FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Dominik Dinh

Implementation of VCM in a fluorescence-capable path tracing framework

Department of Software and Computer Science Education

Supervisor of the master thesis:	Dr. Alexander Wilkie
Study programme:	Computer science
Study branch:	Visual Computing and Game Development

Prague 2024

I would like to thank my supervisor Dr. Alexander Wilkie for his endless patience and insight.

Title: Implementation of VCM in a fluorescence-capable path tracing framework

Author: Bc. Dominik Dinh

Department of Software and Computer Science Education: Department of Software and Computer Science Education

Supervisor: Dr. Alexander Wilkie, Department of Software and Computer Science Education

Abstract: This thesis presents a novel implementation of Vertex Connection and Merging (VCM) in a spectral framework with Hero Wavelength Spectral Sampling (HWSS) support. To the best of our knowledge, this is the first work to successfully achieve this integration. The primary objective was to enhance the efficiency and accuracy of light transport simulations in complex spectral scenarios. Additionally, the system incorporates fluorescence simulation capabilities, although the current implementation captures only a subset of fluorescent effects. Our implementation adds support for fluorescent effects to a light tracer and progressive photon mapper.

Keywords: spectral path tracing vcm vertex connection and merging spectral

Contents

Introduction

When enough light hits the human eye, it creates a perception of the world around us. This perception, or image, allows us to navigate the world and appreciate how colorful it can be. Recent advancements in technology have made it possible to capture a fraction of what we can see, save it, and share it with others. But what if we wanted to create an imaginary, virtual world where imagination is the limit? The answer is computer graphics, namely realistic image synthesis or rendering for short.

The main goal of rendering is to synthesize an image from a description of a virtual world and make it visually indistinguishable from photography. Although recent advancements have made real-time graphics capable of producing breathtaking results, they still cut corners. They cannot reproduce the plethora of interactions that an offline Monte-Carlo renderer can.

To this day, most photorealistic renderers are unidirectional path tracers. Undeniable advantages of a unidirectional path tracer are speed and relatively low complexity. Thanks to that, one can partially rethink math to implement more complex interactions, such as fluorescence. However, it can struggle with certain scenes and light paths. For example, a scene with a light source hidden behind a piece of scene geometry can invalidate most attempts at NEE. Also, modeling caustics or SDS paths can pose a significant challenge since the probability of finding a light source from a pure specular reflection is very low.

Over the years, many successful and viable alternatives have been proposed.

In his famous thesis [Vea97] Dr. Eric Veach proposed a Bidirectional path tracing algorithm that attempts to sample light transport path samples more efficiently. The algorithm's basic premise is to sample light paths from the camera and light sources. MIS is then utilized to connect eye paths to light paths.

Another viable alternative, albeit biased, is photon mapping. The algorithm works in two passes. In the first pass, photons are traced from the light source, creating a 'photon map.' In the second pass, camera rays are traced through the scene, approximating overall light energy at each point by accumulating photons in a certain radius. This merging in a radius is what causes a slight bias and blur.

Georgiev et al. [GKDS12] combined these two algorithms into a single method, Vertex Connection and Merging. While many implementations exists, such as [Sma], to our knowledge, there is no implementation of VCM in a spectral fluorescence-capable renderer.

Consequently, it is the goal of this thesis.

1. Overview

In this chapter, we will cover the essential math and techniques that are prerequisites to this thesis. We've divided this chapter into multiple sections, each introducing a distinct building block this thesis stands on.

First, we will derive Vertex Connection and Merging and define the math needed to derive it. Main focus will be on the path integral formulation 1.1.1. Prerequisite algorithms will be mentioned to keep things complete and consistent.

Secondly, we will cover spectral rendering and hero wavelength spectral sampling.

Lastly, fluorescence will be touched upon.

1.1 Prerequisites to VCM

This section focuses on the prerequisites for deriving Vertex Connection and Merging.

First, we will review path-integral formulation, light tracing, and bidirectional path tracing. These were presented first in the infamous thesis by [Veach], who first proposed the idea of bidirectional path sampling.

Secondly, photon mapping will be revisited and explained.

Lastly, we will discuss the advantages and disadvantages of both techniques and compare them side by side to see why combining the two techniques might be beneficial.

1.1.1 Path-space integration

In 1986, [Kajiya] introduced the rendering equation

$$L(x,\omega_{out}) = L_e(x,\omega_{out}) + \int_\Omega f_r(x,\omega_{in},\omega_{out})L_i(x,\omega_{in})cos(\Theta)d\omega_{in}$$

which states that radiance arriving at the camera from a point x coming in the direction ω_{out} is equal to the point's emitted radiance $L_e(x,\omega_{out})$ plus the integral of incoming radiance $L_i(x,\omega_{in})$ at point x over the hemisphere 1.1.

The function f_r stands for the *bidirectional scattering distribution function* or BSDF for short. It defines how the surface at point x scatters light. For example, figure 1.1 shows a BRDF for a specular surface (in red). Specular surfaces scatter light primarily in the direction of the lobe.

We define

$$L_i(x,\omega_{in}) = L_o(y,-\omega_{in})$$

After a short examination, we can see that the rendering equation is inherently recursive. To get the outgoing radiance at a point x, we first need to calculate the incoming radiance, which is the same as calculating $L_o(y,\omega_{out})$.

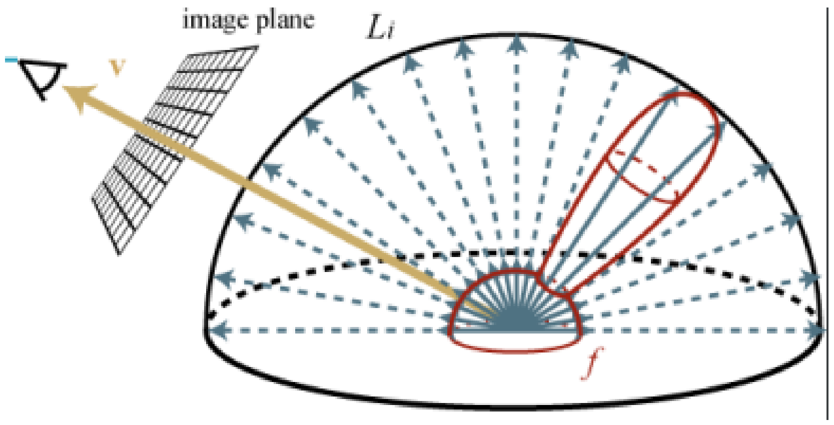


Figure 1.1: Incoming direction over the hemisphere. Image taken from teaching materials by CGG group MMF UK

Eric Veach, in his thesis [Vea97] showed that we can approach the problem from a different perspective and reformulate the problem as an integration problem

$$I_j = \int_{\Omega} f_j(\overline{x}) d\mu(\overline{x}) \tag{1.1}$$

We then define a path of length k as

$$\overline{x} = x_0 x_1 ... x_k$$

The integration domain is a union of sets of paths with the length $1...\infty$

$$\Omega = \bigcup_{k=1}^{\infty} \Omega_k$$

The term μ is an area measure of a path.

$$d\mu(\overline{x}) = dA(x_0)...dA(x_k)$$

While the original formulation of the measurement equation uses solid angle and projected solid angle measures, the new formulation uses an area measure. We can easily convert between the two using

$$d\sigma^{\perp}_{x'}(\omega_i) = G(x \leftrightarrow x^{'}) dA(x)$$

where

$$G(x \leftrightarrow x^{'}) = V(x \leftrightarrow x^{'}) \frac{|cos(\theta_i) \cdot cos(\theta^{'}_o)|}{\parallel x - x^{'} \parallel^2}$$

V is a visibility term which equals **1** if point x is visible from point x', otherwise it is equal to zero. Finally, the last term that we have yet to define is the measurement equation of a path.

$$f_j(\overline{x}) = L_e(x_0 \to x_1) G(x_0 \leftrightarrow x_1) W_e(x_{k-1} \to x_k) \cdot \prod_{i}^{k-1} f_s(x_{i-1} \to x_i \to x_{i+1}) G(x_i \to x_{i+1})$$

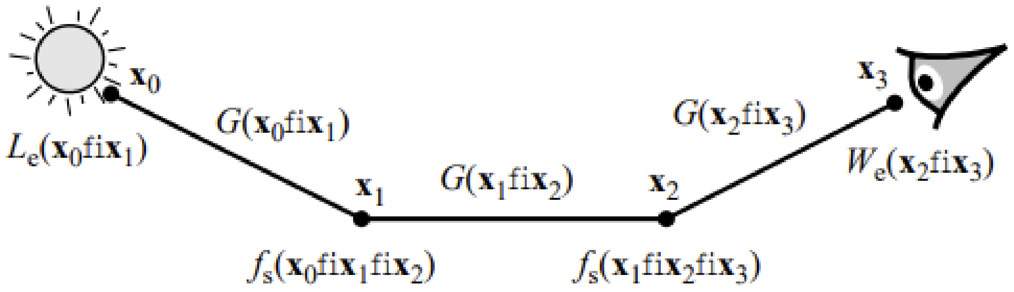


Figure 1.2: Measurement contribution function for a path of length 4. Taken from a thesis by Eric Veach. Image taken from teaching materials by CGG group MMF UK

The last term W_e, or *importance*, will be defined in the later part of the thesis 1.1.2.

Formulating the problem this way has several advantages. Namely, it allows for the usage of well-known integration techniques. Also, it simplifies the measurement into a single expression and removes the recursion.

1.1.2 Light tracing

A standard unidirectional path tracer works by tracing a ray from the camera. This is usually much more effective than tracing rays from light sources, which is how nature does it. Purely because the probability of hitting a camera pixel is very low, and the amount of rays needed is disproportionally higher than in a standard path tracer.

However, certain scenarios, such as caustics or occluded light sources, might require a light tracer to produce better results.

Recall from section 1.1.1 that the measurement equation contains a term W_e, or emitted importance. Intuitively, it describes how important a contribution is to the camera sensor. Similar to radiance, importance can be transported and traced through the scene 1.4. Light tracing can be thought of as solving the measurement equation for importance and path tracing for radiance.

$$W(x,\omega_{out}) = W_e(x,\omega_{out}) + \int_\Omega f_r(x,\omega_{in},\omega_{out})W_i(x,\omega_{in})cos(\Theta)d\omega_{in}$$

A light tracer is usually not used alone; instead, it is the basis for bidirectional methods, such as bidirectional path tracing and photon mapping, both of which will be introduced in the following sections. Differences and shortcomings of LT compared to PT is shown in figure 1.3

1.1.3 Bidirectional path tracing

First introduced by [Vea97], bidirectional path tracing aims to combine the best aspects of both standard path tracing and light tracing. The original paper

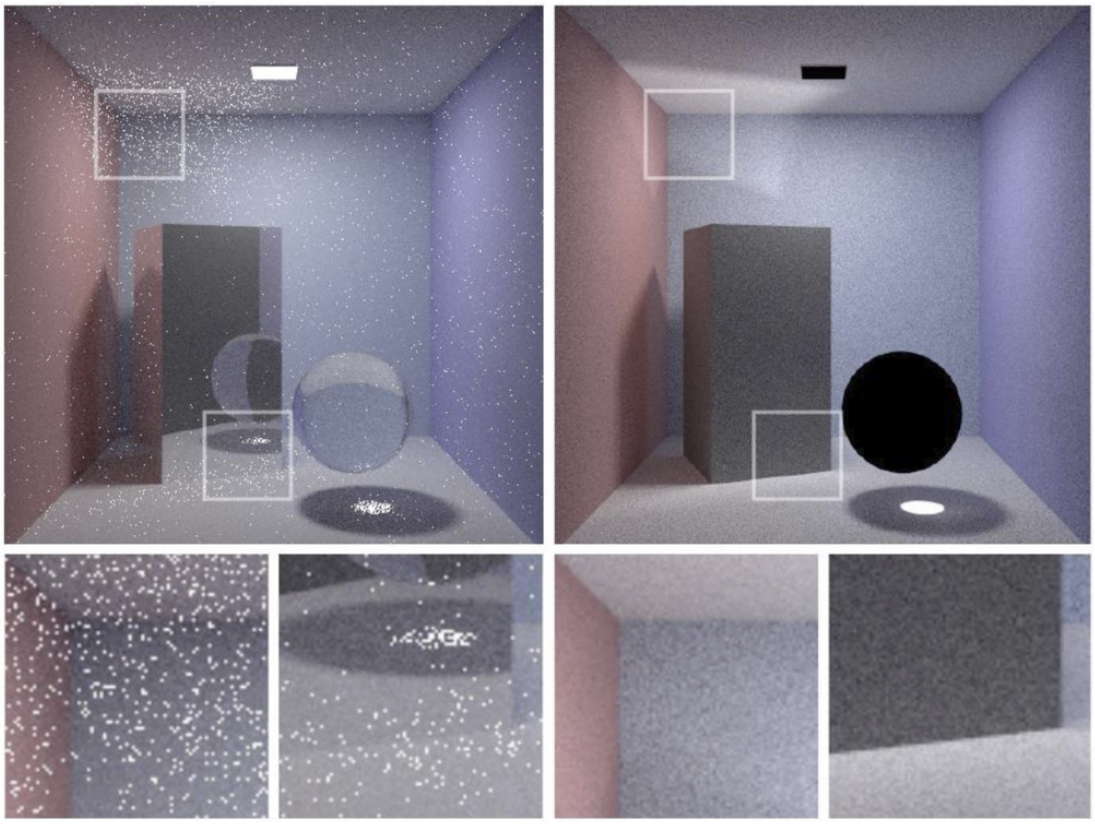


Figure 1.3: Comparison between a path tracer with NEE (left) and a light tracer (right). This image shows the relative strengths and weaknesses of respective algorithms. Light tracer, for example, converges faster at the bright light spots. However, light sources, and especially for a pinhole camera, specular (in this case, glass) objects will be black. Image taken from teaching materials by CGG group MMF UK

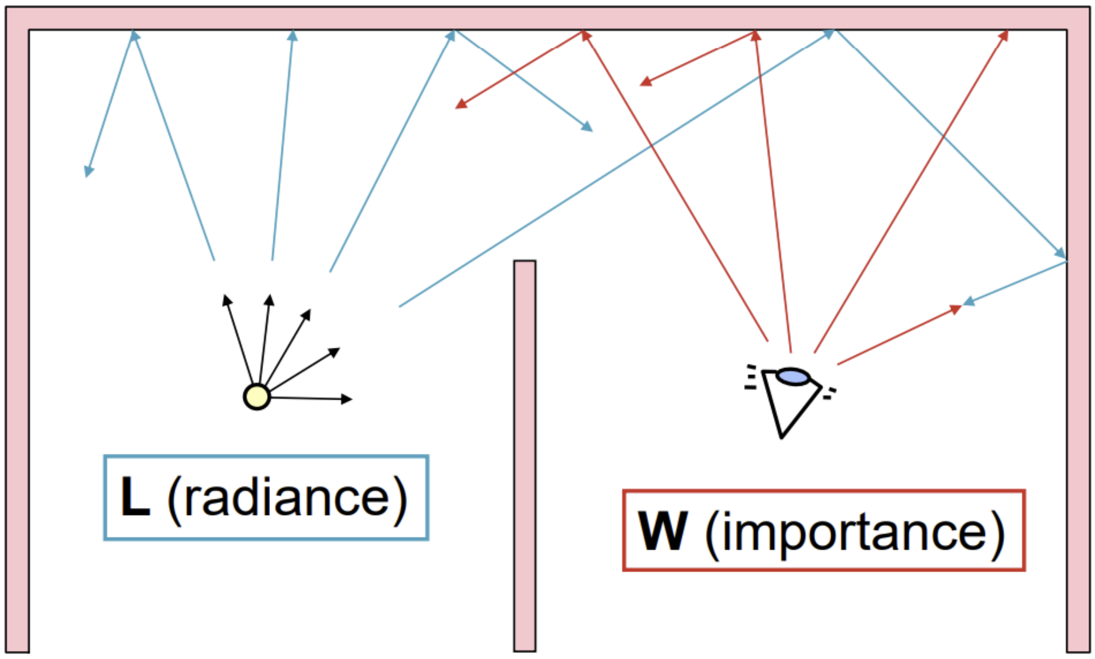


Figure 1.4: The duality of radiance and importance. Image taken from teaching materials by CGG group MMF UK

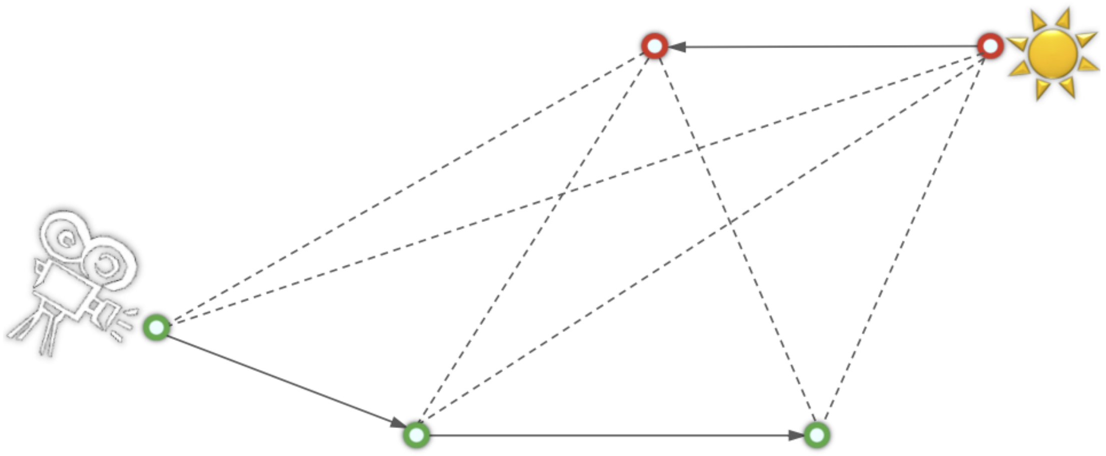


Figure 1.5: Diagram showing the different connections we can make between a camera path and a light path. Image taken from teaching materials by CGG group MMF UK

states that the method is a family of 'importance sampling techniques' to solve the measurement integral 1.1.

