Vertex Connection and Merging with Vulkan

Pedro Miguel Silva Rodrigues

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. João António Madeiras Pereira

Examination Committee

Chairperson: Prof. Paolo Romano

Supervisor: Prof. João António Madeiras Pereira

Member of the Committee: Prof. Fernando Pedro Reino da Silva Birra

November 2021

Dedicated to my family who always believed in me and did everything in their power to help and motivate me. Especially to my brother who always has to surpass what I do, I am waiting for you to show how much better your thesis is.

Acknowledgments

I would like to thank Instituto Superior Técnico for the chance to study in a field that I love and for making me grow into a more mature and capable person. I would also like to thank my thesis supervisor in particular, João António Madeiras Pereira, for all the support and feedback, from beginning to end.

I would also like to thank my mother and my brother. These last few months during the lockdown were hard on the three of us, the whining, the screaming, the pouting, but I could not have finished this thesis without you.

I want to give special thanks to my father, who made the personal sacrifice of going to work abroad so my brother and I could pursue the best academic future without having to worry about money. He showed me that sometimes we have to do what is best for all even if we do not like it and not what we want.

To my friends Diogo Oliveira, José Malaquias, João Santos, Alexandre Abreu and André Godinho, the first two who had to put up with me through the bad times, when the project would not run for god knows why, or when that impossible test was just around the corner I am happy you were always there to make me laugh or make fun of how I was behaving when it was needed. To the five of you, I am grateful for the nights out, the afternoons turned to night in Cervetoria, the trips to Belém and for the nights when we stayed in just talking.

Finally, I would like to thank my aunt and uncle, who are like parents to me. I am grateful for always having you there whether I asked or not. And also to my cousins, Jorge and Nando, for the support, for always pushing me to do and be better and for taking me out of the house when I needed it.

Resumo

Algoritmos de rendering baseados no mundo real são capazes de produzir imagens de ambientes virtuais de grande qualidade. Desde que foi introduzido, o Vertex Connection and Merging tem provado ser um algoritmo consistente, robusto e eficiente. O VCM tira proveito de dois algoritmos, BDPT e PM, para gerar imagens usando os seus pontos fortes.

O problema com este tipo de algoritmos é o tempo que demoram a gerar imagens foto-realistas. Uma solução para esse problema pode ser a utilização da unidade de processamento gráfico em vez da unidade central de processamento. Para pôr em prática esta solução, vamos usar a extensão de Ray Tracing do Vulkan, uma API de baixo-nível desenvolvida para tirar proveito das vantagens do suporte em hardware presente nas novas placas gráficas Nvidia RTX.

Esta tese pretende explora a implementação de Vertex Connection and Merging em GLSL e a sua conversão da versão original desenvolvida em CPU para GPU. No entanto, devido a problemas de implementação tivemos de desenvolver e implementar o Modified Vertex Connection and Merging, um algoritmo que mantém os mesmos conceitos que o VCM mas muda a estrutura. Com os resultados alcançados comprovamos que este algoritmo é uma melhor solução que o BDPT quando o objectivo é a qualidade da imagem final.

Palavras-chave: VCM, VUlkan RTX, GPU, Rendering em Tempo Real

Abstract

Physically-based rendering algorithms are capable of producing high-quality images of virtual environments. Since its introduction, Vertex Connection and Merging (VCM) has proven to be a consistent, robust, and efficient algorithm. VCM takes advantage of two algorithms, Bidirectional Path Tracing (BDPT) and Photon Mapping (PM), to generate an image with the strong points of the two referred algorithms.

The problem with these algorithms is the time they take when generating a photo-realistic image. A solution to this problem may be the use of the Graphics Processing Unit (GPU) instead of the Central Processing Unit (CPU). To implement this solution, we will use the Vulkan Ray Tracing extension, a low-level API that was developed to take advantage of the hardware support for ray tracing present in NVIDIA RTX graphic cards.

This thesis intends to explore the implementation of Vertex Connection and Merging in GLSL and its conversion from the original implementation in the CPU to the GPU. However, due to problems that emerged during the implementation we had to develop the Modified Vertex Connection and Merging, an algorithm that preserves the concepts of VCM but changes its structure. The results we got show that Modified VCM is a better solution than BDPT when the goal is to render an image with the better quality possible.

Keywords: VCM, VUlkan RTX, GPU, Real-time Rendering

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Real-time ray tracing is experiencing an extensive and fast growth, with special hardware dedicated to it being developed by many companies in the field, from the manufacturers of game consoles to the companies specialized in GPU such as Nvidia and AMD.

Even with this technology, full real-time ray tracing is still difficult to achieve. The best algorithms used in ray-tracing trace many rays per pixel or many iterations of the same image to get the best result. This process is very demanding and complex, this way requiring a long runtime. So, the use of denoising algorithms to improve the image quality is becoming an industry standard. Using denoising, we can achieve the same or close results with less rays per pixel and less iterations.

With the introduction of the NVIDIA RTX graphic card and the use of denoising algorithms, real-time ray tracing has become a possibility. However, the algorithms developed until now were optimized to run in the CPU, so it is important to port them and optimize them to the GPU and to find the algorithms best suited to real-time ray tracing while still delivering good photo-realistic results.

1.2 Objectives

The main objective of our project is to implement Vertex Connection and Merging and integrate it in the Lift [1] Framework, an educational interactive Stochastic Ray Tracing Framework with AI-Accelerated Denoiser, which uses the Vulkan ray tracing API in order to find out if it can achieve photo-realist results while keeping a real time performance and to compare it to other global illumination algorithms that do not achieve the same quality results but that are less complex. Furthermore, we also want to improve the Framework by implementing a better bidirectional scattering distribution function following the Phong model [2] and another one capable of rendering microfacets [3].

1.3 Thesis Outline

This thesis is divided into six chapters.

After Chapter 1, the Introduction, Chapter 2 provides an explanation of ray tracing, its objectives and the mathematical formulations used to accomplish it. An overview of the current state of the art in global illumination algorithms, from the first one that started it all to the more complex and recent one.

Chapter 3 provides a more in depth analysis of Vertex Connection and Merging focusing on its mathematical formulation.

Chapter 4 provides an implementation overview of Vertex Connection and of Vertex Merging, their integration in the Lift Framework and of VCM, the resulting algorithm of the combination of VC and VM. It also explains the Bidirectional Scattering Functions implemented.

Chapter 5 offers an analysis of the algorithms implemented, evaluating the quality of the result and their performance.

Chapter 6 covers the conclusion of the thesis, an evaluation of the results obtained and future work that can be done to improve the implementation of VCM and the Lift Framework.

Chapter 2

Background

2.1 Ray Tracing

Photo-realistic rendering is the process of generating an image from a 3D scene description that is indistinguishable from a photograph of the same scene, employing techniques that model the interaction between light and matter using physics principles [4]. The best method to achieve realistic results is ray tracing (RT). Turner Whitted introduced Ray tracing in 1980 [5], and he used the behaviour of light in the real world as inspiration. Where a light source emits photons, and when they hit an object, they can be reflected or refracted and may be absorbed by someone's eyes. However, in Ray Tracing, instead of having a light source as origin, rays originate from the camera, because the probability of a ray that originated on a light source hit the camera is very low. Therefore, it is more efficient to start from the camera. Rays are cast from the camera to the scene, and when they hit an object, they can be reflected or refracted, originating more rays, and using next-even estimation, a shadow ray is cast in the direction of the light source, adding its contribution.

2.2 The Light Transport Equation

The light transport equation or the rendering equation, proposed by Kajiya [6], is a mathematical formula that describes the distribution of light in an equilibrium state. It determines the radiance leaving a point by calculating the sum of the light emitted in direction ω_o at point x, plus the radiance incident from all directions scattered in direction ω_o.

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

Where:

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

2.3 Monte Carlo integration

Monte Carlo integration is used to help evaluate the light transport equation. Solving the integral of the light transport equation is usually impossible due to the high dimension and frequent discontinuities. However, Monte Carlo integration solves this by using random sampling to estimate the values of integrals. This is done by independently sampling random points according to some probability density function (pdf) and then computing the estimate, turning the integral into a discrete sum.

$$F_n = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)} \tag{2.2}$$

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

The major advantages of Monte Carlo integration are that, in order to estimate the value of $\int f(x)dx$, it only needs to be able to evaluate it at an arbitrary point in the domain, making it easy to implement. Additionally, it converges at a rate of $O(N^{-1/2})$ in any dimension, regardless of the continuity of the integral [7].

2.4 Real-time Rendering

Real-time rendering focuses on rapidly making images on the computer. As an image appears on the screen, the viewer acts or reacts, and this feedback affects what is generated next [8]. This cycle of rendering and reacting needs to happen at a fast enough rate that the viewer does not see still images.

Although rasterization is still the most common and efficient way to achieve real-time rendering, ray tracing achieves the results with more realism. Ray tracing light transport methods, path tracing or derived methods such as bidirectional path tracing or vertex connection and merging are usually favoured when implementing a real-time ray tracing renderer due to their scalability and algorithmic complexity [9].

These methods work by solving the light transport equation [6], using Monte Carlo integration, so the more rays they send during the render process, the better the final result, but in real-time rendering there is a limited, finite time to render each image, which will result in noisy images, so a denoiser is usually helpful to improve the final result while keeping it fast and interactive.

Ray tracing has been incorporated in a hybrid solution, where both rasterization and ray tracing are used in games like Metro Exodus, Battlefield V or Shadow of the Tomb Raider. But most remarkably, Quake 2 [10] has become the first game to be fully path traced in real-time using a denoiser. Quake 2 was developed by NVIDIA using Vulkan, and first it had to bring ray tracing support to the Vulkan API, to showcase the performance of the RTX graphic card family.

2.5 Path Tracing

When Kajiya [6] proposed the rendering equation, he also proposed the Path Tracing method to try to solve it. The Path Tracing algorithm is an extension of Ray Tracing, so like Ray Tracing, it is an unbiased method. In this method, rays are traced originating from the camera to the scene: every time a ray intersects an object, the bidirectional scattering distribution function (BSDF) is taken into account. New rays are generated in a random direction, and trace the scene generating new rays if an object is intersected until they find a light source.

2.6 Light Tracing

Light tracing [11] preserves the same ideas as Path Tracing, with the big difference that the rays, instead of having the camera as their origin, have the light sources. Rays originate at the light sources, are traced to the scene, and when they intersect an object, a new ray is traced to the camera. The colour contribution is added to the pixel corresponding to the intersection between the ray and the image plane.

