control the number of vertex connections.

A frame in our implementation is an image consisting of one or more samples per pixel. Allowing multiple samples per frame improves coherence and ray count, and thus traversal performance. There are also other cases where this can cause a significant speed-up. The benefits and pitfalls are discussed in section 4.3.2.

The most important factor for performance is the high-level parallelism. It is discussed in section 4.3.3.

4.3.1 Light Vertex Cache

The light vertex cache was introduced by [Dav+14] as a technique that significantly increases the performance of BPT and VCM implementations on the GPU. Instead of storing entire light paths, the light vertex cache stores only the vertices of the paths. The partial MIS weights, described in section 2.3.5, made this idea possible. Storing all light vertices in a simple array or `std::vector` simplifies the code significantly. As our photon mapping hash grid stores only references to the actual light path vertices, using the vertex cache gave a huge speed-up for large radii, due to simplified iteration over all photons and better memory coherence.

The major benefit of the light vertex cache in a CPU implementation, however, is that the number of connections can be configured. Instead of connecting all the vertices of a camera path to all the vertices of one randomly chosen light path, the camera vertices are connected to a given number of vertices chosen randomly from all the vertices within the cache. This can be thought of as first randomly choosing a light path, with a pdf proportional to the number of vertices in the path, and then selecting one vertex from this path with uniform probability. Mathematically, this means that a conversion from the old

random process, selecting one light path out of N_p paths with uniform probability, to the new one is required. The new pdf for choosing a vertex is simply $p(x) = \frac{1}{N_v}$, where N_v is the number of vertices in the cache. Thus, the conversion factor is given by

$$c_{vc} = \frac{1}{N_c} * \frac{N_v}{N_p} \tag{4.1}$$

where N_c denotes the number of connections that were made, that is, the number of samples that were drawn.

The light vertex cache has a fixed size. This was a necessity when implementing it on the GPU. Considering that memory allocation is costly, it is also a reasonable idea on the CPU. However, the cache has to be large enough to store all vertices created during one iteration. Otherwise, some vertices will have to be discarded, introducing bias. Setting the size of the cache to the average light path length times the number of light paths and adding a ten percent safety margin accomplishes this goal. There are usually more than one million light paths traced per iteration. Statistically, it is almost impossible that the total number of vertices during any iteration is larger than this cache size. [Dav+14] reported that with this size, they never had to discard a single vertex. We confirmed this in our convergence tests, which lasted for many hours. The cache size can be computed once per scene in a preprocessing step. Pseudocode is given in Listing 4.5. A small number of light paths is traced through the scene, and the number of vertices that those paths would create is counted and averaged over the number of paths. This step requires only a few milliseconds.

```
1  def preprocess():
2    rays = sample_light_rays(10000)
3    vertex_count = 0
4    while len(rays) > 0:
5      hits = traverse(rays)
6      rays = []
7      for hit in hits:
8        if hit.valid:
9          vertex_count++
10         if russian_roulette():
11           rays.append(bounce(hit))
12
13   avg_len = vertex_count / 10000
14   allocate_lvc(avg_len * path_count * 1.1)
```

Figure 4.5: *Pseudocode for the function that computes the size of the light vertex cache. The function is called only once per scene. It is quite simple and also very fast.*

Tweaking the number of connections, N_c, allows for a high grade of flexibility. Whereas a small N_c increases the frame rate, larger values reduce the variance and thus the noise in a single frame. However, accessing randomly selected elements of a very large array, especially when done within multiple threads, does not go nicely with the cache-centered performance of the CPU. Due to frequent cache misses, the performance decreases rapidly for $N_c > 1$.

Table 4.1 compares the different numbers of connections for BPT in terms of performance and root-mean-squared error (RMSE) after 30 seconds. Using more than one connection hurts performance. Even for the simple Cornell Box scene, where the costs of traversing the additional shadow rays is negligible, the slow-down is comparable to the more complex Sponza and Still Life scenes. Thus, the problem is not the traversal of the shadow rays, but rather the code that creates those rays. Profiling confirmed that cache misses are the bottleneck. Multiple connections require multiple vertices from the light vertex cache. As it is very unlikely that randomly selecting two vertices yields two adjacent ones, this is likely to cause cache misses.

Using only a single connection increases performance significantly. More connections yield only a small improvement in quality after the same number of samples. The difference between one and eight connections after the same number of samples was only visible in the Sponza scene. However, even for the Sponza scene, the additional time per sample from using more connections yields worse results overall. Thus, we used a single connection in all our tests.

Count	Samples Per Second	RMSE
1	0.98 M	838
2	0.80 M	847
4	0.16 M	909
8	0.41 M	1075

(a) *Cornell Box*

Count	Samples Per Second	RMSE
1	0.68 M	5560
2	0.61 M	5560
4	0.51 M	5687
8	0.39 M	5929

(b) *Still Life*

Count	Samples Per Second	RMSE
1	0.51 M	3991
2	0.45 M	3837
4	0.35 M	3925
8	0.25 M	4208

(c) *Sponza*

Table 4.1: *The tables compare the performance in terms of samples per second to the root-mean-squared error (RMSE) after 30 seconds. Fewer connections yield better convergence rates.*

4.3.2 Multiple Samples

For optimal traversal performance, it is important to have many coherent rays. Rendering more than one sample per pixel during every frame increases both coherence and ray count. Our implementation supports a (theoretically) arbitrarily high number of samples. How many samples can be processed is only limited by the available memory for storing the rays and hit points. By using atomic operations to add the sample contributions to the final image, we can support multiple samples per frame without any (measurable) synchronization overhead.

Because we are processing the rays in parallel, those rays, their states, and their hit points have to be stored in memory at some point. Using multiple samples per frame increases the number of rays, and thus also the memory consumption. Because the memory available on the GPU is often smaller than the CPU memory, we made sure to only allocate as much

memory on the GPU as is needed to traverse the largest possible queue of rays. When using the tile scheduler (described below), for instance, this means that we only allocate enough memory on the GPU to store all the rays of a single tile. This allowed us to use much more samples per frame with the GPU traversal.

Building the hash grid that is used for photon mapping is not parallelized on a per-photon basis. With more than one sample per frame, we also build more than one hash grid per frame. Those hash grids can be built in parallel, which improves the CPU utilization significantly. Figure 4.6 shows the CPU utilization of VCM, using the tile scheduler, with one and with four samples per frame. Building four hash grids in parallel occupies all four cores of our CPU, whereas with a single sample, all cores are idle but one.

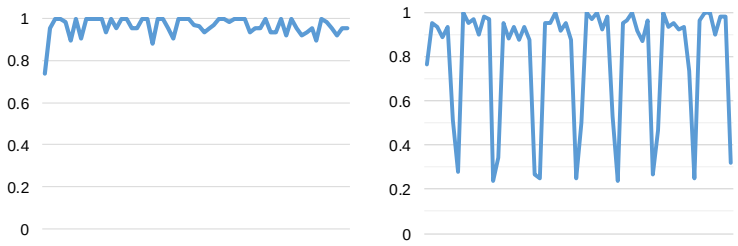


Figure 4.6: *The CPU utilization for VCM with the GPU traversal is significantly better when using multiple samples for one frame (left), compared to using just a single sample (right). Both versions use the tile scheduler and the GPU traversal. Every drop to 25 percent (one core) in the right chart occurs when building a photon map.*

Figure 4.7 shows how the performance, in terms of rays per second, changes by increasing the number of samples. Path Tracing always benefits from additional samples, because the rendering times are dominated by the traversal. VCM, on the other hand, actually got slower when using too many samples. The number of light vertices that have to be stored increases with every sample. Thus, having too many samples causes cache misses or even page faults, which hurts performance. On our hardware, using four samples per frame yielded the best performance for VCM. That was not very surprising, considering that we were using a CPU with four cores. Fewer samples would be less efficient while building the photon map, whereas more samples would produce too many light vertices.

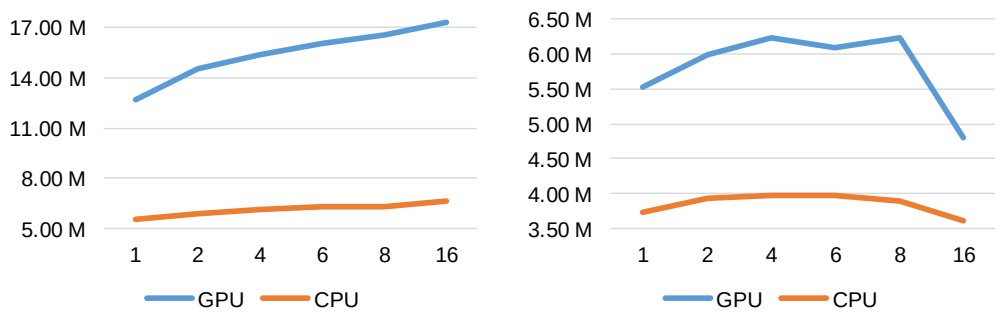


Figure 4.7: *While the performance of the Path Tracer (left) increases with every additional sample per frame, with VCM (right) it is possible to have too many samples.*

4.3.3 Schedulers

The low-level parallelism was realized with Intel Threading Building Blocks (TBB, [Inta]). TBB automatically schedules parallel tasks to the available cores. TBB was chosen because it mixes well with additional parallelism from other sources. The low-level parallelism, from processing hit points in parallel with `tbb::parallel_for`, is of course not enough to fully utilize the CPU when using the GPU traversal. With only the low-level parallelism, the CPU would be idle while traversing. The high-level parallelism, described in this section, occupies the CPU by shading the hit points from another set of rays, while waiting for the output from the GPU traversal.

