GPU-Based Global Illumination

Vertex Connection and Merging using OptiX

Valdis Vilcans

Master of Science in Digital Media Engineering
Kongens Lyngby 2014

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Building 324, 2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

Physically based rendering algorithms are capable of producing high quality images of virtual environments. The rendering time, however, usually is long due to associated computation costs. The GPU processing power is increasing in a relatively faster pace than CPUs and they are specially designed to handle parallelizable workloads. This makes them attractive for accelerating physically based algorithms. In recent years multiple tools such as OpenCL, Nvidia's CUDA and OptiX for harnessing GPU power have appeared.

The goal of the thesis is to provide an overview of the novel Vertex Connection and Merging (VCM) [GSK11] algorithm which combines benefits of Bidirectional Path Tracing and Photon Mapping, and to describe an implementation using Nvidia's OptiX ray-tracing framework [PBD+10].

The taken approach was to extend the OppositeRenderer, an open source OptiX based renderer which contains Progressive Photon Mapping (PPM) implementation. We hoped that would help to speed up the implementation of Vertex Merging through partial reuse of data structures used in PPM and to provide richer feature set in the end. Due to experienced development issues and delays, Vertex Merging was not implemented, however, Vertex Connections including the common parts for both path sampling methods were implemented. Also material handling capabilities were improved to allow flexibility of combining multiple reflection models for single material. The extended version of Opposite Renderer is provided to the public same as the original was.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring an M.Sc. in Digital Media Engineering.

The thesis main title encompasses a very broad area of computer graphics field. This is so because the author had desire to investigate current state of the art algorithms and techniques and explore possible ways of combining them or implementing on GPU without having a concrete one in mind initially. During this initial research which took much longer than anticipated, the author learned about the novel Vertex Connection and Merging (VCM) algorithm introduced by Georgiev et al. [GSK11] which formulates a new bidirectional path sampling technique called *Vertex Merging*. It provides benefits of Photon Mapping [WJ01] and allows to combine it efficiently with traditional Bidirectional Path Tracing (BPT) [Vea98]. This is the algorithm chosen for GPU implementation and is described in this thesis.

Given the limited time the decision was to use the Nvidia OptiX [PBD+10], a CUDA [1] based ray tracing engine that provides acceleration structures and provides optimizations for various GPU architectures. As a staring basis the open source renderer OppositeRenderer [2] was used, it contains an OptiX implementation of Progressive Photon Mapping (PPM) [HOJ08] and was created by Stian Pedersen for his M.Sc. thesis [Ped13].

One of the goals for implementation was to use recursive Multiple Importance Sampling (MIS) weight [Vea98] computation as proposed in [Geo12] that allows

[1] It works only with Nvidia GPUs

[2] `http://apartridge.github.io/OppositeRenderer`

efficient path weigh computation without need to traverse connected subpaths. Another was to use uniform light vertex sampling for connections as suggested in [DKHS14], this would allow to use arbitrary number of connections and to reduce negative effects due to connections to paths with many vertices.

The development using OptiX turned out slow due to problematic debugging, the current versions of OptiX [3] have a bug that often causes crashes when debug printing functions are used. Also multiple other bugs in OptiX [4] and CUDA [5] were encountered and had to be worked around. Moreover, late in the process it was recognized that MIS term computation formulas of [Geo12] should be revisited when uniform light vertex sampling is used, since connections are made to all light subpaths, not just one as in traditional BPT. An effort was made to do that and the author of the algorithm Iliyan Georgiev confirmed the correctness of intended changes, but due to limited time this was not finished.

The result is that within the given schedule the vertex connection part of the VCM algorithm was implemented. This is the main contribution of the project. Also a more flexible material handling system was added that allows easily combining multiple reflectance models and a Glossy material. The implementation is available on GitHub [6]. Given that the core of the algorithm is implemented, vertex merging could be implemented building on grid data structures used in PPM implementation.

The thesis consists of overview of the theoretical background and path space sampling based algorithms, the Vertex Connection and Merging, the description of implementation, the discussion of results, future work and conclusions.

Due to longer than expected time needed for implementation of the algorithm, very little time remained for writing the thesis. Therefore it is as concise as possible, providing links to resources to allow the reader look up additional details if needed. References are provided to all resources and work of other authors that were used. All images present in the thesis are diagrams or renderings made by the author or are used with author's permission.

[3] 3.5 and 3.6 at the time of writing

[4] The callable functions not receiving parameters, buffer out of bounds exceptions not being triggered, infinite loops causing crashes that look like memory corruption when it actually is graphics driver timeout error, only part of the debug output buffer being forwarded to file and only after seconds context launch, and some others

[5] The CUDA 6 fails to compile the solutions due to a problem to inline a function

[6] `http://github.com/voldemarz/OppositeRenderer`

Lyngby, 21-July-2014

Valdis Vilcans

Acknowledgements

I would like to thank my supervisor Jeppe Revall Frisvad for provided guidance. I thank Mircea-Costin Rohat and Janis Siksnans for support and help to identify inconsistencies. Much gratitude goes to Stian Pedersen for sharing his renderer with public and allowing to use some of his graphics figures. Finally, I thank Iliyan Georgiev for his valuable answers to questions about recursive multiple importance sampling terms and for allowing to use graphics from Vertex Connection and Merging papers.

Contents

Chapter 1

Introduction

The global illumination characterizes lighting quality of captured lighting effects in a rendered image of a 3D scene. In particular that means effects such as color bleeding from one object to another, light caustics due to specular reflections, glossy inter-reflections, or difficult lighting scenarios when light source is in another room.

There are two main approaches to global illumination, one is based on sampling the space of light carrying paths (e.g. Bidirectional Path Tracing [Vea98]) and the other is using density estimation (e.g. Photon Mapping [WJ01]). Georgiev et al. [GKDS12] recently introduced a method called Vertex Connection and Merging (VCM) that provided benefits of both above mentioned approaches.

Rendering algorithms are well suited for parallelization since they generally perform same operations on varying input data. GPUs on the other hand contain hundreds (or even thousands) of simple computation units and are specifically designed for parallel workloads. The advance of programmable pipeline gave some opportunities to use GPUs for general processing tasks, and later the introduction of OpenCL and Nvidia's CUDA allowed to view GPUs purely as general purpose computational units for highly parallel tasks. CUDA (Compute Unified Device Architecture) is a proprietary platform introduced by Nvidia in [Cor] in 2007. OpenCL [Gro] was introduced in 2009 as an open standard supported by all major vendors including Nvidia. Currently CUDA is generally the

preferred tool for implementing rendering algorithms due to its better performance for these problems as shown in [KGF+11].

GPU processing power is growing consistently and generally in faster pace than the one of CPUs. The Fig. 1.1 demonstrates the trend of theoretical maximum number of floating point operations (FLOPs) per seconds for Nvidia GPUs and Intel processor architectures.

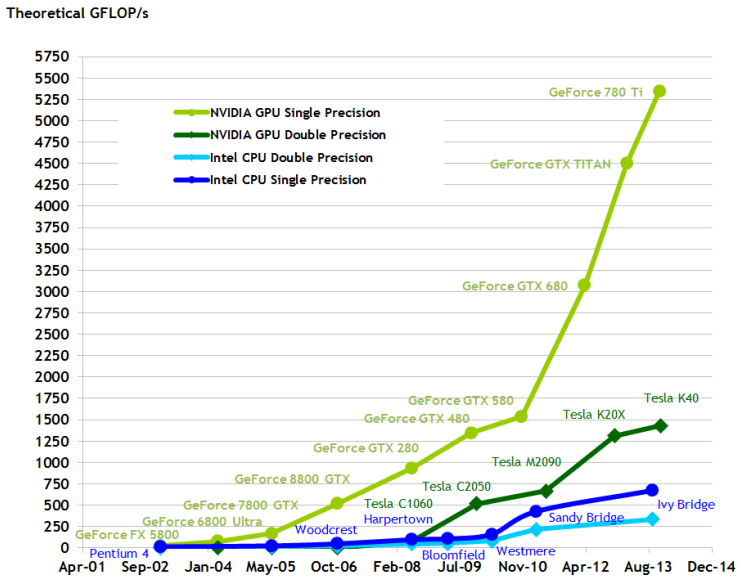


Figure 1.1: Maximum theoretical number of floating operations for Nvidia GPUs and Intel CPUs in GFLOPs per second. Courtesy of Nvidia.

To alleviate developers from the need to optimize for various GPU architectures Nvidia introduced a general purpose ray tracing engine OptiX [PBD+10], that provides essential capabilities for ray tracers such as acceleration structures. It allows to implement arbitrary algorithms by defining a set of programs (similarly as shader programs in OpenGL and DirectX) that can be hooked into various parts of the pipeline.

In this thesis we will focus on implementing the novel VCM algorithm using OptiX. We base our work on the open source renderer OppositeRenderer with implementation of progressive photon mapping (PPM) made by Stian Pederesen for his M.Sc. thesis [Ped13]. It was decided to do so because it gave the potential to speed up implementation of vertex merging by partially reusing grid data

structures used for PPM, also conveniently it provided scene loading using Asset Import Library and an interactive interface.

The thesis is structured as follows:

- Chapter 2 presents work that serves as basis for Vertex Connection and Merging and its GPU implementation
- Chapter 3 reviews theoretical background, important for understanding the implementation
- Chapter 4 describes the design and implementation details
- Chapter 5 analyzes achieved results and compares them with the reference implementation in SmallVCM
- Chapter 6 describes possible areas of future improvements
- Chapter 7 summarizes what has been done and achieved

Chapter 2

Related Work

2.1 Algorithms

The Path Tracing algorithm introduced in 1986 by Kajiya [Kaj86] follows random paths through the scene from camera to light sources. Veach et al. [VGS+95] and Lafortune et al. [LWS93] introduced the Bidirectional Path Tracing (BPT) that creates paths by combining sub-paths starting both from light sources and camera, thus helping to find paths to occluded light sources. Veach [Vea98] formulated rendering as an integration of a pixel measurement function over the space of all paths and showed how to combine various path sampling techniques in a provably good way using Multiple Importance Sampling (MIS).

The Photon Mapping [WJ01] approximates radiance (it is biased, but consistent) at any point via density estimation using light particles (*photons*) that have been distributed from light sources and stored on surfaces in the scene. It is efficient in rendering lightning effects such as caustics, but has difficulties in scenes with glossy surfaces. Vorba [Vor11] addressed this by using multiple importance sampling (MIS) to combine contributions of photons at camera path vertices.
Progressive Photon Mapping (PPM) introduced by Hachisuka [HOJ08] removes the requirement of storing all photons in memory and diminishes bias in the course of computation as density estimation radius is reduced. Stochastic pro-

gressive photon mapping [HJ09] enables PPM to render distribution ray tracing effects such as depth of field. Knaus et al. [KZ11] showed that radius reduction doesn't need to depend on per pixel statistics, effectively removing dependency between iterations in PPM.

Georgiev et al. [GSK11] formulates a new bidirectional path sampling technique called Vertex Merging that conceptually relates to photon mapping. It is combined with traditional bidirectional path tracing (BPT) using multiple importance sampling (MIS) and the resulting algorithm is called Vertex Connection and Merging (VCM). A work of Hachisuka et al. [PJH12] addresses the same problem, but uses significantly different theoretical formulation. The technical report of the VCM reference implementation [Geo12] presents an efficient way to compute MIS weights by recursively computing partial terms at every path vertex that allow to compute full weight once it is connected or merged without traversing the full path. The technique is similar to the one proposed by Van Antwerpen [Ant11].

Efficient ray tracing requires acceleration structures that group geometry together and allow to avoid checking intersection with individual objects if the ray doesn't intersect the group. Thus research in this area focuses on three main issues: the selection of suitable acceleration structure, algorithms for their effective construction and traversal. Common examples of such structures are uniform grids, bounding volume hierarchies (BVHs) and kd-trees. Havran [Hav00] provides an overview of CPU-based acceleration structure construction.

2.2 Applications to GPU

In the survey article from 2012 Ritchel et al. [RDGK12] provides a comprehensive review of current methods for computing global illumination and attempts to provide an approximate comparison using criteria such as speed, quality, dynamism, scalability, ease of implementation, ease of mapping to GPU.