An individual sampling technique is a variation of vertex connections between two independently sampled paths—one from the camera and the other from a light source. Let us denote the number of vertices on the camera path as s and t when on the light path. The total number of vertices in a path is $k = s + t - 1$. The total number of variations we can generate this way is $k + 2$. An illustration is given in figure 1.5.

Each path sampling technique has a density function (i.e. probability) $p_{s,t}$. This is important for an accurate combination using multiple-importance sampling. To compute the MIS weights, for every measurement I_j we define

$$F = \sum_{s \geq 0} \sum_{t \geq 0} w_{s,t}(\overline{x}_{s,t}) \frac{f_j(\overline{x}_{s,t})}{p_{s,t}(\overline{x}_{s,t})} \tag{1.2}$$

where $w_{s,t}$ represents a combination strategy, ie. balance heuristic.

1.1.4 Photon-mapping

Proposed in 1996 in an EGSR paper by Jensen [Jen96], photon mapping is an effective, albeit biased, bidirectional method. Compared to other methods, PM excels at sampling SDS paths (1.6).

PM works in two passes.

In the first pass, photons are traced through the scene, similar to a light tracer. However, at each hitpoint, its path vertex is stored in a photon map. Photon maps are most often implemented as a range structure, such as kD-trees or hash grids, which allow for quick look-ups.

The second pass collects the stored photons. An eye ray is traced, and at each hitpoint, the range structure is queried to gather all photons in a set radius r to determine the overall radiant energy.

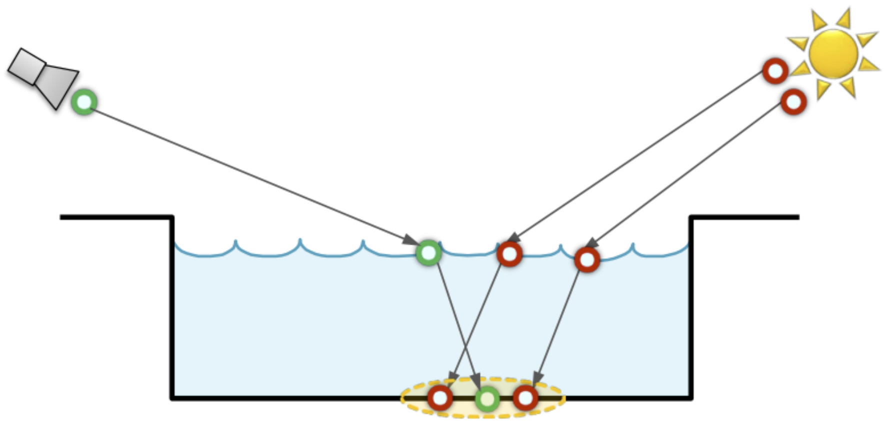


Figure 1.6: Photon mapping density estimation. Tracing methods struggle with connecting SDS paths. Photon mapping does density estimation, which is much more effective, albeit biased.Image taken from teaching materials by CGG group MMF UK

A formula expressing the collection of radiant energy

$$L(x, \omega_{out}) \approx \sum_j K_r(\| x - x_j \|) f_r(\omega_j, x, \omega)\Phi \tag{1.3}$$

Where K_r is 2D kernel with radius r. The sum iterates over all photons found in range r.

One attempt to mitigate some of the disadvantages of base photon mapping and to make it consistent is *progressive photon mapping* [HOJ08]. At each iteration, the merging radius is decreased so that total bias and variance approach zero in infinity.

1.2 Spectral rendering

Most renderers used today are RGB-based. That means they use a simple RGB triplet (equation 1.4) to propagate light information throughout the scene. While simple and computationally efficient, this approach often fails to accurately capture the complex nature of light and color. One example where this approach might fall short is wavelength-dependent effects such as dispersion or fluorescence.

$$\begin{aligned}\begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix}(x, \omega_{out}) = \begin{pmatrix} L_{e,R} \\ L_{e,G} \\ L_{e,B} \end{pmatrix}(x, \omega_{out}) + \\ \int_\Omega \begin{pmatrix} f_R \\ f_G \\ f_B \end{pmatrix}(x, \omega_{in}, \omega_{out}) \begin{pmatrix} L_{in,R} \\ L_{in,G} \\ L_{in,B} \end{pmatrix}(x, \omega_{in}) \cos\theta d\omega_{in}\end{aligned} \tag{1.4}$$

On the other hand, spectral rendering techniques aim to model the full spectrum of light rather than just the three primary colors. This approach has the potential to provide a more accurate representation of color and lighting but at the cost of increased computational complexity. One disadvantage of spectral rendering is the larger data requirements to store and process the full spectrum information.

Adding a spectral element to the standard rendering equation is straightforward:

$$L(x, \omega_{out}, \lambda) = L_e(x, \omega_{out}, \lambda) + \int_\Omega f(x, \omega_i, \omega_o, \lambda) L_{in}(x, \omega_i, \lambda) cos\theta d\omega_{in}$$

To get images in standard sRGB, we can utilize the standard CIE XYZ color space. XYZ corresponds to individual color-matching functions defined as follows (1.5)

$$\begin{aligned} X &= \int_\lambda L(\lambda)\overline{x}(\lambda)d\lambda \\ Y &= \int_\lambda L(\lambda)\overline{y}(\lambda)d\lambda \\ Z &= \int_\lambda L(\lambda)\overline{z}(\lambda)d\lambda \end{aligned} \tag{1.5}$$

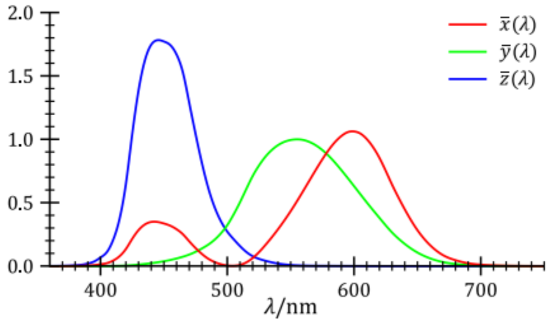


Figure 1.7: Individual color matching functions plotted onto a graph. Image taken from teaching materials by CGG group MMF UK

After acquiring the values for the individual components (X, Y, Z) we can convert them into RGB using a well-defined formula, which will be omitted for brevity.

1.2.1 Fluorescence

Fluorescence refers to a light-matter interaction in which the material absorbs a photon of wavelength λ_i and re-emits a photon with a, usually longer, wavelength λ_o. Although similar to phosphorescence, fluorescence happens 'instantly,' while there is a certain delta time between absorption and emission in the case of phosphorescence. This difference can be easily observed in the real world: phosphorescent materials will continue to glow for Δt, while fluorescent material will quickly return to its default state.

One example where fluorescent phenomena are common is marine life. Many types of corals exhibit fluorescence, especially in very deep waters. A better, more useful example might be a fluorescent jellyfish, which contains a special protein called GFP (1.8). This protein can be used, for example, as a reporter protein to measure pollution.

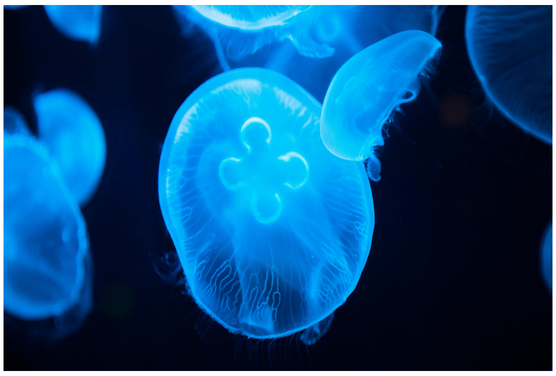


Figure 1.8: Fluorescent jellyfish. Courtesy of Pexels, Ryotaro

Fluorescent BBRRDF

We model fluorescent surfaces with the *bispectral bidirectional reflection and reradiation distribution* function, or *BBRRDF* for short.

$$f(\omega_{in}, \lambda_{in}, \mathbf{x}, \lambda_{out}, \omega_{out}) = \frac{d^2 L_{out}(\mathbf{x}, \omega_{out}, \lambda_{out})}{L_{in}(x, \omega_{in}, \lambda_{in}) \cos\theta_{in} d\sigma(\omega_{in}) d\lambda_{in}} \tag{1.6}$$

BBRRDFs can be fully data-driven by measuring material properties in real life, analytical, or hybrid. Data to model fluorescent behavior is usually saved in a *reradiation matrix*.

Reradiation matrix is an asymmetric matrix in which rows correspond to existent wavelengths and columns correspond to incoming wavelengths. Each element $m_{i,j}$ defines the probability of wavelength shift from λ_{in} to λ_{out}. Intuitively, the diagonal refers to non-fluorescent probabilities. To preserve energy conservation, the sum of elements in each column needs to be less than or equal to 1. Illustration can be seen in figure 1.9.

Fluorescence in rendering

Handling fluorescence is not too difficult in a standard uni-directional path tracer.

As a path is traced through a scene, we can either hit a fluorescent or a non-fluorescent surface. When we hit a non-fluorescent surface, we continue as usual with the wavelength λ_0 sampled at the camera. However, when we arrive at a fluorescent vertex, a new wavelength λ_{11} is sampled from the BBRRDF's emission spectrum. Path tracing then continues with this new wavelength.

An additional component is added to the overall path PDF to account for fluorescence. We express the extended path PDF as follows

$$p_A(x_0, \lambda_0, ..., \lambda_k, x_k) = \prod_{i=0}^{k} p_A(x_i) \cdot \prod_{i=1}^{k} p_\Lambda(\lambda_i) \tag{1.7}$$

where

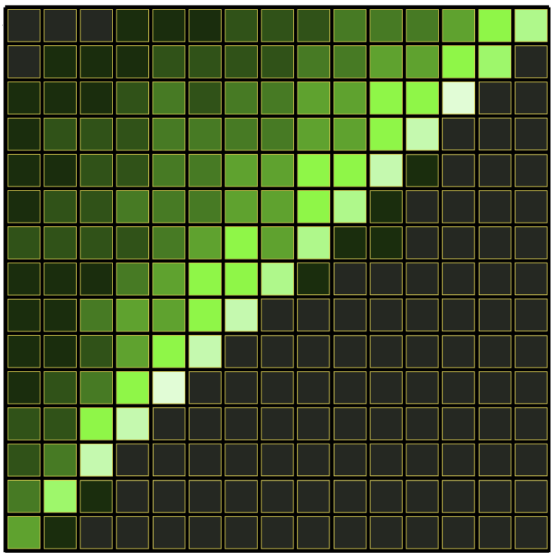


Figure 1.9: Reradiation matrix; the diagonal shows probabilities of a non-fluorescent interaction. The part 'above' the diagonal represent the wavelength shift probability, or, the 'fluorescent' interaction. Taken from [JHD20]

$$p_\Lambda(\lambda_i = \begin{cases} p_{camera}(\lambda_0) & \text{if } i = 1 \\ p_{fluo}(\lambda_i|\lambda_{i-1}) \cdot p_{abs}(\lambda_i|\lambda_{i-1}) & \text{if } x_i \text{ is fluorescent and } \lambda_i \neq \lambda_{i-1} \\ p_{nonfluo}(\lambda_i|\lambda_{i-1}) & \text{if } x_i \text{ is fluorescent and } \lambda_i = \lambda_{i-1} \\ 1 & \text{if } x_i \text{ not fluorescent} \end{cases} \quad (1.8)$$

Recently, new methods to handle fluorescence have been introduced. Notably, [MFW18] managed to combine and implement fluorescence in an HWSS renderer by making individual wavelengths shift freely.

Bidirectional methods got some attention as well. For example, [JHD20] treats fluorescence as a singularity in the spectral domain - similar to how mirrors are singularities in the geometrical domain. It then employs mollification to handle the singularities and allow for path connections between fluorescent and non-fluorescent vertices.

2. VCM

Framework proposed by Georgiev et al. [GKDS12], [Geo12] aims to efficiently combine bidirectional path tracing and progressive photon mapping to alleviate the respective technique weakness. The method builds upon the *path-integral framework* and *multiple-importance sampling* to express a new common mathematical framework to efficiently combine the two techniques.

Vertex Connection and Merging has been widely used in production, namely, the Pixar's RenderMan, V-Ray or Corona. Being employed by such well-known and respected renderers provides proof of how solid the VCM framework is.

2.1 Expressing PM in path-integral framework

The Intuitive approach is to take a photon x_s^* and express its tracing history and collection history as an extended path. Let us define a tracing history of a photon x_s^* as $x_0 x_1 ... x_s^*$, where x_0 is sampled on the light source and shot in the direction of a photon x_1. We then set a collection history, or PM radiance estimate, as $x_s x_{s+1} ... x_k$. A full extended path consists of both sub-paths connected together: $\overline{x}^* = (x_0 x_1 ... x_s^*, x_s x_{s+1} ... x_k)$

While formulating the problem this way enables the use of MIS to weigh the individual contributions of PM, it cannot be directly used in conjunction with BPT. The reason for that is that the extended path has an extra vertex, x_s^*. That means the pdfs of BPT are naturally expressed in relation to a different area measure. Since MIS expects all PDFs to be expressed in relation to the same measure, we need to make a slight adjustment to the extended-path definition.

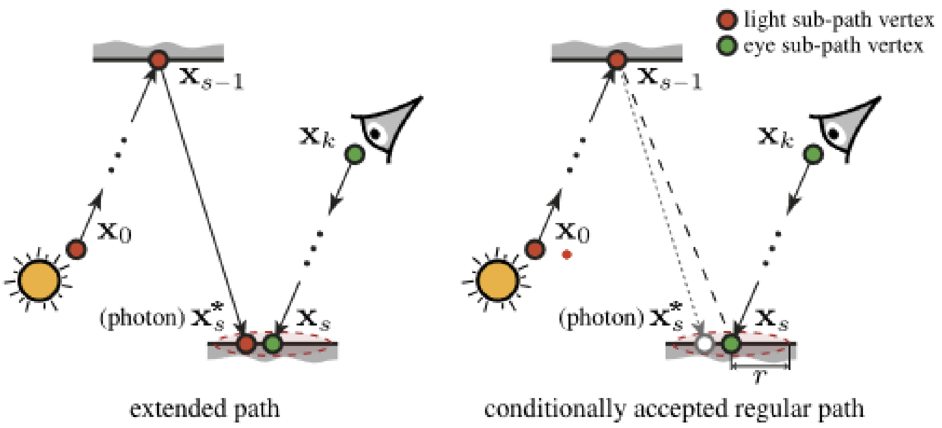


Figure 2.2: A figure showing the difference between an extended path and a regular path. [Geo12]

Instead, we will consider *conditionally accepted regular paths*, where we omit the photon x_s^*. The term *conditionally accepted* comes from the fact that a regular path is accepted if and only if the omitted photon x_s^* falls within the radiance estimate radius around x_s (figure 2.2). The probability of accepting a said path is then defined as

$$P_{acc}(\overline{x}) = Pr(\| x_s - x_s^* < r) \approx \pi r^2 p(x_{s-1} \rightarrow x_s^*) \tag{2.1}$$

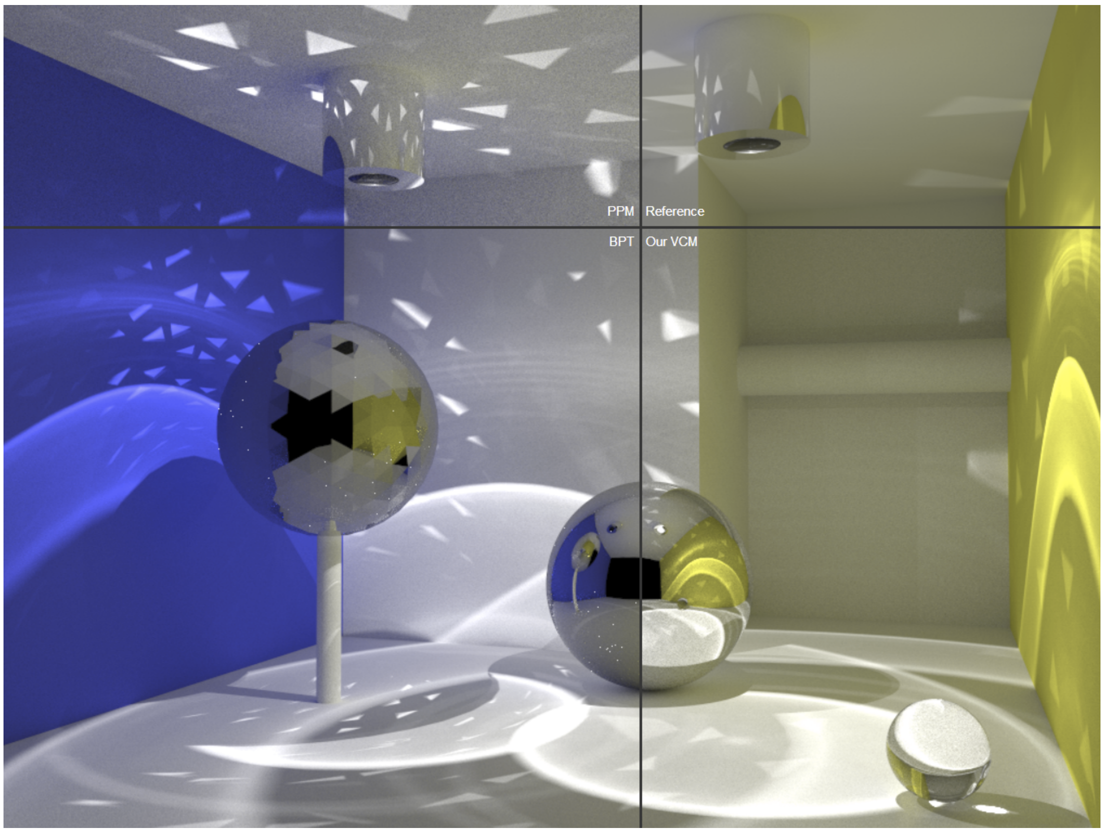


Figure 2.1: Comparison image taken from the VCM supplemental document, which shows the differences between PPM, BPT, and VCM. The image clearly shows how BPT struggles with caustics and how using VCM leads to a more consistent result. Refer to figure 2.4 to see the individual contributions of the respective algorithms.