Light tracing has some drawbacks considering that some pixels on the image plane may never be hit at all. Nonetheless, it is a very efficient method to find caustics and indirect illumination, which is difficult or impossible to find using Path Tracing due to the probability of sampling paths that contribute to these effects being proportional to their impact on the final image.

2.7 Multiple Importance Sampling

Multiple Importance Sampling (MIS) was introduced by Veach in 1995 [12] and is crucial for Monte Carlo methods. It allows combining samples from different techniques while minimizing the variance. To achieve this, it calculates the weight of each technique used by taking into account the probability density function (pdf) values of each technique. The estimator used:

$$I = \sum_{i=1}^{n} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(X_{i,j}) \frac{f(X_{i,j})}{p_i(X_{i,j})} \tag{2.3}$$

Where n is the number of sampling techniques, n_i the number of samples from technique i and w_i the weighting function. The weights have to sum up to one, the weighting function is usually the power heuristic:

$$w_i(X_{i,j}) = \frac{n_i^\beta p_i^\beta(X_{i,j})}{\sum_{t=1}^{n} n_t^\beta p_t^\beta(X_{i,j})} \tag{2.4}$$

Where n_t is the number of samples from technique t, and p_t the probability density function of technique t. If $\beta = 1$ then, it is the balance heuristic that works well in practice and is easy to implement.

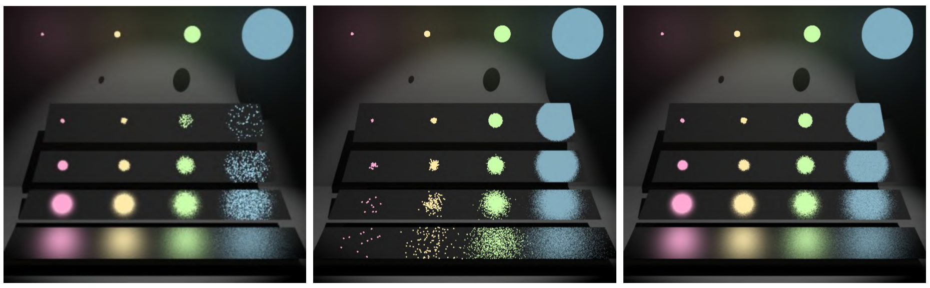


Figure 2.1: Sampling the light source, Sampling the BRDF and Sampling the MIS (Source:Veach 1995 [12])

Figure 2.1 depicts the results that MIS is capable of achieving. By combining the techniques used in the first two images and taking into account their pdf values, it produces an image that retains the best features of each technique, resulting in a final output with higher quality.

In 2019, Grittman introduced the Variance-aware Multiple Importance Sampling [13]. This new technique has three significant improvements: The weight of the techniques with low variance is increased; The weighting functions are close to the probably good balance heuristic when all techniques have high variance; They yield the optimal constant weight when combining techniques with equal effective densities, but different variances. To achieve these improvements, the variance of each technique is used in the weighting function, where v_t is the variance of the technique t:

$$w_i(X_{i,j}) = \frac{v_i^\beta n_i^\beta p_i^\beta(X_{i,j})}{\sum_{t=1}^{n} v_t^\beta n_t^\beta p_t^\beta(X_{i,j})} \tag{2.5}$$

This variance value is calculated by first pre-computing the MIS weight values of each technique for all $X_{i,j}$ points in many iterations. Then it goes through over the calculated weights and calculates the variance values.

2.8 Bidirectional Path Tracing

Bidirectional Path Tracing [14][15] (BDPT) was developed to try and take advantage of the benefits of Path Tracing and Light Tracing without any of the drawbacks. Just as in these two methods, paths are traced originating from both the camera and the light sources, forming the camera path and the light path, respectively. Both paths are initiated simultaneously. At each iteration, the camera path vertices are connected to the vertices of the light path. Then using Multiple Importance Sampling (MIS) [12] each path contribution is added to determine the resulting colour.

Even though BDPT has the advantages of both Path Tracing and Light Tracing, there are still paths that present difficulties. More precisely, specular-diffuse-specular (SDS) paths are very challenging to BPT and other methods because they are hard to find but have a high impact on the final image.

2.9 Photon Mapping

Photon Mapping [16] is a biased two-pass method that, contrasting to BDPT, is very effective at capturing SDS paths. In the first pass, photons are shot originating from the light sources and stored on a photon map when they intersect an object. The second pass is identical to Path Tracing. Paths are traced starting at the camera. Every time it intersects an object, the photon map is accessed to compute an estimation of photon density around the intersection point. From this estimation a bias is introduced.

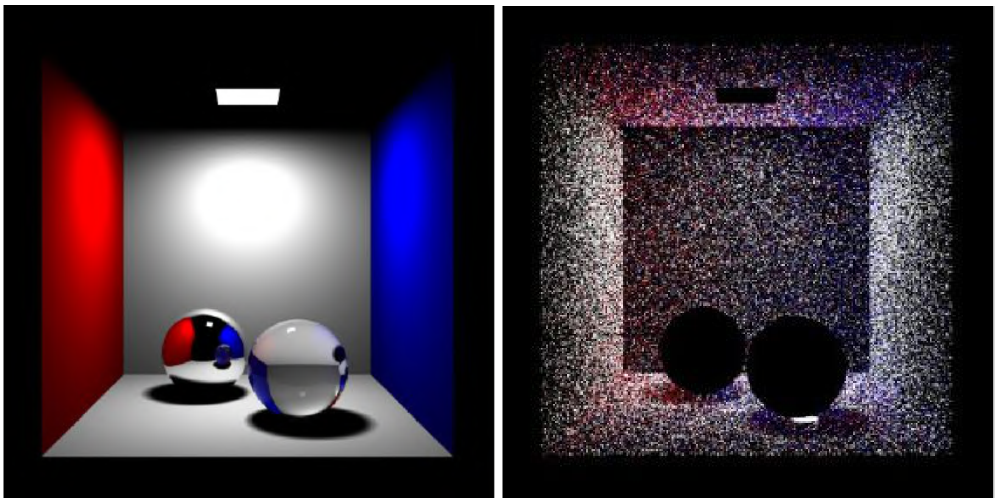


Figure 2.2: Cornell Box with glass and chrome spheres. A ray traced image (left). The photons in the corresponding photon map (right). (Source: H. W. Jensen 1999 [17])

Progressive Photon Mapping, introduced by Hachisuka et al. [18], decreases the bias introduced by reducing the search radius over time. In the first pass of PPM, contrary to PM, the paths originate from the camera. The second pass is an iterative process, where the search radius is reduced at each iteration which shoots photons to the scene originating from the light sources, and whenever they intersect an object, it checks if there is a camera vertex nearby and adds its contribution to the illumination value. As a result, the photon properties do not need to be stored in memory, thus reducing the amount of memory used.

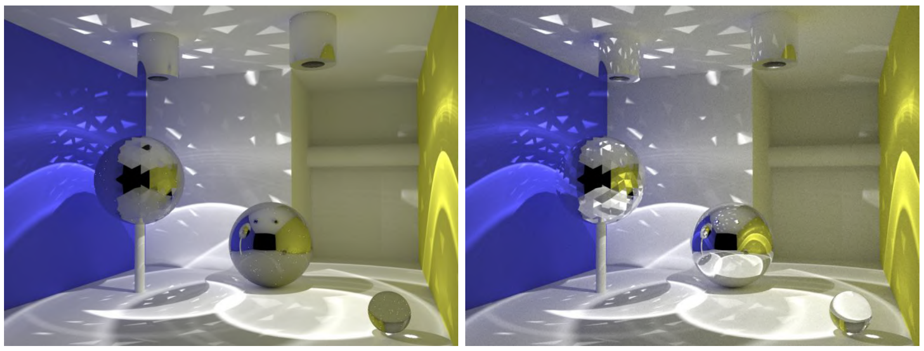


Figure 2.3: A comparison between BDPT (left), and PPM(right). (Source:Georgiev 2012 [19])

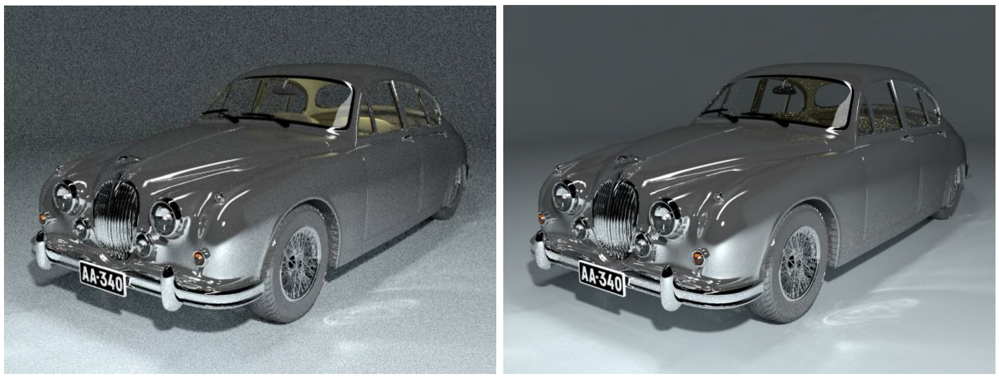


Figure 2.4: A comparison between BDPT (left), and PPM(right). (Source:Georgiev 2012 [19])

Bidirectional Path Tracing and Photon Mapping are two very distinct two-pass algorithms that use light tracing to achieve better final results in different ways. While BDPT improves the amount of indirect light gathered, easily finding paths that carry light, PM is very efficient at rendering caustics and reflected caustics caused by SDS paths. From the figures 2.3 and 2.4, we clearly see the differences of both algorithms. While in Figure 2.3 we can see the caustics reflected in the mirror spheres, in Figure 2.4 (right) we can see the noise in the background of the PM image and this is because PM is inefficient at rendering diffuse lighting.

2.10 Metropolis Light Transport

Metropolis Light Transport [20] is a method of the Markov chain Monte Carlo (MCMC) family, proposed to improve Path Tracing methods by mutating already found paths that carry light. Each new mutated path depends only on the previous path, forming a Markov chain. The mutation can be applied by adding, deleting, or replacing a small set of vertices on the current path. The resulting path still needs to be accepted, so, to improve the efficiency of this method, the mutation strategies need to be carefully designed to guarantee that it not only generates valid paths but that these paths are also significantly different, thus making the implementation of Metropolis Light Transport quite challenging.