Initial attempts to use the programmable GPU pipeline for ray tracing date back to the 2002 work of Carr et al. [CHH02] where they implement ray-triangle intersection as a pixel shader and used uniform grid acceleration structure. More recently, Aila et al. [LA09] [ALK12] showed how to approach theoretical peak ray tracing performance using Spatial Bounding Volume Hierarchies (SBVHs) [SFD09] on Nvidia GPUs using CUDA. They observed that previously known methods were about a factor of 2X off from the theoretical optimum and identified that it was mostly caused by inefficiencies in hardware work distribution. Much of the current research focuses on fast construction of acceleration

structures on GPU to allow interactive rendering of dynamic geometry. For references and details we refer the reader to the survey on progressive light transport algorithms on the GPU by Davidovič et al. [DKHS14].

The survey on progressive light transport algorithms on the GPU presents implementations of the progressive photon mapping (and its bidirectional and stochastic derivatives) and the bidirectional path tracing based on the state-of-art method proposed by Van Antwerpen [VA11] [Ant10]. Both algorithm were implemented within the CUDA ray casting framework based on work of [LA09]. They also review effects due to use of single or multiple CUDA kernels, propose few improvements over state-of-art, present the first GPU implementation of Vertex Connection and Merging, and compare all the implementations.

Upon introduction of OptiX ray tracing engine Nvidia demonstrated [PBD+10] that a well optimized OptiX based solution can achieve performance of around 60% compared to a hand-optimized CUDA solution of [LA09]. However, Ludvigsen [LE10] who explored OptiX the same year concluded that it is 3-5 times slower that hand-optimized CUDA solutions on similar scenes and hardware.

Chapter 3

Theoretical background

This chapter introduces the theoretical background for global illumination, thus laying foundation for understanding the Vertex Connection and Merging algorithm. The focus is on the most essential concepts and references are provided to resources with more detailed discussion.

We start with a brief overview of radiometry and continue with describing surface reflection. Next we discuss basic concepts of probability theory and light transport. We continue by introducing the Monte Carlo as a technique for estimating integrals describing light distribution. Finally, we present an overview of path sampling methods and continue with discussing the Vertex Connection and Merging.

3.1 Radiometry

In order to generate an image of a virtual scene, the amount of reflected light towards the viewer needs to be computed. Radiometry provides the theoretical basis for analyzing the propagation of light and the transfer of radiant energy. It operates at the geometrical optics level, that is, where objects the light interacts with are much larger than its wavelength. For more details we refer reader to

Symbol	Description
x	Surface point
n	Surface normal
M	The set of all surface points
A	Area m^2
ω	Solid angle / solid angle cone direction
S^2	The set of all directions
$H^2(n)$	The hemisphere of directions around surface normal
Φ	Radiant flux J/s (W)
E	Irradiance W/m^2
I	Intensity W/sr^{-1}
L	Radiance $\mathrm{W}/(sr^{-1}m^{-2})$
f_r, f_s	Bidirectional reflectance (r) / scattering (s) distribution function
p	Probability distribution function
$\langle I \rangle$	Monte Carlo estimator for quantity I
f	Measurement contribution function
$\overline{x} = x_0 \dots x_k$	Complete path of length k, x_0 is on light source, x_k is on camera image plane
$\overline{y} = y_0 \dots y_{s-1}$	Light sub-path with first vertex $y_0 \equiv x_0$
$\overline{z} = z_0 \dots z_{t-1}$	Camera sub-path with first vertex $z_0 \equiv x_k$
W_e	Sensor responsiveness / emitted importance
$G(x_i \leftrightarrow x_j)$	Geometry factor between two points
$T(\overline{x})$	Path throughput (product of geometry factors and BSDFs)
w	Multiple importance sampling weight
i	Subscript denotes incoming direction
o	Subscript denotes outgoing direction

Table 3.1: Common notations.

text book on rendering such as [PH10].

3.1.1 Radiant flux

This quantity, also called *Power*, denoted Φ, describes amount of radiant energy per unit time and is expressed in watts (W or J/s). Its a typical way to describe the amount of energy emitted by a light source.

3.1.2 Irradiance and radiant exitance

Irradiance E describes energy arriving at a surface and its units are W/m^2. A related quantity, radiant exitance (M), describes the amount of energy leaving the surface.

3.1.3 Intensity

Before defining intensity we need to establish the notion of *solid angle*. It describes the area subtended by an object projected onto a unit sphere (Fig. 3.1). That is, a small object close to the origin can subtend the same amount of space as a distant object. It is measured in *steradians* (sr) and entire sphere subtends solid angle of 4π, and a hemisphere 2π.

Finally, the intensity I describes flux density per solid angle

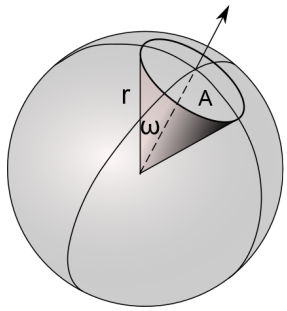


Figure 3.1: Solid angle. An area subtended by an object projected onto sphere. An illustration by Haage licensed under CC BY 3.0.

$$I = \frac{d\Phi}{d\omega}$$

and is only meaningful for point light sources.

3.1.4 Radiance

Radiance L (Fig. 3.2) describes flux density per unit area, per solid angle, $watts/(sr \cdot m^2)$

$$L = \frac{d\Phi}{d\omega dA \cos\theta} \tag{3.1}$$

The cosine term of an angle θ between the surface normal and solid angle cone direction dω accounts for the fact that at grazing angles the incident ray will cover more surface area than for directions close to the surface normal. As per common convention in this thesis we always consider ω to be pointing away from the surface.

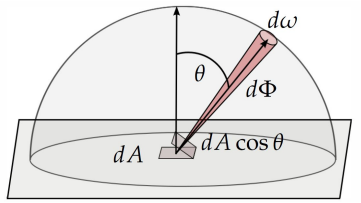


Figure 3.2: Radiance. An illustration by Stian Pedersen.

It is the most common quantity used in rendering since all the other can be computed by integrating over surface areas and directions.

3.2 Surface reflection

The way an object looks is described by its material reflection properties. Some materials like wood reflect incident light in all directions and look the same from all directions. Mirror on the other hand has a perfect reflection. A formalism that allows to describe this behavior is *bidirectional reflectance distribution function* (BRDF) and describes the ratio of differential reflected radiance ω_o and differential irradiance from direction ω_i at a surface point x and is defined as

$$f_r(x, \overrightarrow{\omega_i}, \overrightarrow{\omega_o}) = \frac{dL_o(x, \overrightarrow{\omega_o})}{dE(x, \overrightarrow{\omega_i})} = \frac{dL_o(x, \overrightarrow{\omega_o})}{L_i(x, \overrightarrow{\omega_i}) \cos\theta_i d\omega_i}. \tag{3.2}$$

The outgoing radiance is obtained by integrating over hemisphere of incident directions

$$L_o(x, \omega_o) = \int_{H^2(n)} f_r(x, \omega_o, \omega_i) L_i(x, \omega_i) \cos\theta d\omega_i \tag{3.3}$$

Similarly also is defined the *bidirectional transmittance distribution function* (BTDF) which describes the distribution of the transmitted light. In that case the incident and the outgoing directions are on different sides of the hemisphere. As general case that combines both, it is common to define the *bidirectional scattering distribution function* (BSDF) which describes the outgoing radiance from a point due to incident radiance from all directions. In that case the integral 3.3 is over the sphere of directions S^2 and BSDF is abbreviated as f_s where subscript s stands for scattering.

Physically based BRDFs have two important properties:

1. $f_r(x, \omega_i, \omega_o) = f_r(x, \omega_o, \omega_i)$ (Helmholtz's reciprocity)
2. $\int_{H^2(n)} f_r(x, \omega_o, \omega') \leq 1$ (Energy balance)

The first states that for two given directions, the amount of the reflected light is the same, disregarding of the direction in which the light travels. The second states that the fraction of the total amount of the reflected energy cannot be higher than 1.

The reflection from surfaces can be split into four broad categories:

- Diffuse - scatter light equally in all directions (example - matte paint)
- Perfect specular - scatter incident light in single outgoing perfect reflection direction (examples - mirror and glass)
- Glossy specular - scatter incident light mostly in directions close to perfect reflection direction (example - plastic)
- Retro-reflective - scatter incident light mostly back along incident direction (example - velvet)

In rendering the reflection from a surface is simulated suing varios *reflection models* which can be based on measurement data or use a set of equations that are effective for simulating the look of a given real world material. For example of various reflection models we refer the reader to a book on rendering such as [PH10].

3.3 Probability theory

In this section we provide brief review of some probability theory principles used in the later chapters. For more in-depth discussion the reader could consult [PH10] or books on probability theory.

3.3.1 Random variables

The random variable is described by its *cumulative distribution function* (CDF) and *probability distribution function* (PDF) [PH10]. Having a distribution of random variables X, the CDF describes the probability that a value from variable's distribution is less than or equal to some value x:

$$P(x) = Pr\{X \leq x\}. \tag{3.4}$$

The PDF p(x) is the derivative of the random variable's CDF

$$p(x) = \frac{dP(x)}{dx} \tag{3.5}$$

and describes probability for sampling a random variable with value x from the distribution X.

3.3.2 Expected value and Variance

The *expected value* E_p[f(x)] of a function is defined as the average value of the function over some distribution of values $p(x)$ over it's domain D [PH10]

$$E_p[f(x)] = \int_D f(x)p(x)dx. \tag{3.6}$$

The variance of a function is the expected deviation of the function from its expected value and is defined as [PH10]

$$V[f(x)] = E[(f(x) - E[f(x)])^2]. \tag{3.7}$$

In rendered images it presents itself as noise. It is a fundamental method in quantifying the error produced by a Monte Carlo estimator that will be introduced in next section.

3.4 Light transport and measurement

Here we briefly introduce concepts of light transport and measurement functions following Veach [Vea98].

3.4.1 Light transport equation

The light transport equation, also known as the *rendering equation*, was introduced by Kajiya [Kaj86] and it describes the *equilibrium* distribution of radiance in a scene, which means constant energy as emitted radiance is continuously absorbed due to conservation of energy. Another assumption is that light travels instantly and hence the equilibrium state is achieved instantly. The equation states that the outgoing radiance from a point is equal to the sum of the emitted radiance and the reflected radiance which is obtained by integrating over all incident directions

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_{H^2(n)} f_r(x,\omega_o,\omega_i) L_i(x,\omega_i) \cos\theta_i d\omega \tag{3.8}$$

The incident radiance L_i is obtained using

$$L_i(x,\omega) = L_o(t(x,\omega), -\omega) \tag{3.9}$$

where $t(x,\omega)$ is the ray casting function returning the first point x' visible from x.

3.4.2 Measurement equation

The goal in rendering is to compute *measurements* I of light incident at an image pixel (camera sensor). The measurement corresponds to the output of a hypothetical sensor that responds to incident radiance $L_i(x,\omega)$. The response may vary depending on position and direction at which light strikes, this is characterized by *sensor responsiveness* $W_e(x,\omega)$. For more details we refer the reader to [Vea98]. Therefore the total response is determined by integrating the product $W_e L_i$

$$I = \int W_e(x,\omega) L_i(x,\omega) \cos\theta dA(x) d\omega \tag{3.10}$$

The subscript "e" for the sensor responsiveness W_e stands for "emitted". This is because conceptually this measure can be seen as emitted from the camera

and propagated through the scene following the same rules as for the radiance (with some exceptions). In that case this quantity is called *importance* and its propagation – *importance transport*. This concept is used in the *path integral formulation*.

The exceptions apply to non-symmetric BSDFs as indicated by Veach [Vea98]. They do not adhere to the reciprocity property, that is, for two directions w_o and w_i, a different amount of light is scattered in each direction $f_s(w_o, w_i) \neq f_s(w_i, w_o)$. An example of such case is refraction. We refer the reader to work of Veach for additional details.