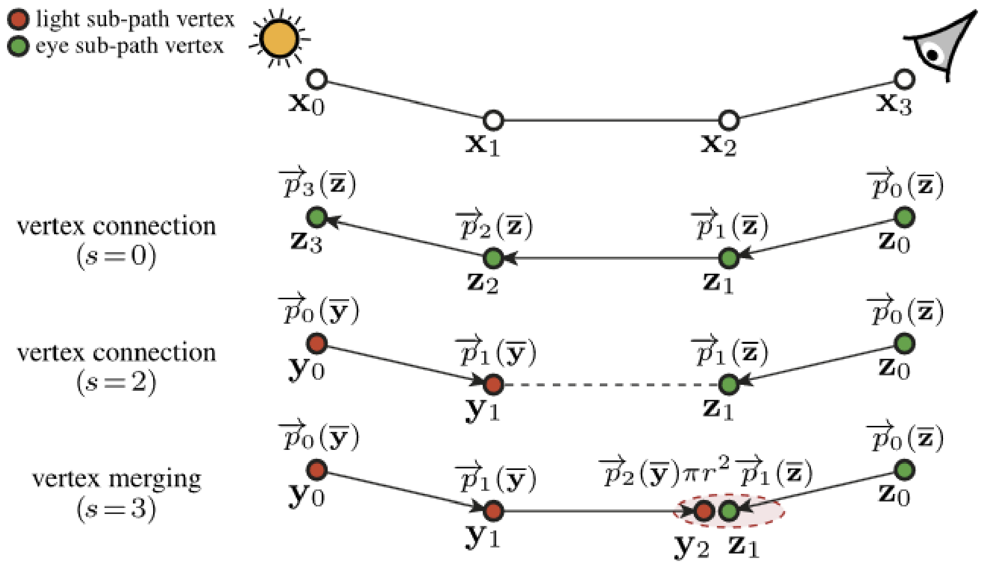


Figure 2.3: Figure showing the entire process of vertex connection and merging. Up to the vertex connection part, the illustration is equivalent to the one used in [Vea97].

and the pdf of a full regular path is

$$p_{VM}(\overline{x}) = P_{acc}(\overline{x})p_{VC}(\overline{x}) \tag{2.2}$$

Term p_{VC} corresponds to the BPT technique, which (could have) sampled the path by connecting the two vertices. We now have a new path sampling framework called ”Vertex Merging”.

2.2 Combined framework

The whole idea behind deriving the new path sampling technique, *vertex merging*, was so that we can combine vertex merging and vertex connection (or BPT) and apply MIS to weigh the contributions. Vertex merging adds *k-1* new techniques to the existing *k+2* techniques provided by vertex connection.

With this definition in place, we can now derive the combined estimator, *vertex connection and merging.*

$$\begin{aligned} I_{VCM} = C_{VC} + C_{VM} = \\ \frac{1}{n_{VC}}\sum_{l=1}^{n_{VC}}\sum_{s\geq 0,t\geq 0} w_{VC,s,t}(\overline{x}_l)I_{VC}(\overline{x}_l) + \\ \frac{1}{n_{VM}}\sum_{l=1}^{n_{VM}}\sum_{s\geq 2,t\geq 2} w_{VM,s,t}(\overline{x}_l)I_{VM}(\overline{x}_l) \end{aligned} \tag{2.3}$$

It is the original BPT estimator as first introduced in extended to also handle vertex merging. For each pixel, we trace an eye path which is to be connected to n_{VC} light paths and merged with at most n_{VM} light paths.

2.3 Optimizations and path reuse

The naive approach to evaluating the path weights involves independent computation of all path PDFs in the denominator, which can be computationally inefficient.

[Vea97] proposed an improved scheme for BPT by exploiting the fact that many terms are canceled out in path PDF calculations. This method loops once over the light and eye sub-path vertices. While this approach can be easily extended to VCM, the efficiency is still sub-optimal as many redundant computations exist.

In a standard approach, every subpath vertex stores its throughput—the accumulated contributions from all preceding subpath vertices. This allows for quick evaluation of the unweighted contribution when connecting two vertices.

The proposed solution by [GKDS12] aims to do the same for path weight evaluation. They first reformulated the sums in the weight formula as recursive quantities that can be incrementally computed and cached at the sub-path vertices during random walks. For brevity, we will include the derived formulas only.

We can reformulate the balance heuristic weight as

$$w_{v,r,t} = \frac{1}{\frac{n_{\mathrm{VC}}}{n_v}\sum_{j=0}^{k+1}\frac{p_{\mathrm{VC},j}}{p_{v,s}} + \frac{n_{\mathrm{VM}}}{n_v}\sum_{j=2}^{k}\frac{p_{\mathrm{VM},j}}{p_{v,s}}} \tag{2.4}$$

where $p_{v,j}$ is the pdf for sampling a full path (of length k) using a light subpath with j vertices and $v \in \{VC, VM\}$

We can further simplify the weight calculation to

$$w_{v,s,t} = \frac{1}{w_{v,s}^{light} + 1 + w_{v,s}^{eye}} \tag{2.5}$$

where

$$\begin{aligned}
w_{\mathrm{VC},s}^{\mathrm{light}} &= \overline{w}_{\mathrm{VC},s-1}^{\mathrm{VC}}(\overline{y}) + \eta_{\mathrm{VM}}\overline{w}_{\mathrm{VC},s-1}^{\mathrm{VM}}(\overline{y}) \\
\overline{w}_{\mathrm{VC},0} &= \frac{\overleftarrow{p}_0}{\overrightarrow{p}_0} \\
\overline{w}_{\mathrm{VC},i} &= \overleftarrow{p}_i\left(\eta_{\mathrm{VCM}} + \frac{1}{\overrightarrow{p}_i} + \frac{1}{\overrightarrow{p}_i}\overline{w}_{\mathrm{VC},i-1}\right)
\end{aligned} \tag{2.6}$$

for vertex connection. For vertex merging, the process is equivalent

$$\begin{aligned}
\overline{w}_{\mathrm{VM},1} &= \frac{1}{\overrightarrow{p}_1}\left(\frac{1}{\eta_{\mathrm{VCM}}} + \overleftarrow{p}_0\frac{1}{\eta_{\mathrm{VCM}}\overrightarrow{p}_0}\right) \\
\overline{w}_{\mathrm{VM},i} &= \frac{1}{\overrightarrow{p}_i}\left(\frac{1}{\eta_{VCM}} + \overleftarrow{p}_{i-1} + \overleftarrow{p}_{i-1}\overline{w}_{VM,i-1}\right)
\end{aligned} \tag{2.7}$$

There is still a problem with the reverse probabilities $\overleftarrow{p}_i(\overline{y})$, which are not yet known at the time of sampling a vertex i. To mitigate this, computations of individual MIS weights are split into three distinct quantities, d_i^{VCM}, d_i^{VC}, d_i^{VM}, which are accumulated during a random walk. This way, the computation of the reverse probabilities at $\overleftarrow{p}_i(\overline{y})$ can be postponed until its values can be computed.

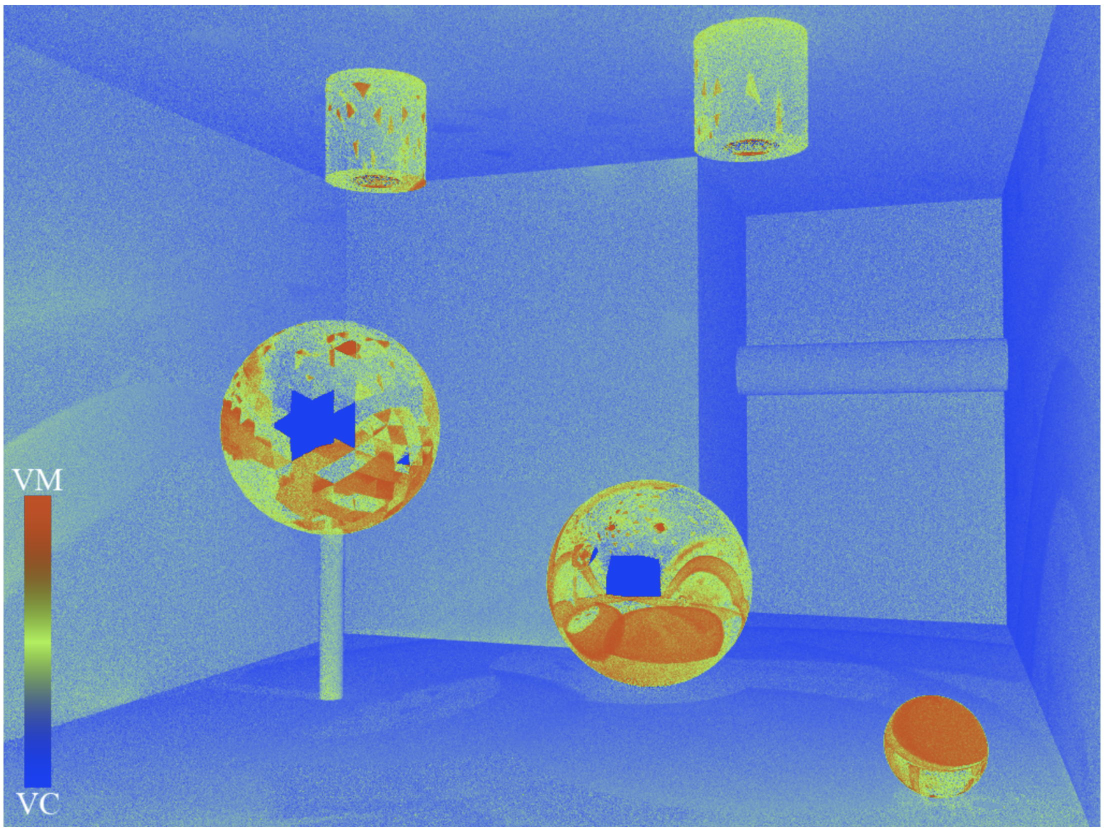


Figure 2.4: Image showing the relative contributions of vertex connection (BPT) and vertex merging (PPM). As expected, vertex merging contributes more in case of SDS paths. Image by [GKDS12]

$$\begin{aligned}\overline{w}_{VC,i} &= \overleftarrow{p}_i \left(\eta_{VCM} + \underbrace{+\frac{1}{\overrightarrow{p}_i}}_{d_i^{VCM}} + \underbrace{\frac{1}{\overrightarrow{p}_i}\overline{w}_{VC,i-1}}_{\overleftarrow{p}_{\sigma,i-1} d_i^{VC}} \right) \\ \overline{w}_{VM,i} &= \underbrace{\frac{1}{\overrightarrow{p}_i}}_{d_i^{VCM}} \frac{1}{\eta_{VCM}} + \overleftarrow{p}_{\sigma,i-1} \underbrace{\frac{\overleftarrow{g}_{i-1}}{\overrightarrow{p}_i}(1+\overline{w}_{VM,i-1})}_{d_i^{VM}}\end{aligned} \tag{2.8}$$

Great thing about the VCM framework is that it natively allows us to use light tracing, bi-directional path tracing or progressive photon mapping on their own by ignoring some of the quantities.

2.4 Pseudocode

Now that we have defined the base analytical derivation of the VCM framework, all that's left is to provide an overview of the actual implementation. This section will briefly review the VCM process and provide a short pseudocode (2.4) for quick reference. We had to rethink the process during our implementation, but we will assess that in the following chapters.

The reference VCM implementation separates the rendering process into two distinct parts, not unlike other techniques like PM.

A light tracing with 'NEE' takes place in the first part. At every vertex $x_i, i \geq 1$; we first check if it's a specular / mirror surface; in case it is not, we store the vertex in a range structure to allow for vertex merging, mark it eligible for connection and do NEE. Like NEE in a standard path tracer, we connect the vertex to a pixel on the camera, calculate the MIS weights, and write the contributions in a frame buffer.

During the second part, vertex connections and vertex merging happen. For each eye sub-path $p_{e,i}$ we trace, we connect each non-specular vertex to vertices on a light sub-path $p_{l,i}$. After the connection, we query a range structure to gather all neighboring vertices (photons) in a radius r and perform vertex merging.

```
function RENDER(r)
    # Stage 1: Light path sampling
    lightPaths = TRACELIGHTPATHS(pixelCount)
    CONNECTTOEYE(lightVertices)
    BUILDRANGESEARCHSTRUCT(lightVertices)

    # Stage 2: Eye path sampling and pixel estimator
       construction
    for i = 1 to pixelCount do
        eyeVertex = TRACERAY(SAMPLEPIXEL(i))
        while eyeVertex is valid do
            # Unidirectional sampling (US)
            if eyeVertex is emissive then
                ACCUM(eyeVertex, US, r, i)
            end if

            # Vertex connection (VC)
            for lightVertex in lightPaths[i] ∪
               SAMPLELIGHTPOINT() do
                ACCUM(CONNECT(eyeVertex, lightVertex), VC, r,
                   i)
            end for

            # Vertex merging (VM)
            for lightVertex in RANGESEARCH(eyeVertex, r) do
                ACCUM(MERGE(eyeVertex, lightVertex), VM, r, i)
            end for

            eyeVertex = CONTINUERANDOMWALK(eyeVertex)
        end while
    end for
end function

# Accumulates the pixel measurement estimate due to a given
   path
function ACCUM(path, technique, r, i)
    contrib = MEASUREMENTCONTRIBUTION(path, technique, r)
    pdf = PDF(path, technique, r)
    weight = POWERHEURISTIC(path, technique, pdf)
    image[i] += weight * contrib / pdf
end function
```

3. Implementing VCM in ART

This chapter will describe how we implemented VCM in a spectral renderer with HWSS (hero wavelength super sampling) support. Extending ART to fit the VCM framework was not trivial. However, fitting the VCM framework in such a complex renderer system proved to be the bigger challenge. Consequently, to our knowledge, this work is the first to make VCM fully compatible with spectra **and** HWSS. This chapter is divided into multiple sections, each covering a step of the implementation process,

3.1 Preliminary research

Given the complexity of ART, a significant initial phase of this thesis involved an in-depth exploration and hands-on experimentation with the codebase. This preparatory stage was crucial for comprehensively understanding the system's architecture and functionality.

An intermediary step was undertaken to build a solid foundation and ensure an understanding of the underlying concepts. This involved speed-running a whole university course on global illumination. The goal was to develop a simplified renderer and implement both path tracing and light tracing algorithms with next-event estimation (NEE). Successfully implementing the renderer provided hands-on experience and served as a stepping stone before diving into a complex system - like ART.

Luckily, VCM is very well documented. In addition to the main paper that was published, a technical report and a reference implementation (SmallVCM) exist, which immensely helped with the development and testing. Furthermore, we have used images generated by SmallVCM to check that we are correctly weighing the contributions of the respective methods, i.e., if the contribution of VM and VC align with the reference.

3.2 Brief overview of ART

ART, or Advanced Rendering Toolkit, is a physically based research renderer which first came about at TU Wien in 1996. It is currently maintained and developed mainly by people at Charles University. Although not used in production, ART offers many unique features that are absent in most (or all) systems used today.

Core structures and types are written in ANSI-C. Rest is written in Objective-C.This makes it slightly more complicated to develop, compared to C++ for example, since in recent years Objective-C is predominantly used in the Apple ecosystem. However, we have found that Arch-based distributions and the CLion IDE from JetBrains make the development streamlined and native. **As such, this thesis was developed partly on MacOS (XCode) and Arch Linux (CLion)**.

A more detailed structure illustration is provided in figure 3.1.

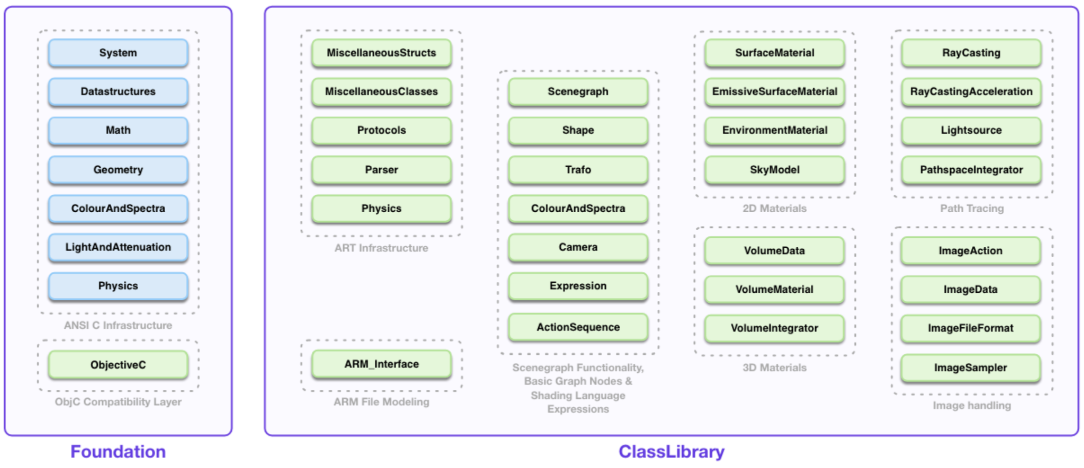


Figure 3.1: A class structure of ART.

It is one of the few renderers with full spectral and HWSS support. Even more commendable is that it is still the only renderer to handle fluorescent materials and polarising effects. All of these features make it a perfect choice not only for researchers but also for production renderers.

As such, ART does not support RGB. Everything is expressed in relation to spectral data. While this creates complexities in certain scenarios, it allows for physical correctness and supports wavelength-dependent phenomena.

For example, path throughput in a standard RGB renderer is relatively trivial - path pdf, material reflectance, and light intensity can be easily stored in a single 3D vector of doubles. In a complex system such as ART, this is not possible.

Refer to figure 3.2, which shows how ART encodes attenuation (i.e., reflectance). It is similar to a reradiation matrix in which the diagonal contains non-fluorescent base reflectance values, and below the diagonal is crosstalk, which describes the crosstalk found in fluorescent materials. In polarisation mode, individual reflectance values are replaced by a Muller matrix.

Attenuation is one of many rather complex structures that must be managed. Another example might be an Intersection class, which stores the hitpoint, directions, material handles, and so on.

Also, a note about HWSS. Since HWSS works by tracing a batch of n wavelengths, classes such as attenuation samples, PDFs, light samples, and so on contain n distinct values for each of the wavelengths. While in a standard renderer, PDF is a single value of type double, in ART, PDF is a n-dimensional vector.

The consequence is that storing hundreds of thousands of vertices, which need to store path attenuations, hitpoints, PDFs, and others, becomes almost unmanageable. The later parts of the thesis will provide more details on how we decided to overcome the problem.