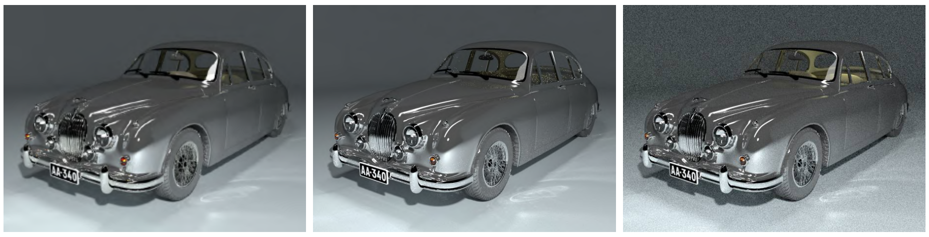


Figure 2.5: MLT (left), BPT (middle) and PPM(right) outputs of the same scene.(Source:Georgiev 2012 [19])

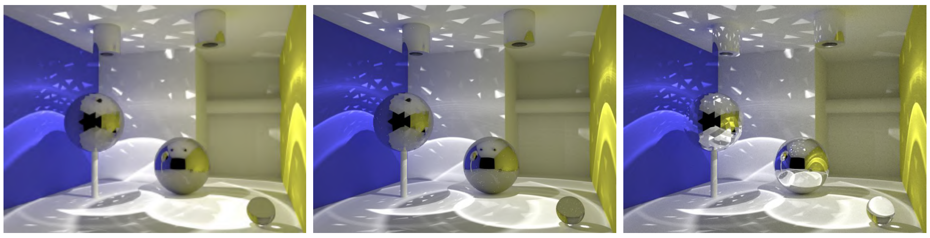


Figure 2.6: MLT (left), BPT (middle) and PPM(right) outputs of the same scene.(Source:Georgiev 2012 [19])

Figure 2.5 clearly shows that MLT finds more and better light paths than BDPT. However, sometimes it can get stuck in an area leading to brighter zones, as we can see in the front window of the car.

Figure 2.6 represents the difficulties MLT has in producing reflected caustics, as it is not able to find SDS paths, and it can overexplore bright zones leading to very bright caustics, as we can see on the floor in the picture on the left.

2.11 Multiplexed Metropolis Light Transport

Multiplexed Metropolis Light Transport [21] (MMLT) has as base a new mutation strategy, where the state itself is a set of random numbers that are used in the generation of a path. This has the two following advantages; Any multi-dimensional random number generates a valid path (similar numbers generate similar paths); The last mutations can easily be made symmetric, thus facilitating the computation of the acceptance probability. However, MMLT, instead of connecting all the vertices, it only connects the endpoints of the eye path and the light path. Additionally, each path sampling has its own MIS weight w. This way, MMLT keeps exploring the path sampling with the greater weight and, if one of the endpoints cannot be connected, the weight is actualized to zero.

The image on the right is a clear evidence that MMLT still has problems producing reflected caustics, not being able to find SDS paths and it can overexplore bright zones leading to the bright caustics on the floor.

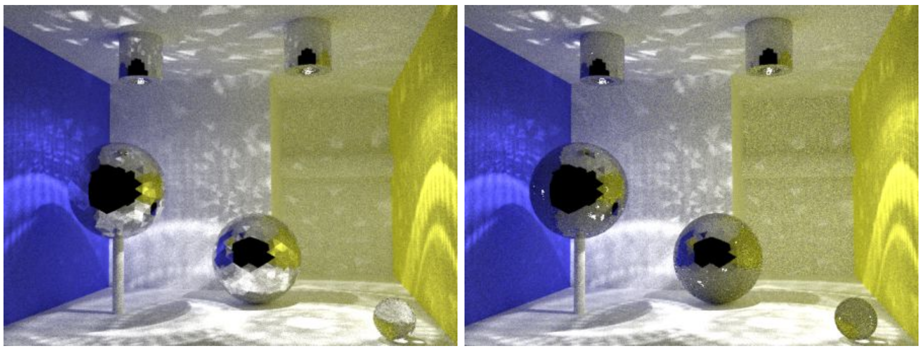


Figure 2.7: VCM (left) and MMLT(right). As is clear in the image MMLT still has problems producing reflected caustics and it over explores bright zones, leading to the bright caustics on the floor. (Source:Hachisuka 2016 [22])

2.12 Vertex Connection and Merging

Vertex Connection and Merging [19] (VCM) is a two-pass algorithm developed to try and take advantage of the benefits of photon mapping and bidirectional path tracing, keeping the high asymptotic performance from BDPT for most light path types while having PM efficiency to reproduce specular-diffuse-specular lightning effects. In the first stage, it traces the light paths, connects them to the camera, and stores the light vertices in a hash grid. In the second stage, it performs three techniques and, using multiple importance sampling, adds their contributions to the pixel resulting colour. It starts by tracing a camera path to each pixel and then, for each node in the camera path it connects them to the light source, it connects them to the light vertices processed in the first stage and it merges them with the light vertices within a predefined merge radius. VCM can be implemented in an iterative manner where for every iteration the merge radius is reduced, decreasing the bias introduced in the final image.

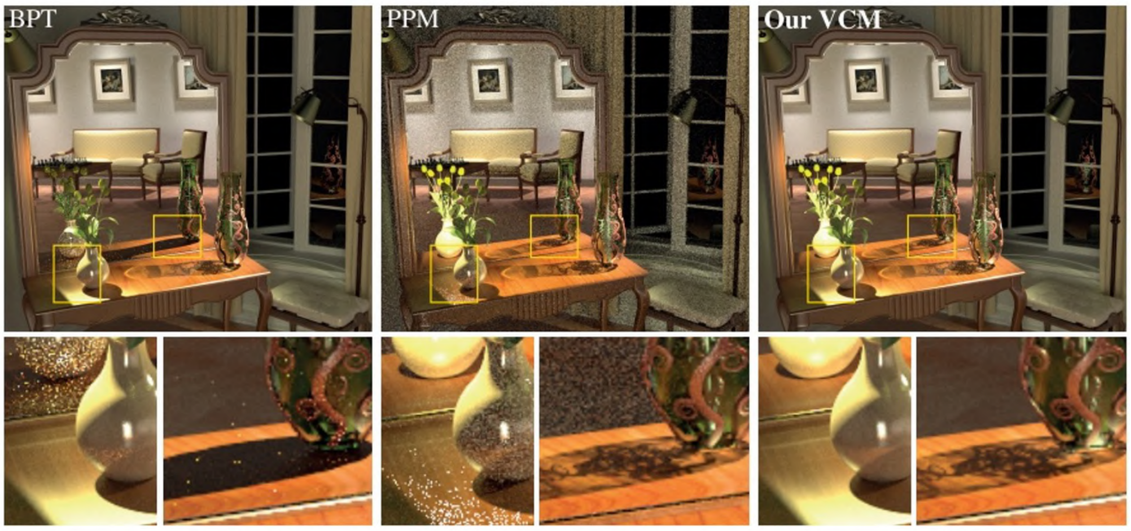


Figure 2.8: Comparison between VCM, BDPT and PM results of the same scene. (Source:Georgiev [19])

As we have discussed, BDPT suffers with specular-diffuse-specular paths and PM with diffuse illumination. The goal of Vertex Connection and Merging is to use BDPT to deal with diffuse illumination and to use PM to find SDS paths. This combination was thought to be impossible, but as we can see in Figure 2.8, VCM successfully combined both techniques. This was only possible thanks to a reformulation of the process to calculate the MIS weight of each path by Gerogiev [23].

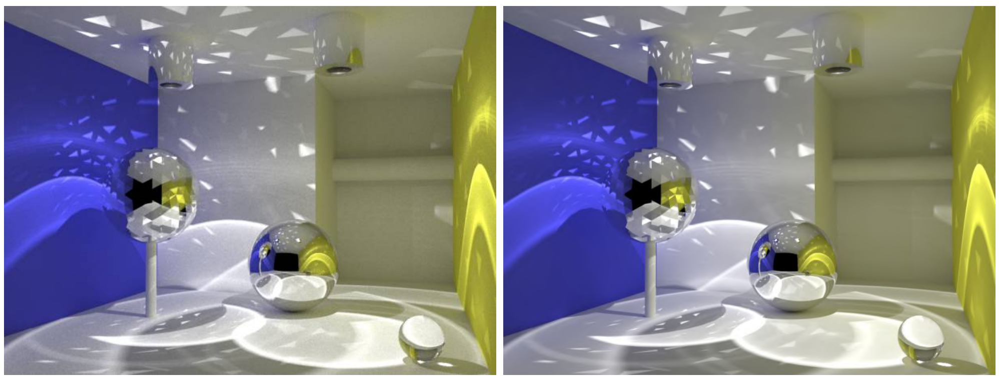


Figure 2.9: VCM (left) and Reference (right). (Source:Georgiev 2012 [19])

2.13 Metropolised Bidirectional Estimators

Metropolised Bidirectional Estimators [22] (MBE) was developed to try to handle two of the most significant issues that hinder efficient light transport: light transport due to multiple highly glossy or specular interactions and scenes with complex visibility between the camera and the light sources. To achieve this, it combines Markov Chain Monte Carlo (MCMC), which handles the first issue, with Vertex Connection and Merging, which handles the second one.

To efficiently combine both techniques, MCMC is only used to generate the light paths, since using it to sample the eye paths compromises good image plane stratification. The specific MCMC sampler used is a Metropolis-Hastings sampler that operates in the primary space. Instead of applying mutations directly on paths, it applies them on a vector of random numbers, which helps to reduce the complexity of the algorithm. To guide the light path generation, there are two Markov Chains with complementary target functions. One of them is path contributions and the other one is a visibility chain. The two chains interact via replica exchange, and the samples produced are combined using MIS.

Although MBE achieves better results than VCM, it is an extremely complex algorithm that does not seem feasible to implement in real-time and it is not clear that the complexity overhead it introduces is compensated by its results. But it is interesting to understand that new global illumination algorithms seem to be a combination of old ones.

Chapter 3

Vertex Connection and Merging

This chapter consists of a more detailed analysis of Vertex Connection and Merging algorithm and how it calculates the MIS weights for each type of path. It will begin with an overview and will finish with the mathematical formulations of some of the paths used in VCM [23].

3.1 Algorithm Overview

VCM consists of two stages. The first stage is responsible for calculating the light nodes and storing them in a range search structure. The second stage is the one responsible for the calculation of the pixel final colour, by constructing camera paths for each pixel and connecting each vertex in the path to the light nodes stored in the range search structure and merging the vertex with all light nodes within range.