3.5 Monte Carlo integration

Monte Carlo is a numerical integration method that allows to evaluate the value of an arbitrary integral by sampling it with random numbers. The estimator for an integral $I = \int f(x)dx$ is [PH10]

$$\langle I \rangle_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \tag{3.11}$$

where $p(X_i)$ is the probability density function that describes possibility of sampling the random variable X_i from some distribution X with restriction that it must be non zero for all x where $|f(x) > 0|$. The number of samples N can be chosen arbitrarily and its convergence rate $O(\sqrt{N})$ is independent of dimensionality of the integral. This makes it the only practical numerical integration algorithm for integrals with high (or infinite) dimensionality which are common in rendering.

3.5.1 Bias and consistency

An estimator $\langle I \rangle$ is *unbiased* if its expected value is equal to the value of the integral being estimated. Otherwise the amount of *bias* is

$$\beta = E[\langle I \rangle] - \int f(x)dx \tag{3.12}$$

and the estimator is said to be *biased*.
An estimator $\langle I \rangle$ is *consistent* if it converges to the correct value as the number of samples N increases. Formally that can be expressed as

$$\lim_{N \to \infty} Pr[|\langle I \rangle_N - I| > \varepsilon] = 0 \tag{3.13}$$

where is ε is some small fixed value. The equation states that as the number of samples approaches infinity, the probability of the error being greater than ε approaches zero. Having $\varepsilon = 0$ means that estimator approaches the exact answer.
Unbiased estimators produce the correct result *on average* and increasing the number of samples will typically reduce the variance. Biased, but consistent estimators approach the correct value only in the limit.

3.5.2 Russian Roulette

The Russian Roulette [PH10] is a technique that increases efficiency of the estimator. It addresses the problem of evaluating samples that have small contribution to the final result. It is applied by introducing a termination probability q and checking it before computation of some expensive function. The computation is continued with probability $1 - q$, but the result is scaled by a factor $\frac{1}{1-q}$. The expected value of the estimator is unchanged, but its variance increases, therefore the choice of q is important and should be inversely proportional to the importance of the evaluation to the final result.

3.5.3 Importance Sampling

Importance sampling is a variance reduction technique that exploits the fact that Monte Carlo estimators converge quicker if samples are taken from the distribution $p(x)$ that is similar to the function $f(x)$ [PH10]. It can be demonstrated using a simple example. Suppose we are trying to evaluate some integral $\int f(x)dx$. Since $p(x)$ can be freely chosen, suppose $p(x) = cf(x)$, this means

$$c = \frac{1}{\int f(x)dx}$$

In this case the estimate would have the value

$$\frac{X_i}{p(X_i)} = \frac{1}{c} = \int f(x)dx$$

which is the exact value we were trying to estimate. Of course it is not realistic to find such PDF since that would require to know the value for the integral that we are trying to estimate, but it shows how the PDF affects the estimate. Similarly, applying this concept to the light transport equation 3.8, it would be beneficial to choose $p(x)$ proportional, for example, to BRDF or incident light distribution.

3.5.4 Multiple Importance Sampling

It is difficult to find a PDF that works well in all cases for importance sampling presented in the previous section due to the product of multiple functions in the integral, L_i and f_r. If we were to sample using a density function that matches the BSDF, but the incident radiance is large from an unimportant direction according to the BSDF probability density, then variance would also be large.

Multiple importance sampling (MIS) introduced by Veach [Vea98] solves this by weighting the samples from each technique and so helps to eliminate large variance spikes due to mismatch between integrand's value and the sampling density. The *multi-sample estimator* for m different distributions (or sampling techniques), each given by its probability density function p_i is

$$\langle I \rangle_{MIS} = \sum_{i=0}^{m} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(X_{i,j}) \frac{f(X_{i,i})}{p_i(X_{i,j})} \tag{3.14}$$

where n_i is a number of samples for given technique, $\mathrm{X}_{i,j}$ are independent random variables with distribution p_i and w_i is the weighting function for i-th sampling technique. To produce the correct result the weighting functions must satisfy two conditions:

- $\sum_{i=1}^{n} w_i(x) = 1$ iff $f(x) \neq 0$
- $w_i(x) = 0$ whenever $p_i(x) = 0$

There are many weighting functions that satisfy these conditions, but Veach [Vea98] showed that in a general case the *power heuristic*

$$w_i(x) = \frac{(n_i p_i(x))^\beta}{\sum_{k=1}^{n} (n_k p_k(x))^\beta} \tag{3.15}$$

with β=1 (*balance heuristic*) is the best option. In some cases β=2 is beneficial, we refer reader to the original work for the details.

3.6 Path sampling methods

3.6.1 The path integral formulation

Veach reformulated the light transport equation in form of an integral over paths (*path integral framework*) [Vea98] that allows to express the pixel j measurement

as a non-recursive integral:

$$I_j = \int_{\Omega} f_j(\overline{x}) d\mu(\overline{x}) \tag{3.16}$$

In the equation $\overline{x} = x_0 \dots x_k$ is a light path with k≥1 edges, where the first vertex x_0 is on the light source and the last vertex x_k on the camera image plane. The Ω represents the set of all paths of any length (*paths space*), $d\mu_(\overline{x}) = dA(x_0) \dots dA(x_k)$ is the *differential area product measure*, and f is the *measurement contribution function* for the pixel.
The measurement contribution function $f(\overline{x})$ gives the energy flowing through the path (Fig. 3.3) per unit area product measure and is defined as

$$f(\overline{x}) = L_e(x_0) G(x_0 \leftrightarrow x_1) \left[\prod_{i=1}^{k-1} f_s(x_i) G(x_i \leftrightarrow x_{i+1}) \right] W_e(x_k). \tag{3.17}$$

The terms in the equation are:

- $L_e(x_0) = L_e(x_0 \leftrightarrow x_1)$ is the radiance emitted from a light source
- $W_e(x_k) = W_e(x_{k-1} \rightarrow x_k)$ is the pixel responsiveness to light arriving at x_k from direction of x_{k-1}
- $f_s(x_i) = f_s(x_{i-1} \rightarrow x_i \rightarrow x_{i+1})$ is the bidirectional scattering function (BSDF)
- $G(x_{i-1} \leftrightarrow x_i) = V(x_{i-1} \leftrightarrow x_i) \frac{|\cos\theta_{i-1,i}||\cos\theta_{i,i-1}|}{||x_{i-1} - x_i||^2}$ is the geometry factor where $V(x_{i-1} \leftrightarrow x_i)$ is the visibility term, and the rest are due to change from integration over solid angle to surface area.

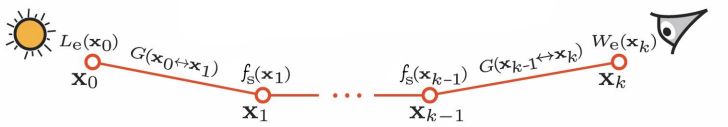


Figure 3.3: Path from the light source to camera. Used with permission from [GKDS12]

The product of all the geometry terms and BSDF factors describes the fraction of light reaching the sensor and is called *throughput*:

$$T(\overline{x}) = G(x_0 \leftrightarrow x_1) \prod_{i=1}^{k-1} f_s(x_i) G(x_i \leftrightarrow x_{i+1}). \tag{3.18}$$

This allows to express the measurement function as

$$f(\overline{x}) = L_e(x_0) T(\overline{x}) W_e(x_k). \tag{3.19}$$

The PDF of the light $p(\overline{x}) = x_0 \dots x_k$ is the product of individual probabilities of conditional vertex PDFs $p(\overline{x}) = p(x_0) \dots p(x_k)$. The PDF of a vertex x_i characterizes probability to be sampled on some surface from the previous vertex x_{i-1}. For the path starting point x_0 it is the probability of being sampled on some light source.

As the equation 3.19 shows we may start creating the path at the camera, this can be seen as propagation of the sensor responsiveness W_e as it was mentioned in 3.4.2. In that case BSDF $f_s(x_i) = f_s(x_{i+1} \leftarrow x_i \leftarrow x_{i-1})$.

3.6.2 Path Tracing

The Path Tracing is an unbiased rendering algorithm originally proposed by Kajiya as a solution to the *rendering equation* [Kaj86]. Using the path integral framework it is defined by the following estimator

$$\langle I \rangle = \sum_{t \geq 2} \frac{f(\overline{z}_t)}{p_t(\overline{z}_t)} \tag{3.20}$$

where z_t is a camera path with t vertices. The paths are constructed unidirectionally staring from a sampled point within a pixel on an image plane and then tracing an outgoing ray to find first intersection. At the intersection point a path is completed by connecting to randomly sampled position on a light source (*next event estimation*), then an outgoing ray direction is sampled for next path vertex. Since the estimate contains an infinite sum of path samples and constructing a path is an expensive process involving path vertex visibility checking, the standard approach is to reuse previously generated path as the starting point for the next one, instead of creating completely new path that is one segment longer. To avoid constructing paths infinitely, the path generation is possibly stopped by Russian Roulette technique 3.5.2, for example, by having high stopping probability when path throughput is low.

3.6.3 Bidirectional Path Tracing

The path tracing algorithm is simple, but has difficulty finding paths to light sources that are occluded by objects, handling light caustics (paths where light is scattered by specular interactions before landing on a diffuse surface). Bidirectional path tracing (BPT) [LWS93] [Vea98] addresses these issues by combining light paths and camera paths, we will call them *sub-paths*.

Connecting the light and camera sub-paths potentially results in multiple possible ways (techniques) to create a path of a given length. For example having a light sub-path $\overline{y} = y_0 y_1$ and camera sub-path $\overline{z} = z_0 z_1$ both of length 1, there are two ways of creating a path of length 2, $= y_0 y_1 z_0$ and $= y_0 z_1 z_0$. The paths with varying number of light vertices s and camera vertices t, *techniques (s,t)*, correspond to different density functions $p_{s,t}$ on the path space, they effectively account for different lightning effects. [Vea98] To take advantage of this Veach proposed to use MIS to combine them by computing the pixel measurement I as

$$\langle I \rangle = \sum_{s \geq 0, t \geq 0} w_{s,t}(\overline{x}_{s,t}) \frac{f(\overline{x}_{s,t})}{p_{s,t}(\overline{x}_{s,t})} \tag{3.21}$$

where $p_{s,t}(\overline{x}_{s,t})$ is the probability density for generating a path $\overline{x}$ using the technique (s,t) and $w_{s,t}(\overline{x}_{s,t})$ is the MIS weight for this path. The sum over all lengths of s and t, represents all possible connection techniques, but use only one sample from each technique, therefore it represents only the inner sum in the MIS equation 3.14.

The standard approach to implement BPT algorithm is to generate 1 light and camera sub-path per pixel and then connect all pairs of their vertices. The connection process involves computation of MIS weight which requires to iterate over all light and camera vertices. This process can be made more effective by computing partial MIS terms for each vertex while doing a random walk through the scene, so that all required terms for MIS weight computation are available for any 2 given vertices. [Geo12] [Ant11]

3.7 Vertex Connection and Merging

Bidirectional path tracing in combination with multiple importance sampling is effective in diminishing weight for sampling techniques that are inappropriate, e.g., has low PDF and would cause variance spikes. This weighting is only possible if alternative sampling techniques exist (non zero PDF). Specular-diffuse-specular paths can only be found unidirectionally, since it is not possible to connect to a vertex of a specular surface where reflected direction is strictly defined. Vertex Merging is a path sampling technique that addresses this problem. It is conceptually related to the Photon Mapping [WJ01].

Georgiev et al. [GKDS12] [GSK11] introduced the Vertex Connection and Merging (VCM) algorithm that combines vertex connection path sampling technique used in BPT and vertex merging using multiple importance sampling 3.5.4. Vertex merging can intuitively be thought to concatenate two sub-paths by welding

their endpoints if they lie within a given distance r to each other (Fig. 3.4). The work of Hachisuka et al. [PJH12] addresses the same problem but with significantly different theoretical formulation.