Since ART is already a fully functional and advanced toolkit, we could take advantage of existing functionality and extend it instead of writing everything from scratch. Of course, some parts had to be adapted, but no massive rewriting of the core foundation was needed.

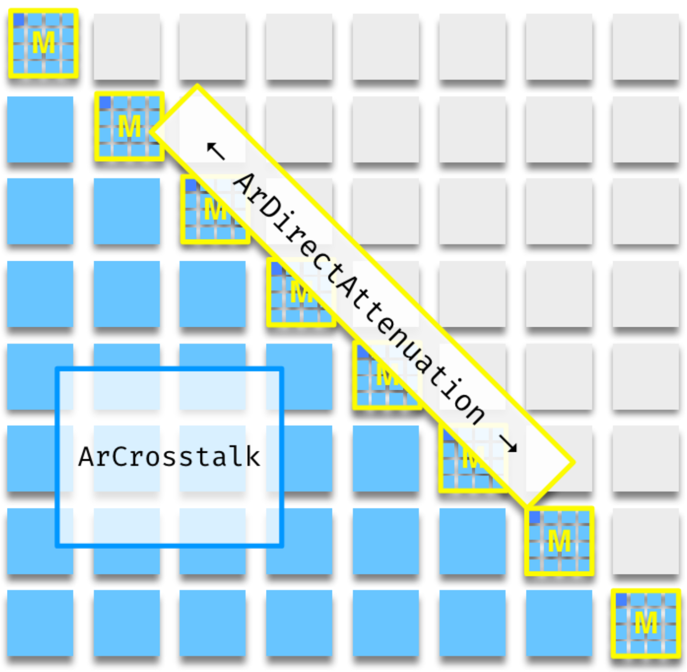


Figure 3.2: The ArAttenuation struct

3.3 Camera implementation

There are many ways one can implement a camera model for a renderer. The most used, and arguably most versatile, is the projection matrix. It also makes it easy to determine which pixel is world point covering by projecting it back onto the camera using an inverse matrix. This can be precomputed ahead of time and does not pose any significant performance issues.

Since the default pinhole camera implementation in ART does not use this approach, we had to slightly rethink the camera model and implement the matrix workflow. However, there is another aspect that makes this worthwhile - this will allow us to unify scene (and camera) descriptions between ART and SmallVCM. This aids in implementation, but mainly in testing, as it sets a common base. We can then easily compare rendered images, assess the correctness of the implementation, and possibly find irregularities.

The VCM framework, similarly to BPT, requires a camera PDF to be used in MIS. In VCM, we calculate the surface factor and the camera PDF as follows

$$\begin{aligned}
&\cos\theta = c_f \cdot -d_{cam} \\
&img\_dist = c_{plane\_distance} \div \cos\theta \\
&img\_dist\_solid\_angle = img\_dist^2 \div \cos\theta \\
&surface\_factor = img\_dist\_solid\_angle \cdot |\cos\phi| \div distance^2
\end{aligned} \tag{3.1}$$

And we set the

$$PDF_{cam} = surface\_factor \tag{3.2}$$

Where c_f is the camera forward vector, d_{cam} is the direction to the camera and ϕ is the angle between the surface normal and d_{cam}.

3.4 Range structure - uniform grid

Due to their lookup speed, photon maps are usually implemented as a range structure. Many existing range structures exist, from kD-trees to hash grids.

The original VCM implementation used a hash grid, in which points are assigned a hash, which is then used for lookup.

However, after some experimentation with hash grids, we have opted for uniform grids to save time and simplify matters. Uniform grids divide the space into n^3 cells. Implementation-wise, it is a 3D static array of integers. We are not storing individual vertices as it is a big waste of space, especially when we already have storage for vertices (the dynamic array). The better solution is to store indexes into that dynamic array for easy look-up.

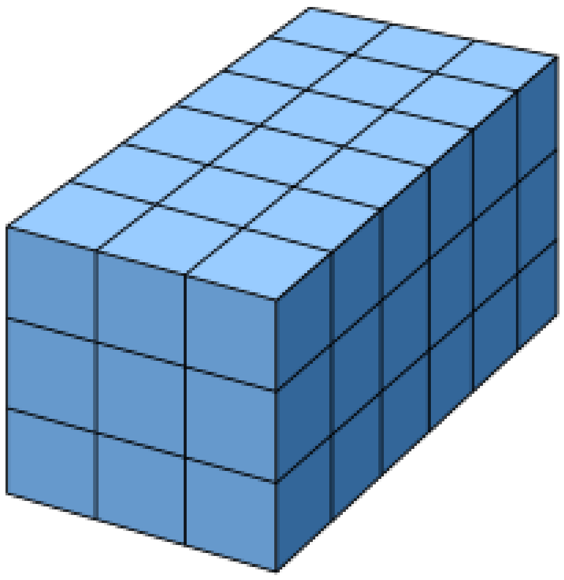


Figure 3.3: A graphical representation of a uniform / regular grid. Taken from a wikipedia page

A regular (uniform) grid is characterized by a parameter n for each dimension, which determines the number of cells along that dimension. To construct the grid:

- Determine the extent of each dimension (e.g., x, y, and z) by calculating the difference between the global maximum and minimum coordinates of all vertices in that dimension.
- For each dimension i, the size of the dimension is: $dimensionSize_i = maxCoordinate_i - minCoordinate_i$
- The cell size for each dimension is then calculated as: $cellSize_i = dimensionSize_i/n$

The cell size in either dimension gives us imaginary breakpoints in the coordinate extent. To illustrate this fact, we can look at Figure 3.3, where each black line represents a breakpoint, and the blue rectangles represent the "inside" of a cell.

It is trivial to get the index of a cell a vertex belongs to. For each dimension i, we can get its index as

$$(pos_i - minCoordinate_i)\ /\ cellSize_i \tag{3.3}$$

However, more than just one cell is required. If the vertex falls close to the edge of a cell, neighboring cells might still contain vertices within the acceptance radius. The simplest method to mitigate this is to assume a 3x3x3 neighborhood around the cell.

3.5 Image sampler and path-space integrator

One of the central core classes used in the rendering process is the ArnImageSampler, derived from ArnImageSamplerBase. This class handles some initial set-up and framebuffer allocations but mainly spawns worker threads and assigns work to each of them by assigning them an execution of the 'renderProc' function. This function is defined in the base class declaration and contains the core rendering routine, varying by the definition of the concrete class.

It is also a place where normalization happens, i.e. averaging the result by the number of generated pixel samples. In a default implementation, this is mostly equal to the number of pixel samples one (artist, user) sets as a program argument. However, for our implementation, this had to be slightly altered. Instead of the per-pixel count, we normalize the result by the number of thread samples. The thread sample is calculated simply as *SPP / THREADCOUNT*. The reason for that is the fact that the per-pixel count is distributed non-uniformly in a light tracer.

Equally as important are the descendant classes of ArnPathspaceIntegrator. As the name suggests, their job is to find a solution to the path integral 1.1. To achieve this, they perform various tasks, such as

- Random walks: They perform random walks, construct new paths by interacting with surfaces, accumulating path pds, throughput and so on
- Path evaluation: Contribution to the final image of each path is evaluated
- Sampling strategies: Such as next-even estimation, importance sampling BSDF

To summarize this, path-space integrators contain the all the logic, that a standard path tracer uses to compute the contributions.

In the current state of ART, there is one main path-space integrator—a path tracer. This thesis aims to add support for **four** more integrators: a light tracer, a bidirectional path tracer, photon mapping, and VCM.

These two families of classes are closely intertwined. While path-space integrators handle the core rendering logic (they are workers), image samplers handle work distribution, scheduling, state, and overall management (they are managers).

3.6 Stochastic image sampler

A sub-class of the base ArnImageSampler, ArnStochasticImageSampler is an existing class that manages the default path tracer.

Each spawned render thread iterates over the pixels in a defined order. In the case of the StochasticImageSampler, a thread performs iterations in scanlines. A scanline in this context is a row. For each pixel, it first generates a new hero wavelength batch and a ray going through that pixel. It sends both items to an integrator along with a framebuffer to perform computations. After getting a result back from the integrator, the sampler checks the validity of contributions and, if valid, splats them onto an image buffer using a splatting kernel - if defined.

After all the measurements are gathered for all iterations, the render threads get destroyed and the parent ImageSampler normalizes the result and writes it to the final image. A pseudocode is provided in 3.6

```
function RENDER(r)
    for y = 0 to height do
        for x = 0 to width do

            #framebuffer to store the current contribution
            pixelBuffer = new FRAMEBUFFER
            hWavelength = GENERATE_RANDOM_WAVELENGTHS()
            ray = GENERATE_RAY_WORLDSPACE(x, y)

            integrator->CALCULATE_CONTRIBUTION(ray, wavelength
               ,&pixelBuffer)

            if pixelBuffer is valid then
                if USE_SPLAT_KERNEL then
                    SPLAT_ONTO_IMAGE_KERNEL(pixelBuffer,
                       wavelength)
                else
                    SPLAT_ONTO_IMAGE(pixelBuffer, wavelength)
                end if
            end if
        end for
    end for
end function
```

3.7 Backwards sampler

The StochasticImageSampler is unsuitable for VCM or other integrators, as it does not allow for intermediate steps.

Therefore, a new image sampler supporting VCM and derived integrators (LT, BPT, PPM) was needed. The new sampler bears the name ArnStochasticBackwardsSampler to emphasize that VCM (and the other integrators mentioned) always starts by tracing paths from light sources.

The architecture behind the BackwardsSampler was iteratively changed multiple times during the implementation. Overall, we can summarize the generations into two distinct architectures, which will be explained and compared.

3.7.1 Base architecture

The first architecture is a straightforward and somewhat intuitive extension of the StochasticSampler.

First, shared variables are initialized.

A struct containing the η_{VCM} weights is one of the most essential shared variables. They are used to properly weigh the contributions, as discussed in an earlier chapter, but can also be used to switch between VC, VM, or VCM.

A dynamic array to store all eligible light vertices is allocated. Light vertices are created at each hitpoint and, if the surface is not specular, get stored in the array for later use. This is needed to perform vertex connection and vertex merging.

In VCM, each eye sub-path gets assigned exactly one light sub-path. A shared array is used to achieve this. Each array element stores the index of the last vertex of the i^{th} light subpath. When eye sub-paths are traced, the light sub-path to be used is simply queried by using the index of the current eye sub-path. A concrete light vertex can be retrieved from the light vertex dynamic array.

A group of shared variables is only initialized when a specific integrator is used. For example, a hash grid is only allocated and built when PPM or VCM is used.

Pseudocode is again provided at 3.7.1 to illustrate the workings of this sampler. First, light tracing occurs, which results in an array of light vertices. We iterate over each of the vertices and check if it has any contribution (2.4) - i.e. if we should splat it onto the image. In the second stage, we again go over the image pixels in scan lines and compute contributions for each pixel.

This approach's main advantage is that it is relatively trivial to implement. For simpler RGB renderers, this might be the preferred solution (the SmallVCM renderer has a similar implementation). Each thread performs the whole rendering job, and their results are normalized.

However, this solution is far from ideal for more complex spectral renderers like ART. The memory overhead of this approach is staggering. To put it into perspective, running a render process to generate an image with a resolution of *512x512* with just 11 threads on a machine with 32 cores took up *32GB* of RAM. The main reason is how much memory the array of vertices takes up. We will provide more information on this in the section explaining the main VCM logic.

```
function SAMPLE(iter)
    ligthVertices = DYNARRAY_ALLOC()
    hashgrid = HASHGRID_ALLOC()
    vcm_weights = GET_VCM_WEIGHTS_FOR_ITER(iter)

    for i = 0 to LIGHT_SUB_PATH_COUNT do
            hWavelength = GENERATE_RANDOM_WAVELENGTHS()
            ray = GENERATE_RAY_WORLDSPACE(x, y)

            integrator->CALCULATE_CONTRIBUTION(ray,
               wavelength, lightVertices)
    end for

    for i = 0 do
        if lightVertex->pixelBuffer is valid then
                if USE_SPLAT_KERNEL then
                    SPLAT_ONTO_IMAGE_KERNEL(pixelBuffer,
                       lightVertex->arrivingWavelength)
                else
                    SPLAT_ONTO_IMAGE(pixelBuffer,
                       wavelength)
                end if
        end if
    end for

    if USE_VCM or USE_PPM then
        hashgrid->SET_UP(lightVertices)
    end if

    for y = 0 to height do
        for x = 0 to width do
            #framebuffer to store the current contribution
            pixelBuffer = new FRAMEBUFFER
            hWavelength = GENERATE_RANDOM_WAVELENGTHS()
            ray = GENERATE_RAY_WORLDSPACE(x, y)

            integrator->CALCULATE_CONTRIBUTION(ray,
               wavelength, &pixelBuffer, hashGrid)

            if pixelBuffer is valid then
                if USE_SPLAT_KERNEL then
                    SPLAT_ONTO_IMAGE_KERNEL(pixelBuffer,
                       wavelength)
                else
                    SPLAT_ONTO_IMAGE(pixelBuffer,
                       wavelength)
                end if
            end if
        end for
    end for
end function
```

3.7.2 Tiling

To mitigate the shortcomings of the first architecture, we had to devise a new strategy to fully utilize the system's resources—that is, to take advantage of all the CPU cores and significantly reduce memory overhead. To simplify matters, in this section, we will assume a computer with a 32-core (32-thread) CPU and that we fully utilize the CPU.

This proved to be non-trivial, as it requires balancing BPT and PPM simultaneously. Without PPM, we can easily generate one full light subpath, trace a ray from the camera, and connect the two subpaths.

However, if we assume the full VCM, this is not possible. Vertex connections and merging are calculated during the same camera subpath walk; vertex merging requires an already-built hash grid.

The first iteration of this method fully respected the above restriction and tried to cut a few corners to optimize resource usage. Instead of generating a complete set of light paths for each render thread, we've split the work; each render thread now generates only a subset of the full set. These subsets are then merged into one, which is then used in the second stage of the VCM algorithm - as previously explained. This translates to generating one set of light paths instead of 32 sets.

While this fixed the memory overhead, it introduced a set of new problems. Variance has increased significantly. For example, recall that instead of having 32 unique sets to perform VCM on, we only have one set shared across the 32 render threads to perform VCM. Furthermore, the overall speed-up was minor, making it not worth the variance increase.

In the second iteration, we tried to keep the initial idea of splitting work into subsets or 'buckets' while mitigating the constraints.

After much thought, we've noticed that we can further split the second stage into two separate phases. Therefore, we can create two camera passes, one that handles vertex connection and the other vertex merging. But this creates an opportunity to join the light pass, and vertex connection passes into a single stage, as is the case with many implementations of BPT. More details are presented later. With this in mind, we can now describe how the second architecture works.

Again, we have utilized tiling to distribute work across threads. We divide the resulting image into multiple tiles whose size is set in a macro. For example, an image of size 512x512 and a tile size of 32x32 can be divided into 256 tiles. These 256 tiles are then evenly distributed across the threads. The index of the first tile is set to be the same as the thread index. The iteration step is then set to be equal to the thread count. A thread with an index of 2 would first process a tile of index 2. Next, it would add the thread count to the index, which, in this case, would result in 34. This continues until the index number is higher than the number of tiles.

Recall from the first architecture the existence of an array that holds light path indices to be later used in the vertex connection. We can now scratch this array. Vertex connections are now handled one by one. After generating a single light sub-path, the light pass is done, and we can start tracing an eye sub-path. However, we still need to save all the light paths to build a range structure for vertex merging. This can be done by having a global two-dimensional dynamic array. The first dimension represents the number of threads, and the second represents the buckets for each thread. After the vertex connection stage, we put a thread barrier, set one thread to merge the buckets and build the range structure. The barrier is there to ensure that all threads operate over the same data. With the range structure in place, we apply tiling, trace eye subpaths, and perform vertex merging where possible.

```
#define TILE_SIZE = 32

class ArnStochasticBackwardsSampler
    lightVertices = ARRAY_ALLOC(THREAD_COUNT, DYNARRAY_ALLOC()
    hashgrid = HASHGRID_ALLOC()
end class

function SAMPLE(iter)
    vcm_weights = GET_VCM_WEIGHTS_FOR_ITER(iter)
    tile_count = (imageSize->x / TILE_SIZE) * (imageSize->y /
       TILE_SIZE)

    for i = THREAD_INDEX to TILE_COUNT step THREAD_COUNT do
            pixelBuffer = new FRAMEBUFFER
            hWavelength = GENERATE_RANDOM_WAVELENGTHS()
            ray = GENERATE_RAY_WORLDSPACE(x, y)
            path = integrator->GENERATE_LIGHT_PATH(ray,
               wavelength, pixelBuffer, lightVertices[i])

            integrator->CALCULATE_CONTRIBUTION(ray, wavelength
               , &pixelBuffer, path)

            if lightVertex->pixelBuffer and pixelBuffer is
               valid then
                if USE_SPLAT_KERNEL then
                    SPLAT_ONTO_IMAGE_KERNEL(lightVertex->
                       pixelBuffer, hWavelength)
                    SPLAT_ONTO_IMAGE_KERNEL(pixelBuffer,
                       hWavelength)
                else
                    SPLAT_ONTO_IMAGE_KERNEL(lightVertex->
                       pixelBuffer, hWavelength)
                    SPLAT_ONTO_IMAGE(pixelBuffer, wavelength)
                end if
        end if

    end for


    if USE_VCM or USE_PPM and THREAD_COUNT == 0 then
        hashgrid->SET_UP(lightVertices)
    end if

    THREAD_BARRIER

    for i = THREAD_INDEX to TILE_COUNT step THREAD_COUNT do
```

```
        #framebuffer to store the current contribution
        pixelBuffer = new FRAMEBUFFER
        hWavelength = GENERATE_RANDOM_WAVELENGTHS()
        ray = GENERATE_RAY_WORLDSPACE(x, y)

        integrator->CALCULATE_CONTRIBUTION(ray, wavelength, &
           pixelBuffer, hashGrid)

        if pixelBuffer is valid then
            if USE_SPLAT_KERNEL then
                SPLAT_ONTO_IMAGE_KERNEL(pixelBuffer,
                   wavelength)
            else
                SPLAT_ONTO_IMAGE(pixelBuffer, wavelength)
            end if
        end if
    end for
end function
```

3.8 The VCM path-space integrator

This section will introduce the path-space integrator. The main logic behind VCM happens here. However, there were cases where a specific functionality resulted in modifying or extending the existing ART codebase. Such instances will be explained in this section as well.