The additional high-level parallelism is also beneficial when using the CPU traversal, although rays and hit points are already processed in parallel. Partitioning the rays into groups and processing those groups in parallel allows for better load balancing and latency hiding, thus further improving the CPU utilization.

There is a trade-off between the amount of high-level parallelism and the performance of the traversal. Whereas having a large amount of small sets of rays would be ideal for load balancing, traversal is only efficient for large numbers of coherent rays, especially on the GPU.

The high level parallelism is implemented by the schedulers. A scheduler manages a set of ray queues, fills them with primary rays, and invokes traversal or shading on them. We experimented with two different approaches to scheduling, which are described in the following sections.

Tile Scheduler

When partitioning camera rays, it is generally a good idea to split an image into (equally sized) tiles. The camera rays of neighboring pixels are very coherent and so are the hit points they produce. For load balancing, it is beneficial to have as many tiles as possible.

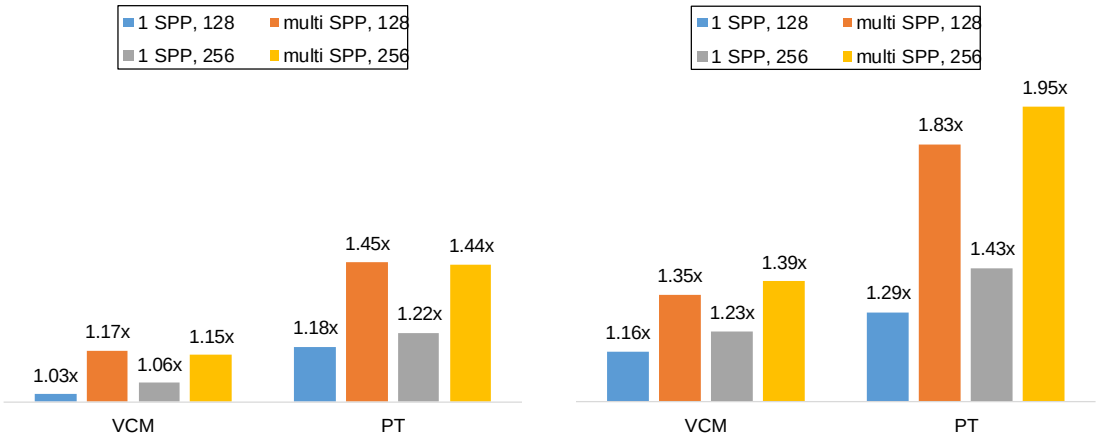


Figure 4.8: *Relative speed-up from the tile scheduler for the CPU traversal (left) and the GPU traversal (right). Different tile sizes and samples per pixel (SPP) are compared for Path Tracing and VCM. For VCM, one or four samples where used. Path Tracing used either one or sixteen samples.*

To maintain a high ray count at the same time, multiple samples per pixel can be used per frame. By adjusting the size of the tiles, the number of rays within a partition is easily controlled.

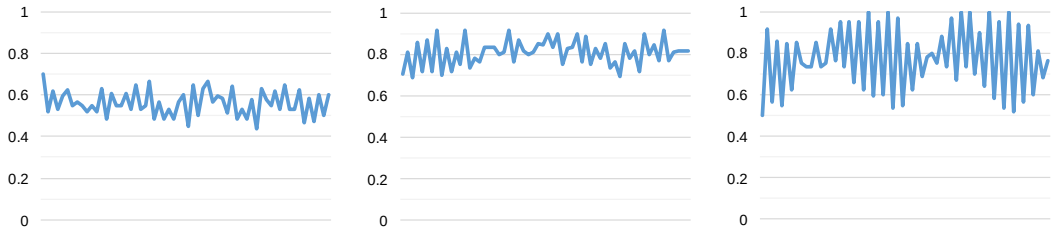
Something to keep in mind is that light paths are not directly related to the pixels of the rendered image. Thus, partitioning light paths into tiles does not yield more coherent rays. That means that the tile based scheduler is not necessarily the best solution for tracing light paths. However, in our experiments, the tile scheduler was also the better choice for light paths.

The tile scheduler maintains a pool of threads. Each thread selects the next available tile, by increasing an atomic counter. The rays within this tile are then traversed and shaded, until all paths are completed. The threads have been implemented using `std::thread`, and not TBB, because TBB tasks are not efficient for code that is frequently blocking or waiting for IO [Intb], which is the case when using the GPU traversal.

Figure 4.8 shows the speed up from using the tile scheduler, with varying tile sizes and samples per pixel, with the CPU traversal and the GPU traversal. Performance was measured for the Still Life scene, the behavior in other scenes is similar. While the Path Tracer benefits a lot from the tile scheduler, the benefit for VCM is smaller. The speed-up is larger when using the GPU traversal, as would be expected, because the CPU is no longer idle while traversing. The speed-up that was achieved with the CPU traversal is also surprisingly high.

The CPU utilization when using the tile scheduler for Path Tracing is plotted in Figure 4.9(b) and 4.11(b), for the GPU traversal and the CPU traversal respectively. The measurements were made on a CPU with four cores, using 256^2 tiles and one sample per frame. The plotted curves show that the tile scheduler offers a significant improvement. With the CPU traversal, the utilization is almost at 100%. There is still room for improvement with the GPU traversal, but it is very unlikely to achieve full CPU utilization, because the shading part is quite cheap during Path Tracing. The utilization for VCM is plotted in Figure 4.10(b) and 4.12(b). The regular drops to 25 percent utilization are caused by the single-threaded construction of the photon map. While the photon map is being built, nothing else can be done. Figure 4.6 shows the utilization if multiple photon maps are built in parallel. Note that the average CPU utilization is much higher with VCM than with PT when using the GPU traversal, even if no high-level parallelism is used. The reason for the higher utilization is that shading is way more expensive during VCM. VCM benefits less from the high level parallelism, because it already benefits a lot from the low level parallelism during shading.

The number of rays within each tile decreases over time. That reduces the efficiency of the traversal, especially on the GPU. A simple way to keep the ray count in every thread as high as possible for as long as possible is to merge tiles. Whenever the ray count drops below some threshold (50 percent has proven a decent value), another tile is acquired, if available, and the primary rays from that tile are added to the current rays. However, although this increases the number of rays and thus also reduces the calls to traversal, coherence is not ideal. In our test scenes, we experienced only a small speed-up (less than ten percent) from the merging. Merging only improved performance in combination with small tiles and only few concurrent samples. For larger tiles and more samples, there was no measurable change in performance.



(a) *No Scheduling* **(b)** *Tile Scheduler* **(c)** *Queue Scheduler*

Figure 4.9: *Path Tracing using the GPU traversal utilizes the CPU only for around 50%, since shading is quite cheap. The tile scheduler increases the utilization to 80%. With the queue scheduler, the utilization is fluctuating much more, but roughly the same on average.*

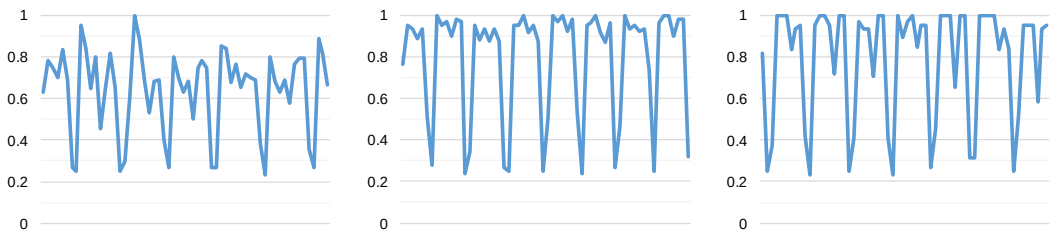
The memory consumption when using the tile scheduler increases with the number of threads, because each thread has to store all rays, states and hit points of the current tile. Every thread has three ray queues available: the currently processed queue, a queue for continuation rays and a queue for shadow rays. Thus, the memory usage is roughly three times the number of rays within one tile, per thread.

Queue Scheduler

The queue scheduler was an attempt to optimize performance with the GPU traversal even further. It is much more complex than the tile scheduler. Instead of having a fixed number of threads, each rendering a part of the image, the queue scheduler maintains a pool of queues, each of which can be either empty, in use, ready for shading, ready for traversal, or ready for shadow ray traversal. Queues are traversed in the main thread and shading is parallelized with TBB tasks.

The main benefit of this approach is that all rays are still managed in the main thread. This allows for many potential improvements. We experimented with traversing all available queues at once. Although that increased the GPU utilization, the increased time until more work became available for shading outweighed all benefits. Furthermore, the memory usage can be controlled by altering the number and size of the queues within the pool.