Every modern global illumination algorithm has the objective of efficiently solving the formulated light transport equation. Veach[7] formulated light transport over a three dimensional scene as a pure integration problem. The integral is solved by a process called Monte Carlo integration, previously discussed in section 2.3. The final colour of each pixel is expressed as different integration problems. The values estimated by algorithms like PT, LT, BDPT or VCM are all approximations of the real value of the integral, and have some bias (the difference between the estimated value and the real value of the integral). Unbiased algorithms are usually favoured over biased ones, since they provide more accurate results.

Vertex Connection and Merging is the combination of BDPT and PM, so it uses techniques from both algorithms. Like BDPT and PM, its first pass is light tracing. It traces light paths from the light sources to the scene, connects the light nodes to the camera and stores the light nodes in a range structure. The second pass of VCM is where all computations to estimate the pixel final colour are. It starts by shooting a ray from the camera to the scene and every time it intersects an object, it accesses the stored light nodes to calculate the final colour using two different techniques, Vertex Connection and Vertex Merging.

```
VCMRenderer ()
 {
    Light Paths
    lightPaths [numPixels];
    for (int i = 1; i <= numPixels ; i++)
    {
        LightNode lNode = TraceRay(Pixel(i));
        while ( lNode.isValid() ){
            if ( !lNode.Specular ){
                lightPaths.storeLighNode(lNode);
                ConnectToCamera(lNode);
            }
            lNode = SampleScatter();
        }
    }
    BuildRangeStructure(lightPaths);
    Camera Paths
    Color color;
    for (j = 1; j <= numPixels ; j++)
    {
        CameraNode cNode = TraceRay(Pixel(i));
        while ( cNode.isValid() )
        {
            if( intersected with emissive material )
            {
                Vertex connection emissive Lighting.
                color += cNode.color * GetLightRadiance();
                break;
            }
            Ignores purely specular materials
            if( intersection with material not specular )
            {
                Vertex Connection.
                color += ConnectVertices();
                Vertex Merging.
                color += MergeVertices();

            }
            cNode = SampleScatter();
        }
    }
 }
```

Listing 3.1: VCM algorithm.

Vertex Connection (Stage 2 a) in Figure 3.1), connects the camera vertex to all visible light nodes from the chosen path, just like in BDPT. This technique allows VCM to find paths that carry light in a fast and efficient way, easily dealing with diffuse lighting.

Vertex Merging (Stage 2 b) in Figure 3.1), merges the camera vertex with all light nodes within merging range, just like PM. With a high number of light nodes it is important to use a range search structure to speed up this process. This technique is capable of finding SDS paths easily. By calculating the photon density around the camera vertex, it increases the weight of these kind of paths that are very hard to find but that have a big impact on the final result.

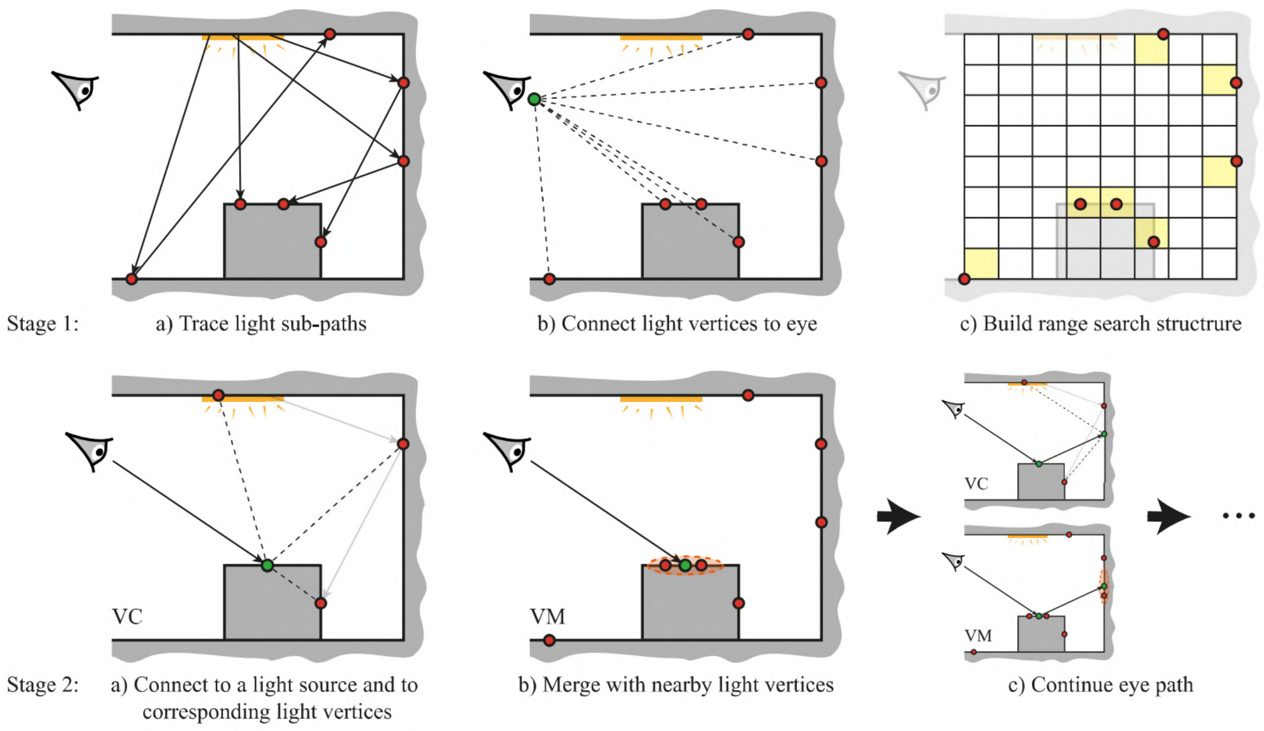

Stage 1:
a) Trace light sub-paths
b) Connect light vertices to eye
c) Build range search structrure
VC
VM
VC
VM
Stage 2:
a) Connect to a light source and to corresponding light vertices
b) Merge with nearby light vertices
c) Continue eye path

Figure 3.1: VCM overview. (Source:Georgiev 2012 [19])

3.2 Mathematical Formulation

VCM estimates the colour of the final pixel by adding the contributions of Vertex Connection C_{VC} and Vertex Merging C_{VM}. The contribution of each technique is equal to its final colour multiplied by its MIS weight, w_v. What made the combination of these techniques possible is the new process developed by Georgiev [23] to calculate the Multiple Importance Sampling weight of VC and VM.

$$I_{VCM} = C_{VC} + C_{VM} \tag{3.1}$$

$$C_{VC} = w_{VC} I_{VC} \tag{3.2}$$

$$C_{VM} = w_{VM} I_{VM} \tag{3.3}$$

The contributions C_{VC} and C_{VM} present in equations (3.1) and (3.2) are obtained over a set of paths represented by s and t. For each path, its colour contribution, $I_{s,t}$, is multiplied by the corresponding MIS path weight $w_{s,t}$. So the final equation for each technique colour contribution is [19, 23]:

$$C_{VC} = \frac{1}{n_{VC}} \sum_{j=1}^{n_{VC}} \sum_{s\geq 0, t\geq 0} w_{VC,s,t}(\bar{x_j}) I_{VC,s,t}(\bar{x_j}) \tag{3.4}$$

$$C_{VM} = \frac{1}{n_{VM}} \sum_{j=1}^{n_{VM}} \sum_{s\geq 0, t\geq 0} w_{VM,s,t}(\bar{x_j}) I_{VM,s,t}(\bar{x_j}) \tag{3.5}$$

- n_{VC}: number of light sub-paths considered to perform Vertex Connection;
- n_{VM}: number of light sub-paths considered to perform Vertex Merging;
- s: Length of a light sub-path;
- t: Length of a camera sub-path;
- $\bar{x}$: light path.

The power heuristic weight w for technique v, s, t has the usual form:

$$w_{v,s,t}(\bar{x}) = \frac{n_v^B p_{v,s,t}^B(\bar{x})}{n_{VC}^B \sum_{s'\geq 0, t'\geq 0} p'^B_{VC,s',t'}(\bar{x}) + n_{VM}^B \sum_{s'\geq 0, t'\geq 0} p'^B_{VM,s',t'}(\bar{x})} \tag{3.6}$$

- v: is the technique being evaluated, VC or VM;
- n_{VC}: number of vertex connections evaluated per camera path;
- n_{VM}: number of light vertices we could merge with per camera path;

Georgiev [23] developed a recursive formulation that simplifies the process to calculate the MIS weight of each technique. By reformulating the process to calculate the weight of each path, in a way that it follows the progression of the path, we only need to keep track of three variables: $dVCM$, dVM and dVC, that will be stored along the path at each node.

With these alterations we are able to simplify the equation for the path weight calculation and, since the MIS heuristic used is the balance heuristic, we know that $B = 1$.

$$w_{v,s} = \frac{1}{w_{v,s}^{light} + w_{v,s}^{camera}} \tag{3.7}$$

Now, we only need to know how to calculate the light sub-paths weight, $w_{v,s}^{light}$, and the camera sub-paths weight, $w_{v,s}^{camera}$. The calculations needed for these depend on the technique being evaluated, Vertex Connection or Vertex Merging.

$$w_{v,s,t} = \frac{1}{\bar{w}_{v,s-1}(\bar{y}) + 1 + \bar{w}_{v,t-1}(\bar{z})} \tag{3.8}$$

Equation (3.8) is a reformulation of equation (3.7) but rewritten taking into account the evolution in a forward manner. This way, instead of iterating backwards from the sub-paths endpoints, we can calculate the MIS weight in the order the sub-paths are being generated. In this reformulation, the weight of the light and camera sub-paths are represented as recursive formulations $\bar{w}_{v,s-1}(\bar{y})$ and $\bar{w}_{v,t-1}(\bar{z})$, respectively. These variables are updated as we trace the sub-paths y and z.