3.8.1 Tracing the light paths

Extending the light sources

Before implementing a functional light tracer, we first needed to extend the functionality of the light source interaction. By default, there was no support for generating a ray from a point on the light source into the scene. This is to be expected since a path tracer does not need such a feature. We will only assume area light sources, as they are a) plausible in the real world b) arguably most versatile c) intuitive to implement.

First, we needed to sample a point on a light source, which will serve as the origin of the generated ray. Much of this code was taken from the existing sampling method, as there was no reason to develop a new approach. The result is a point on a light source along with a normal vector.

Secondly, we had to add a new utility function to generate a cosine-weighed direction sample, i.e., a direction along a hemisphere. For the direction to be usable, we also needed to transform it w.r.t the normal vector. That is because the generated direction is oriented w.r.t to the word up-vector.

Finally, we need the PDF. Two probabilities are at play here: a direct PDF and an emission PDF. A direct pdf is simply

$$\frac{1}{area} \tag{3.4}$$

Where *area* is the area of the light source. The emission pdf is slightly more complicated. It is a joint probability of generating the direction (cosine weighed) and the direct pdf:

$$\frac{\sqrt{r_2}}{\pi} * \frac{1}{area} \tag{3.5}$$

The term r_2 is a random generated number.

A path vertex

One of the most essential pieces of the puzzle is the path vertex. A path vertex stores all the information accumulated during random walks, such as throughput. Essentially, path vertices are the intersection points of a random path walk.

In ART, a path vertex must store a non-trivial amount of data:

- Intersection
- Path attenuation
- Light sample
- Camera contribution
- Path PDF
- Incoming and outgoing wavelengths
- VCM sub-weight values d_{VC}, d_{VCM}, d_{VM}

ArcIntersection in ART is a class that gives us access to the world hit point, the normal vector, distance traveled, function handles to interact with the surface material, etc.

In a simpler (usually RGB-based) renderer, path throughput can be expressed easily using a 3D vector of type double since its value is cumulatively calculated at each intersection point as

$$\begin{aligned} &thrtpt *= \frac{reflectance}{pdf} \\ &\text{or} \\ &thrpt *= light \end{aligned} \tag{3.6}$$

As discussed in the introductory piece about ART, this approach is unusable. Therefore, the throughput is split into three parts combined later: attenuation, light sample, and pdf.

The camera contribution plays the role of a framebuffer, which stores the contribution of $s = 1$ paths, as briefly discussed in the section about samplers 3.7.1.

Light tracing logic

With this in place, we can finally describe how light tracing is implemented.

Now that we have extended the light source sampling logic, we can generate a light sample with a direction. We then use the sampling PDFs to initialize the MIS sub-weights.

We trace a ray until (or if) we hit a surface. If we hit a light source, we immediately terminate the light tracing algorithm and return back to the sampler.

In the case of a valid surface, we query the intersection to get the cosine theta of the fixed direction. The cosine is used to update the MIS sub-weights

$$\begin{aligned} d_{VCM} &\mathrel{*}= dist^2 \\ d_{VCM} &= \frac{d_{VCM}}{\cos\theta} \\ d_{VC} &= \frac{d_{VC}}{\cos\theta} \\ d_{VM} &= \frac{d_{VM}}{\cos\theta} \end{aligned} \tag{3.7}$$

For non-specular surfaces, we first calculate the camera contribution, i.e s=1 paths, which will differ depending on the algorithm used. In the case of a pure light tracer, no MIS weights are calculated. In either case, we have to query the camera for the surface factor and camera PDF.The surface factor is multiplied by the light subpath count and used to normalize the contribution.

The w^{light} is computed as

$$\frac{\overrightarrow{pdf_{cam}}}{n^{light}} \cdot (\eta_{VM} + d_{VCM} + d_{VC} \cdot \overleftarrow{pdf_{bsdf}}) \tag{3.8}$$

and the w_{VC} as

$$w_{VC} = \frac{1}{w^{light} + 1} \tag{3.9}$$

This formula begs a question: how do we plug in the probabilities? Recall that ART has full HWSS support, which means that all values, such as PDF or attenuation, are n-dimensional vectors, and not single values, as in other renderers. Do we evaluate the weight for each wavelength?

The answer to that question stems directly from the philosophy behind HWSS: *All decisions are made based on the HERO wavelength.* Therefore, we can safely calculate the MIS weight using only the HERO probability.

It is important we do not forget to include HWSS weights in the calculation. The HWSS weight is calculated from the current path PDF and appended to the MIS weights.

$$w^{hwss} \cdot w^{mis} \tag{3.10}$$

Contribution normalization happens by dividing by

$$\frac{n^{light}}{factor} \tag{3.11}$$

Then, the information gathered so far is saved into a new path vertex, which is then pushed into the dynamic array. This vertex will then be potentially used for vertex connection or vertex merging.

Surfaces causing singularities, i.e., specular surfaces, don't allow for vertex connection or merging, so we don't calculate the camera contribution, nor do we save the path vertex.

The final step of an iteration is performing a random walk, during which we sample the BSDF to get continuation direction and the pdfs to update the MIS sub-weights.

This alone gives us a fully functional light tracer with an NEE equivalent. But mainly, it sets up the base for the whole VCM framework.

3.8.2 Connections and collections

Having established the foundation of our rendering system with light sub-path generation, we now turn our attention to a more advanced technique that builds upon this groundwork. The next phase of our implementation involves integrating VC and VM.

As previously discussed, our implementation splits up the camera subpath generation stage. Camera paths for connections and merging are now traced separately.

The render loop of the camera tracer is more complicated than the loop of a light tracer. That is not surprising—it is the stage where we collect all the light vertices, perform connections on them, and query the space for all neighboring vertices.

We begin by casting rays from the camera into the scene. At each intersection point along these camera paths, several important computations are performed:

- Update of MIS sub-weights. This is the same as in light tracing.
- Direction sampling: A light source is directly hit, we need to weigh the contribution to the image, as so-called t=0 paths
- NEE: At each point, t=1 paths are also to be evaluated. This is standard NEE, in which we sample a point on a light source and perform an explicit connection
- Vertex connection: The last step is to the current vertex and try to connect it with the light sub-path vertices.
- Random walk: sampling the BSDF, updating the MIS sub-weights. In this case, pretty much identical to the light tracer

Vertex merging is excluded from this list on purpose. While in the original implementation, vertex merging is just another step after the Vertex connection, it stands alone in our implementation. More specifically, a separate camera tracing loop is utilized to perform vertex merging. The reasons for that were already discussed.

Direction sampling

At each stage of the loop, there exists a non-trivial probability of hitting an area light source. Obviously, for point lights, this probability is zero. For MIS purposes, we again need the direct pdf and the emission pdf. In this context, one is the probability of hitting the light source, which intuitively is

$$\frac{1}{area} \tag{3.12}$$

and the other is the probability of emitting a photon in the reverse of the incident direction

$$\cos\theta \cdot \frac{1}{\pi} \cdot \frac{1}{area} \tag{3.13}$$

where θ is the angle between the normal and the direction.

NEE

Next event estimation, or direct light sampling, is a popular technique for variance reduction and is frequently used in path tracers. In the context of BPT, they play the role of $t = 1$ paths, where we have a camera subpath with s vertices and try to append the generated light vertex.

Again, a light source is sampled for a point, spectral intensity, and PDFs. The emission pdf remains the same. In the case of direct PDF, we have to include a Jacobian factor to convert between measures:

$$\frac{1}{area} \cdot \frac{\text{distance}^2}{\cos\theta} \tag{3.14}$$

Finally, BSDF is evaluated at the surface, and MIS weight is calculated.

For w_{light} we get

$$w_{light} = \frac{\overrightarrow{pdf}}{p_{light} \cdot pdf_{direct}} \tag{3.15}$$

and for w_{camera}

$$w_{camera} = \frac{pdf_{emission} \cdot \cos\theta}{\cos\phi \cdot pdf_{direct}} \cdot (\eta_{VCM} + d_{VCM} + d_{VC} \cdot \overleftarrow{pdf_{bsdf}}) \tag{3.16}$$

where θ is an angle at the surface and ϕ is an angle at the light source. Putting it all together, we get

$$\begin{aligned} w_{VC} &= \frac{1}{1 + w_{light} + w_{camera}} \\ w_{VC} &= w_{VC} \cdot w_{hwss} \end{aligned} \tag{3.17}$$

Connections

Now, we can make connections for non-specular surfaces (vertices). As discussed, a light subpath is passed from the sampler to make connections.

We iterate over all light subpath vertices for the current eye vertex and try to perform connections. First, we need to check for any occlusion; if there is, we skip it. Secondly, we have to make sure that the hero wavelengths match between vertices. This is always true for non-fluorescent scenes since we are tracing the eye path with the same wavelength batch as the light path.

Next, we evaluate the BSDF at both surfaces to get attenuations, PDFs, and reverse PDFs. Then, we convert the forward PDFs from the solid angle measure to the area measure, calculate the geometry term between the two surfaces, and combine all values using MIS.

$$\begin{aligned} w_{light} &= \overrightarrow{pdf^{A}_{camera}} \cdot (\eta_{VCM} + d_{VCM} + d_{VC} \cdot \overleftarrow{pdf^{\sigma}_{light}}) \\ w_{camera} &= \overrightarrow{pdf^{A}_{light}} \cdot (\eta_{VCM} + d_{VCM} + d_{VC} \cdot \overleftarrow{pdf^{\sigma}_{camera}}) \\ w_{VC} &= \frac{w_{hwss}}{1 + w_{light} + w_{camera}} \end{aligned} \tag{3.18}$$

The resulting MIS weight must also be weighed by the HWSS weight, which we only evaluate for the eye subpath pdf. The reason we don't have to evaluate

HWSS weights for the light subpaths is that they are already weighed during light tracing.

Merging

The last step to make VCM functional is to allow for vertex merging/photon mapping.

The path generation loop in the case of vertex merging is much simpler than in the case of vertex connection. This is because we only calculate the contributions of vertex merging and direct radiance, for cases when we hit a light source.

At each intersection, we update the MIS sub-weights, query the range structure, and perform a random walk In case pure PPM is used, we stop the loop at the first non-specular surface.

As discussed in 3.4, we need to query the range structure to get the nearby vertices. The implementation itself is done so that we can pass a single vertex, and the grid will return the collected contribution. Essentially, we are distilling all collected photons into a single weighted photon.

For each photon in an acceptance radius r, we weight it by w_{VM} and accumulate its contribution:

$$\begin{aligned} w_{light} &= d_{VCM}^{photon} \cdot \eta_{VCM} + d_{VM}^{photon} \cdot \overrightarrow{pdf_{camera}} \\ w_{camera} &= d_{VCM}^{camera} \cdot \eta_{VCM} + d_{VM}^{camera} \cdot \overleftarrow{pdf_{camera}} \\ w_{VM} &= \frac{w_{hwss}^{light}}{1 + w_{camera} + w_{light}} \end{aligned} \tag{3.19}$$

The resulting contribution is then weighed again by the eye subpath throughput and HWSS weight - not to be confused with the HWSS weight used in equation 3.19, which was calculated from the light subpath PDF!

Handling wavelength mismatch

However, wavelength mismatch can happen even in non-fluorescent scenes, unlike in VC. That is to be expected; each traced light subpath is assigned a randomly generated wavelength, and in a single neighborhood, there might be higher thousands of vertices. The natural consequence is that we may discard most of the vertices, which introduces significant variance. Since the contributions are normalized by the number of light subpaths (recall section about VCM), the brightness is also affected.

Now, there are two potential fixes.

The first is arguably more straightforward. Instead of generating a random wavelength batch for each subpath, we generate one wavelength batch at the start of each sample, i.e., each sample is assigned a *global* wavelength. For scenes without fluorescence, this will make all vertices eligible for merging.

Sadly, this might introduce color variance and require much higher sample counts to distribute the queried wavelengths along the spectrum properly.

The second solution required extending the renderer functionality but may produce better results. It is inspired by the paper in which Wilkie et al.[WND⁺14] introduced HWSS.

For each potential merge, we check whether any of the light vertex's wavelengths in a batch match the hero wavelength of the camera subpath. If they do, we cyclically rotate the batch to match the hero wavelengths and proceed with the merge. Matching a wavelength perfectly is almost impossible, so we allow an error margin. The absolute difference between the two wavelengths cannot exceed 5NM.

To support this attenuation, PDFs and light samples had to be extended so that we could rotate the elements. This required implementing new interface handles and support for piece-wise element assignment.

Indeed, this significantly increases the number of mergeable vertices, but it is still relatively low. However, we still need to include the probability of being able to perform the rotations.

The entire visible spectrum in ART is 320 NM and since we allow 5NM error margin, the probability of any two wavelengths matching is

$$\frac{2d}{320} = \frac{10}{320} = \frac{1}{32} \tag{3.20}$$

where d is the margin. We also have to factor in that we are not tracing one wavelength; hwss works by tracing wavelength batches. The size of the batch is a changeable parameter, n_h.

$$\frac{10 \cdot n_h}{320} = \frac{n_h}{32} \tag{3.21}$$

The default n_h count in ART is four (this allows for SIMD operations), which would make this probability simply

$$\frac{1}{8} \tag{3.22}$$

Combining this probability with the rotated light vertex path PDF then yields the desired results.

3.8.3 Making fluorescence work

Implementing fluorescence in a renderer is a complex task. As such, fluorescence is relatively underresearched compared to other branches of computer graphics. Just months before this thesis, a dissertation that resulted in a BPT handling fluorescence using mollification was finally published.

After consulting with the authors of [JHD20], the most significant challenge would seem to be correct calculations of MIS weights, with added dimensionality of VM. As such, adding fluorescence into a complete VCM framework would require more time to conduct research.

However, despite the limitations, we have added support for some fluorescent phenomena to a light tracer and a photon mapper. The photon mapper has one drawback: it does not support HWSS when handling fluorescence.

ART is already fully fluorescent capable, which significantly helped with the implementation. Most fluorescence-related calculations are greatly abstracted. Fluorescent probabilities or wavelength shifts all happen in the background when we interact with a surface.

For example, when we sample a BSDF in ART, it returns a new wavelength (the same as the previous one in case of non-fluorescent interactions), direction, attenuation, and PDF. This is standard behavior. We then use these values to update the path throughput and continue the random walk. In the case of a fluorescent surface, the probability of shifting a wavelength is already included in the returned PDF.

The only time we have to explicitly ask a surface for a new wavelength or a probability of shifting from wavelength a to b is when we are *evaluating* a BSDF, i.e., in NEE. Why? Evaluating a BSDF in ART requires two wavelengths (incoming and outgoing), and we don't have the outgoing wavelength. Therefore, we ask the surface for a new wavelength (again, for non-fluorescent surfaces, it's just the incoming wavelength), and along with a new wavelength, we also get the probability. Reminding the reader that fluorescent probabilities are taken from the reradiation matrix of a fluorescent surface 1.9.

We will now present our propositions for handling fluorescence in LT and PPM. Their validity is supported by their consistency in output and visual similarity to results produced by a reference path tracer, disregarding noise. The solution for photon mapping is especially prone to color variance, due to omitting HWSS.

Fluorescent light tracing

As we have defined previously, light tracer functionality in ART is closely similar to a path tracer with NEE. As such, we can take inspiration. In a path tracer, wavelength shift probabilities are combined with the sample PDF (direct pdf) we get from a light source.

In our implementation of a light tracer, the surface factor plays the role of an NEE PDF. As such, we can now write

$$\frac{n^{light}}{factor} \rightarrow \frac{n^{light}}{factor \cdot pdf_{shift}} \tag{3.23}$$

Fluorescent photon mapping

Due to the complexity that HWSS introduces in this case, we've decided to make photon mapping work in plain spectral mode as the minimal solution. However, this introduces significant color noise. That is to be expected and is the main motivation for techniques like HWSS to exist in the first place.

This solution was inspired by the work of Jung et al. [JHD20], in which they solved fluorescence in a standard BPT. The main idea behind their method is to mollify a BSDF, which effectively extends the range at which wavelengths can be connected.

This solution might also aid in future work of extending it to VCM.

Similar to how the collection radius in VCM (PPM) is iteratively reduced, the mollification range is also shrunk with each iteration.

First, we define d_0 as the initial mollification range. The actual range is then defined as

$$d = d_0 \cdot n^{-s} \tag{3.24}$$

where n is the current iteration and s is the shrinkage factor. We have set $d_0 = 50nm$ and $s = \frac{1}{4}$.

At each potential merge, we check that the distance between the traced wavelength and the photon's wavelength is at most d.

$$|w_0 - w_p| \leq d \tag{3.25}$$

The probability of such a connection is then intuitively defined as

$$\frac{2d}{320} \tag{3.26}$$

where 320 is the range of the visible spectrum.

Results are presented in Figures 4.15 and 4.8.

4. Results

In this chapter we present the results of implementing VCM. Most of the images will focus heavily on SDS paths, as they are the main motivator for using VCM.

One such scene can be seen in Figure 4.1. In a standard CornellBox, we made the floor and the back wall with OrenNayar materials; the rest are Lambertian. There are four spheres in total, each of them specular. The sphere near the light source is glass. The grooved scenes add great variety in light scattering. Figure 4.1 and Figure 4.4 show a difference between a perfectly smooth glass sphere and such grooved spheres. Thanks to the placement of the sphere, the difference between PT and VC is text-book-like, where the path tracer struggles with sampling SDS paths and caustics. Figure 4.3 then shows a zoomed in comparison.

Both Figure 4.1 and Figure 4.4 clearly illustrate individual techniques' relative strengths and weaknesses. For example, in the light-traced image, we can see how it handles caustics much better than a path tracer. However, it also clearly shows the main shortcoming of LT - light sources are black, as we cannot splat specular surfaces onto an image.