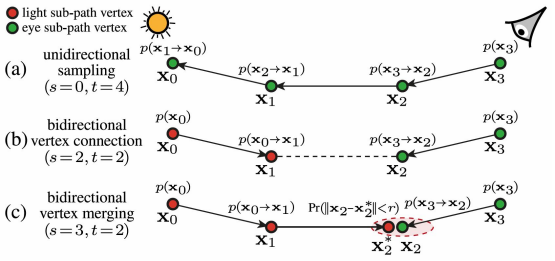


Figure 3.4: Different techniques for sampling a light path, with the corresponding PDF terms associated with each vertex. For paths with k edges (here k=3) bidirectional path tracing provides $k+2$ sampling techniques. Vertex merging brings $k-1$ new techniques corresponding to merging at the k-1 different interior path vertices. Used with permission from [GKDS12]

3.7.1 Vertex Connections

The vertex connection part of the algorithm is the same as vertex connections in BPT. Here we will just define PDF of a vertex connected path with s light and t camera vertices as

$$p_{vc,s,t}(\overline{x}) = p_s(\overline{y})p_t(\overline{z}) \tag{3.22}$$

where $\overline{y}$ and $\overline{z}$ are light and camera sub-path PDFs.

3.7.2 Vertex Merging

3.7.2.1 Path PDF

Vertex merging path sampling technique determines if light and camera sub-path vertices (end therefore the sub-paths leading up to these vertices) should be merged by checking if camera path vertex is within the acceptance radius (Fig. 3.5 right). We will consider the light path vertex where this is evaluated as x_s^*, the endpoint of light sub-path with vertices $x_0 x_1 \dots x_{s-1} x_s^*$

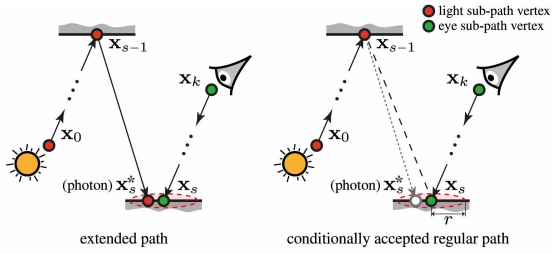


Figure 3.5: **Left:** Photon mapping can be considered to sample *extended paths* $\overline{x}^*$ of length k that has k+2 vertices, photons x_s^* in neighborhood of x_s are used to estimate incident radiance. **Right:** To remain compatible with the path integral framework for BPT, vertex merging is interpreted to sample *regular paths* $\overline{x}$ of same length k, but with k+1 vertices. Path is accepted if x_s^* lies within distance r from x_s. Used with permission from [GKDS12]

Consider a vertex x_s of the camera sub-path $x_s \dots x_k$ with t vertices where x_k is on the camera sensor and x_s has landed near the light sub-path vertex x_s^*. If x_s^* is within the acceptance radius then the following path is constructed

$$\overline{x}_k = x_0 \dots x_{s-1} x_s \dots x_k \tag{3.23}$$

and the path PDF is

$$p_{vm,s,t}(\overline{x}) = p_{vc,s,t}(\overline{x}) \cdot P_{acc,s,t}(\overline{x}) \tag{3.24}$$

where $P_{acc,s,t}(\overline{x})$ is the path acceptance probability. Note that there is no x_s^* in the path. PDF was expressed this way to allow using this technique together with vertex connection sampling in MIS estimator which requires both PDFs to be expressed with respect to the same area measure. x_s^* is used as the acceptance condition, it is interpreted as the Monte Carlo sample used to estimate the integral and serves as the Russian Roulette random variable. [GKDS12]
The acceptance probability is defined as

$$\begin{aligned} P_{acc}(\overline{x}) = Pr(\|x_s - x_s^*\| < r) &= \int_{A_M} p(x_{s-1} \to x) \\ &\approx \pi r^2 p(x_{s-1} \to x_s^*), \end{aligned} \tag{3.25}$$

where $A_M = \{x \in M | \ \|x_s - x\| < r\}$ is the set of surface points within distance r of x_s. $p(x_{s-1} \to x_s^*)$ is the probability of sampling a vertex x_s^* from x_{s-1} and is multiplied with the circle area around x_s. The result is obtained using common assumptions used in progressive photon mapping that the PDF within the circle area (points in A_M) is constant and that the surface is locally flat [GKDS12].

This approximately gives the probability of having sampled any point within the circle. In case x_s^* comes from a specular vertex x_{s-1} the $P_{acc} = 1$.

3.7.2.2 Contribution function estimator

Here we briefly describe derivation of the *vertex merging contribution function* estimator. For more detailed derivation and discussion we refer to the Appendix A in the original work [GKDS12].

The measurement contribution function for vertex merging does account for x_s^* and considers contribution from an *extended path*(Fig. 3.5 left)

$$\overline{x}_k = x_0 \dots x_{s-1} x_s^*, x_s \dots x_k. \tag{3.26}$$

It is extended in the sense that it contains an additional vertex x_s^* compared to *regular paths* (all complete paths discussed before). The *measurement contribution function for extended paths* (note, not *vertex merging measurement contribution*) is defined as

$$\begin{aligned} &f(\overline{x}^*) = \\ &L_e(x_0)T(x_0 \dots, x_s^*)K_r(||x_s^* - x_s||)f_s(x_s^*, x_s)T(x_s \dots, x_k)W_e(x_k) \end{aligned} \tag{3.27}$$

where $f_s(x_s^*, x_s) = f_s(x_{s-1} \to x_s^*; x_s; x_s \to x_{s+1})$ denotes the BSDF at x_s evaluated for incoming direction $x_{s-1} \to x_s^*$ and outgoing direction $x_s \to x_{s+1}$. $K_r(||x_s^* - x_s||)$ is the density estimation kernel with support radius r similarly as in photon mapping [WJ01] that weights photons depending on their proximity to estimation point. We will use a simple $K_r = \frac{1}{\pi r^2}$ that weights all particles (photons) equally.

Due to the extra vertex x_s^* compared to regular paths, the *vertex merging measurement contribution function* is defined as an integration of extended path measurements over the scene surfaces M:

$$f_{vm} = \int_M f(\overline{x}^*) dA(x_s^*). \tag{3.28}$$

One-sample Monte Carlo estimate of f_{vm} can be obtained using x_s^* (photon) as

$$\langle f \rangle_{vm}(\overline{x}) = f(\overline{x}^*) \Big/ \left[\frac{p(x_{s-1} \to x_s^*)}{\int_{A_M} p(x_{s-1} \to x)} \right] = f(\overline{x}^*) \pi r^2, \tag{3.29}$$

where $p(x_{s-1} \to x_s^*)$ is the probability of sampling the x_s^* on a given surface which is normalized using integral over surface points within circle area to obtain a valid surface PDF, since only points within merging radius survive the

Russian Roulette. The end result is obtained using same assumptions as for path acceptance probability 3.25.

Finally the vertex merging estimator can be defined using the VM path PDF $p_{vm}(\overline{x})$ as

$$\langle I \rangle_{vm}(\overline{x}) = \frac{\langle f \rangle_{vm}(\overline{x})}{p_{vm}(\overline{x})} = \frac{f(\overline{x}^*)\pi r^2}{p_{vm}(\overline{x})} = \frac{f(\overline{x}^*)}{p(x_{s-1} \to x_s^*)p_{vc}(\overline{x})}, \tag{3.30}$$

where the end result is obtained by observing that πr^2 is present also in p_{vm} and therefore cancels out. Notice that measurement contribution function uses an extended path, but p_{vc} uses a regular path, thus allowing VM estimator to be combined using multiple importance sampling with BPT. The potential efficiency of vertex merging stems from the fact that we can merge large number number of vertices without the need of checking visibility between vertices as for vertex connections.

3.7.3 Combined VCM estimator

Having defined vertex merging as a sampling technique for regular paths, it can be combined with vertex connection (BPT) using the MIS multi-sample estimator as

$$\begin{aligned} \langle I \rangle_{VCM} = & \frac{1}{n_{vc}} \sum_{l=1}^{n_{vc}} \sum_{s \geq 0, t \geq 0} w_{vc,s,t}(\overline{x}_{s,t}) \langle I \rangle_{VC}(\overline{x}_{s,t}) + \\ & \frac{1}{n_{vm}} \sum_{l=1}^{n_{vm}} \sum_{s \geq 2, t \geq 2} w_{vm,s,t}(\overline{x}_{s,t}) \langle I \rangle_{VM}(\overline{x}_{s,t}) \end{aligned} \tag{3.31}$$

where n_{vc} is the number of light sub-paths whose vertices are connected to vertices of one camera sub-path for a given pixel estimate (number of samples or path pairs for the VC/BPT estimator 3.21), n_{vm} is the total number of light sub-paths (samples for VM estimator), $w_{vc,s,t}$ and $w_{vm,s,t}$ are the power heuristic weights. The sum variables for VM estimator start at 2 since both light and camera sub-paths need to have at least 2 vertices (no need to merge vertices on light source and camera). The power heuristic weight for the technique (v, s, t)

where $v \in \{vc, vm\}$ is

$$
\begin{aligned}
w_{v,s,t}(\overline{x}) &= \frac{n_v^\beta p_{v,s,t}^\beta(\overline{x})}{n_{vc}^\beta \sum_{s'\geq 0, t'\geq 0} p_{vc,s',t'}^\beta(\overline{x}) + n_{vm}^\beta \sum_{s'\geq 2, t'\geq 2} p_{vm,s',t'}^\beta(\overline{x})} \\
&= \frac{1}{\frac{n_{vc}^\beta}{n_v^\beta} \sum_{s'\geq 0, t'\geq 0} \frac{p_{vc,s',t'}^\beta(\overline{x})}{p_{v,s,t}^\beta(\overline{x})} + \frac{n_{vm}^\beta}{n_v^\beta} \sum_{s'\geq 2, t'\geq 2} \frac{p_{vm,s',t'}^\beta(\overline{x})}{p_{v,s,t}^\beta(\overline{x})}}
\end{aligned}
\tag{3.32}
$$

The weight takes into account all possible ways of sampling $\overline{x}$ with vertex connection (left side sum) and merging (right side sum) by considering all pairs of light and camera sub-paths with s' and t' vertices. If path constructed by other (s,t) technique will have a larger PDF, the weight of the given path will get scaled down.

As pointed out in [GKDS12] the images produced by the combined algorithm will contain systematic error (bias) in the form of blur due to vertex merging. It can be made consistent 3.5.1 by progressively reducing the merging radius r and accumulating the resulting images. The proposed radius reduction technique by the original paper is

$$r_i = r_1 \sqrt{i^{\alpha-1}} \tag{3.33}$$

where r_1 is the initial radius and $\alpha \in (0, 1)$ is a user parameter that controls radius reduction rate. The authors show that optimal convergence rate is achieved using $\alpha = 2/3$.

3.7.4 Efficient path weight evaluation

As it can be seen in the power heuristic formula 3.32 the weight evaluation requires computing PDFs for all other paths. In the technical report of the reference implementation Georgiev et al. [Geo12] proposed an efficient recursive approach to accomplish this. It is quite extensive, therefore we will not describe it here, we will only point to the key factors useful for understanding it.

The approach proposed by [Vea98] for BPT (the VC sum, on the right in the denominator of 3.32) was to iterate once over light and camera sub-path vertices and accumulate PDF fractions. This approach is suboptimal since every time a sub-path is reused, the same terms are recomputed (and accessed in memory). For example, for light sub-paths with 3 and 4 vertices, the PDFs of first 3 vertices will be the same. This could be a significant overhead for vertex merging which relies heavily on sub-path reuse.