3.2.1 Vertex Connection

For Vertex Connection, these are the equations to calculate the MIS weight for connections at vertex s.

$$w_{vc,s}^{light} = \sum_{j=0}^{s-1} \frac{p_{vc,j}}{p_{vc,s}} + \frac{n_{vm}}{n_{vc}} \sum_{j=2}^{s} \frac{p_{vm,j}}{p_{vc,s}} \tag{3.9}$$

$$w_{vc,s}^{camera} = \sum_{j=s+1}^{k+1} \frac{p_{vc,j}}{p_{vc,s}} + \frac{n_{vm}}{n_{vc}} \sum_{j=2}^{s} \frac{p_{vm,j}}{p_{vc,s}} \tag{3.10}$$

To calculate $w_{vc,s}^{light}$ we iterate over each vertex on the light sub-path and calculate the proportion of the pdf of the connection happening at vertex s, $\sum_{j=0}^{s-1} \frac{p_{vc,j}}{p_{vc,s}}$. Then, we iterate again over each vertex on the light sub-path, this time taking into account where a merge could occur and add the ratio of the pdf of the merge happening at vertex j with the pdf of the merge happening at vertex s, $\frac{n_{vm}}{n_{vc}} \sum_{j=2}^{s} \frac{p_{vm,j}}{p_{vc,s}}$. The process to calculate $w_{vc,s}^{camera}$ is similar. We make the same calculations but for the vertices on the camera sub-path.

Focusing now on the recursive formulation, it is clear that every time a new vertex is added to a sub-path, the MIS weight of that sub-path changes. The weight of the first vertex on a vertex connection sub-path is calculated by equation (3.12) and the weight for each vertex added after the first one is calculated by equation (3.13):

$$n_{vcm} = \frac{n_{vm}}{n_{vc}} \pi r^2 \tag{3.11}$$

$$w_{vc,0} = \frac{\overleftarrow{p_0}}{\overrightarrow{p_0}} \tag{3.12}$$

$$w_{vc,i} = \overleftarrow{p_i}(n_{vcm} + \frac{1}{\overrightarrow{p_i}} + \frac{1}{\overrightarrow{p_i}} w_{vc,i-1}) \tag{3.13}$$

- the forward pdf ($\overrightarrow{p_i}$) is the area pdf for each vertex to connect to its adjacent vertices ;
- the reverse pdf ($\overleftarrow{p_i}$) is the area pdf at each vertex of the path.

Environment and Emitting materials

These calculations happen when no intersection occurs or when a ray hits an emitting material, $s = 0$. These paths are only composed by camera nodes so the MIS light path weight is zero. (3.14):

$$w_{VC,s,t} = \frac{1}{1 + \bar{w}_{VC,t-1}(\bar{z})} \tag{3.14}$$

$$\bar{w}_{VC,t-1}(\bar{z}) = p_{t-1}^{connected} d_{t-1}^{VCM} + p_{t-1}^{trace} \overleftarrow{p}_{w,t-2} d_{t-1}^{VC} \tag{3.15}$$

Direct Illumination

Direct illumination happens when the camera sub-path vertex z_{t-1} is connected to a vertex y_o on a light source, $s = 1$. This type of paths corresponds to next event estimation, a technique where, at each new camera node, a ray is traced to all the light sources to add their contributions. The MIS weight equations are:

$$w_{VC,s,t} = \frac{1}{\bar{w}_{VC,0}(\bar{y}) + 1 + \bar{w}_{VC,t-1}(\bar{z})} \tag{3.16}$$

$$\bar{w}_{VC,0}(\bar{y}) = \frac{\overleftarrow{p}_0}{p_0^{connect}} \tag{3.17}$$

$$\bar{w}_{VC,t-1}(\bar{z}) = \frac{p_0^{trace}(\bar{y})}{p_0^{connect}(\bar{y})} \overleftarrow{p}_{t-1}(n_{VCM} + d_{t-1}^{VCM} + \overleftarrow{p}_{\sigma,t-2} d_{t-1}^{VC}) \tag{3.18}$$

Light tracing

Light tracing happens when the light sub-path vertex y_{s-1} is connected to a vertex z_0 on the camera lens, $t = 1$. This event occurs when the light paths are being created. Therefore, no camera path has been generated yet, so its MIS weight contribution is zero and the MIS weight equations are:

$$w_{VC,s,t} = \frac{1}{\bar{w}_{VC,s-1}(\bar{y}) + 1} \tag{3.19}$$

$$\bar{w}_{VC,s-1}(\bar{y}) = \frac{p_0^{trace}(\bar{z})}{p_0^{connect}(\bar{z})} \frac{\overleftarrow{p}_{s-1}}{n_{light}}(n_{VCM} + d_{s-1}^{VCM} + \overleftarrow{p}_{\sigma,s-2} d_{s-1}^{VC}) \tag{3.20}$$

Where n_{light} is the total number of light sub-paths.

Vertex Connection

Each new vertex in the camera sub-path can be connected with all existing light nodes, leading to a large number of potential paths. These paths are generated by connecting the light vertex y_{s-1} to the eye vertex z_{t-1}, when $s > 1$ and $t > 1$. The MIS weight equations of these paths are:

$$\bar{w}_{VC,s-1}(\bar{y}) = \overleftarrow{p}_{s-1}(n_{VCM} + d_{s-1}^{VCM} + \overleftarrow{p}_{\sigma,s-2} d_{s-1}^{VC}) \tag{3.21}$$

$$\bar{w}_{VC,t-1}(\bar{z}) = \overleftarrow{p}_{t-1}(n_{VCM} + d_{t-1}^{VCM} + \overleftarrow{p}_{\sigma,t-2} d_{t-1}^{VC}) \tag{3.22}$$

3.2.2 Vertex Merging

Vertex Merging operations take place on every node in the light and camera path where $s > 1$ and $t > 1$. The merge radius r is taken into account to evaluate if the photons being merged are within range or if the operation is discarded. The MIS weight equations of these paths are:

$$w_{VM,s,t} = \frac{1}{\bar{w}_{VM,s-1}(\bar{y}) + 1 + \bar{w}_{VM,t-1}(\bar{z})} \tag{3.23}$$

$$\bar{w}_{VM,s-1}(\bar{y}) = \frac{d_{s-1}^{VCM}}{n_{VCM}} + \overleftarrow{p}_{\sigma,s-2} d_{s-1}^{VM} \tag{3.24}$$

$$\bar{w}_{VM,t-1}(\bar{z}) = \frac{d_{t-1}^{VCM}}{n_{VCM}} + \overleftarrow{p}_{\sigma,t-2} d_{t-1}^{VM} \tag{3.25}$$

3.2.3 Sub-Path Vertex Data

The three partial terms d_k^{VCM}, d_k^{VC} and d_k^{VM} were used by Georgiev [23] to calculate the full area pdf of a vertex. These values are stored in each node k of the camera $\bar{z}$ and light $\bar{y}$ sub-paths. Their formulas are the same for all camera and light nodes except when $z = 1$ or $y = 1$, and since we do not store the sub-paths origins, $y0$ and z_0, the values of the data we store at each node are:

First camera node $z = 1$:

$$d_1^{VCM} = \frac{p_0^{connect}}{p_0^{trace}} \frac{n_{light}}{\overrightarrow{p}_1} \tag{3.26}$$

$$d_1^{VC} = 0 \tag{3.27}$$

$$d_1^{VM} = 0 \tag{3.28}$$

First light node $y = 1$:

$$d_1^{VCM} = \frac{p_0^{connect}}{p_0^{trace}} \frac{1}{\overrightarrow{p}_1} \tag{3.29}$$

$$d_1^{VC} = \frac{\overleftarrow{g}_0}{p_0^{trace} \overrightarrow{p}_i} \tag{3.30}$$

$$d_1^{VM} = \frac{\overleftarrow{g}_0}{p_0^{trace} \overrightarrow{p}_i n_{VCM}} \tag{3.31}$$

All nodes where $y > 1$; $z > 1$:

$$d_i^{VCM} = \frac{1}{\overrightarrow{p}_i} \tag{3.32}$$

$$d_i^{VC} = \frac{\overleftarrow{g}_{i-1}}{\overrightarrow{p}_i} (n_{VCM} + d_{i-1}^{VCM} + \overleftarrow{p}_{\sigma,i-2} d_{i-1}^{VC}) \tag{3.33}$$

$$d_i^{VM} = \frac{\overleftarrow{g}_{i-1}}{\overrightarrow{p}_i} (1 + \frac{d_{i-1}^{VCM}}{n_{VCM}} + \overleftarrow{p}_{\sigma,i-2} d_{i-1}^{VM}) \tag{3.34}$$

Chapter 4

Implementation

In this chapter we show the details of the two shading models implemented. We implemented a Bidirectional Scattering Distribution Function (BSDF) using the Phong model [2] and a BSDF using the Cook-Torrance model [3] capable of rendering Microfacet materials. Then, we will show the implementation of our BDPT. We describe the implementation of our BDPT light transport algorithm since the version integrated in Lift framework (Gonçalo 2020 [1]) was incomplete. And finally, we describe implementation of Modified VCM algorithm and its integration in Lift.

4.1 Vulkan

Vulkan is a standard API that provides high-efficiency, cross-platform access to modern GPUs. It is hardware-agnostic, meaning that it does not require any special hardware adaptations nor its extension VKRay. One important feature we used in this thesis is shader programming. In our solution, we used GLSL to implement VC, VM and VCM.

In recent years, Vulkan has taken the place of OpenGL as the industry standard for 3D graphics, improving on its predecessor by providing more control to the programmer. Like OpenGL, Vulkan works around the concept of handles, where VulkanObjects are recognized at the API level as unique IDs passed by the drivers. Vulkan is located at a lower-level than OpenGL, which means it can use pointers to the hardware memory buffers being able to access VRAM from the CPU side [24].

4.1.1 Vulkan Ray-Tracing extension VKRay

Vulkan Ray-Tracing pipeline is different from the traditional rasterization pipeline. The most simple programmable rasterization pipeline contains two programmable shaders, the Vertex shader and Fragment shader. The Vertex shader handles the procesing of individual vertices that are then combined and passed to the Fragment shader where the primitive colour will be calculated. While in the ray-tracing pipeline there are five programmable shaders, see Figure 4.1, every shader on the pipeline must be available for execution at any time, and the one executed is chosen at runtime. The five different types of ray-tracing shaders are: Ray Generation, Intersection, Any Hit, Closest Hit, Miss.