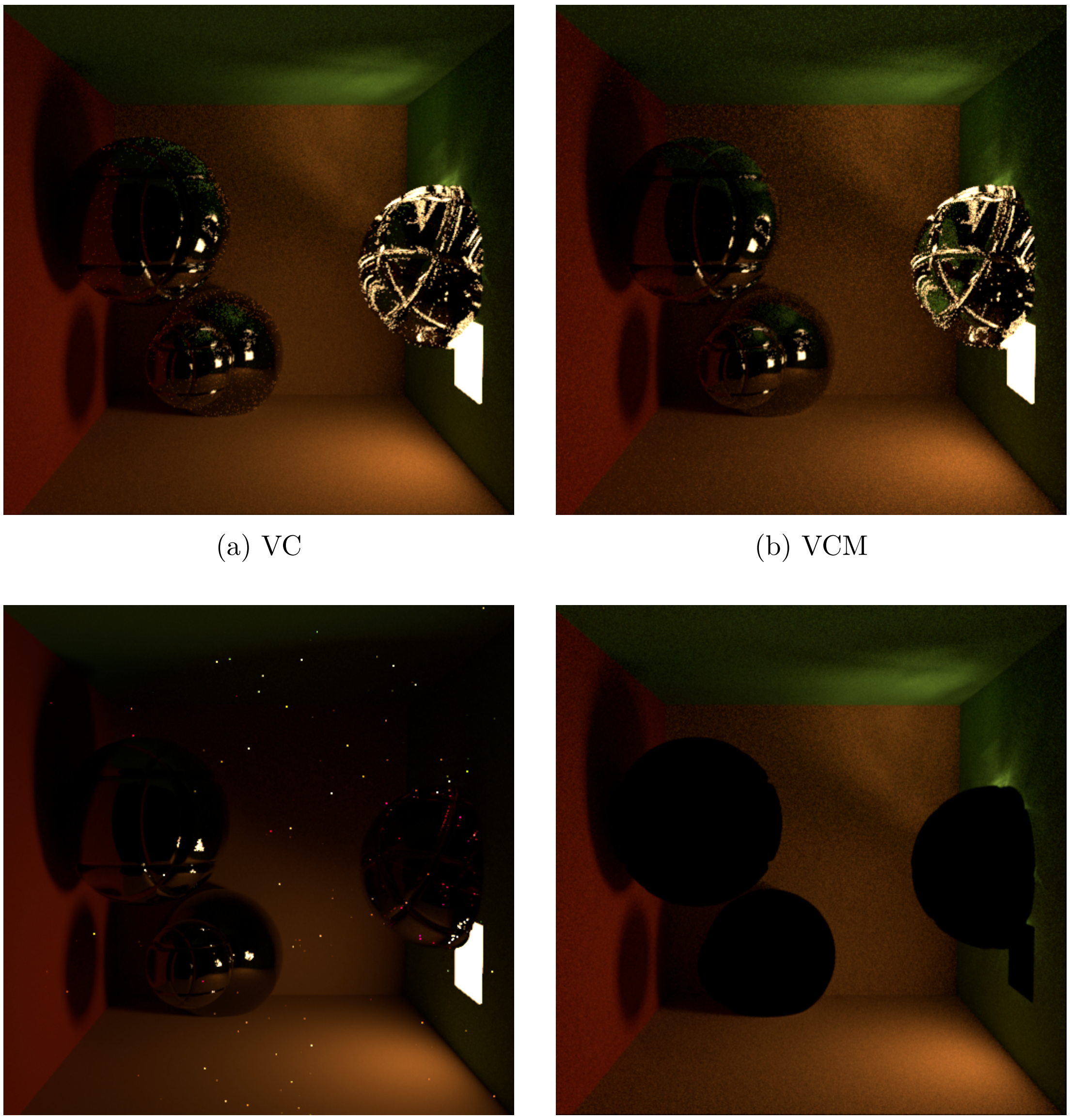
Figure 4.1 (and Figure 4.4) also show that VCM (or PPM for that matter) handles SDS paths better then BPT. Figure 4.2 shows zoomed-in images for more clarity.

To properly compare and test VC vs VCM, we've set up two scenes which should produce sharp caustics: 4.9, 4.10, 4.13 and 4.12. A zoomed in detail view is provided in 4.11.

We've taken inspiration from the original paper and generated an image similar to 2.4, i.e, plot out the relative contributions of individual techniques 4.6. We can see that this is consistent with figures 4.2 or 4.4.

Interesting results were achieved in Figures 4.15 and 4.8, where we show how our implementations of LT and PPM handle fluorescent surfaces. Figure 4.15 contains two grooved spheres with synth fluorescent surfaces enclosed in a Cornell box with white diffuse surfaces. Besides variance or color noise in the case of PPM, results seem to be consistent. In Figure 4.8, we've swapped the diffuse surface with the same synth fluorescent material.

Surprising might be the slow convergence. All of these images were rendered for almost 2 hours, but there is still visible color noise. Especially in PT in 4.8. Unsurprisingly, omitting HWSS, lead to significant color noise for images rendered with PPM.



(a) VC (b) VCM

(c) PT (d) LT

Figure 4.1: Images from the left: BPT, VCM, PT with NEE, and LT. Comparison of 4 different rendering methods, with 2000 SPP (divided among the threads). Path tracer was given to have a few hundred samples more, which is why the floor is much smoother. This was done for various reasons. Mainly to show how much more effective the other algorithms are in certain scenarios.

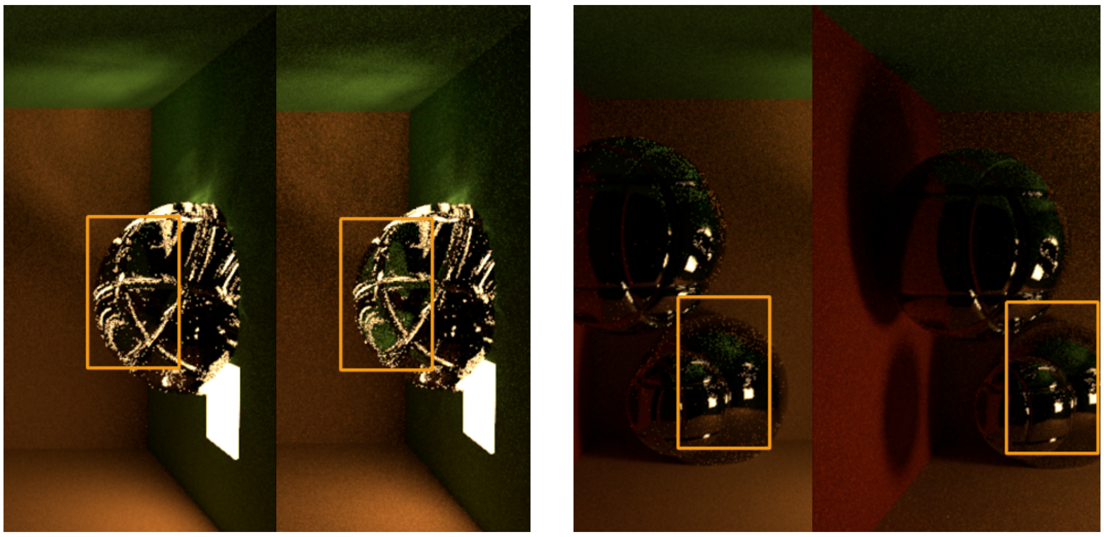


Figure 4.2: Zoomed in images of VC and VCM from figure 4.1 to show the convergence rate difference between them on a specular surface. This is consistent with the results presented in the original paper 2.1

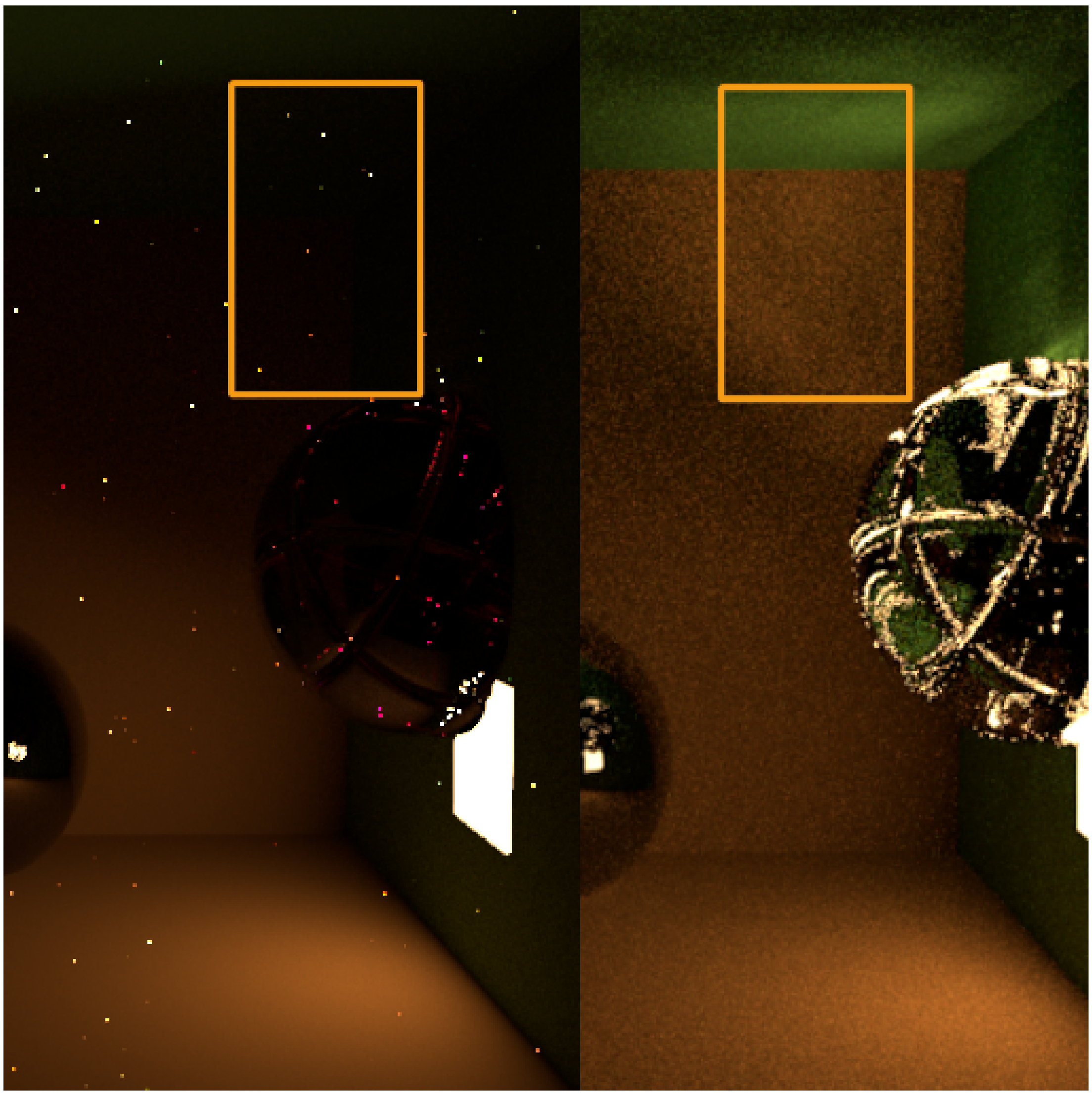
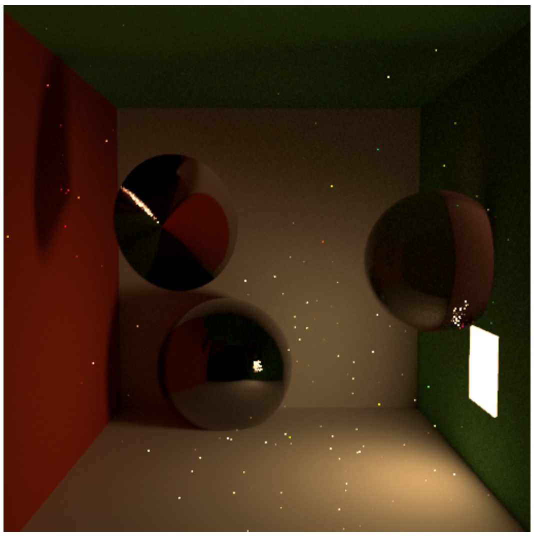
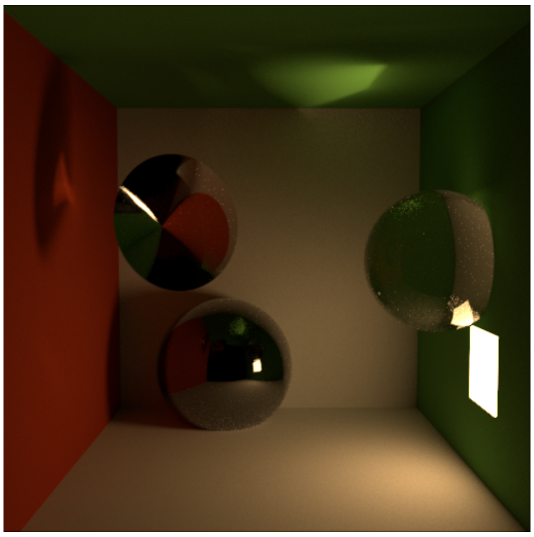


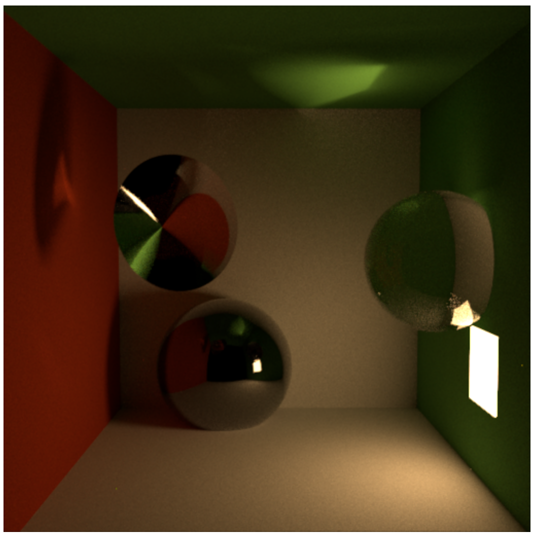
Figure 4.3: Great illustration of how efficiently VCM samples SDS paths. Although we have given the PT an edge, which can be seen on the bright spot on the floor, PT did a very bad job in this instance.



(a) PT



(b) VC



(c) VCM

Figure 4.4: Images from the left: BPT, VCM, PT with NEE, and LT. SDS comparison with smoother specular objects. This scene contains a perfectly smooth glass sphere and a cone plus a mirror sphere down on the floor.

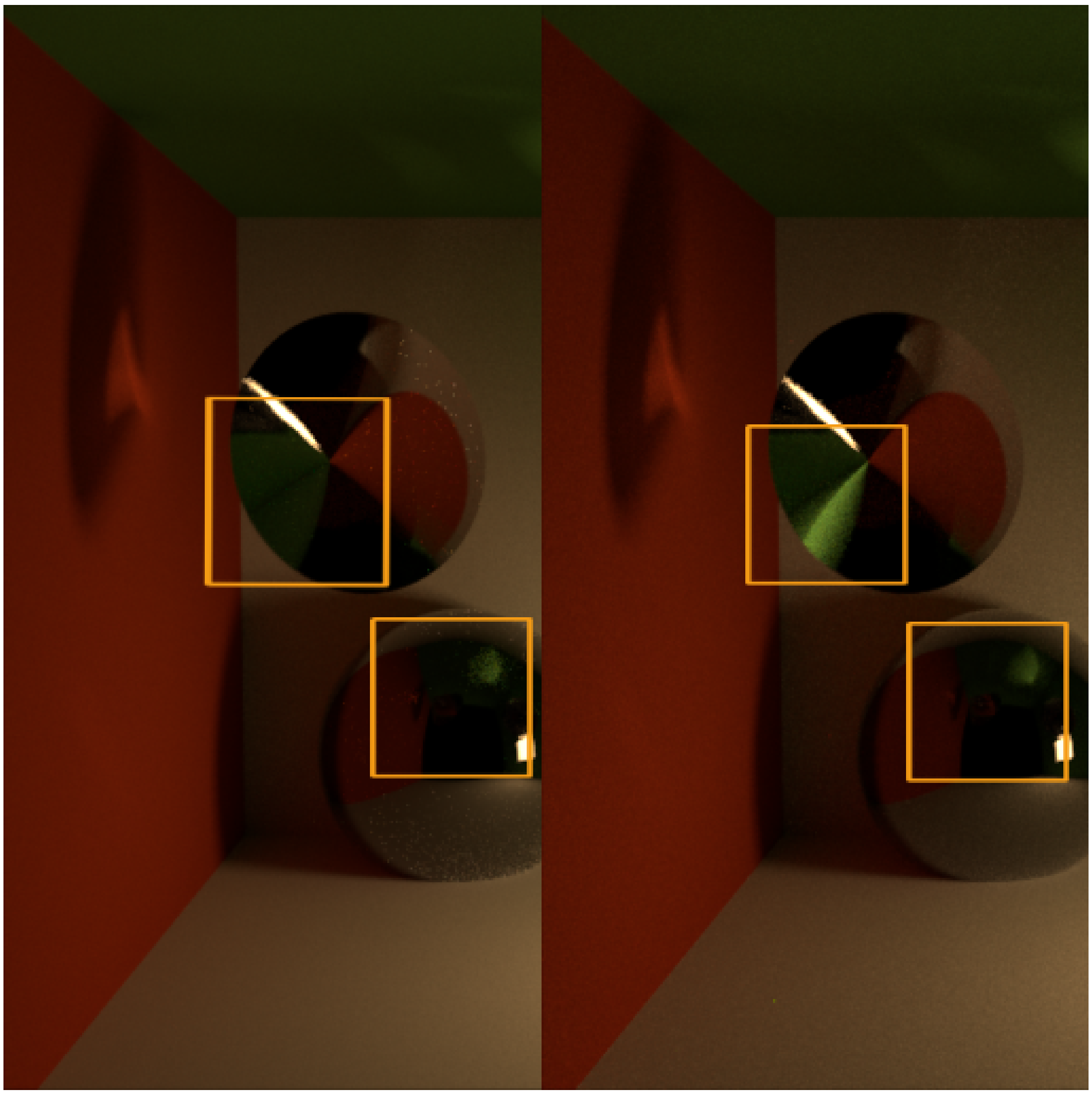


Figure 4.5: Another Comparison between BPT (VC) and VCM. In this example, BPT struggles with sampling the caustics in the cone.