Suppose we have sampled a path with $s = 4$ light and $t = 2$ camera vertices

$$\overline{y}_s\overline{z}_t \equiv y_0y_1y_2y_3z_1z_0 \equiv x_0x_1x_2x_3x_4x_5x_6 \equiv \overline{x}_k. \tag{3.34}$$

The PDF for such path is

$$\overrightarrow{p}_0(\overline{y})\ldots\overrightarrow{p}_3(\overline{y})\overrightarrow{p}_1(\overline{z})\overrightarrow{p}_0(\overline{z}) \tag{3.35}$$

where $\overrightarrow{p}$ denotes a *forward* PDF with respect to the path "growth" direction (light and camera paths grow towards each other, i.e. in opposite directions). As 3.34 shows, the light and the camera path abbreviations y and z are only for convenience. We can similarly express z_1z_0 as y_4y_5, and use *reverse* PDFs $\overleftarrow{p}_4(\overline{y})\overleftarrow{p}_5(\overline{y})$ to describe their sampling probabilities (they still are part of the camera path). The complete path PDF with respect to light path growth (light propagation) direction then is

$$\overrightarrow{p}_0\overrightarrow{p}_1\overrightarrow{p}_2\overrightarrow{p}_3\overleftarrow{p}_4\overleftarrow{p}_5 \tag{3.36}$$

Consider a possible alternative path created using these same vertices with technique with $s' = 3$ $t' = 3$, its PDFs then is

$$\overrightarrow{p}_0\overrightarrow{p}_1\overrightarrow{p}_2\overleftarrow{p}_3\overleftarrow{p}_4\overleftarrow{p}_5 \tag{3.37}$$

The VC sum fraction (right sum in the denominator of 3.32) for these paths then is

$$\frac{\overrightarrow{p}_0\overrightarrow{p}_1\overrightarrow{p}_2\overleftarrow{p}_3\overleftarrow{p}_4\overleftarrow{p}_5}{\overrightarrow{p}_0\overrightarrow{p}_1\overrightarrow{p}_2\overrightarrow{p}_3\overleftarrow{p}_4\overleftarrow{p}_5} = \frac{\overleftarrow{p}_3}{\overrightarrow{p}_3} \tag{3.38}$$

Similarly the fraction term for alternative technique with $s' = 2$ and $t' = 4$ is

$$\frac{\overrightarrow{p}_0\overrightarrow{p}_1\overleftarrow{p}_2\overleftarrow{p}_3\overleftarrow{p}_4\overleftarrow{p}_5}{\overrightarrow{p}_0\overrightarrow{p}_1\overrightarrow{p}_2\overrightarrow{p}_3\overleftarrow{p}_4\overleftarrow{p}_5} = \frac{\overleftarrow{p}_2\overleftarrow{p}_3}{\overrightarrow{p}_2\overrightarrow{p}_3} \tag{3.39}$$

We can observe that most of the terms cancel out and the ones that remain are for vertices that use PDFs of different sampling directions. If we only consider alternative techniques with $s' < s$ all the VC fraction sum is

$$\frac{\overleftarrow{p}_0\overleftarrow{p}_1\overleftarrow{p}_2\overleftarrow{p}_3}{\overrightarrow{p}_0\overrightarrow{p}_1\overrightarrow{p}_2\overrightarrow{p}_3} + \frac{\overleftarrow{p}_1\overleftarrow{p}_2\overleftarrow{p}_3}{\overrightarrow{p}_1\overrightarrow{p}_2\overrightarrow{p}_3} + \frac{\overleftarrow{p}_2\overleftarrow{p}_3}{\overrightarrow{p}_2\overrightarrow{p}_3} + \frac{\overleftarrow{p}_3}{\overrightarrow{p}_3} \tag{3.40}$$

For techniques with $s' > s$ the VC fraction sum is

$$\frac{\overrightarrow{p}_4\overrightarrow{p}_5}{\overleftarrow{p}_4\overleftarrow{p}_5} + \frac{\overrightarrow{p}_4}{\overleftarrow{p}_4} \tag{3.41}$$

The sums 3.40 and 3.41 represent the terms that can be partially evaluated when tracing respectively light and camera sub-paths. Partially because some

of the terms depend on vertices that are not yet known during the tracing stage, for example $\overrightarrow{p}_4$ in the camera sum 3.41 represents the probability of sampling a camera path vertex $y_4 \equiv z_1$ from the light path endpoint y_3.
The sum 3.40 can be expressed recursively as

$$\begin{aligned} w_{vc,0} &= \frac{\overleftarrow{p}_0}{\overrightarrow{p}_0} \\ w_{vc,i} &= \frac{\overleftarrow{p}_i}{\overrightarrow{p}_i}(1 + w_{vc,i-1}) \end{aligned} \tag{3.42}$$

A similar idea as demonstrate above is used in [Geo12] to derive the full recursive quantities for both the VC and VM path weights. As discussed in the example with $\overrightarrow{p}_4$, there are terms that depend on yet unknown quantities, therefore it is not possible to fully precompute these sums for each sub-path as we trace. But it is possible to split the computation and precompute using all the terms that are known and complete the evaluation of these recursive quantities once the vertices are connected and merged. The resulting approach is to precompute 3 recursive quantities dVC, dVM and dVCM for each path vertex that enables efficient computation of path weight using only information stored at the vertices being connected or merged.

Chapter 4

Implementation

4.1 OptiX

OptiX is a general purpose ray tracing engine provided by Nvidia that builds on CUDA's parallel computation capabilities [PBD+10]. We chose to use it for our implementation since allows to focus on the algorithm rather than optimizations for GPU architectures. OptiX requires developers to provide a set of programs (similarly to shader programs in OpenGL and DirectX) that get called at specific places in the pipeline and allows to implement desired algorithms. The main types of programs are:

- **Ray generation** - is the entry point of the pipeline. It is expected to generate rays and possibly provide arbitrary payload data for them. There can be multiple entry points with different ray generation programs that can be used to implement multi-pass algorithms;
- **Intersection** - gets called to determine intersections with arbitrary types of geometry;
- **Bounding box** - gets called to compute axis-aligned bounding box for geometry during acceleration structure build;
- **Closest hit** - gets called when closest ray intersection has been found. This is the place where object shading typically will be done;

- **Any hit** - allows to check if a ray intersects anything. Typically used for shadow rays;
- **Miss** - gets called if a ray does not hit any geometry;
- **Exception** - gets called when exception such as stack overflow or an invalid ray has been encountered.

OptiX consists of host-side and device-side APIs. The host side API provides functions to create and configure OptiX context, load and bind programs, create materials, geometry, to define acceleration structure, create data buffers and textures and eventually launch the ray generation program. The devide-side API used in OptiX programs provides access to special functions such as reporting ray-object intersections, tracing a ray, and also many optimized utility functions common in rendering algorithms such as hemisphere sampling routines. These programs are CUDA kernels and therefore can leverage also functionality provided by CUDA framework.

The programs get compiled into Parallel Thread Execution functions (.PTX file) which is a virtual assembly language and implements low-level virtual machine similar to the open-source LLVM [1] [PBD+10]. The PTX is defined from the perspective of a single thread, thus gives the OptiX runtime the ability to manipulate and optimize the resulting code. This is performed before the first context launch after all OptiX programs have been loaded. OptiX runtime performs Just-In-Time (JIT) compilation, this involves combining all provided programs into one or more kernels, performing optimizations specific to GPU architecture used on a given machine, analyzing scene graph to identify data-dependent optimizations.

As shown by Aila [LA09] [ALK12], currently a monolithic mega-kernel is considered the best approach since it minimizes context launch overhead. OptiX currently favors this approach and requires all function calls to be inlined and may fail at runtime during JIT compilation if done otherwise.

4.2 OppositeRenderer

Instead of writing everything from scratch it was decided to extend an existing OptiX based renderer OppositeRenderer [2] that contains implementations of the progressive photon mapping (PPM) and the path tracing (PT), scene loading capabilities, an interactive interface and a support for network based multi-workstation rendering. It was hoped that it may be possible to leverage GPU

[1] `http://llvm.org/`
[2] `http://apartridge.github.io/OppositeRenderer/`

the grid data structures used in PPM for light vertex storage for use in Vertex Merging. Here were briefly describe the renderer and its project structure. For more details and the user guide we refer the reader to the work of the author Stian Pedersen [Ped13]. Figure 4.1 shows the interface of the renderer.

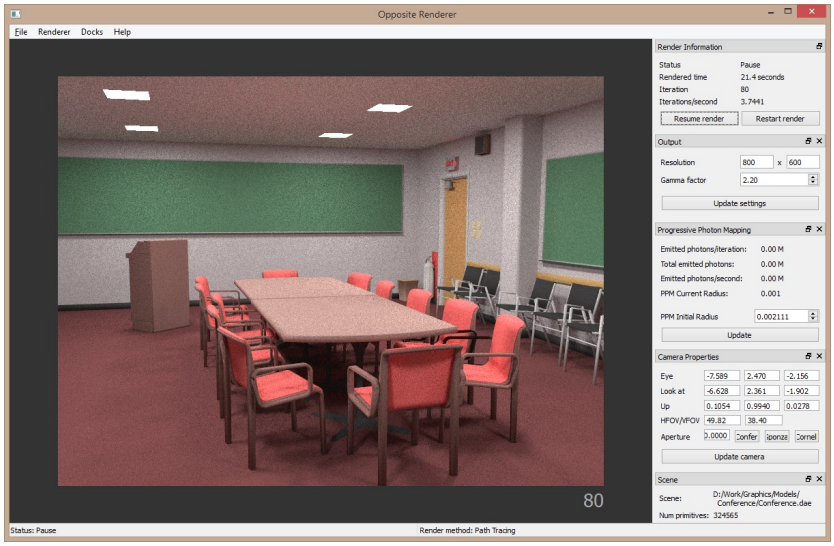


Figure 4.1: OppositeRenderer iterface

The solution for the Windows application is built using Microsoft Visual Studio and uses QT Framework [3]. It is modular and consists of five C++ projects:

- **RenderEngine** - a library that contains all the functionality responsible for rendering an image using a single GPU;
- **Standalone** - an application that uses RenderEngine to render a scene
- **Server** - an application that uses RenderEngine to render a scene requested by a connected Client and sends the renderer image to it over the network. There can be multiple Server instances running on a machine, thus allowing to leverage multiple GPUs;
- **Client** - an application that connects to one or more Servers over the network and controls the rendering process;
- **Application** - contains common functionality for Standalone, Server and Client projects, such as parts of GUI and scene loading using Asset Import Library (Assimp) [4].

[3] `http://qt-project.org`
[4] `http://assimp.sourceforge.net`

The core rendering related functionality of the RenderEngine is in the OptixRenderer class. It initializes and binds OptiX programs, buffers and launches context entry point for the desired algorithm. The Scene class loads textures, creates materials, geometry and the OptiX scene graph from scene representation of Assimp.

4.3 The VCM algorithm

In this section we provide an overview of the VCM algorithm implementation. Due to various issues experienced during the development we were severely delayed, as a consequence we managed to implement the vertex connection part of the algorithm which is a bidirectional path tracer, but vertex merging part is missing, although some parts essential for it such as recursive MIS weight computation are done. Some of the issues are accounted in the Preface , therefore we will not repeat them again. The author plans to continue working on the project and implement the vertex merging as well, therefore in this section for completeness we consider the full VCM algorithm. The Fig. 4.2 and the psoudocode listing 1 gives an overview of the algorithm.

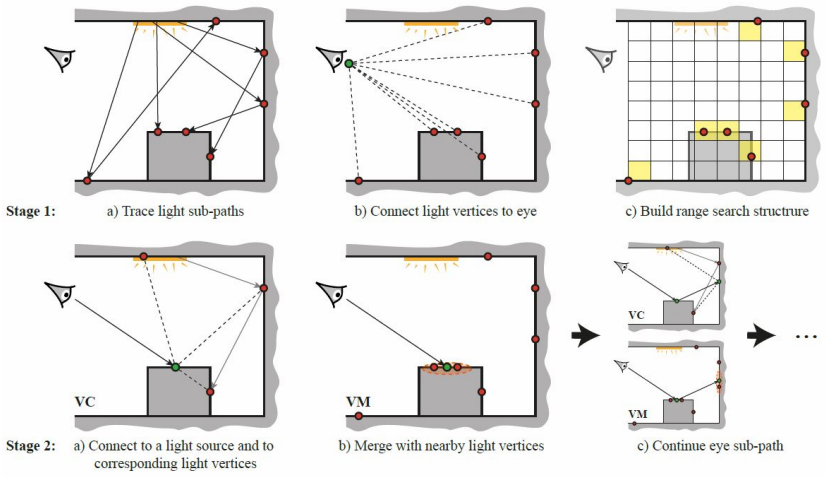


Figure 4.2: Stages of the VCM algorithm. Used with permission from [Geo12]

The rendering is performed in two stages. In the first stage we trace light subpaths from light sources and store their vertices (lines 3-17) by performing a