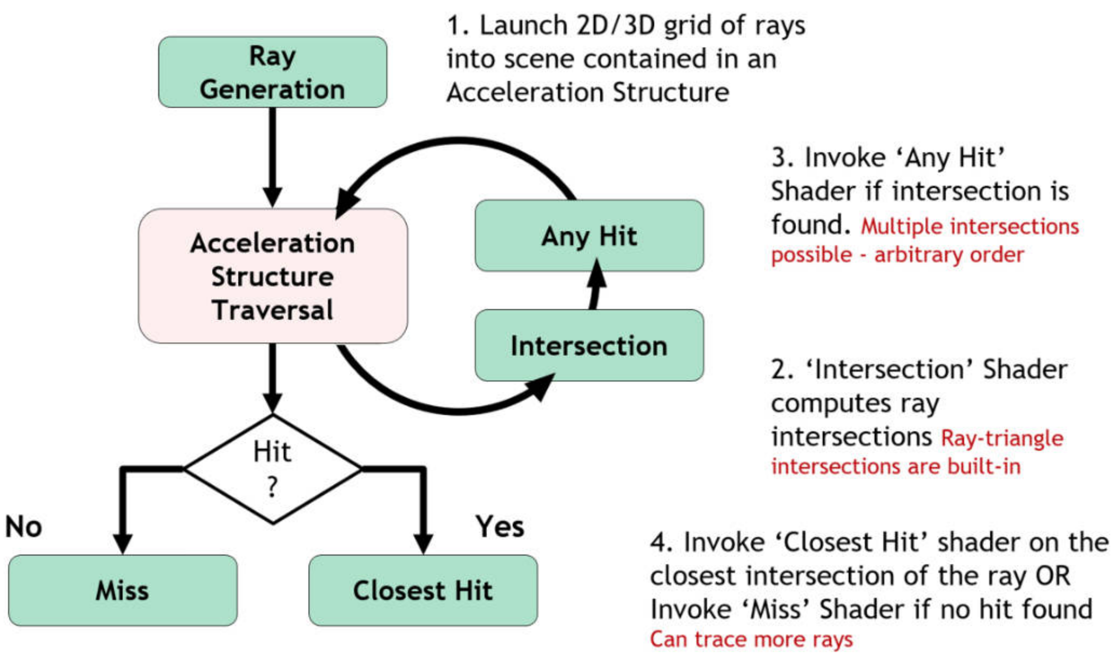

Ray Generation
1. Launch 2D/3D grid of rays into scene contained in an Acceleration Structure
3. Invoke 'Any Hit' Shader if intersection is found. Multiple intersections possible - arbitrary order
Acceleration Structure Traversal
Any Hit
Intersection
2. 'Intersection' Shader computes ray intersections Ray-triangle intersections are built-in
Hit ?
No
Yes
Miss
Closest Hit
4. Invoke 'Closest Hit' shader on the closest intersection of the ray OR Invoke 'Miss' Shader if no hit found Can trace more rays

Figure 4.1: Vulkan Ray tracing Shaders domains and relationships. (Source: Nuno Subtil 2018[25])

Ray Generation shader:

- This shader is the starting point of the ray-tracing pipeline. It runs on a 2D Grid of threads and it launches the ray into the scene and stores the final colour on the image buffer.

Intersection shader:

- This shader is used to detect an intersection between the ray and a primitive that does not have built-in support, like spheres. Triangle primitives are the only ones that have built-in support.

Any Hit shader:

- Hit shaders are responsible for computing the interactions that happen at an intersection point. Any Hit shaders are called on all intersections of a ray with a primitive, in an arbitrary order.

Closest Hit shader:

- Closest Hit shaders are responsible for computing the interactions that happen at an intersection point. Closest Hit shaders are called only on the first intersection between a ray and a scene primitive.

Miss shader:

- Miss shaders are called when there is no intersection along the ray path.

Ray-Tracing requires testing each ray against all scene primitives, a slow and expensive process. So Vulkan uses acceleration structures that store the geometric information of the primitives in the scene necessary to render, spatially sorted from the camera location, in order to reject potential intersections in a fast and straightforward way.

VKRay uses acceleration structures modelled as a two-level structure composed by the bottom level structures and the top level structures, see Figure 4.2. Bottom level acceleration structures contain geometry data of the objects in the scene, while Top level acceleration structures contain a list of references to bottom level nodes. Bottom level nodes are stored in a tree where its root is a Top level node representing an object, and every leaf is a Bottom level node representing the primitives that constitute that object.

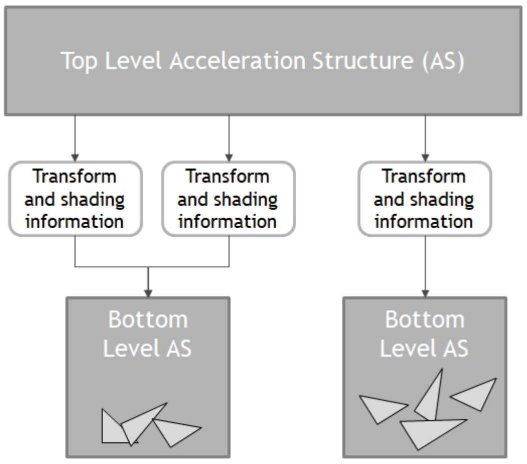

Top Level Acceleration Structure (AS)
Transform and shading information
Transform and shading information
Transform and shading information
Bottom Level AS
Bottom Level AS

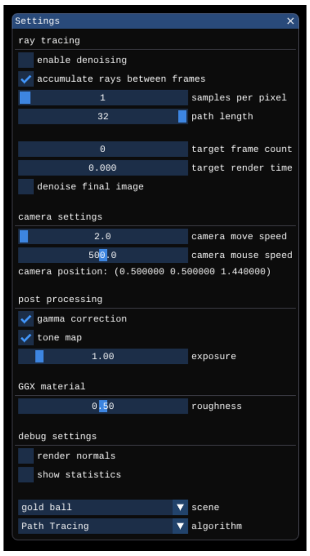
Figure 4.2: Relationship between Top-level and Bottom-level acceleration structures.

4.2 Lift Framework

The implementation of VCM algorithm was integrated in the Lift framework developed by Gonçalo Soares[1], and is available in his GitHub [26]. Lift was designed around a rendering architecture with progressive refinement that allows the choice of one of the two light transport techniques, PT or BDPT, the selection of the input scene and the enabling of a feature to denoise every frame being generated or just the last one [27].

Lift, as we can see in Figure 4.3, gives the user several options: It has the option to activate or deactivate the denoiser and to accumulate rays between frames; It gives the user the option to define a target number of accumulated samples or a desired rendering elapsed time, which can be used to evaluate the different algorithms integrated in the framework. The algorithms integrated are Path Tracing and Bidirectional Path Tracing.

We integrated two more algorithms: Modified VCM and Complete BDPT [28]. To compare these global illumination algorithms, Lift provides a different set of statistics: Frame Size, Frame Rate, Total Frame Duration, Denoiser Duration, Total Samples and Total Elapsed Time.



```
Settings
ray tracing
[ ] enable denoising
[✓] accumulate rays between frames
1        samples per pixel
32       path length

0        target frame count
0.000    target render time
[ ] denoise final image

camera settings
2.0      camera move speed
500.0    camera mouse speed
camera position: (0.500000 0.500000 1.440000)

post processing
[✓] gamma correction
[✓] tone map
1.00     exposure

GGX material
0.50     roughness

debug settings
[ ] render normals
[ ] show statistics

gold ball     ▼ scene
Path Tracing  ▼ algorithm
```

Figure 4.3: Lift Interface.

Lift also lets the user change the camera settings, the camera move speed and the mouse move speed.It has some post-processing features that can be activated or deactivated, Gamma Correction and Tone Map, and lets the user change the exposure level of the scene. We changed the BRDF used in the framework and added Phong shading and GGX materials [29]. To let the users understand the impact of the roughness value in GGX materials, we added the option so that users could change the roughness value of this type of materials. To add value to Lift, we also added more scenes to the list with different characteristics: ”Cornell Box Dragon 2”, the scene shown in Figure 4.3, that shows the impact of diffuse lighting and SDS paths; ”Veach Ajar”, a scene where the only light source is out of the scene and can only be accessed through a semi-closed door, very difficult to render as we will show in Section 5.5.1; ”Gold Ball”, the scene shown in Figure 4.4 that has as main element a goldball rendered using GGX materials.

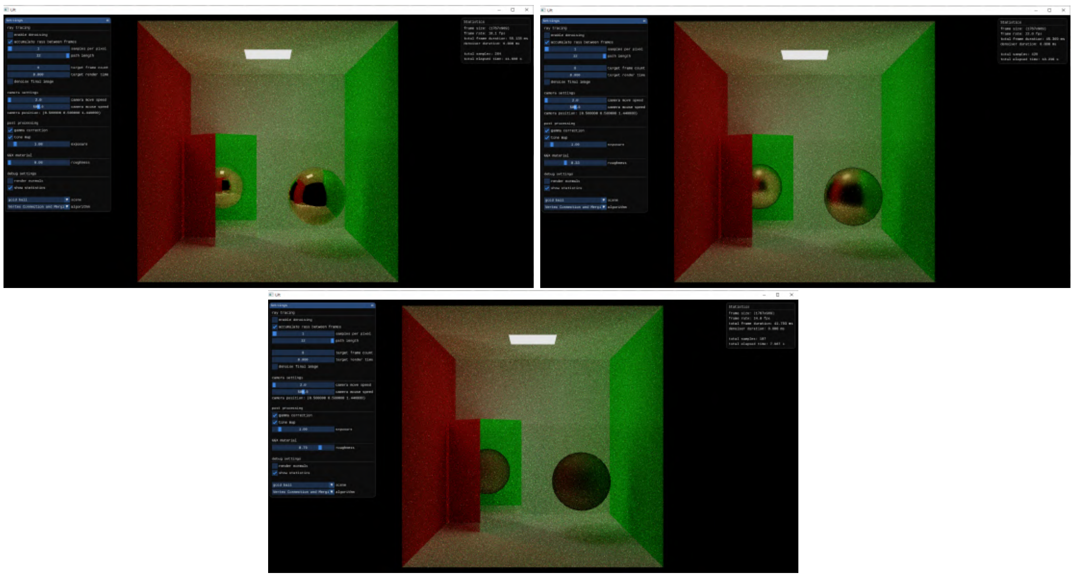


Figure 4.4: Gold ball with GGX material rendered with different roughness values.

4.3 SmallVCM

We used as reference the CPU-based implementation done by the VCM's authors [19], the SmalIVCM. This algorithm was developed in C++ and uses OpenMP to concurrently render many frames all with different merging radius, see Figure **??**. We will use SmallVCM as comparison to our VCM implementation too.