The queue based approach cannot be done entirely synchronization free, as the tile based one could. Because we only use a limited number of queues, to prevent excessive mem-



(a) *No Scheduling* **(b)** *Tile Scheduler* **(c)** *Queue Scheduler*

Figure 4.10: *For VCM with the GPU traversal, the tile scheduler increases the CPU utilization significantly. Again, the queue scheduler fluctuates more but performs similar on average. The sudden drops to 25 percent (single core) are caused by the single threaded build of the photon map.*

(a) *No Scheduling* **(b)** *Tile Scheduler* **(c)** *Queue Scheduler*

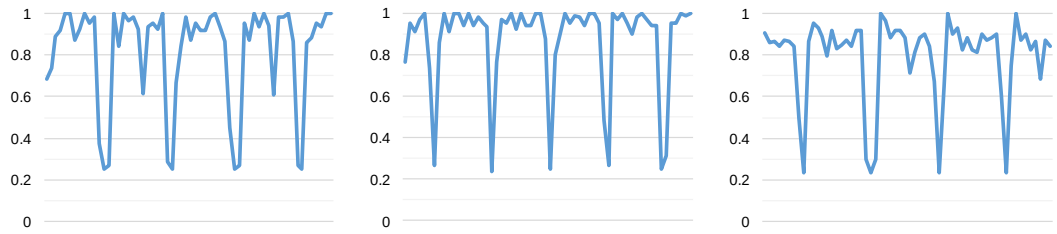
Figure 4.11: *Path Tracing with the tile scheduler and the CPU traversal achieves almost full utilization of all cores (90-100%). The sudden drops in performance that occur when not using any high-level parallelism are almost completely gone. The queue scheduler does not perform well in combination with the CPU traversal.*

ory usage, we often have to wait for queues to become available. This waiting proved a bottleneck. Only by setting the number of queues large enough such that we never had to wait for empty queues, could we achieve a level of performance that was comparable to the tile scheduler.

The speed-up from using the queue scheduler in the Still Life scene is given in Table 4.2. Overall, the performance with the GPU traversal is slightly worse than that from the tile scheduler. In combination with the CPU traversal, the performance can even be worse than not using any high-level parallelism at all. The queue scheduler consumes more memory than the tile scheduler, thus the possible number of samples per frame is also more limited. In combination with the Path Tracer, the queue scheduler achieved slightly faster frame rates. This can be seen by comparing the speed-up with only one sample per frame. Still, this minor increase in performance was not really worth the effort.

SPP	**Speed-Up GPU**		**Speed-Up CPU**	
	PT	*VCM*	*PT*	*VCM*
1	1.45x	1.10x	1.04x	0.96x
4	1.57x	1.00x	1.10x	0.96x

Table 4.2: *Speed-up when using the queue scheduler. The queue scheduler only works with the GPU traversal. With the CPU traversal, it is sometimes even worse than not using any scheduler at all.*



(a) *No Scheduling* **(b)** *Tile Scheduler* **(c)** *Queue Scheduler*

Figure 4.12: *Using the tile scheduler for VCM with the CPU traversal greatly increases the time spent at 100% utilization. The drops from building the photon map cannot be eliminated with high level parallelism. In combination with VCM and the CPU traversal, the queue scheduler achieves even worse utilization than not using any high-level parallelism.*

There are many potential improvements to the queue scheduler. However, they are very difficult and time consuming to implement, and it is uncertain whether the results could outperform the tile scheduler, at least in a CPU implementation.

4.3.4 Random Number Generator

Another minor improvement is to use a simpler random number generator than for instance a Mersenne Twister. The random number generator we used, MWC64X [Tho], can be implemented with only a few lines of code and requires only a small state (64 bits). Thus, the generator is especially useful our implementation, since we associate every ray on every path with its own random number generator state.

The MWC64X produces high quality random numbers, according to [Tho]. The convergence rates compared to the Mersenne Twister form the C++ standard library were the same in our experiments. Although the random number generator by itself is much simpler and faster, the overall impact on performance was negligible.

One major benefit from this random number generator is that it can easily be used on the GPU. Thus, in a future implementation in Impala, the same code for sampling can be used on both the CPU and the GPU.

Chapter 5

Discussion

We conclude the thesis by presenting the performance and convergence results. We analyze the results, determine the bottlenecks and propose improvements for future work. The Path Tracer and the VCM implementation both achieve interactive frame rates. Our analysis shows that images of decent quality can be rendered within five minutes, for all our scenes, using VCM. Additionally, the performance scales well with the number of cores.

5.1 Testing Setup

The computer used for testing was equipped with an Intel i5-4570, running at 3.20GHz, 16GB RAM, and a NVIDIA GeForce GTX 660 with 2GB of memory. Both the CPU and the GPU are unfortunately quite old, thus an increase in performance is to be expected when running on newer hardware.

Figure 5.1 shows the scenes that were used for testing. We only considered indoor scenes in our tests. Outdoor scenes are usually quite large and dominated by direct illumination. Thus, VCM is not really a good choice for them. Path Tracing is more efficient for this kind of scenes.

The **Still Life** scene is the most complex. It was created for the purpose of this thesis as a scene that is challenging for all algorithms, except VCM. The scene consists of 475K polygons. It features a combination of diffuse, glossy, and specular materials with a small number of textures. The scene is very challenging to render, because of the complex caustics. The caustics that are seen through the empty wine glass, for instance, are created by paths with at least ten specular and one diffuse bounce.

The **Sponza** scene is probably one of the most famous computer graphics test scenes. It consists of 262K polygons, with many textures and bump maps. There are no specular or glossy materials. The part of the scene that was used in our renderings is dominated by indirect illumination, leaking through small gaps around the curtains. Of the algorithms used in this thesis, BPT performs best in this scene.

Because shading performance does not depend on the geometric complexity, we also use multiple variations of the **Cornell Box** scene for testing. Each variant clearly showcases

a setting that is difficult for one or more algorithms, which gives interesting insights into the shading performance.

Figure 5.1: *The scenes used for testing. From top to bottom and left to right: Still Life, Sponza, and Cornell Indirect, Original, Specular Close, Specular, and Water*

5.2 Performance Results

The performance measured in rays per second and frames per second. Rays per second are less useful for comparing Photon Mapping and VCM, because the performance of those algorithms does not only depend on the number of rays. Since the resolution of all our

images is approximately one megapixel, the number of frames per second allows for a better comparison.

Table 5.1 summarizes the performance for all scenes and algorithms. As would be expected, Path Tracing is the fastest and VCM the slowest algorithm. The speed-up when switching from the CPU traversal to the GPU traversal is also given in the table. The following sections summarize the level of interactivity and the rendering times that were achieved. Afterwards, we discuss the scalability of our implementation and the remaining bottlenecks.

Scene	**Algo.**	**Rays per Second**		**FPS**		**Speed-Up**
Cornell Box	pt	21.93 M	16.83 M	3.4	2.6	1.30x
Cornell Box	bpt	13.38 M	11.65 M	0.8	0.8	1.09x
Cornell Box	vcm	6.50 M	6.04 M	0.4	0.4	1.06x
Cornell Box	ppm	6.57 M	5.94 M	1.1	1.0	1.09x
Cornell Specular	pt	18.67 M	13.61 M	2.3	1.7	1.35x
Cornell Specular	bpt	11.05 M	8.76 M	0.7	0.6	1.17x
Cornell Specular	vcm	5.74 M	5.07 M	0.4	0.3	1.33x
Cornell Specular	ppm	6.66 M	5.52 M	1.0	0.8	1.25x
Cornell Specular Close	pt	20.91 M	14.13 M	2.3	1.6	1.44x
Cornell Specular Close	bpt	13.29 M	9.89 M	1.0	0.7	1.43x
Cornell Specular Close	vcm	6.75 M	5.81 M	0.5	0.4	1.25x
Cornell Specular Close	ppm	8.28 M	6.61 M	1.1	0.9	1.22x
Cornell Indirect	pt	24.43 M	19.03 M	3.8	3.0	1.26x
Cornell Indirect	bpt	13.82 M	12.00 M	0.8	0.7	1.14x
Cornell Indirect	vcm	6.50 M	6.08 M	0.3	0.3	1.07x
Cornell Indirect	ppm	6.60 M	5.92 M	0.8	0.8	1.05x
Cornell Water	pt	17.57 M	9.59 M	2.7	1.5	1.83x
Cornell Water	bpt	11.83 M	6.52 M	0.8	0.4	2.00x
Cornell Water	vcm	7.07 M	4.79 M	0.5	0.3	1.66x
Cornell Water	ppm	7.02 M	4.49 M	0.9	0.6	1.50x
Sponza	pt	11.58 M	4.69 M	1.4	0.6	2.33x
Sponza	bpt	6.23 M	2.98 M	0.5	0.2	2.50x
Sponza	vcm	4.21 M	2.39 M	0.3	0.2	1.50x
Sponza	ppm	4.61 M	2.46 M	0.7	0.4	1.75x
Still Life	pt	12.37 M	5.36 M	1.8	0.8	2.25x
Still Life	bpt	9.55 M	5.01 M	0.7	0.4	1.75x
Still Life	vcm	6.49 M	4.03 M	0.5	0.3	1.66x
Still Life	ppm	7.42 M	4.59 M	0.7	0.5	1.44x

Table 5.1: *Performance measurements across all scenes for all algorithms, using the tile scheduler. The frames per second (FPS) speed-up from using the GPU traversal instead of the CPU traversal is given in the rightmost column. Test were run with an i5-4570 @3.20GHz, 16GB RAM, and a GTX 660 with 2GB.*