Algorithm 1 Pseudocode for VCM algorithm given a merging radius r and $numLightPaths = numPixels$, estimate 3.31 sample count $n_{vc} = 1$ and $n_{vm} = numLightPaths$

```
1: procedure VertexConnectionAndMerging(r)
2:     ▷ 1. Light path sampling
3:     for i = 1 to numPixels do
4:         lightVertex = TraceRay(SampleLightPoint())
5:         while lightVertex is valid do
6:             if lightVertex is not specular then
7:                 if lightVertex is not first then
8:                     lightPaths[i] += lightVertex
9:                 end if
10:                ▷ Vertex connection (VC) - project light vertex to camera pixel
11:                lightVertexOnCamera = ConnectToCamera(lightVertex)
12:                Accum(lightVertexOnCamera, VC, r, GetPixel(lightVertexOnCamera))
13:            end if
14:            lightVertex = ContinueRandomWalk(lightVertex)
15:        end while
16:    end for
17:    rangeStruct = BuildRangeSearchStruct(lightPaths)
18:    ▷ 2. Camera path sampling
19:    for j = 1 to numPixels do
20:        cameraVertex = TraceRay(SamplePixel(j))
21:        while cameraVertex is valid do
22:            ▷ Vertex connection (VC) - random light hit, unidirectional sampling
23:            if cameraVertex is emissive then
24:                Accum(cameraVertex, VC, r, j)
25:            end if
26:            ▷ Vertex connection (VC)
27:            for lightVertex in lightPaths[j] ∪ SampleLightPoint() do
28:                Accum(Connect(cameraVertex, lightVertex), VC, r, j)
29:            end for
30:            ▷ Vertex merging (VM)
31:            for lightVertex in RangeSearch(rangeStruct, cameraVertex, r) do
32:                Accum(Merge(cameraVertex, lightVertex, r), VM, r, j)
33:            end for
34:            cameraVertex = ContinueRandomWalk(cameraVertex)
35:        end while
36:    end for
37: end procedure
38:
39: ▷ Accumulates the pixel measurement due to a given path
40: procedure Accum( path, technique, r, i )
41:    contrib = MeasurementContribution(path, technique, r)
42:    pdf = Pdf(path, technique, r)
43:    weight = PowerHeuristic(path, technique, pdf)
44:    if technique = VC then
45:        image[i] += weight * contrib/pdf
46:    else
47:        image[i] += weight * contrib/(pdf * numLightPaths)
48:    end if
49: end procedure
```

random walk. We use the same number of light sub-paths as the number of pixels. The vertices are stored only on non-specular surfaces since only these can be connected or merged, we also do not store the first path vertex to reduce correlation (lines 6-9), instead connection points on light sources are sampled directly. Specular BSDFs contain delta distribution [PH10] which means that the PDF for any arbitrary direction is zero, except for exactly the one of specular reflection direction. We also need to connect the light vertices to the camera (lines 11-12), this corresponds to path sampled with technique $(s > 0, t = 1)$ (one camera path vertex). This step possibly could be done after all light sub-paths are traced as shown in Fig. 4.2, but we choose to do it while tracing each path. We project the vertex to camera image plane to obtain the origin position of the ray that would have hit the current light vertex position. The procedure *Accum* (lines 40-49) computes the path contribution estimate, evaluates MIS weight and contribution to the pixel. After the tracing the range search data structure should be built for the vertices for use in vertex merging (line 18). Same as the reference implementation of VCM [Geo12] we ignore light paths that randomly hit camera lens, technique $(s > 0, t = 0)$, since the probability of this happening is very low or even zero in case of pinhole camera model where the camera aperture is described as a point.

In the second stage we trace camera sub-path for each pixel. Upon sampling a vertex on a light source, we accumulate the emitted radiance (lines 21-23), this corresponds to a path sampled with the technique $(s = 0, t > 0)$. If the sampled vertex is not on a light source, it is connected to all vertices of one light sub-path (lines 27-29), it corresponds to the technique $(s > 1, t > 1)$. Since we do not store the first vertex of a light path, we connect to a randomly sampled point on a light source, it corresponds the technique $(s = 1, t > 1)$. Finally we merge vertices that lie within the radius r from the $cameraVertex$ (lines 31-33), it corresponds to the technique $(s > 1, t > 1)$.

4.4 BPT with recursive MIS weights

In this section we present the implementation of the bidirectional path tracing with recursive MIS weight computation that was implemented as part of work on VCM. This can be seen as a special case of the full VCM algorithm when having $n_{vm} = 0$ samples from vertex merging estimator 3.31. Since our goal was the complete VCM, this abbreviation is used every in the source code. Our extended version of the OppositeRender can be found on GitHub [5]. To allow evaluating the extent of author's contributions, the author's name was added to the contributors list in headers of modified files.

[5] `http://github.com/voldemarz/OppositeRenderer`

4.4.1 Structure

We briefly described the structure of the OppositeRenderer solution in the section 4.2 and mentioned that the rendering related code in is RenderEngine project. The core of our added code is located in the "RenderEngine/renderer/vcm" subdirectory of the renderer. We also made numerous additions, modifications and fixes all around in the existing solution files.

Our implementation follows quite closely the reference implementation of VCM in SmallVCM [6]. We also used the recursive MIS term computation proposed in the technical report of SmallVCM [Geo12], it requires tracking of 3 quantities dVC, $dVCM$, dVM (dVM is not used for BPT) with the path state and storing with each path vertex, this later allows to compute MIS weight without traversing all path vertices. All the "tech. rep. (#)" comments with equation numbers refer to that paper. We also forked the SmallVCM project on GitHub [7] and used as a verification tool for our implementation by setting up same scene dimensions, light and material properties.

The following list summarizes main files and functions implementing the algorithm and how they relate to the pseudocode:

- VCMLightPass.cu - light pass entry point program, contains the while loop that corresponds to the first stage described in pseudocode 1;
- VCMCameraPass.cu - camera pass entry point program, contains the while loop that corresponds to the second stage described in pseudocode 1;
- mis.h - recursive MIS quantity initialization and update functions;
- vcm.h - contains the core parts of the implementation:
  - lightHit() - is called from *closest hit* programs (material shaders), corresponds to the inner part of the light tracing loop. If a ray survives the Russian Roulette, a new ray direction is sampled using the surface BSDF;
  - connectCameraT1() - is called from lightHit(), corresponds to lines 11-12 in the pseudocode;
  - cameraHit() - is called from *closest hit* programs (material shaders), corresponds to the inner part of the camera tracing loop. If a ray survives the Russian Roulette, a new ray direction is sampled using the surface BSDF;
  - connectLightSourceS0() - is called from *DiffuseEmitter* material *closest hit* program (pseudocode line 24);
  - connectLightSourceS1() - is called from cameraHit(), connects camera sub-path vertex to sampled point on a light source (psudocode

[6] `http://www.smallvcm.com`
[7] `http://github.com/voldemarz/SmallVCM`

lines 27-28);
- connectVertices() - is called in from cameraHit() from a loop over light sub-path vertices being connected to the given camera sub-path vertex (psudocode lines 27-28)
- sampleScattering() - is called by lightHit() and cameraHit() to sample a new ray direction using the BSDF. The tracing is possibly stopped here using the Russian Roulette and a corresponding flag set in the ray payload (psudocode lines 14 and 34).

When OptiX launches an entry point program, it creates many threads that execute the same program in parallel. The number of threads is determined by the launch dimensions, which in our case is equal to output image dimensions. Both light and camera pass entry point programs initialize a ray and its payload, and then traces the ray in a loop until the stopping flag is set in the ray payload. The ray payload is a data structure that is passed along with the ray and the main information it contains is

- ray origin;
- ray direction;
- path throughput;
- path length;
- accumulated full path contribution due to vertex connections;
- recursive MIS quantities;
- boolean flag identifying if tracing should be stopped.

The ray origin and direction is updated in lightHit() and cameraHit() functions called by *closest hit* programs (materials shaders) if it survives the Russian Roulette. The tracing is also stopped if the ray fails to hit geometry, in this case a *miss* program is called that sets the stopping flag.

We used an iterative approach for casting rays from the loop in entry point program instead of doing recursively and casting new rays from closest hit programs to avoid issues due stack size limitations. Doing it recursively requires increasing stack size for threads, which significantly degrades performance and increases memory requirements. An iterative approach is generally the preferred way in OptiX and most OptiX SDK samples use the same approach.

4.4.1.1 Passing buffers

As described above the core of the implementation is in the vcm.h header file. The functions defined in this file need access to various buffers. OptiX doesn't allow using buffer pointers since they are subject to being moved if required for

memory optimization, instead buffer data is accessed via handles which are not allowed to be passed as function parameters. Fortunately since version 3.5 it is possible to construct a buffer handle using buffer ID, which can then be used to access data [Cor14]. We used this new feature to pass buffer IDs to functions defined in vcm.h. This allowed to keep all related code together, thus it is easier to manage and comprehend.

4.4.2 BSDF and BxDF classes

As we described in section 3.2 the reflection of a surface is described by its BRDF (or BSDF in general case that includes transmission). Scattering from realistic surfaces if often best described by a mixture of multiple BSDFs (reflection models). The existing material implementations in the renderer performed contribution computation in the closest hit programs. To implement vertex connections we had to be able to store material information with the vertex and evaluate BSDF when connecting vertices. Therefore, we added two separate classes that allow us to characterize a material by combining multiple reflection models and store the necessary information together with vertices:

- BxDF - a base class that should be derived by an implementation of a reflection model
- BSDF - a class that describes a material reflection by combining multiple BxDFs

The BxDF and BSDF classes are customized versions of those found in PBRT [8], the companion renderer for Physically Based Rendering textbook [PH10]. These classes provide methods for evaluating the reflected radiance and the PDF for given incoming and outgoing directions, and also to sample a new direction. When the BSDF sampling method is called the reflection model for sampling is chosen uniformly.

To add support for materials present in the renderer (Diffuse, Glass, Mirror, Texture) we implemented the *Lambertian*, *Specular Reflection* and *Specular Transmission* reflection models following [PH10]. We also added the *Modified Phong* reflection model [LW94] similarly as in SmallVCM and used it in combination with *Lambertian* to create a Glossy material. Currently, it only works with our BPT implementation, the path tracer and photon mapper just use the diffuse component.

[8] `http://www.pbrt.org`

To simplify the implementation we have the VcmBSDF class derived from the BSDF class that additionally stores the incident direction (this simplifies reverse PDF evaluation). It also picks BxDFs to sample with probability relative to their reflectance.

One caveat is that CUDA doesn't support virtual functions, therefore to work around this and make BSDF able to call the actual reflectance model implementation class methods we used a preprocessor macro that at compilation time expands to an if-else block which checks BxDF type ID, casts pointer to the target type and then calls the correct method.

When a ray hits geometry, its material closest hit (shading) program is called. At this point an instance BSDF is created and initialized using BxDFs that describe the material. Here we can potentially initialize any sophisticated BxDF, we could even pass an ID of a data buffer if it needs one for evaluation and sampling. This BSDF instance is then passed to *lightHit()* or *cameraHit()* functions as explained in section 4.4.1.

4.4.3 Light pass

As noted in section 4.4.1, the entry point program for the light path tracing pass casts rays and material closest hit programs call *lightHit()*. It updates the ray payload MIS terms, stores a light vertex in a buffer and connects to a camera if the surface is not specular using function *connectCameraT1()*. The camera connection results in a contribution added directly to an image output buffer, the receiving pixel is determined by projecting the vertex onto the image plane. The path is possibly stopped using the Russian Roulette, if it survives a new direction is sampled using BSDF and is set in the ray payload.

The light vertex is stored in a common buffer which is guarded by an atomic index counter. Vertex index pointing to the data is stored in sub-path vertex buffer. The light vertex data contains hit point, BSDF, throughput, path length and recursive MIS quantities. We also have a buffer where we count the number of vertices each path has stored.

Storing vertices randomly in a common buffer instead of allocating space each sub-path, allows to lower memory requirements and removes the need for path length limitation (at least for scenes that do not contain only mirrors). In the end we did set a maximum path length limit to 10 for performance reasons, so the result is not unbiased. This could be made as a user interface option to set path length limit if any.

The size of the vertex buffer is estimated by doing a light sub-path length estimate pass which is exactly the same as normal light pass, except no light vertices are stored. Then we scale it up to guarantee some extra safety margin and use it to evaluate necessary buffer size, unless it is larger than allowed maximum length.