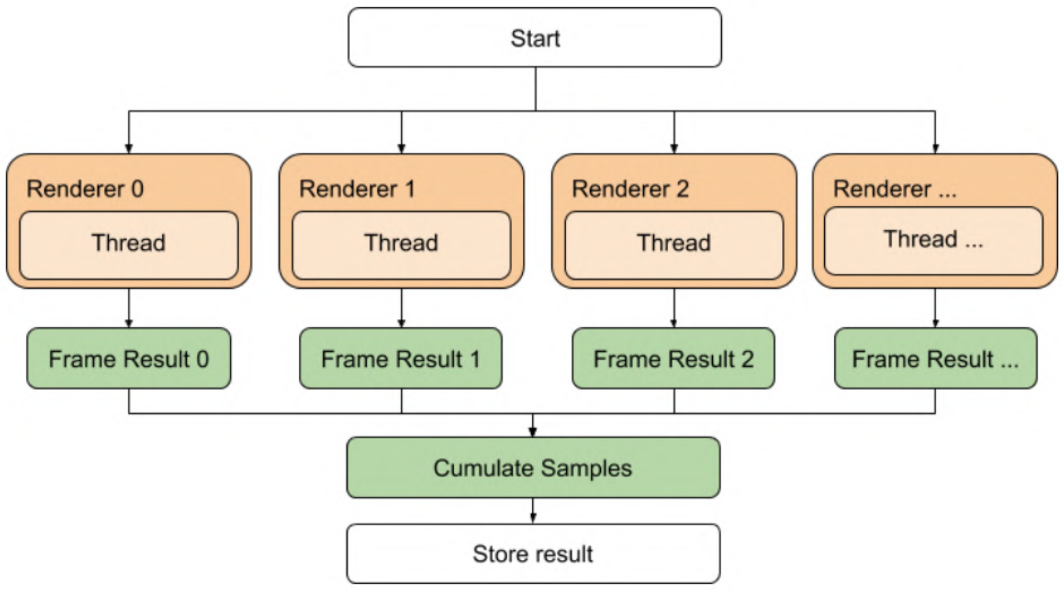


Figure 4.5: SmallVCM parallel execution model.

SmallVCM is a physically based renderer developed by the authors of VCM [19] that implements the Vertex Connection and Merging and Bidirectional Path Tracing algorithms. This renderer is executed exclusively on the CPU and uses OpenMP to have multi threading support. SmallVCM uses a number of renders equal to the number of cores of the CPU, the number of frames to render are distributed equally through the renders and after each iteration the frame is stored in a buffer and combined with the other frames to get the final image.

SmallVCM starts the render process by calculating the photon map. However, Vulkan initiates the ray-tracing rendering by sending a 2D grid of rays from the camera plane to scene, so our shaders must be developed taking in mind the camera path instead of the calculation of the frame. This introduces some complications, namely if and how to reduce the merge radius and how to handle the light path. Since our solution needs to be integrated in Lift framework that has the goal of being able to change the scene being rendered and the algorithm being used without the need to recompile it, we implemented the light path in a non optimized way but that can assure the goal of the framework.

4.4 Shading Models

4.4.1 Phong Shading

"The bidirectional scattering distribution function (BSDF) radiometrically characterizes the scatter of optical radiation from a surface as a function of the angular positions of the incident and scattered beams"[30]. BSDF can be decomposed into two components, BRDF bidirectional reflectance distribution function when referring only to the reflected component and BTDF bidirectional transmittance distribution function, referring to the transmitted component through a material.

There are a lot of BSDF implementations and, in this section, we will explain our implementation by going over each component of BSDF surface interaction.

Sampling Multiple Lobes BSDF

Most of the time, when light intercepts a surface, it produces scattering with multiple different characteristics at the same time. When light intersects a glass like object, it showcases reflection and refraction simultaneously with different weights for each component.

The way light interacts with surfaces can be divided into different components, which are generally called lobes: Diffuse, Specular, Reflection and Refraction.

Modified VCM functions using path tracing and light tracing, meaning that every time a path intercepts an object, it can only create one new path instead of several paths, creating a tree where paths divide themselves at each intersection. To ensure that only one new path is created, it is calculated the probability of each lobe for every material, taking into consideration the contribution of each lobe to the final result.

Getting the Probabilities

To get the probability of each lobe, we calculate the total albedo, the reflection coefficient (the amount of light reflected when intersecting an object). To get the total albedo, we calculate the luminance of the diffuse and specular components using the standard conversion to the Y component of the CIE XYZ colour space:

$$luminance(\vec{c}) = 0.212671C_R + 0.715160C_G + 0.072169C_B \tag{4.1}$$

To calculate the albedo of reflection and refraction components, we use the Fresnel equations. The Fresnel equations describe the way light behaves when moving between different media. There are two major types of materials that present different behaviours, Conductors and Dielectrics. Both are evaluated by the same set of Fresnel equations, but the dielectrics benefit from indices of refraction that, when are guaranteed to be real-valued, lead to simpler equations. We use the Fresnel Dielectric equation to calculate the index of refraction and then multiply it by the luminance of the mirror reflectance to get the reflection albedo.

$$albedo(refraction) = 1 - indexofrefraction \tag{4.2}$$

Finally, we calculate each component probability by dividing each albedo by the total albedo:

$$P(component_D) = \frac{albedo(component_D)}{\sum_i^N albedo(component_i)} \tag{4.3}$$

Lobes and Shading Models

The Phong BSDF is composed by a Lambertian term for diffuse reflection plus the Phong specular term for glossy reflection:

$$f_r(\omega_i, \omega_o) = \frac{\rho_D}{\pi} + \rho_S \frac{\alpha + 2}{2\pi} (\omega_o \cdot R)^\alpha \tag{4.4}$$

To sample the diffuse term the importance distribution used is a cosine-weighted distribution, while the one used to sample the specular term is derived from its expression. It does not use the same distribution because of the spiked shape of the glossy term.

Diffuse

When sampling the Diffuse lobe we have to calculate the new scattered direction, the probability density function (pdf) value and the BSDF factor. The BSDF factor is calculated using the DiffuseReflectance of the material. The direction ω_i is generated using a cosine-weighted density function on the hemisphere and is then used to calculate the pdf.

$$\omega_i = (cos(2\pi\xi_1)\sqrt{1-\xi_2}, sin(2\pi\xi_1)\sqrt{1-\xi_2}, \sqrt{\xi_2}) \tag{4.5}$$

$$bsdfFactor_D = \frac{DiffuseReflectance}{\pi} \tag{4.6}$$

$$pdf_D(\omega) = \frac{\omega \cdot N}{\pi} \tag{4.7}$$

After calculating these values we still need to add the secular contribution of the material, updating the BDSF factor and the pdf value. These two values are updated taking into consideration the direction previously calculated.

Specular

When the Specular lobe is the one being sampled, the direction ω_i is generated using a power cosine-weighted density function on the hemisphere. The direction is then used to calculate the probability density function value and the BSDF factor.

$$\omega_i = (cos(2\pi\xi_1)\sqrt{1-(\xi_2^{\frac{1}{1+\alpha}})^2}, sin(2\pi\xi_1)\sqrt{1-(\xi_2^{\frac{1}{1+\alpha}})^2}, \xi_2^{\frac{1}{1+\alpha}}) \tag{4.8}$$

$$bsdfFactor_S = \frac{PhongReflectance \times (\alpha + 2)}{2\pi}(\omega \cdot \overrightarrow{Ray})^\alpha \tag{4.9}$$

$$pdf_S(\omega) = \frac{\alpha + 1}{2\pi}(\omega \cdot \overrightarrow{Ray})^\alpha \tag{4.10}$$

As we did when sampling the Diffuse lobe, both lobes need to be evaluated to correctly calculate the BSDF and pdf values. So after calculating the Specular contribution we update these values adding the Diffuse contribution using the previously calculated direction.

Reflection and Refraction

When the Reflection or the Refraction lobe is chosen, the calculus of the pdf and BSDF factor are simpler, more straightforward. The most complex part of the process is generating the direction ω_{Refr} when the Refraction lobe is chosen, using Snell's law.

4.4.2 Cook-Torrance BSDF

Cook-Torrance BSDF [3], just like the BSDF function we have just explained, can be divided into different components. The difference between both BSDF function lies in the Specular component. Cook-Torrance BSDF is capable of rendering microfacet models that describe the roughness of surfaces as a compilation of small microfacets, very small perfect reflectors described by three different functions D, F and G.

$$f_{specular} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} \tag{4.11}$$

D is the distribution function. It statistically describes the amount of microfacets that are aligned with the halfway vector, which is the vector between the surface normal and the light direction. There are many distribution functions that can be used to calculate D and we used the Trowbridge-Reitz GGX [29] distribution function:

$$D(n, h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2} \tag{4.12}$$

Where α is the roughness of the surface and $0 \leq \alpha \leq 1$.

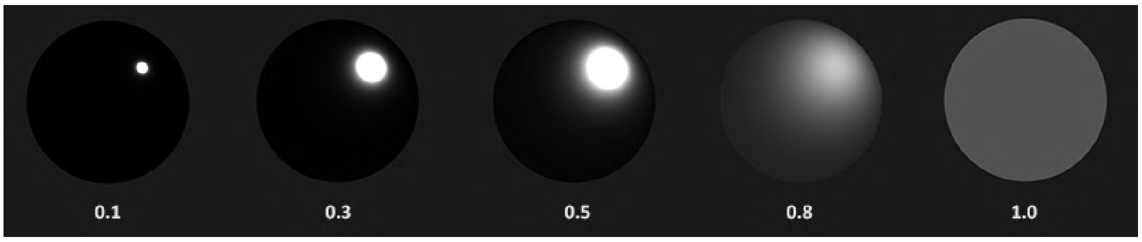


Figure 4.6: GGX Distribution. Roughness values. (Source: Joey de Vries [31])

When the surface is smooth, its roughness value us closer to zero, which means that there is a high number of microfacets concentrated in a small radius pointing to the halfway vector. And as the roughness increases, so does the number of microfacets pointing to the halfway vector, as we can see in Figure 4.6.

The geometry function, G, is used to describe how the microfacets shadow each other, leading to an attenuation of the light. It calculates the probability of a given point in a surface being shadowed by the surrounding microfacets. The geometry function we will use is derived from the distribution function:

$$G_{GGX}(n, w_i, k) = \frac{n \cdot w_i}{(n \cdot w_i)(1-k)+k} \tag{4.13}$$

$$k = \frac{(\alpha+1)^2}{8} \tag{4.14}$$

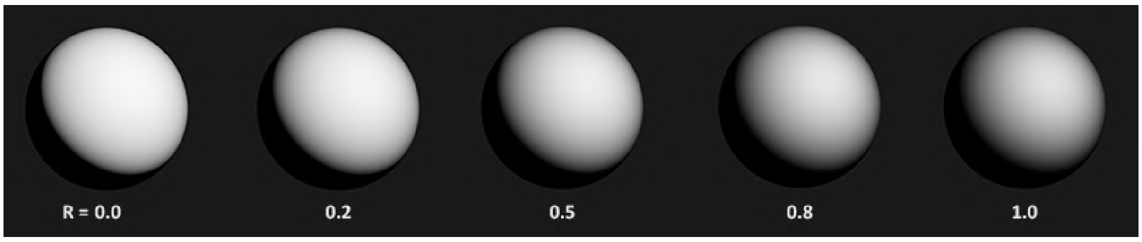


Figure 4.7: GGX Geometry. Roughness values. (Source: Joey de Vries [31])

The geometry function returns a value between [0,1] and as we can see in Figure 4.7, the higher the roughness of the surface, the lower the value returned from the geometry function. This is because the surfaces with higher values of roughness will have more microfacets shadowing each other, leading to darker colours.