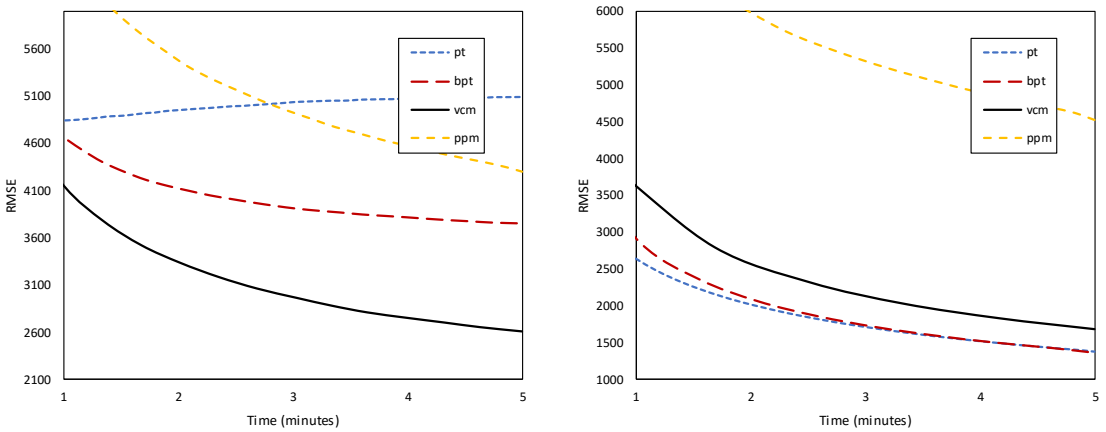
5.2.1 Interactivity

With both the CPU traversal and the GPU traversal, our implementation achieves frame rates that are high enough to be considered interactive, especially when using the Path Tracer. All our scenes were rendered at a resolution of roughly one megapixel. The Path Tracer ran at up to four frames per second, whereas VCM ran at around 0.5 frames per second. Considering that a single frame from VCM often looks much better than a single frame rendered with the Path Tracer, this result is acceptable. Half a frame per second is an acceptable amount of interactivity for quite a few applications. Newer hardware is very likely to increase the performance significantly. Furthermore, simple techniques, like reducing the resolution while moving the camera could be used to improve the frame rate.

5.2.2 Rendering Times

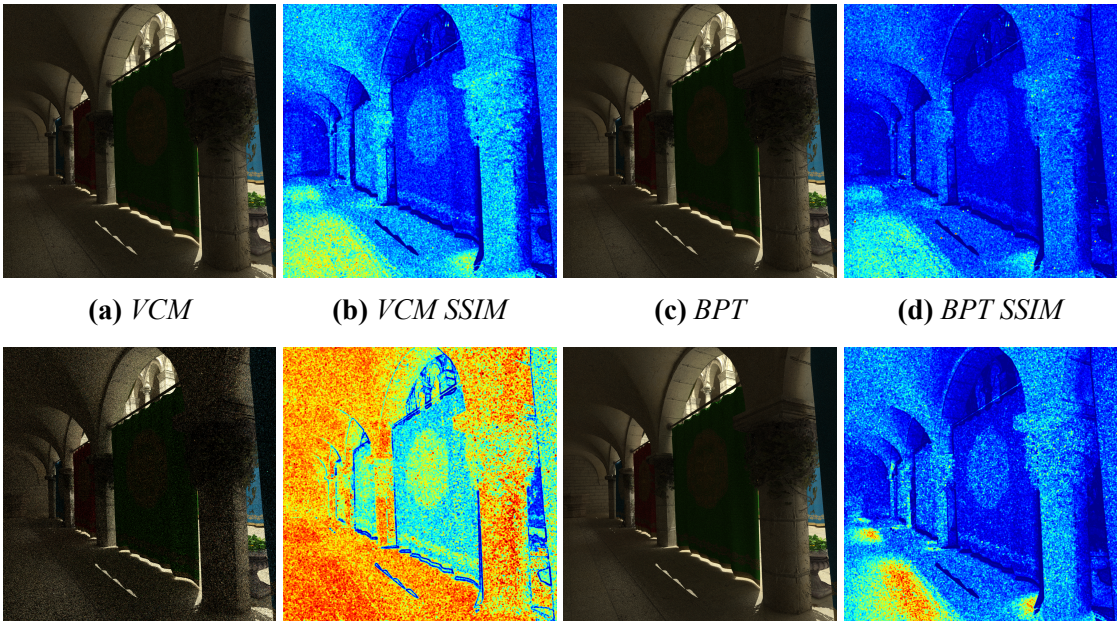
As could be seen in Table 5.1, Path Tracing is several times faster per sample, and per ray, than all other algorithms. Thus, it is still the best choice for maximum interactivity. However, the convergence curves in Figure 5.2 and 5.12 show that the other algorithms, in particular VCM, are often a better choice for offline rendering. The convergence charts, renderings, and structural similarity difference images in this chapter illustrate why implementing VCM was worth the effort. The images were rendered for five minutes, and the root-mean-squared error (RMSE) over time was computed. The differences in the RMSE convergence rates, as well as the visible differences in the images, show that VCM is a good choice, even in scenes without any specular materials.

In the Still Life scene, which was constructed to be difficult for all algorithms but VCM, the results are quite clear. VCM converges quickly, while the RMSE for Path Tracing is even increasing over time, due to the firefly noise from the caustics. The images after five minutes from the Still Life scene are shown in Figure 5.8, along with their structural similarity (SSIM) difference images. The image from VCM is already very close to the



(a) *Still Life* **(b)** *Sponza*

Figure 5.2: *Root mean squared error (RMSE) within five minutes for Still Life and Sponza. VCM outperforms all other algorithms in the Still Life scene. In the Sponza scene, VCM is not the best algorithm, but it is performing well enough.*



(a) *VCM* **(b)** *VCM SSIM* **(c)** *BPT* **(d)** *BPT SSIM*

(e) *PPM* **(f)** *PPM SSIM* **(g)** *PT* **(h)** *PT SSIM*

Figure 5.3: *The Sponza scene is handled well by all algorithms except for PPM. This is a good proof that VCM is not only efficient in scenes with complex caustics and SDS paths.*

reference. The caustics and most other parts of the image from BPT are also very converged. However, the glass between the bottles shows very clearly that BPT struggles with SDS paths: the refractions of the caustic are not visible, even after five minutes. While, according to the SSIM values, the SDS paths in the Progressive Photon Mapping (PPM) image are even further converged than in the VCM image, the glossy surfaces look horrible and the diffuse surfaces and their reflections are also very noisy. The worst result comes from Path Tracing. Here, the caustics consist only of firefly noise. Overall, the only algorithm that can efficiently handle the Still Life scene is VCM.

The images from the Sponza scene after five minutes, shown in Figure 5.3, prove that VCM is also efficient for scenes that do not contain any specular surfaces at all. PPM is still extremely noisy after five minutes, and Path Tracing is also suffering from a lot of noise in some areas. Although the RMSE of the image from VCM is higher than that from Path Tracing, the noise is more evenly distributed and thus less objectionable. Hence, VCM performs only slightly worse than BPT.

The images and RMSE curves for the Cornell Box scenes are at the end of the chapter. According to the RMSE over time, VCM is not always the best algorithm for the Cornell Box scenes. Yet, the difference to the best algorithms for those scenes is not very big.

Overall, VCM seems like a decent choice for rendering arbitrary scenes. Although for many scenes there are algorithms that converge slightly faster than VCM, VCM is never much slower than the best algorithm. In contrast, for scenes where VCM performs best, all other algorithms perform significantly worse.

5.2.3 Scalability

The combination of high-level parallelism from the schedulers and low-level parallelism from processing rays and hit points in parallel scales nicely. Curves for using up to four cores with the CPU traversal and with the GPU traversal are given in Figure 5.4 and Figure 5.5 respectively. With the CPU traversal, both the traversal and the shading benefit from the additional CPU cores. With the GPU traversal, however, the scaling is significantly worse, because additional CPU cores only improve the performance of the shading part. For scenes where connecting and merging vertices happens often, that is, scenes with only a few specular materials, we still experience a very good scaling for VCM and BPT, because shading is very expensive in such scenes.