Without the path length limitation, some paths reached lengths of more that 40 segments. This largely is because as a continuation probability in Russian Roulette (RR) we use the largest RGB component of the reflectance, and at least one of them often is close to 1. We choose the probability this way following the SmallVCM reference implementation with vertex merging in mind. The reasoning behind it is to ensure that particle weight does not rise after Russian Roulette. It is important in vertex merging (and photon mapping) for particles to have similar weights, otherwise particles with significantly larger weight than others cause noticeable spots in the resulting image [PH10]. We tried to use luminance as an alternative approach also proposed in SmallVCM, but it caused noticeably more noise.

4.4.4 Camera Pass

During the camera pass material hit programs call *cameraHit()*. The MIS terms are updated similarly as during the light pass. If the surface is not specular the camera vertex is first connected to a random light source using the function *connectLightSourceS1()* and contribution is added to the ray payload.

Next the camera sub-path vertex is connected to vertices of a light sub-path with a matching ID using *connectVertices()* and the resulting path contributions are accumulated in the payload. The ID of a path is the thread index which corresponds to 2D index within context launch dimensions and in our case both passes have same dimensions, i.e., number of threads. We first retrieve the number of vertices the light path has stored. Then in a loop we retrieve the vertex indices and next the vertices themselves.

If a ray hits the *DiffuseEmitter* material, its closest hit program calls *connectLightSourceS0* which computes the contribution from the light source.

When tracing is stopped due to Russian Roulette or missing any geometry, the entry point program exits the loop and stores the accumulated path contributions in the output buffer.

4.4.5 Shading with geometric normals

Our current implementation is using geometric normals for shading even if shading normals are available. We do so since the use of shading normals causes non-symmetric BSDFs as pointed out by Veach in the section 5.3 of [Vea98]. Veach also demonstrates how to solve this.

4.4.6 Final notes

The readme.md file in the root directory of the solution contains instructions on how to set up the development environment, the dependencies and build the solution. The solution is configured to use Visual Studio 2010 toolchain for Win32 build and Visual Studio 2012 toolchain for x64 to match the available precompiled versions of the Qt Framework 5.2.1/5.3 with OpenGL.

The minimal required OptiX version is 3.5 and the required CUDA Tookit version is 5.5. CUDA Toolkit 6 fails to inline VcmBSDF class sampling function call and eventually fails to compile.

Chapter 5

Results

This chapter presents the results obtained by our implementation of bidirectional path tracer. We first focus on the quality of the results and compare with the reference implementation and afterwards discuss the performance. All renderings were performed on a PC running Windows 8.1 x64 with Intel i7-4770K 4.0 GHz CPU, 16 GB RAM, Nvidia GeForce GTX 770 GPU (1536 CUDA cores, 1137/1189 MHz core clock , 2GB memory, 3.493 GFLOPS peak floating point performance).

5.1 Test scenes

As mentioned in the previous chapter we used the SmallVCM[1], a reference CPU implementation of VCM, to check correctness of our implementation. We forked the project on GitHub[2] and slightly modified. We rounded the coordinates of vertices defining scene geometry so it is easier duplicate the same in the OppositeRenderer, and to allow easier comparison of values when using the debugger. We also added few more scene variations. We duplicated the scenes from SmallVCM with area light and point light in the OppositeRenderer. These can be accessed through the "File→Open built-in scene" menu.

[1] `http://www.smallvcm.com`
[2] `http://github.com/voldemarz/SmallVCM`

Additionally we used two scenes that the author of OppositeRenderer used. One is the *Conference Room* scene containing diffuse surfaces except for a two glass exit signs. It contains eight area light sources and two point light behind exit signs. The other is *Crytek Sponza* scene that consists of textured diffuse surfaces and is illuminated by a single point lights that is positioned 60 meters above the ground. The height of the structure is about 11 meters.

5.2 Analysis

We used identically defined scenes and compared the output of our implementation with the output from SmallVCM using the image processing tool ImageMagick [3] for subtracting images from each other.

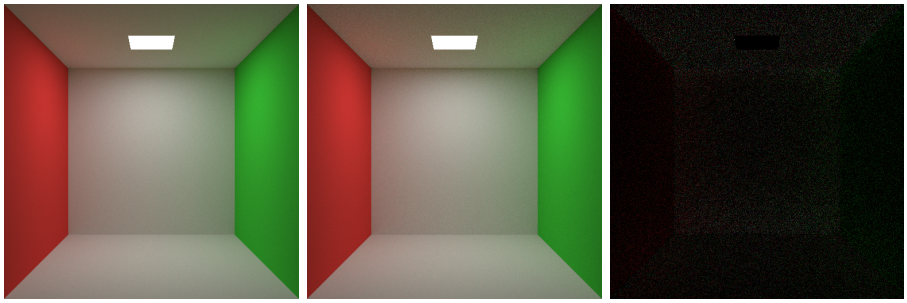
5.2.1 Empty Cornell Box

The Figure 5.1 shows an empty Cornell Box scene images generated by both renderers using 200 iterations. For both renderers we account the time spent and the number of iterations per second, for our implementation we also list the number of light vertices stored per light path and the average execution time of each pass. By observing results closely it is possible to see that, surprisingly, the SmallVCM image exhibits more noise and took about 15 % more time to compute. The subfigure 5.1c shows a 5x scaled difference image.

5.2.2 Cornell Box with spheres

The Figure 5.2 shows the Cornell scene from the SmallVCM with glass and mirror spheres that we also added to the OppositeRender.

[3] `http://www.imagemagick.org`

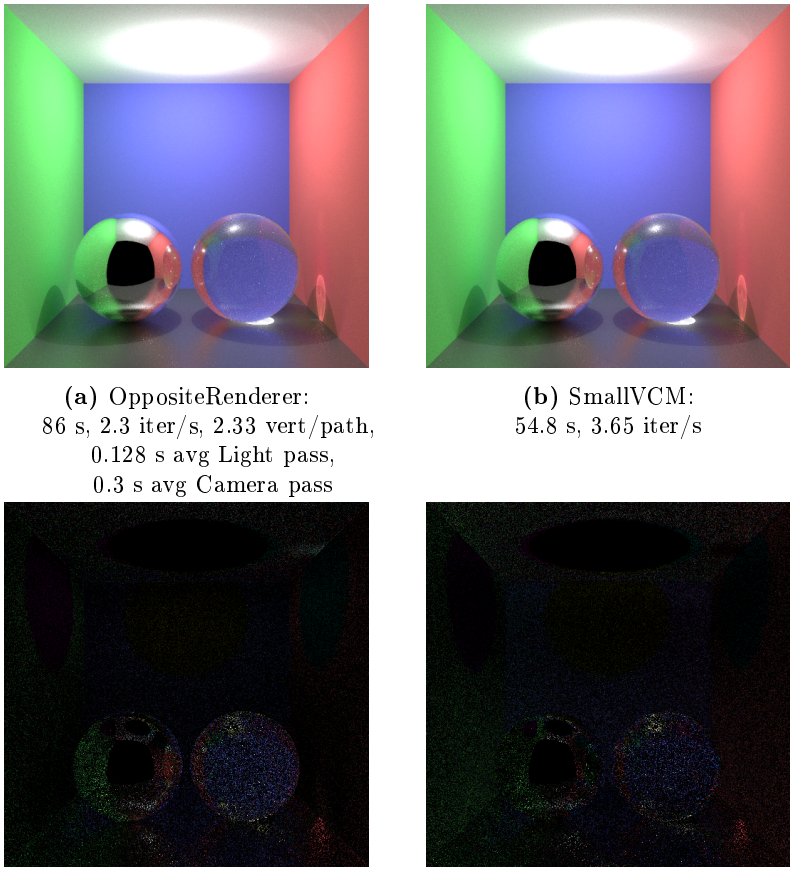


(a) OppositeRenderer: 61 s, 3.3 iter/s, 2.41 vert/path, 0.075 s avg Light pass, 0.228 s avg Camera pass

(b) SmallVCM: 69.6 s, 2.87 iter/s

(c) SmallVCM - OppositeRenderer 5x scaled

Figure 5.1: Comparison of empty Cornell Box scene (200 iterations, 512x512 resolution)



(a) OppositeRenderer: 86 s, 2.3 iter/s, 2.33 vert/path, 0.128 s avg Light pass, 0.3 s avg Camera pass

(b) SmallVCM: 54.8 s, 3.65 iter/s

(c) OppositeRenderer - SmallVCM 5x scaled

(d) SmallVCM - OppositeRenderer 5x scaled

Figure 5.2: Comparison of Cornell Box with spheres (200 iterations, 512x512 resolution)

In the Figure 5.2 we can observe that both renderers produce very similar result and that differences most likely are due to noise since none of the subtracted images is significantly brighter than the other suggesting that it is not due to a some systemic error. However, the ceiling is noticeably more noisy for SmallVCM image. Interestingly performance of the GPU implementation significantly decreased compared to the empty Cornell Box scene, but for the CPU renderer it increased. It seems that more complicated computations due to Phong, Specular Reflection and Refraction reflection models cause significant slowdown for GPU. The speedup for CPU could be due the fact that the glossy floor helps rays bouncing from wall to escape the scene.

5.2.3 Crytek Sponza and Conference

The Figures 5.4 and 5.3 show the Conference and the Crytek Sponza scenes after 300 iterations. In the Sponza scene point the light is quite far from building and therefore very few rays enter the scene, slowing down the convergence.

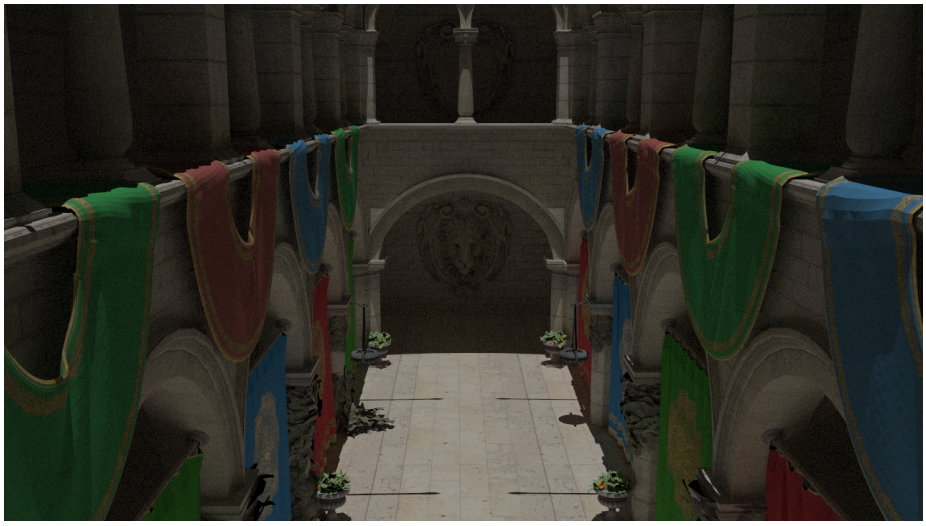


Figure 5.3: Crytek Sponza : 1280x720 res, 300 iter, 306 s, 0.98 iter/s, 0.541 vert/path, 0.185 s avg Light pass, 0.81 s avg Camera pass

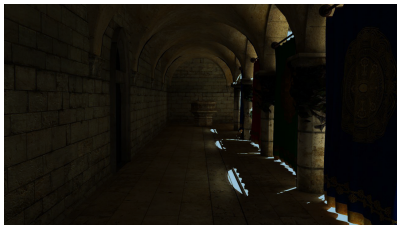
Figure 5.4: Conference : 1280x720 res, 300 iter, 300 s, 1 iter/s, 1.47 vert/path, 0.37 s avg Light pass, 0.72 s avg Camera pass

5.2.4 Performance

Although Cornell Box is a simple scene, it allows us to directly compare our Optix GPU implementation with the reference SmallVCM. As we saw, the quality of both solutions is similar with our implementation having less noise, but performance of the GPU implementation degrades with more complicated BSDFs. It might be related to the scheme we used to emulate virtual functions that result in branches and therefore thread divergence. This requires deeper investigation and possibly testing using BSDFs with more than two BxDFs to see the effect.