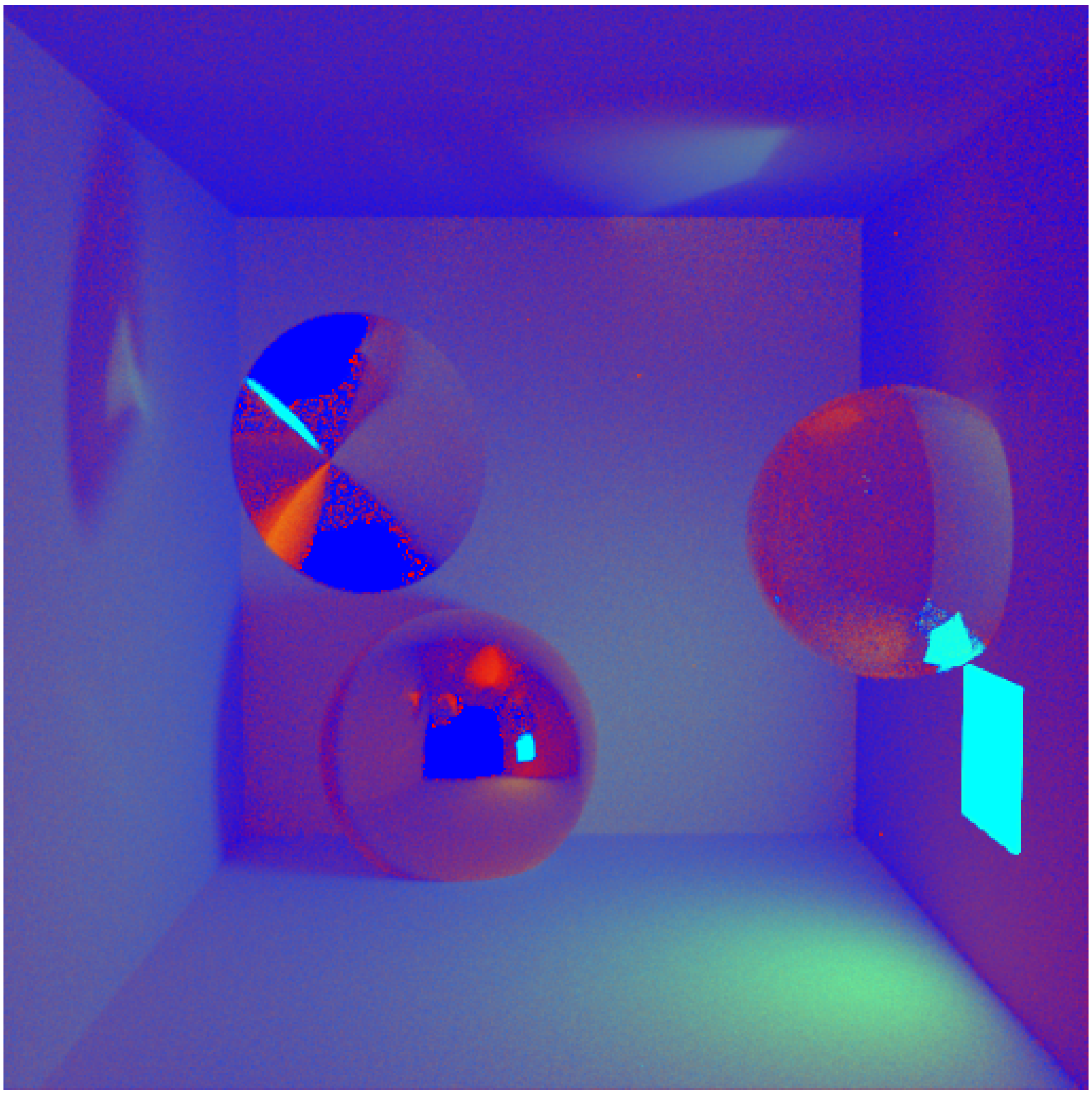
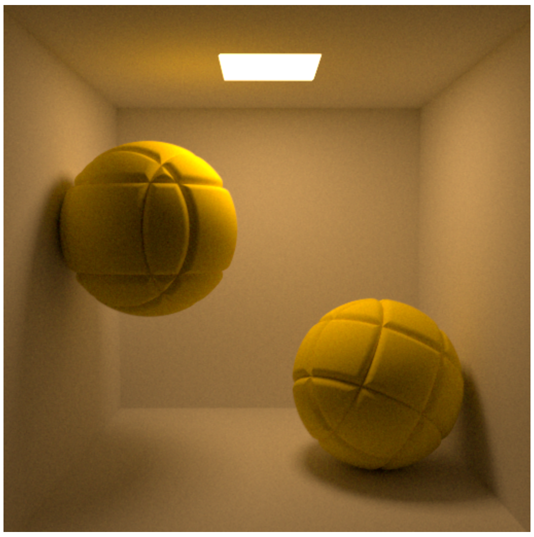
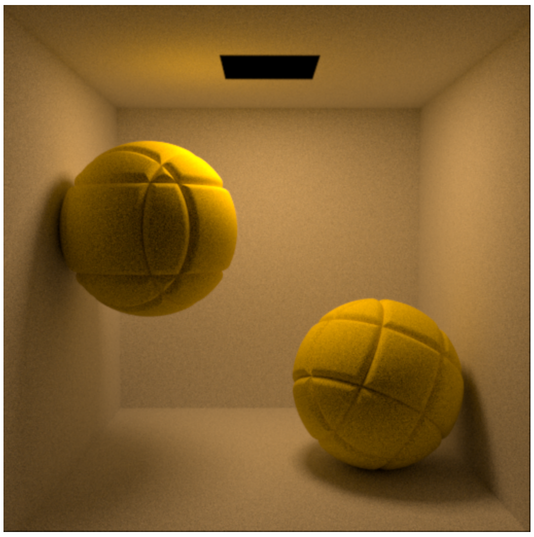


Figure 4.6: Relative contribution of VC vs VM. **Red** is more VM and **Blue** is more VC


(a) PT


(b) LT

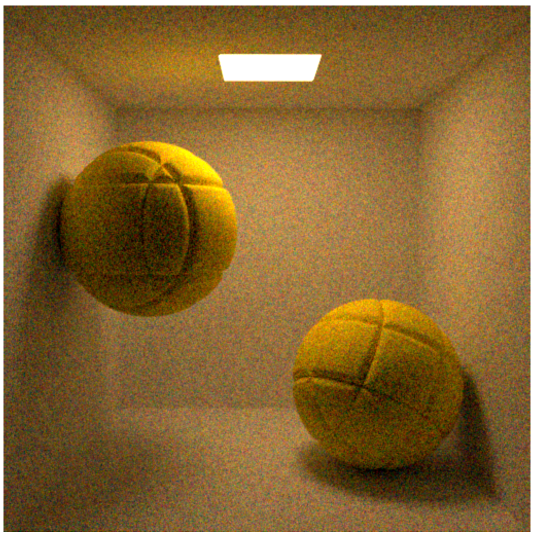
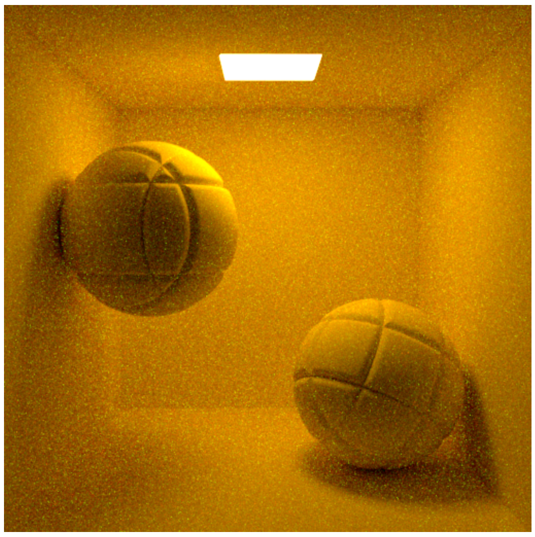
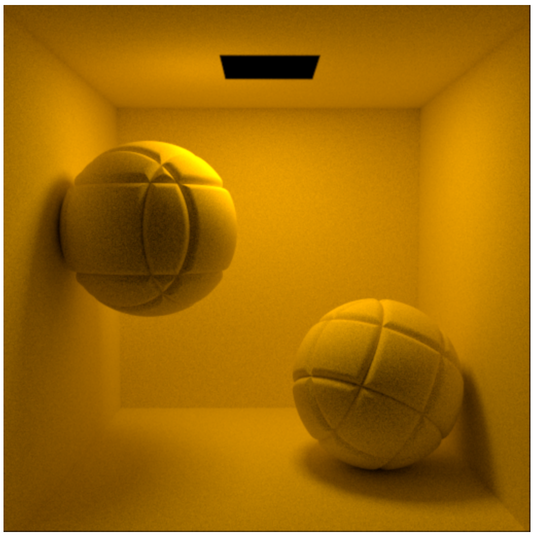

(c) PPM

Figure 4.7: Sample count set to 12000 SPP. The walls are white Lambertian surfaces, and the spheres have a synthetic fluorescent material assigned. Besides the visible noise, the results seem to be consistent with PT. Notice the color noise in the case of PPM. That is a consequence of not using HWSS but tracing a single wavelength at a time.


(a) PT


(b) LT

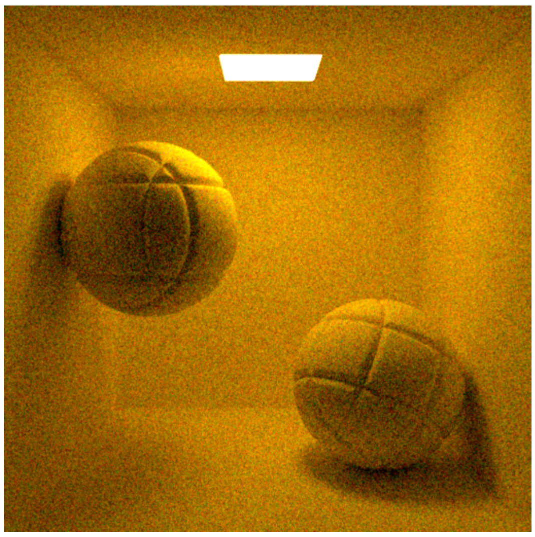

(c) PPM

Figure 4.8: Same setting as with 4.15, but all objects (and walls) have the same synth fluorescent material. The PT solution produced significant variance (fireflies). Again, PPM struggles with color noise.

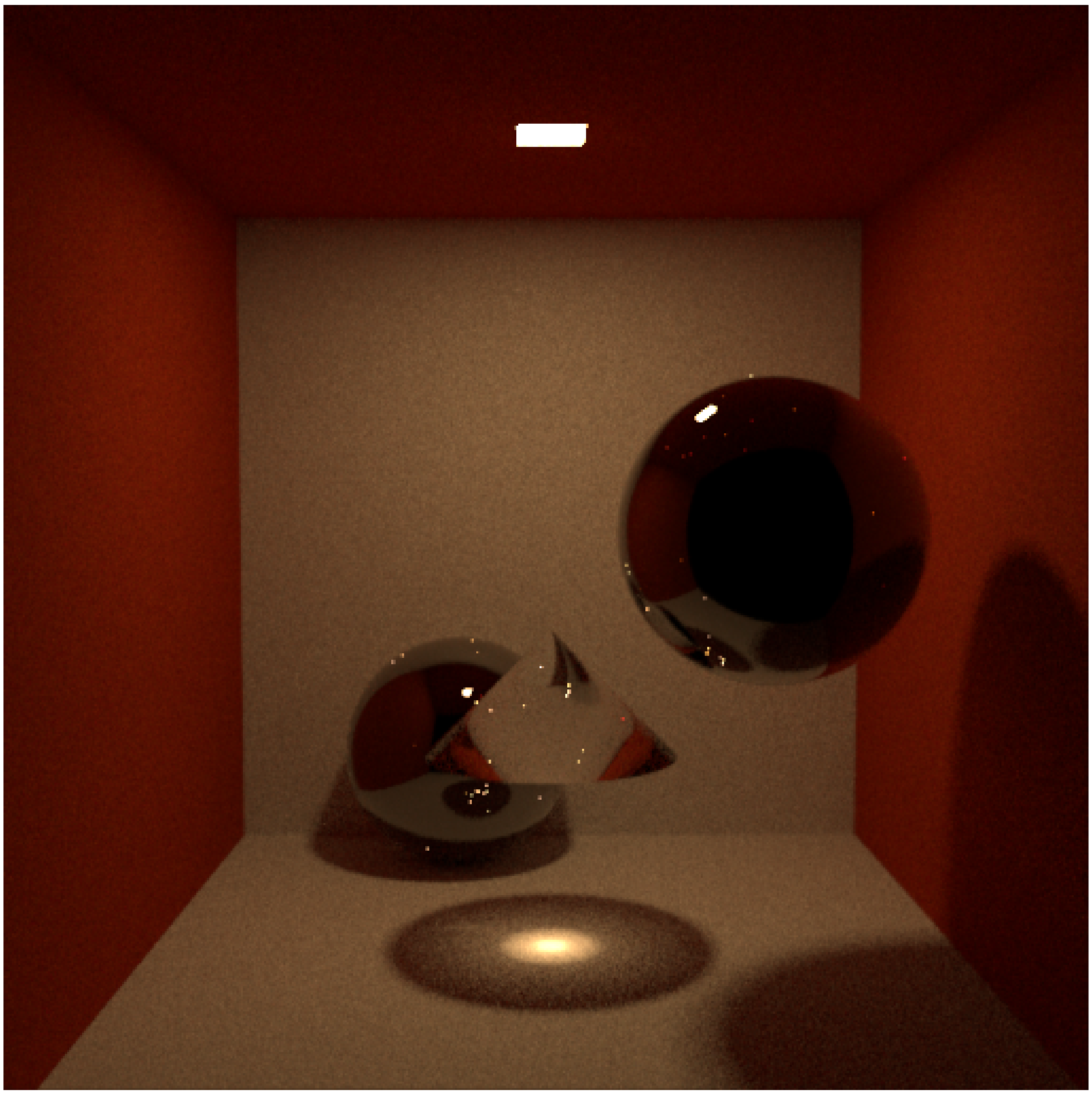


Figure 4.9: Scene with a cone in the middle of the scene, causing a focused caustic on the floor. As expected, the BPT struggles with caustics reflection.

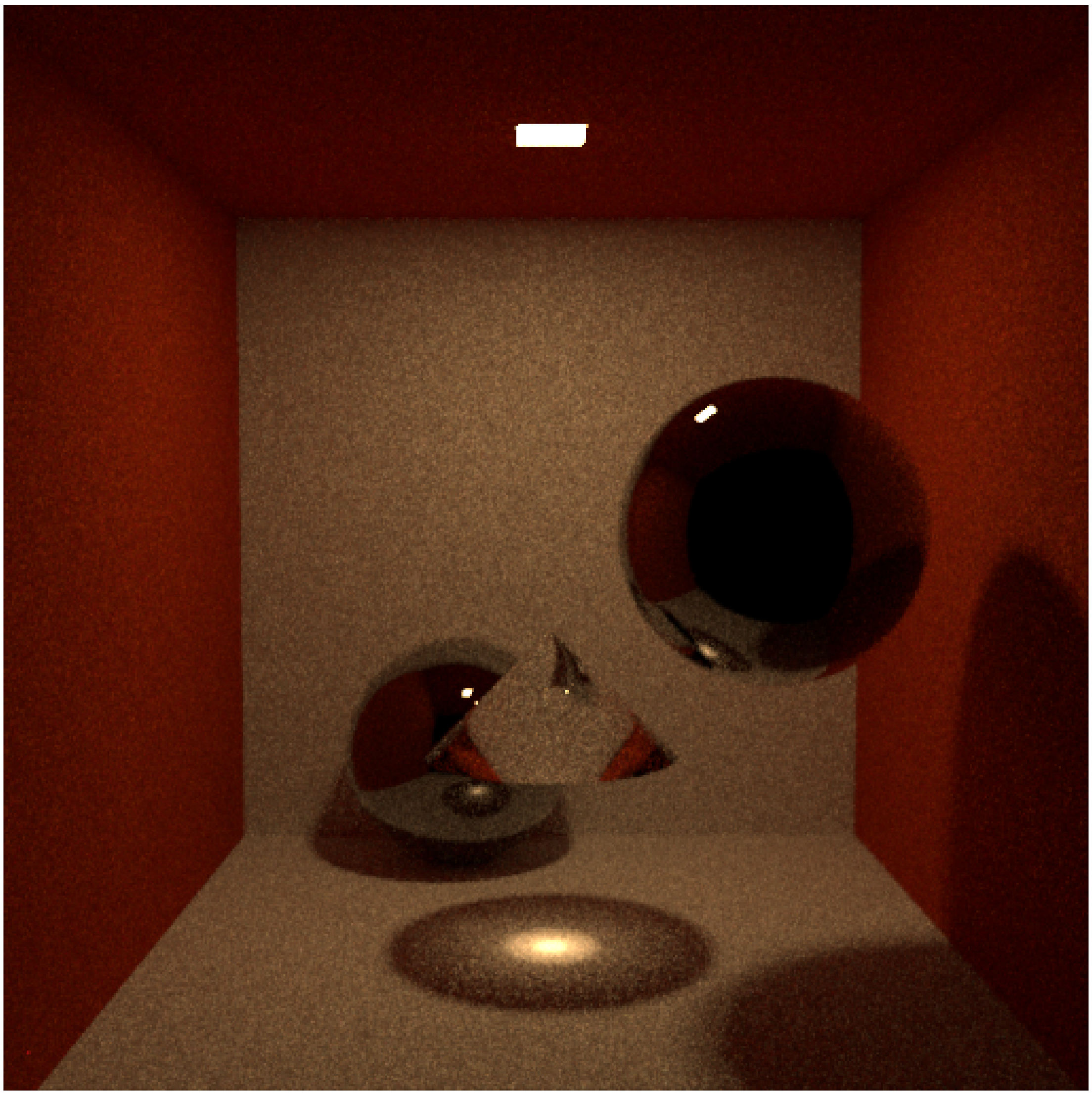


Figure 4.10: Same scene as 4.9, but rendered using VCM, showing that it handles caustics much better.