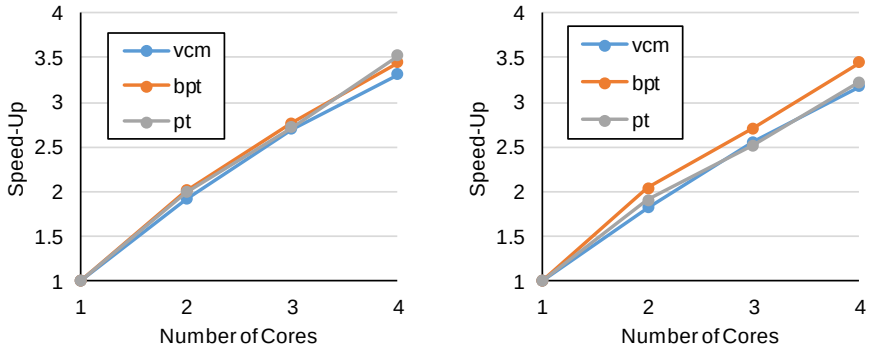


Figure 5.4: *Scaling curves for Sponza (left) and Still Life (right), using the CPU traversal. Because both, traversal and shading, are running on the CPU, all algorithms scale nicely.*

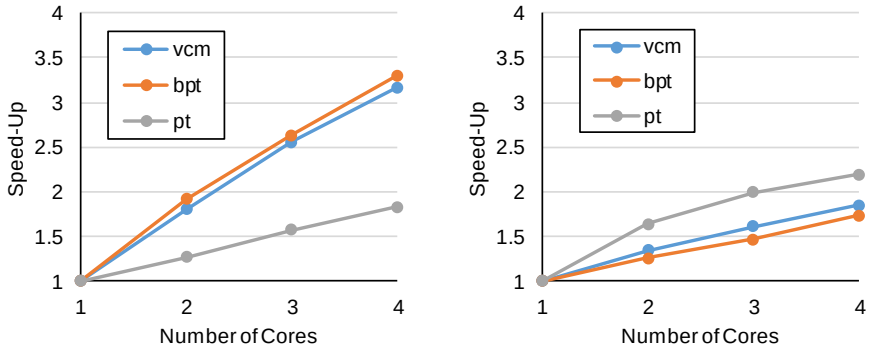
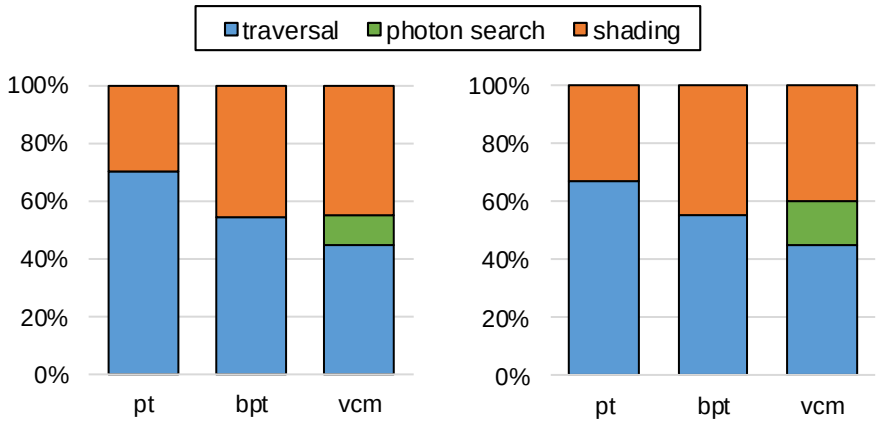


Figure 5.5: *Scaling curves for Sponza (left) and Still Life (right), using the GPU traversal. Traversal does not benefit from additional CPU cores. Thus, Path Tracing performance does not scale well. Especially in scenes with little or no specular surfaces, the shading costs for VCM and BPT are very high. Thus, the increase in performance from additional CPU cores is almost as big as with the CPU traversal.*

5.2.4 Bottlenecks and Possible Improvements

The amount of time spent shading, traversing, and building and searching the photon map was measured with the CPU traversal. The results are visualized in Figure 5.6 for the

Sponza and Still Life scenes. Path Tracing on one hand is mostly dominated by the traversal. Thus, Path Tracing benefits the most from using the GPU traversal. For VCM on the other hand, traversal makes up for less than 50% of the total rendering time. The shading is a bottleneck for VCM performance.



(a) *Still Life* **(b)** *Sponza*

Figure 5.6: *The figures show the amount of time, in percent, that was spent shading, traversing and photon mapping. Measurements where made using the CPU traversal. Whereas Path Tracing is dominated by traversal times, Bidirectional Path Tracing and VCM spend significantly more time in the shading code. The Photon Mapping step is more expensive if fewer specular materials are present.*

Figure 5.7 shows how the performance (in pixel samples per second) changes for different scenes. For Path Tracing, the results are closely related to the geometrical complexity, as would be expected. The other algorithms, however, have very interesting results. With VCM, the Cornell Water scene is actually faster to render than the simple diffuse Cornell Box, although it consists of more than 1000 times the number of polygons. The reason for this phenomenon is that connecting and merging vertices makes no sense on specular surfaces and is thus not performed. In particular, connecting is very slow because it suffers from cache misses. The slow-down from merging is slightly smaller. Hence, the results for Progressive Photon Mapping are somewhat closer to those from Path Tracing.

The results show that the cache misses from random accesses to the light vertex cache, when connecting and merging vertices, are indeed a problem. Reducing these cache misses could increase the performance significantly. It might be possible to randomly shuffle the contents of the light vertex cache in a way, that allows neighboring light vertices to be used for connecting, without introducing bias. Another option would be to alter the probabilities for choosing vertices in a way that they favor nearby vertices.

Currently, the hit points are processed in one big for loop. This loop computes the contributions from direct illumination and from connecting and merging vertices, and it samples a direction for the continuation ray. Splitting this loop into parts would increase the probability that the same memory is used at the same time, and thus increase cache performance. Whether or not this will have a (positive) impact on performance remains to be seen.

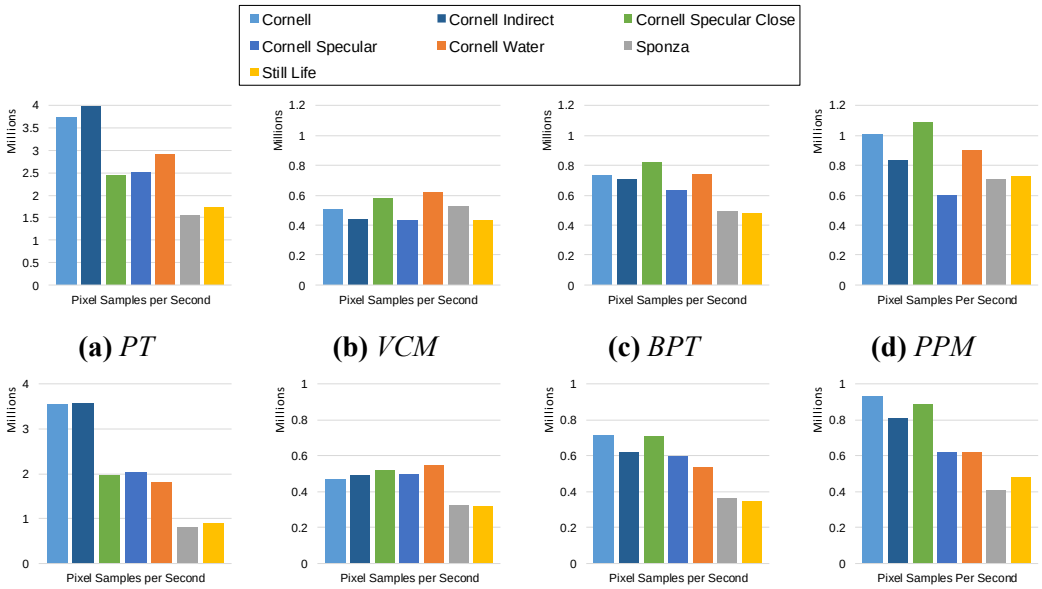


Figure 5.7: *The performance in terms of samples per second with the GPU traversal (top) and the CPU traversal (bottom) does not depend on the geometric complexity much, except with Path Tracing. That also proves that the shading code is the bottleneck for BPT, VCM and PPM.*

5.3 Future Work

Apart from the aforementioned possible improvements regarding cache performance, there are also some other interesting topics for future work.

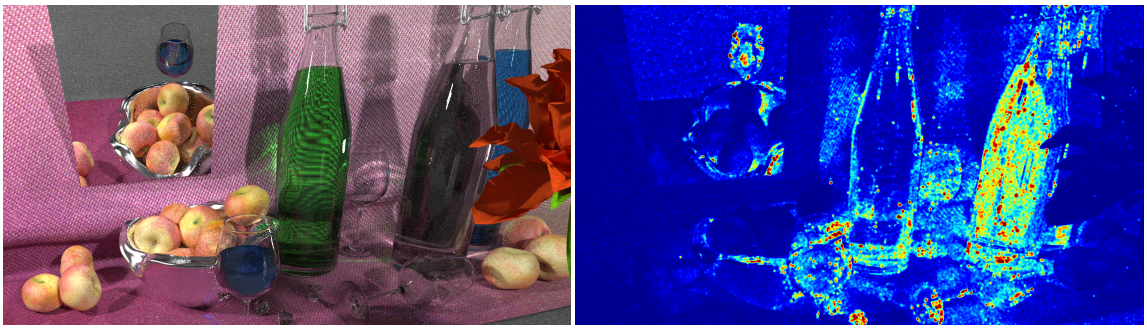
Using SIMD for the shading part could improve performance significantly ([Kar+10] for instance reported a speed-up factor of four with SSE). However, implementing SIMD by hand is very cumbersome and the goal of AnyDSL is to simplify that. Hence, it was left for future work.

Traversal performance in our implementation could benefit from sorting rays to increase coherence, especially on the GPU. Also, sorting hit points by material might increase performance, at least if using SIMD shading. The performance increases reported by others for sorting rays and hits [Eis+13] [LKA13] show that this would be a worthwhile direction for future work.