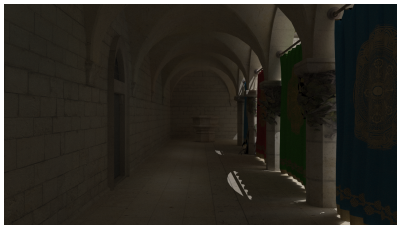
We observed that the GPU and the reference CPU implementation perform roughly the same, which means that unfortunately the GPU implementation did not bring the expected speedup, although we have to admit that the given CPU is also rather powerful. The VCM GPU implementation in [DKHS14] that used highly optimized CUDA ray shooting framework, outperformed reference CPU implementation by factor of 6-10x. In the same paper authors present the implementation of BPT that was based on the state-of-art streaming BPT approach of Van Antwerpen [VA11] with thread compaction and sample regeneration. The test scene was also Crytek Sponza, though with slightly different texture set and also it is not clear what was the configuration of light sources. However we can still attempt to compare their results to ours.

Davidovič [DKHS14] proposed improvements in form of uniform light vertex sampling and optimized loop routines. The resulting algorithm outperformed the state-of-are roughly by a factor of 2.5, using the same GPU (GTX 580, Fermi architecture) and test scene (Crytek Sponza). The proposed approach produced 8.66 million samples (light-camera sub-path pairs) compared to the previous state-of-art 3.64 million samples. Authors also tested the newer Kepler architecture GPU GTX 680, which reached 84% of the GTX 580 performance. This allows to roughly estimate that Van Antwerpen's implementation would reach about 3.06 million samples on the given GPU. The results are summarized in the Figure 5.5a.



GPU (arch)	10^6 **samples**
GTX 580 (Fermi)	3.64 [VA11]
GTX 580 (Fermi)	8.66 [DKHS14]
GTX 680 (Kepler)	7.27 [DKHS14]
GTX 680 (Kepler)	3.06 [VA11] expected

(a) Streaming BPT



GPU (arch)	10^6 **samples**
GTX 770 (Kepler)	0.53

(b) Our BPT

Figure 5.5: StreamingBPT performance in Crytek Sponza for Fermi and Kepler GPU architectures compared to our OptiX BPT

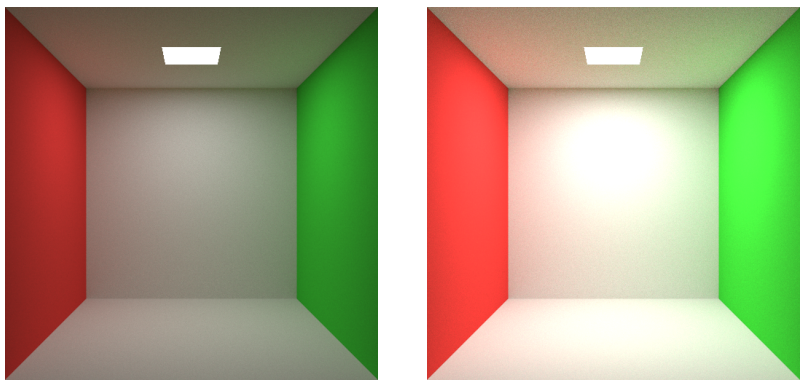
The GTX 680 used in [DKHS14] is very similar to GTX 770 used by us. They both are based on the Kepler architecture, have the same number or CUDA cores, and perform very similarly in benchmarks. The only notable difference is generally slightly higher memory frequency on GTX 770 (varies model by model). This allows us to compare our result to that achieved in [DKHS14]. We placed the camera and the light source roughly in the same position and achieved the performance of 0.53 million samples per second. That is roughly 6 times slower than would be expected for Van Antwerpen's implementation on the given GPU.

Our result is in line with Van Antwerpen's findings on SIMD efficiency. He

observed that the average fraction of active threads in a GPU warp (a set threads executing the same kernel) for the naive BPT with no thread compaction and sample regeneration was 17% during connection phase and 29% during random walk phase. It's not clear how well OptiX handles thread compaction, but it seems that exactly the poor SIMD efficiency is causing the most of performance difference. Some of it likely is due to less coherent memory access patterns than in state-of-art solutions.

5.3 Final comments

For anyone trying using this version of the renderer, we would like to note that while our results are close to renderings of SmallVCM, the existing Path Tracing and Progressive Photon Mapping implementations in the OppositeRenderer seem to produce images that are too bright which suggests there are issues in their implementations. The Figure 5.6 shows the comparison.



(a) OppositeRenderer - BPT **(b)** OppositeRenderer - PT

Figure 5.6: Comparison of empty Cornell Box scene (200 iterations, 512x512 resolution)

We also would like to comment on performance of on the first context launch after the renderer is started which can take 2 minutes or more, so that potential users are aware. During the development we had set the environment variable CUDA_CACHE_DISABLE=1 which disables compiled kernel caching by CUDA runtime. It also affects OptiX and therefore it was always performing new JIT compilation every time renderer application was restarted instead of reusing compiled versions from cache. This way we tried to identify what is

causing slow the slow context compilation. On our relatively powerful workstation the first launch can take even up to two minutes. It seems to happen when scene uses multiple materials (such as scene with two sphere and glossy floor) and OptiX tries to analyze/optimize something in regards to them.

Chapter 6

Future Work

There are multiple opportunities for improvement of the current implementation. The most obvious of course is addition of vertex merging. As indicated in the section 4.4.5 the current implementation does not handle non-symmetry caused by shading normals. This could be solved following Veach [Vea98].

A potential improvement could come from use of uniform vertex sampling as proposed in [DKHS14]. This would mean connecting some predetermined number of random light vertices from the set of all vertices, instead of connecting to vertices of one light sub-path. The number of connections could be the average number of vertices in a light sub-path. This could improve performance by avoiding cases when most of the threads are stalling and waiting on some others to finish connecting to light sub-paths that are much longer than others. Also there would be fewer memory accesses since we would use only one buffer instead three (vertex buffer, vertex index buffer for a path, path vertex count buffer) in the current solution.

The same paper [DKHS14] also proposes loading light vertex buffer using texture units in Array of Structure (AoS) layout for optimal performance.

As mentioned in the section 4.4.4 we added a path length limitation since Russian Roulette was not very effective in stopping the path due to use of maximum component of the reflectance as the continuation probability. As a solution to

this we could use the approach suggested in [PH10, page 811] to use the ratio of a particle weight luminance after the scaling with a BSDF and the particle weight luminance before scaling with the BSDF. In case the new weight is much lower, the continuation probability will be low as well. Also the particle weight should regain its original value after the division with the ratio based probability if it survives Russian Roulette.

There are also multiple improvements that could be added to the OppositeRenderer. One is to add support for loading Glossy materials as part of external scenes. Also it should be implemented for path tracing and photon mapping. It should be simple to add support for using our BPT solution in Client-Server setup. We could add a configuration dock that allows to control the max path length, number of light sub-paths, types of techniques (s,t) for which to account contributions. Also OptixRenderer class could be made as an interface class, and move every algorithm to individual classes, that could improve initial context launch time which currently can be quite long. Finally, using the new BSDF and BxDF classes, more sophisticated materials could be added.

Chapter 7

Conclusions

The thesis has provided an overview of theory related to global illumination concepts and described the Vertex Connection and Merging algorithm. We also discussed how to efficiently compute Multiple Importance Sampling weights recursively that allows to compute a full path MIS weight using only any two given vertices being connected or merged. In the chapter 4 we described the result of our work towards implementation of the VCM in the form of Bidirectional Path Tracer with recursive MIS weight computation. It is available online in GitHub[1]. The chapter 5 presented the achieved results in form of sample renderings and comparison with the reference SmallVCM renderer. It shows that our results are practically identical to SmallVCM and differences seem to be due to noise. The performance of the implementation is about the same as the reference CPU implementation, which is slower than expected. However it is in line with Van Anwerpen's [VA11] observations about SIMD inefficiencies for BPT implementations without thread compaction and sample regeneration. In our case it is handled by OptiX framework. The chapter 6 lists multiple possible improvements for the current implementation such as use of uniform vertex sampling which could improve SIMD efficiency by avoiding thread stalling in cases when some threads connect to much longer light sub-paths than average.

[1] `https://github.com/voldemarz/OppositeRenderer`

Bibliography

[ALK12] Timo Aila, Samuli Laine, and Tero Karras. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, June 2012.

[Ant10] Dietger Van Antwerpen. Unbiased physically based rendering on the gpu, 2010.

[Ant11] Dietger Van Antwerpen. Recursive mis computation for streaming bdpt on the gpu. Technical report, 2011.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. *SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 37–46, 2002.

[Cor] Nvidia Corporation. About cuda. http://developer.nvidia.com/about-cuda.

[Cor14] Nvidia Corporation. *NVIDIA® OptiX™ Ray Tracing Engine Programming Guide*, June 2014.

[DKHS14] Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, and Philipp Slusallek. Progressive light transport simulation on the gpu: Survey and improvements. *ACM TRANSACTIONS ON GRAPHICS*, 33(3):–, 2014.

[Geo12] Iliyan Georgiev. Implementing vertex connection and merging. Technical report, Saarland University, 2012.

[GKDS12] Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič, and Philipp Slusallek. Light transport simulation with vertex connection and merging. *ACM TRANSACTIONS ON GRAPHICS*, 31(6):–, 2012.

[Gro] Khronos Group. Opencl. http://www.khronos.org/opencl.

[GSK11] Iliyan Georgiev, Philipp Slusallekz, and Jaroslav Křivánek. Bidirectional light transport with vertex merging. *SIGGRAPH Asia 2011 Sketches, SA'11*, page 27, 2011.

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HJ09] Toshiya Hachisuka and Henrik Wann Jensen. Stochastic progressive photon mapping. *ACM TRANSACTIONS ON GRAPHICS*, 28(5):–, 2009.

[HOJ08] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *International Conference on Computer Graphics and Interactive Techniques*, pages 1–8, 2008.

[Kaj86] James T. Kajiya. The rendering equation. In *Computer Graphics*, pages 143–150, 1986.

[KGF+11] Arjan Kuijper, Thomas Gierlinger, Dieter Fellner, Andre Stork, and Rafael Huff. A comparison of xpu platforms exemplified with ray tracing algorithms. *Proceedings - 2011 13th Symposium on Virtual Reality, SVR 2011*, pages 1–8, 2011.

[KZ11] Claude Knaus and Matthias Zwicker. Progressive photon mapping: A probabilistic approach. *ACM TRANSACTIONS ON GRAPHICS*, 30(3):–, 2011.

[LA09] Samuli Laine and Timo Aila. Understanding the efficiency of ray traversal on gpus. *Proceedings of the HPG 2009: Conference on High-Performance Graphics 2009*, pages 145–150, 2009.

[LE10] H. Ludvigsen and A. C. Elster. Short paper real-time ray tracing using nvidia optix, 2010.

[LW94] Eric P. Lafortune and Yves D. Willems. Using the modified phong reflectance model for physically based rendering. Technical report, 1994.

[LWS93] E. P. Lafortune, Y. D. Willems, and H. P. Santo. Bi-directional path tracing. 1993.

[PBD+10] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[Ped13] Stian Aaraas Pedersen. Progressive photon mapping on gpus. Master's thesis, 2013.

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

[PJH12] Jacopo Pantaleoni, Henrik Wann Jensen, and Toshiya Hachisuka. A path space extension for robust light transport simulation. *ACM Transactions on Graphics*, 31(6):–, 2012.

[RDGK12] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz. The state of the art in interactive global illumination. *COMPUTER GRAPHICS FORUM*, 31(1):160–188, 2012.

[SFD09] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 7–13, New York, NY, USA, 2009. ACM.

[VA11] Dietger Van Antwerpen. Improving simd efficiency for parallel monte carlo light transport on the gpu. *Proceedings - HPG 2011: ACM SIGGRAPH Symposium on High Performance Graphics*, pages 41–50, 2011.

[Vea98] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford, CA, USA, 1998. AAI9837162.

[VGS+95] E. Veach, L. Guibas, G. Sakas, P. Shirley, and S. Muller. Bidirectional estimators for light transport. 1995.

[Vor11] Jiří Vorba. Bidirectional photon mapping. Technical report, Charles University, 2011.

[WJ01] H. Wann Jensen. *Realistic image synthesis using photon mapping*. AK Peters, 2001.