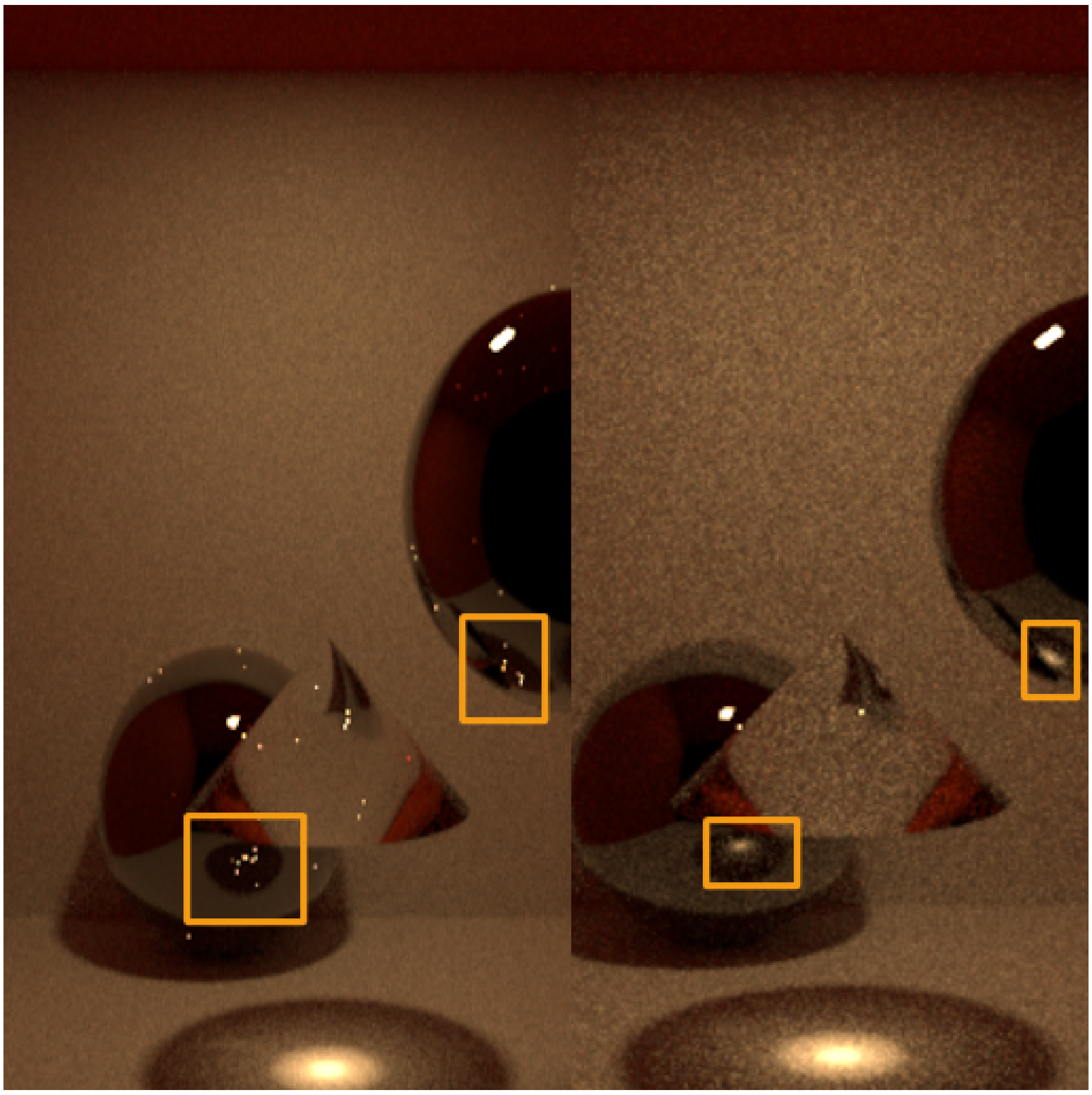


Figure 4.11: Same scene as 4.9, but rendered using VCM, showing that it handles caustics much better.

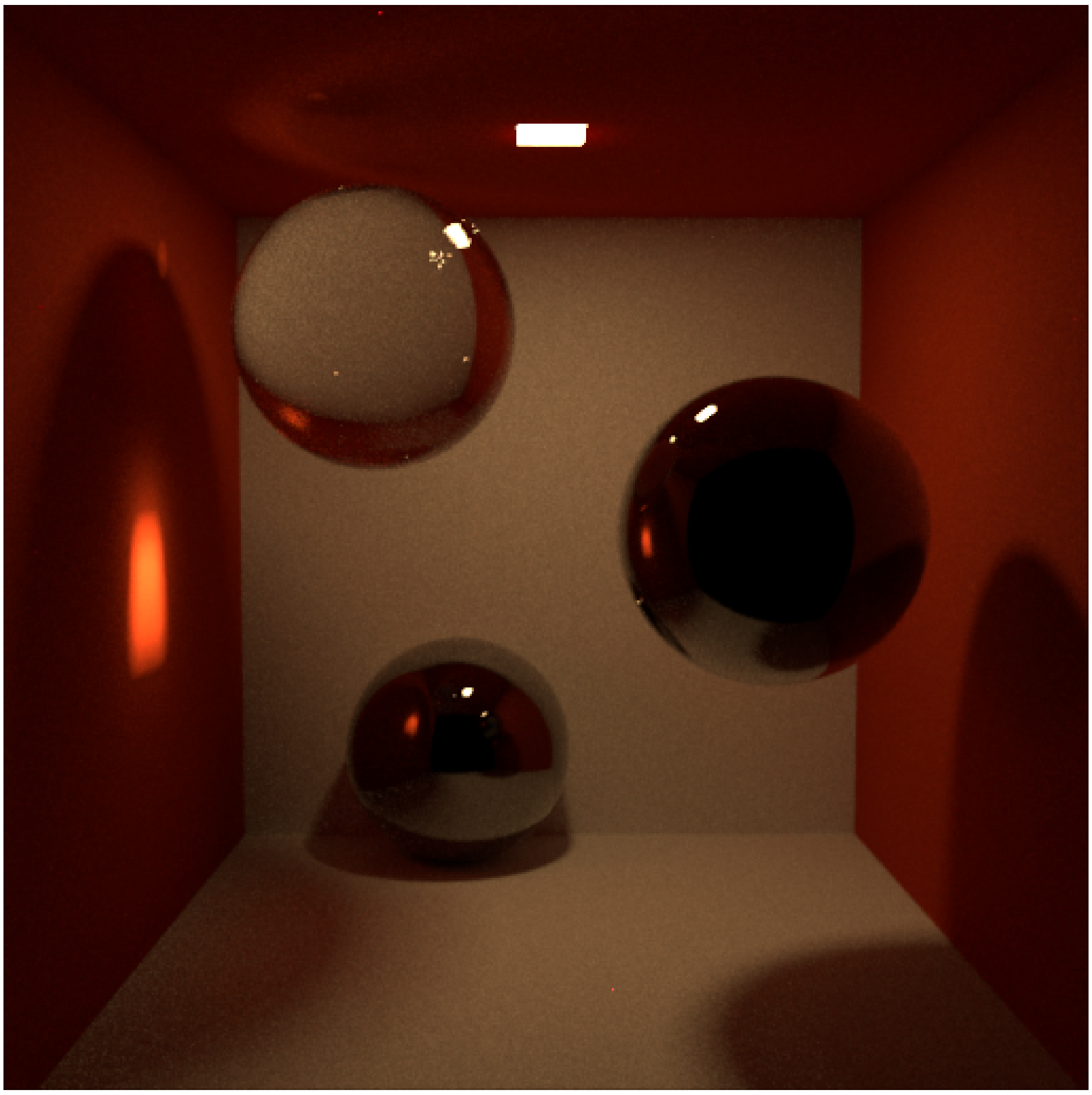


Figure 4.12: Scene with a smaller light source, rendered using VCM. Again, to simulate SDS paths and compare VC and VCM.

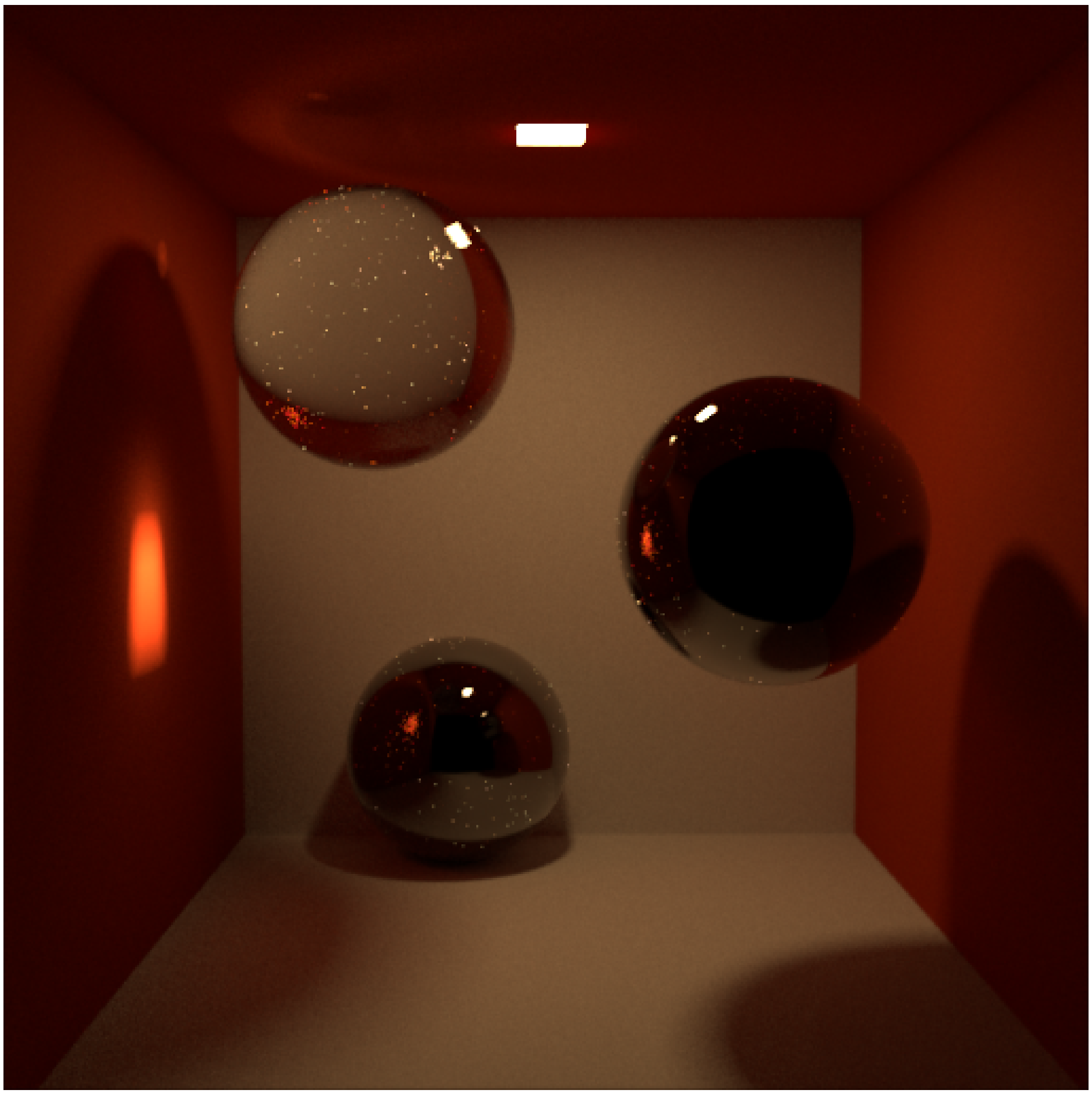
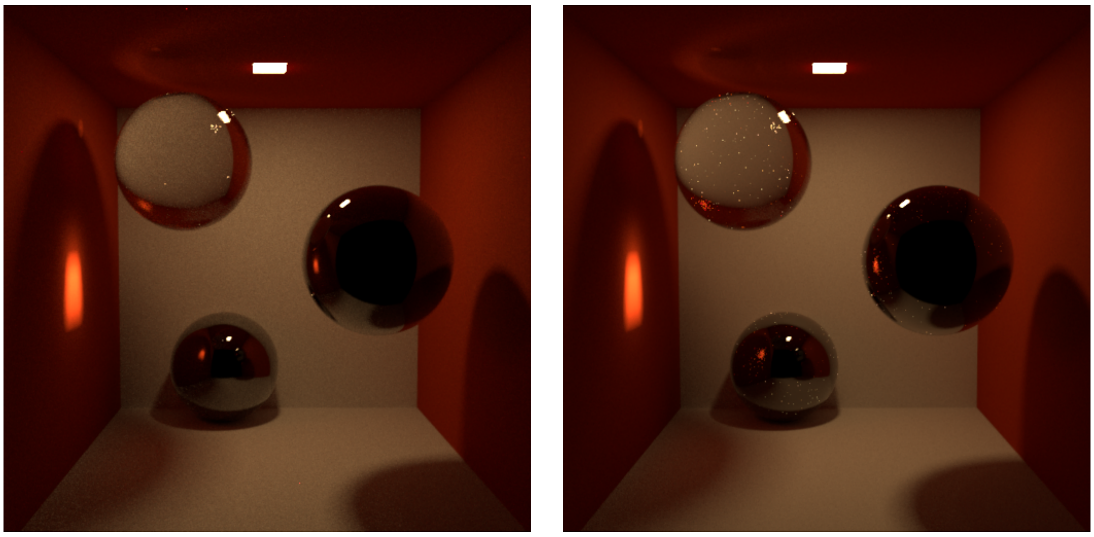
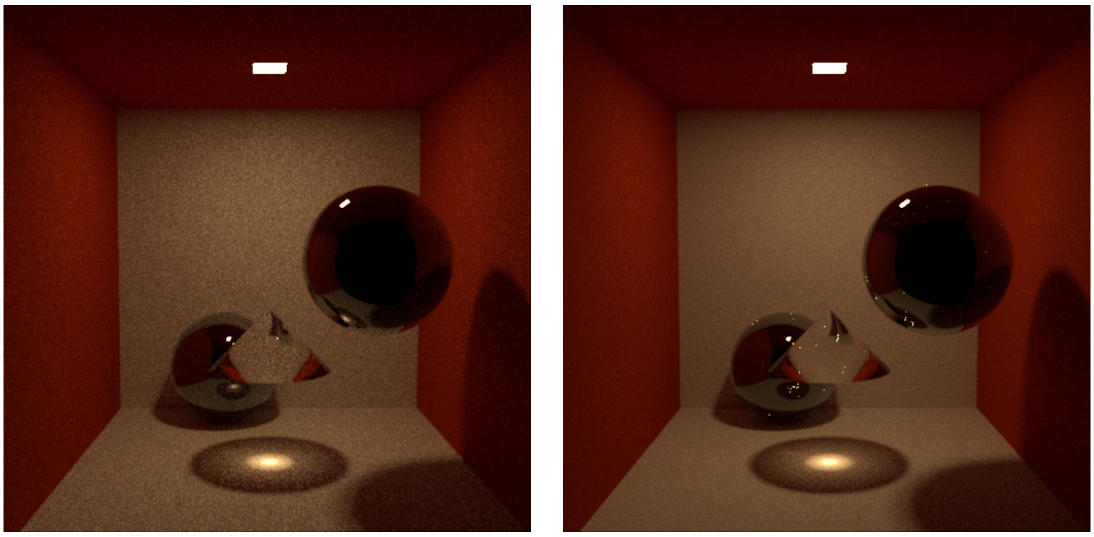


Figure 4.13: Scene with a smaller light source, rendered using VC. Again, to simulate SDS paths and compare VC and VCM.



(a) VCM (b) VC

Figure 4.14: Side by side comparison of Figures 4.13 and 2.1



(a) VCM (b) VC

Figure 4.15: Side by side comparison of Figures 4.9 and 4.10

5. Conclusion

This thesis presents an implementation of Vertex Connection and Merging (VCM) in a spectral renderer, incorporating Hero Wavelength Spectral Sampling (HWSS). This is the first work to successfully integrate these advanced light transport techniques within a spectral framework.

The implementation demonstrates VCM's strength compared to path tracing or BPT and the benefits of combining VCM's robust light transport simulation with the accuracy of spectral rendering and the efficiency of HWSS. This integration allows more physically accurate representations of complex light interactions.

Furthermore, we have made advances in incorporating fluorescence into advanced light transport algorithms. While full integration of fluorescence into VCM remains an unsolved challenge, we successfully implemented fluorescence in light tracing and progressive photon mapping. This lays the groundwork for future research in this area.

Key contributions of this work include:

- The first known implementation of VCM in a spectral renderer
- Successful integration of HWSS with VCM, enhancing rendering efficiency significantly
- Partial incorporation of fluorescence effects in advanced light transport techniques

While we have not achieved all goals we've set, we believe we've made a big enough contribution to the rendering research.

Bibliography

[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 Trans. Graph.*, 31(6):192:1–192:10, November 2012.

[HOJ08] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. 2008.

[Jen96] Henrik Wann Jense. Global illumination using photon maps. 1996.

[JHD20] A. Jung, J. Hanika, and C. Dachsbacher. Spectral mollification for bidirectional fluorescence. 2020.

[MFW18] M. Mojzík, A. Fichet, and A. Wilkie. Handling fluorescence in a uni-directional spectral path tracer. 2018.

[Sma] http://www.smallvcm.com/.

[Vea97] Eric Veach. Robust monte carlo methods for light transport simulation. 1997.

[WND$^+$14] A. Wilkie, S. Nawaz, M. Drozke, A. Weidlich, and J. Hanika. Hero wavelength spectral sampling. 2014.

List of Figures

A. Attachments

A.1 ART source code

Source code of the Advanced Rendering Toolkit (ART) with a functional VCM implementation is provided in the attachment. To compile it, follow the instructions provided in the ART Handbook
`https://cgg.mff.cuni.cz/ART/assets/ART_Handbook.pdf`.

It is recommended to try and compile the code under Arch Linux (or any subsequent distro). Any Linux distribution should be functional, but the installation process might not be as streamlined.

A scene presented in the thesis 4.10 is provided with the source code in **Gallery/CornellBox/CornellBox.arm**.

To do a test render, one can run

```
artist /$art_loc$/Gallery/CornellBox/CornellBox.arm
```

which will save an .exr image in the same location. To open EXR files, we recommend using **TEV**.