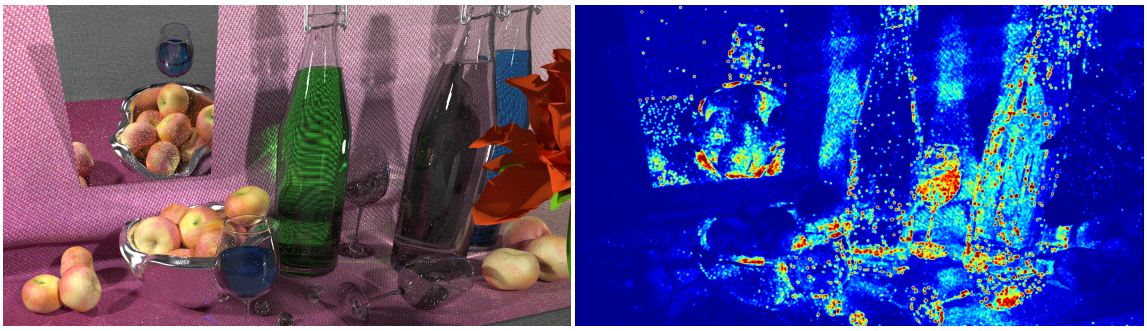
Although photon mapping makes up only around 15% of the rendering time in our scenes, it might be interesting to see how much a better photon mapping step could improve performance. For instance, by using the (rectified) stochastic hash grid, described in [Dav+14]. Building the photon maps for multiple iterations in parallel yielded a speed-up of up to 50%. Parallelizing the build on a per photon basis will likely improve this speed-up further and also increase the frame rates that can be achieved with VCM.

Achieving high performance for production scale scenes, maybe even with textures loaded over the network, would also be very interesting. Huge textures and models that cannot be kept in memory require an intelligent streaming approach. Finding an approach that fits well into the parallel design used by our renderer could be very interesting.

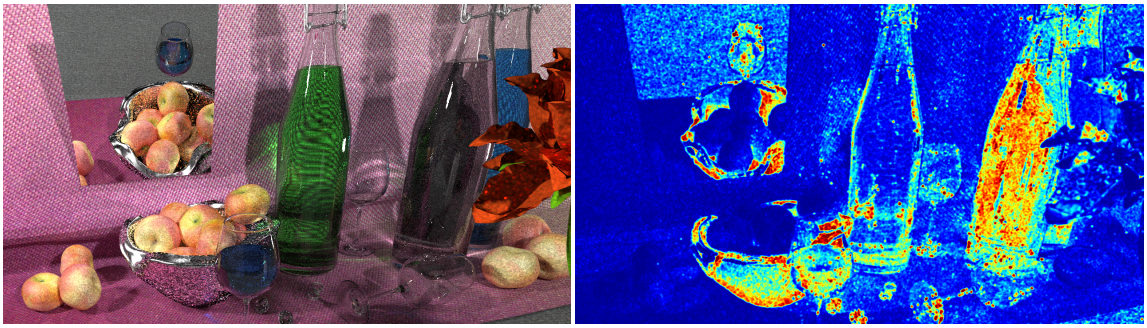
Apart from performance improvements, it is of course also desirable to extend the set of supported features. More complex materials should be easy enough to implement with



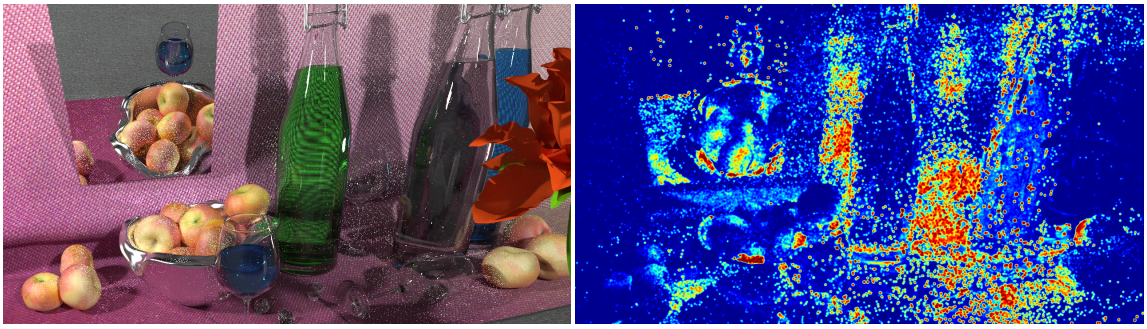
(a) *VCM* **(b)** *VCM SSIM*



(c) *BPT* **(d)** *BPT SSIM*



(e) *PPM* **(f)** *PPM SSIM*



(g) *PT* **(h)** *PT SSIM*

Figure 5.8: *The Structural Similarity (SSIM) values on the right show very well where the difficulties of the individual algorithms lie. VCM handles the scene very well overall. BPT struggles with SDS paths, PPM has problems with glossy and diffuse surfaces, and PT cannot render the caustics efficiently.*

our flexible material system. Other features, like volume rendering, still pose a problem for high performance implementations and may require more research.

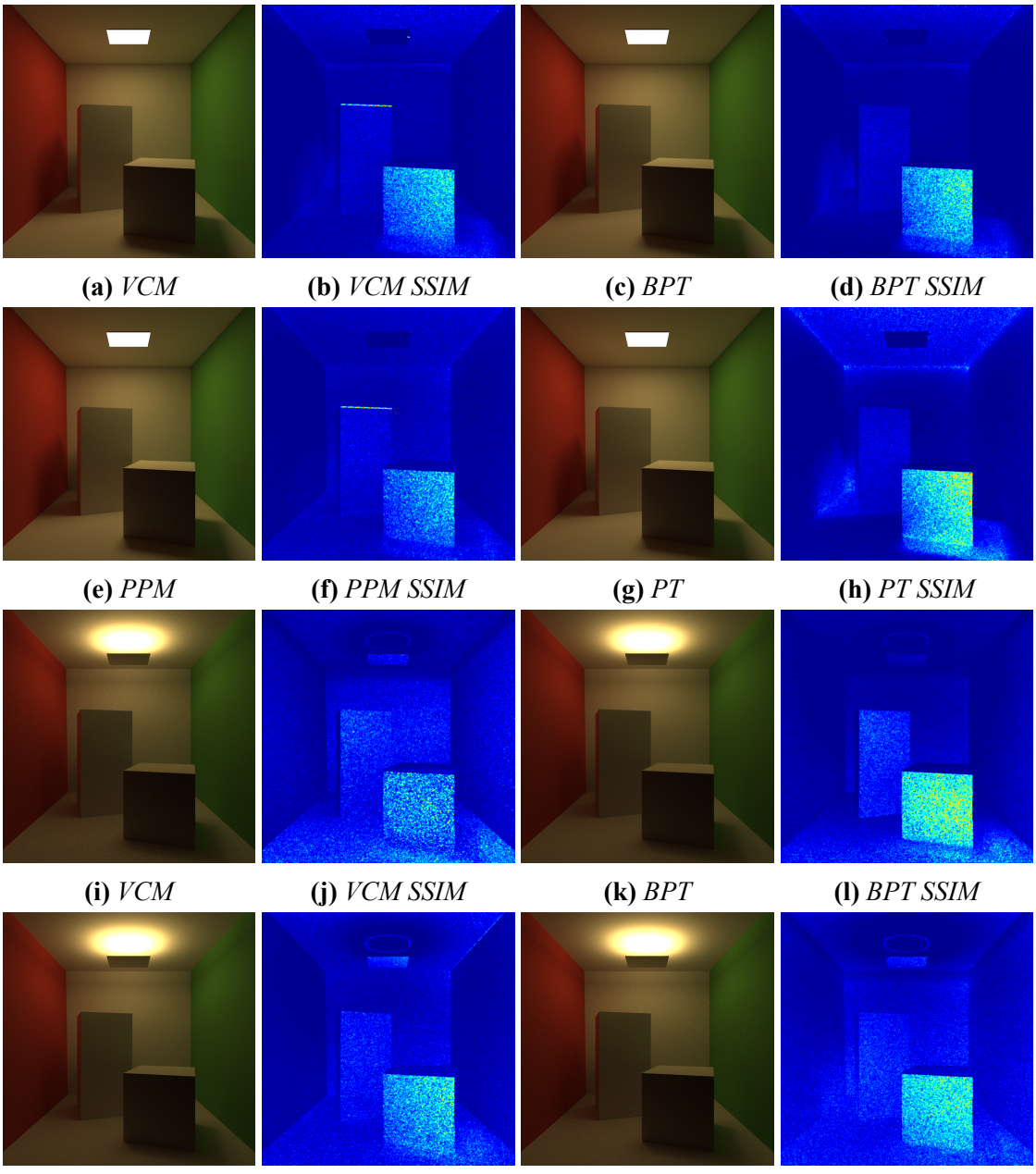
5.4 Conclusion

In this thesis we presented a parallel implementation of a renderer that achieved interactive frame rates for a variety of scenes, by making efficient use of a fast traversal library. By implementing a robust rendering algorithm, VCM, we made sure that the renderer will be able to handle a large variety of scenes. Interactive frame rates have been achieved with Path Tracing, and the VCM implementation is also very close to being interactive on our (quite old) hardware. High quality images can be rendered in less than five minutes for all test scenes, using VCM.

Different approaches to make efficient use of the traversal have been tried and compared. With carefully designed scheduling it was possible, to make the most use out of the GPU traversal. Even though data transfer between the CPU and the GPU is very costly, we still experience a speed-up factor of up to three, when switching from the CPU traversal to the GPU. This would not have been possible without a technique to keep the CPU occupied while waiting for the results from the GPU.

The design of our renderer can also be mapped to a GPU implementation. Thus, it will be a good reference when implementing a renderer in Impala that will run on both the GPU and the CPU. By making sure that all our high-level design choices and most of the low-level details, like the random number generator, map nicely to the GPU as well, the need for special case code in such a multi-platform implementation is greatly reduced.

Our results showed that it is worthwhile to use VCM for interactive rendering as well as for offline rendering. Using a robust algorithm ensures that the convergence rate is good for most scenes. Path Tracing, for instance, is significantly simpler to implement and to parallelize, and achieves faster frame rates. Thus, using Path Tracing in an interactive renderer is in some cases a better choice, especially if there are only few specular surfaces present in the scenes. Path Tracing and VCM both benefit from the parallelization scheme and other optimizations that were described in this thesis.



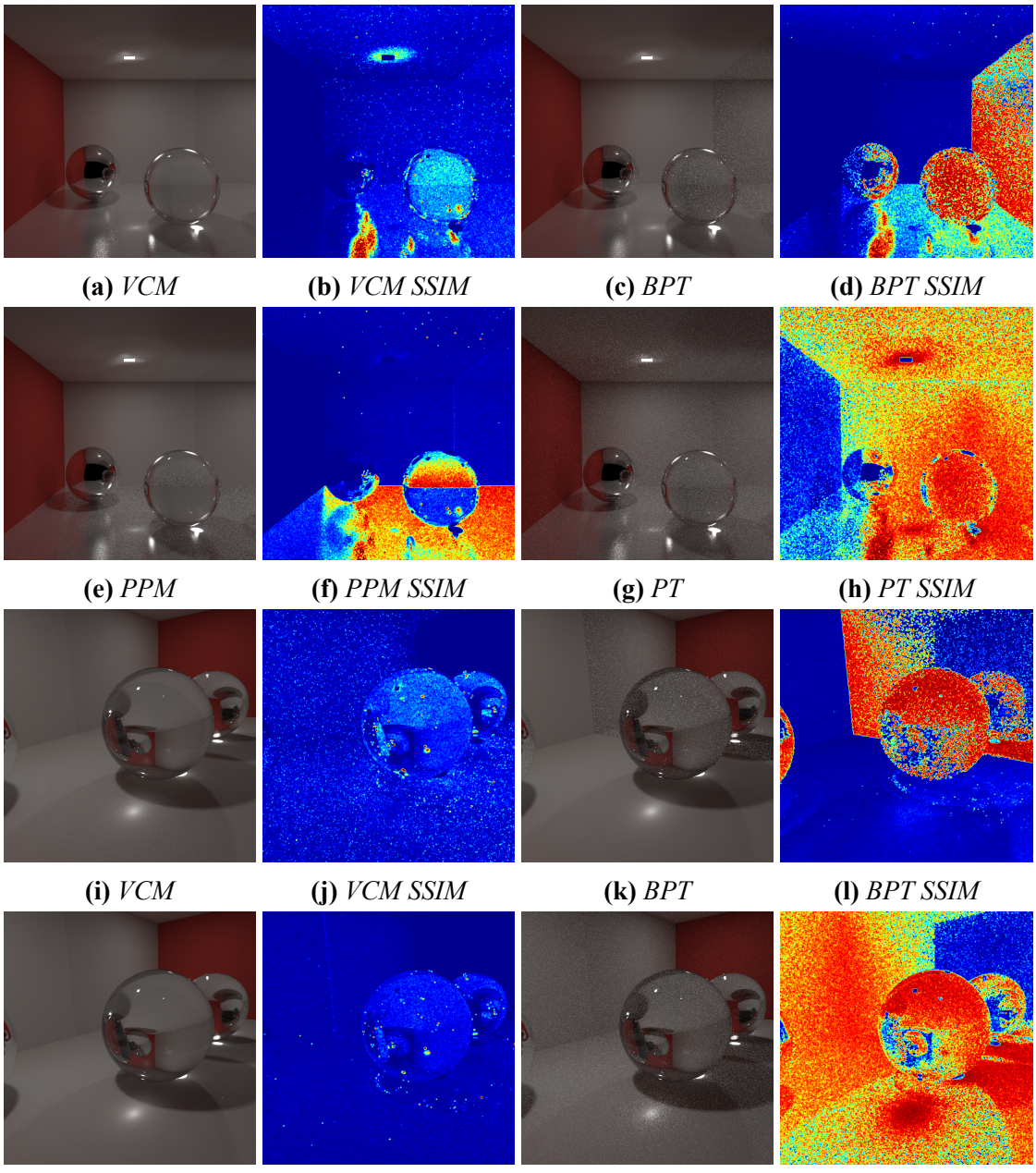
(a) *VCM* **(b)** *VCM SSIM* **(c)** *BPT* **(d)** *BPT SSIM*

(e) *PPM* **(f)** *PPM SSIM* **(g)** *PT* **(h)** *PT SSIM*

(i) *VCM* **(j)** *VCM SSIM* **(k)** *BPT* **(l)** *BPT SSIM*

(m) *PPM* **(n)** *PPM SSIM* **(o)** *PT* **(p)** *PT SSIM*

Figure 5.9: *Images of the diffuse Cornell Box scenes from all algorithms after five minutes.*



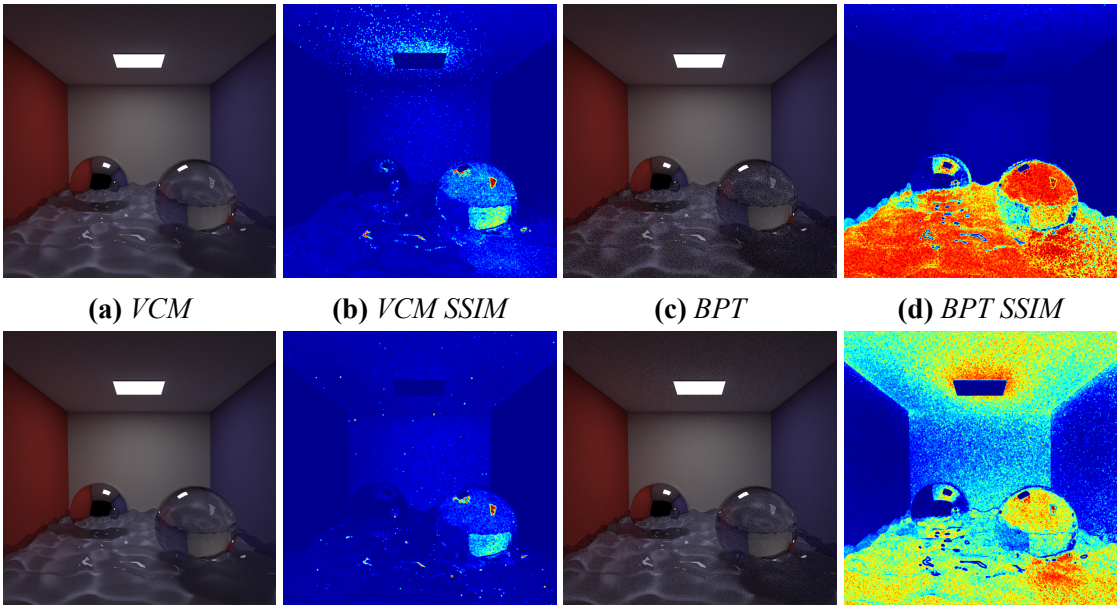
(a) *VCM* **(b)** *VCM SSIM* **(c)** *BPT* **(d)** *BPT SSIM*

(e) *PPM* **(f)** *PPM SSIM* **(g)** *PT* **(h)** *PT SSIM*

(i) *VCM* **(j)** *VCM SSIM* **(k)** *BPT* **(l)** *BPT SSIM*

(m) *PPM* **(n)** *PPM SSIM* **(o)** *PT* **(p)** *PT SSIM*

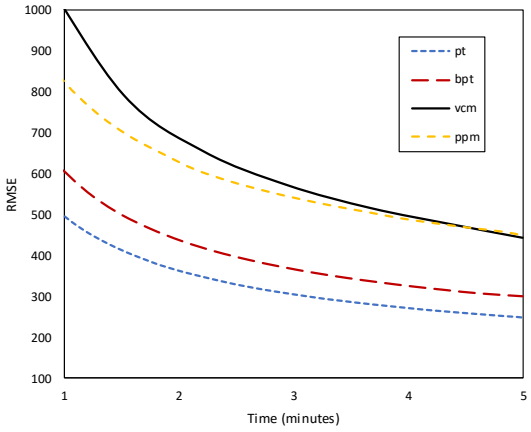
Figure 5.10: *Images of the specular Cornell scenes from all algorithms after five minutes.*



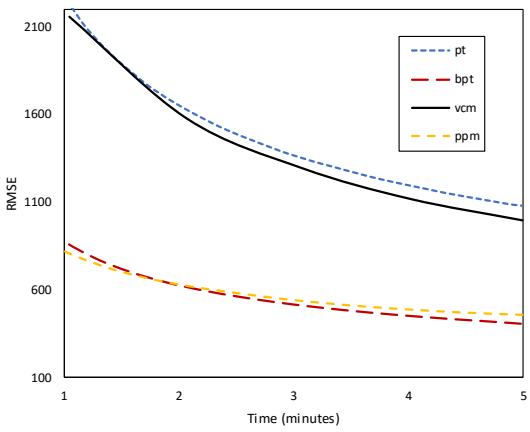
(a) *VCM* **(b)** *VCM SSIM* **(c)** *BPT* **(d)** *BPT SSIM*

(e) *PPM* **(f)** *PPM SSIM* **(g)** *PT* **(h)** *PT SSIM*

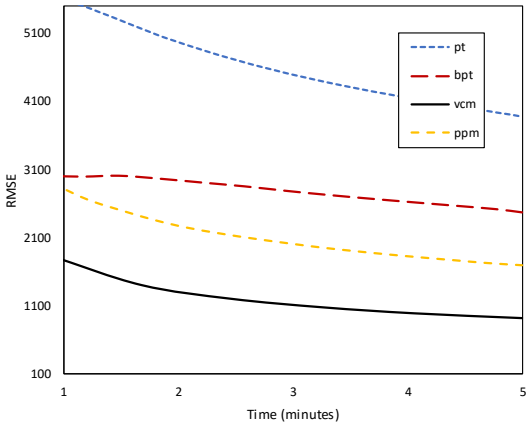
Figure 5.11: *Images of the Cornell Water scene from all algorithms after five minutes.*



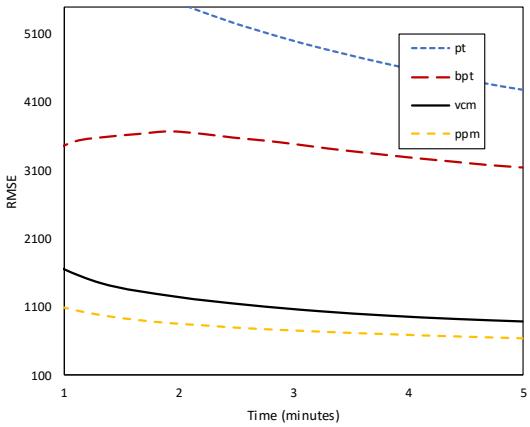
(a) *Cornell Box*



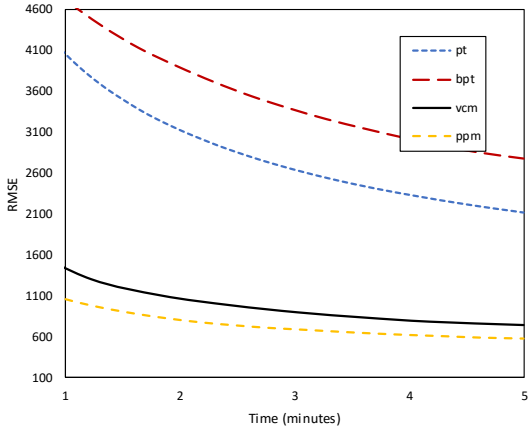
(b) *Cornell Indirect*



(c) *Cornell Specular*



(d) *Cornell Specular Close*



(e) *Cornell Water*

Figure 5.12: *The root mean squared error (RMSE) over time in the cornell box scenes also confirms that VCM is a good algorithm on average.*

Bibliography

[Any] AnyDSL. *AnyDSL*. URL: https://anydsl.github.io/index.html (visited on 05/11/2015).

[Dav] Tomáš Davidovič. *GitHub - SmallVCM*. URL: https://github.com/SmallVCM/SmallVCM (visited on 04/30/2016).

[Dav+14] Tomáš Davidovič et al. "Progressive light transport simulation on the GPU: Survey and improvements". In: *ACM Transactions on Graphics (TOG)* 33.3 (2014), p. 29.

[DLW93] Philip Dutré, Eric P. Lafortune, and Yves D. Willems. "Monte Carlo light tracing with direct computation of pixel intensities". In: *3rd International Conference on Computational Graphics and Visualisation Techniques*. Alvor, Portugal, Dec. 1993, pp. 128–137.

[Eis+13] Christian Eisenacher et al. "Sorted Deferred Shading for Production Path Tracing". In: *Proceedings of the Eurographics Symposium on Rendering*. EGSR '13. Zaragoza, Spain: Eurographics Association, 2013, pp. 125–132.

[Geo+12] Iliyan Georgiev et al. "Light Transport Simulation with Vertex Connection and Merging". In: *ACM Trans. Graph.* 31 (6 Dec. 2012). SIGGRAPH Asia 2012, pp. 1–10.

[Geo12] Iliyan Georgiev. *Implementing Vertex Connection and Merging*. Tech. rep. Saarland University, 2012. URL: http://www.iliyan.com/publications/ImplementingVCM.

[Geo15] Iliyan Georgiev. "Path sampling techniques for efficient light transport simulation". eng. PhD thesis. Universität des Saarlandes, 2015. URL: http://scidok.sulb.uni-saarland.de/volltexte/2015/6152.

[GS08] Iliyan Georgiev and Philipp Slusallek. "RTfact: Generic Concepts for Flexible and High Performance Ray Tracing". In: *To appear in the IEEE/Eurographics Symposium on Interactive Ray Tracing 2008*. Aug. 2008.

[Hec90] Paul S Heckbert. "Adaptive radiosity textures for bidirectional ray tracing". In: *ACM SIGGRAPH Computer Graphics*. Vol. 24. 4. ACM. 1990, pp. 145–154.

[HJ09] Toshyia Hachisuka and Henrik Wann Jensen. "Stochastic Progressive Photon Mapping". In: *ACM SIGGRAPH Asia* (2009).

[HOJ08] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. "Progressive photon mapping". In: *ACM Transactions on Graphics (TOG)* 27.5 (2008), p. 130.

[HPJ12] Toshiya Hachisuka, Jacopo Pantaleoni, and Henrik Wann Jensen. "A path space extension for robust light transport simulation". In: *ACM Transactions on Graphics (TOG)* 31.6 (2012), p. 191.

[Inta] Intel. *Threading Building Blocks*. URL: https://www.threadingbuildingblocks.org/ (visited on 04/30/2016).

[Intb] Intel. *When Task-Based Programming Is Inappropriate*. URL: https://software.intel.com/en-us/node/506101 (visited on 05/20/2016).

[Jen96] Henrik Wann Jensen. "Global illumination using photon maps". In: *Rendering Techniques' 96*. Springer, 1996, pp. 21–30.

[Kaj86] James T. Kajiya. "The Rendering Equation". In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 143–150. ISBN: 0-89791-196-2.

[Kar+10] Ralf Karrenberg et al. "AnySL: Efficient and Portable Shading for Ray Tracing". In: *Proceedings of the Conference on High Performance Graphics*. HPG '10. Saarbrucken, Germany: Eurographics Association, 2010, pp. 97–105. URL: http://portal.acm.org/citation.cfm?id=1921479.1921495.

[LKA13] Samuli Laine, Tero Karras, and Timo Aila. "Megakernels considered harmful: wavefront Path Tracing on GPUs". In: *Proceedings of the 5th High-Performance Graphics Conference*. ACM. 2013, pp. 137–143.

[LKH15] Roland Leißa, Marcel Köster, and Sebastian Hack. "A Graph-based Higher-order Intermediate Representation". In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '15. San Francisco, California: IEEE Computer Society, 2015, pp. 202–212. ISBN: 978-1-4799-8161-8.

[LW93] Eric P Lafortune and Yves D Willems. "Bi-directional Path Tracing". In: *Proceedings of CompuGraphics*. Vol. 93. 1993, pp. 145–153.

[Par+10] Steven G. Parker et al. "OptiX: A General Purpose Ray Tracing Engine". In: *ACM Trans. Graph.* 29.4 (July 2010), 66:1–66:13.

[PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123750792, 9780123750792.

[SFD09] Martin Stich, Heiko Friedrich, and Andreas Dietrich. "Spatial splits in bounding volume hierarchies". In: *Proceedings of the Conference on High Performance Graphics 2009*. ACM. 2009, pp. 7–13.

[Tho] David Thomas. *The MWC64X Random Number Generator*. URL: http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html (visited on 05/10/2016).

[Vea98] Eric Veach. "Robust Monte Carlo Methods for Light Transport Simulation". PhD thesis. Stanford, CA, USA, 1998. ISBN: 0-591-90780-1.

[VG94] Eric Veach and Leonidas Guibas. "Bidirectional estimators for light transport". In: *Eurographics Workshop on Rendering*. 1994.

[VG95] Eric Veach and Leonidas J. Guibas. “Optimally Combining Sampling Techniques for Monte Carlo Rendering”. In: *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’95. New York, NY, USA: ACM, 1995, pp. 419–428. ISBN: 0-89791-701-4.

[Vor11] Jirı Vorba. “Bidirectional photon mapping”. In: *Proc. of the Central European Seminar on Computer Graphics (CESCG’11)*. 2011.

[Wal+01] Ingo Wald et al. “Interactive rendering with coherent ray tracing”. In: *Computer graphics forum*. Vol. 20. 3. Wiley Online Library. 2001, pp. 153–165.

[Wal+14] Ingo Wald et al. “Embree: A Kernel Framework for Efficient CPU Ray Tracing”. In: *ACM Trans. Graph.* 33.4 (July 2014), 143:1–143:8.

[WBB08] Ingo Wald, Carsten Benthin, and Solomon Boulos. “Getting rid of packets-efficient simd single-ray traversal using multi-branching bvhs”. In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE. 2008, pp. 49–57.

List of Figures

List of Tables