The fresnel function, F, simulates the way light interacts with a surface at different angles, calculating the percentage of light that gets reflected. Every material has a base level of reflectivity when looked directly upon. But when looking at it from different angles, its reflections become more apparent. Theoretically all surfaces reflect light when seen from a 90 degree angle. This is called the fresnel effect. However, the fresnel equations are very complex and they differ from conductive to dielectrics materials so we will use an approximation introduced by Shlick [32]:

$$F_{Shlick}(h, w_i, F_0) = F_0 + (1-F_0)(1-(h \cdot w_i)^5) \tag{4.15}$$

Where F_0 represents the base reflectivity, and is calculated using the index of refraction (IOR) of the medias present at the intersection point.

4.5 BDPT

To better evaluate our VCM implementation we also implemented BDPT using the recursive formulation developed by Georgiev [23], having as base the BDPT implementation in SmallVCM. We decided to do a new implementation of BDPT instead of using the one integrated in Lift because that implementation was incomplete, the Multiple Importance Sample weights were not properly calculated, and this way we can also evaluate the results of a complete implementation of BDPT.

4.5.1 Light Tracing

```
void lightTracing () {

    LighPath ray[] = generateRay(Random);
    //initiate sub-path vertex data
    ray[0].dVCM = directPdfA / emissionPdf;
    ray[0].dVC = lightCos / emissionPdf;

    for (int bounce = 0; bounce < lighPath_lenght ; bounce ++) {
        ray[bounce] = traceRay();
        if(ray.hit && !Specular){
            ray[bounce].dVCM /= cosTheta;
            ray[bounce].dVC /= cosTheta;
        }
        ray[bounce+1] = sampleScattering();
        ray[bounce+1].dVCM = 1/bsdfDirPdf;
        ray[bounce+1].dVC = (cosThetaOut / bsdfDirPdf) * (ray[bounce].dVC * bsdfRevPdfW +
     ray[bounce].dVCM + mMisVmWeightFactor);
        ContinuePath(ray);
    }
}
```

Listing 4.1: Light Path

Our implementation of the BDPT algorithm has the same structure as our Modified VCM. During the light tracing loop it calculates the sub-path vertex data of each light node stores them in a vector, see Listing 4.1. Both algorithms have the problem of having a low number of light vertices available when calculating the colour of the pixel. The only light vertices available belong to the single light path built during this stage. This is mitigated by the possibility to accumulate rays between frames provided by the framework Lift.

4.5.2 Camera Tracing

Camera path tracing has the same structure as light tracing, but after calculating the camera vertex data and before continuing on the path, we need to update the pixel colour. To get the pixel colour, we will use the paths described in section 3.2. In our implementation no scene had a background light so the paths described in section 3.2.1 are discarded except when the ray hits a light source.

```
void pathTracing () {

    vec3 colour = Vec3(0);

    CameraPath ray = generateRay(Random);
    //initiate sub-path vertex data
    ray.dVCM = mLightSubPathCount / cameraPdfW;
    ray.dVC = 0;

    for (int bounce = 0; bounce < cameraPath_lenght ; bounce ++) {
        ray[bounce] = traceRay();
        if(ray.hit && !Specular){
            ray[bounce].dVCM /= cosTheta;
            ray[bounce].dVC /= cosTheta;

            if(hit.emissive)
                // Emitting material s = 0
                colour += ray.colour * GetLightRadiance();

            //Direct Illumination, s = 1
            colour += ray.colour * DirectIllumination();

            if(!Specular)
                for (int i = 0; bounce < lighPath_lenght ; bounce ++)
                    colour += ray.colour * lihtPath[i].colour * ConnectVertices(ray,
    lightPath[i]);

        }
        ray[bounce+1] = sampleScattering();
        ray[bounce+1].dVCM = 1/bsdfDirPdf;
        ray[bounce+1].dVC = (cosThetaOut / bsdfDirPdf) * (ray[bounce].dVC * bsdfRevPdfW +
     ray[bounce].dVCM + mMisVmWeightFactor);
        ContinuePath(ray);
    }}
```

Listing 4.2: Camera Path

Our goal implementing this version of BDPT is to evaluate the impact the technique Vertex Merging has in the performance and quality results Modified VCM achieves. Comparing these two algorithms we will be able to conclude if adding VM has a real impact or not in the final image and if its impact in the performance of the algorithm is compensated by the results it gets.

From the images rendered, shown in Figure 4.8, we believe that VM has a significant impact on the final image when rendering caustics and reflected caustics. The impact on the performance, as we will see later, is insignificant which leads us to believe that our Modified VCM is a viable solution when the light paths cannot be constructed beforehand.

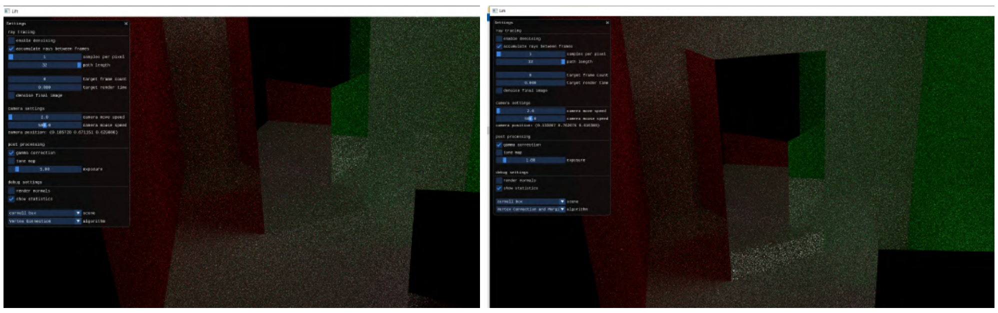


Figure 4.8: Zoom in scene of Complete BDPT and Modified VCM rendered using only 8 frames.

4.6 Our Solution

4.6.1 First Stage

VCM is divided in two stages, like PM. In the first stage photons are traced through the scene, originating from the light sources, and are stored in a range search structure to be used in the second stage. To properly implement VCM we need to execute this stage every time a new scene is loaded, before calling the ray tracing pipeline, since the light sources are static we do not need to run this stage every time we move in the scene.

```
void buildGrid () {

    Hashgrid grid;

    LighPath ray = generateRay(Random);
    //initiate sub-path vertex data
    ray.dVCM = directPdfA / emissionPdf;
    ray.dVC = lightCos / emissionPdf;

    for (int bounce = 0; bounce < lighPath_lenght ; bounce ++) {
        ray[bounce] = traceRay();
        if(ray.hit && !Specular){
            ray[bounce].dVCM /= cosTheta;
            ray[bounce].dVC /= cosTheta;
        }
        ray[bounce+1] = sampleScattering();
        ray[bounce+1].dVCM = 1/bsdfDirPdf;
        ray[bounce+1].dVC = (cosThetaOut / bsdfDirPdf) * (ray[bounce].dVC * bsdfRevPdfW +
     ray[bounce].dVCM + mMisVmWeightFactor);
    }
    if (!Specular)
        grid.insert(ray[bounce]);

    ContinuePath(ray);}
```

Listing 4.3: Hash Grid construction.

To create the Hashgrid we implemented a simple Ray Genaration shader that would create a random light path per pixel in the final image and store it in the range search structure, see listing 4.3. This structure would be stored and sent to the ray tracing pipeline. This shader is used in a pipeline that is executed every time a new scene is loaded, when the camera moves it does not change the state of the light sources, so the light paths do not need to be recalculated. And is executed before the rendering pipeline.

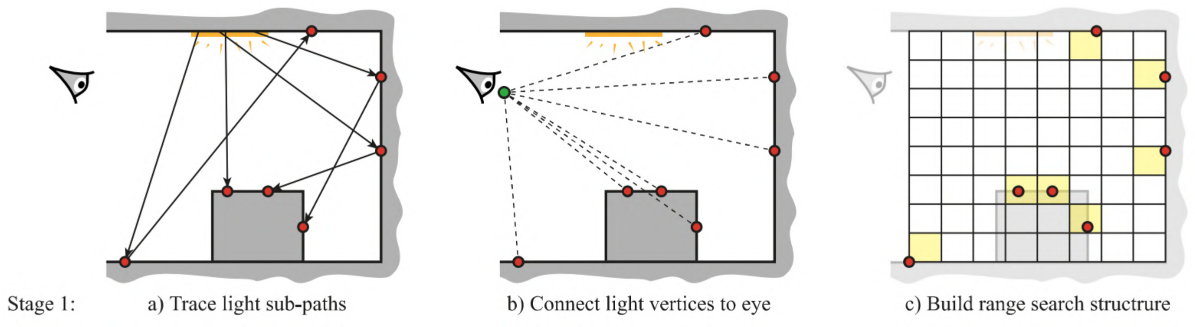

Stage 1:
a) Trace light sub-paths
b) Connect light vertices to eye
c) Build range search structrure

Figure 4.9: VCM First Stage.

4.6.2 Second Stage

The second stage of VCM corresponds to the camera path loop. A camera path is traced for every pixel and the colour is updated at each node using Vertex Connection and Vertex Merging. We implemented a Ray Generation shader to be used in the ray tracing pipeline that uses that HashGrid built in the first stage to do these operations.

```
void cameraPaths () {

    vec3 colour = Vec3(0);

    CameraPath ray[] = generateRay(Random);
    //initiate sub-path vertex data
    ray[0].dVCM = mLightSubPathCount / cameraPdfW;
    ray[0].dVC = 0;
    ray[0].dVM = 0;

    for (int bounce = 0; bounce < cameraPath_lenght ; bounce ++) {
        ray[bounce] = traceRay();
        if(ray.hit && !Specular){
            ray[bounce].dVCM /= cosTheta;
            ray[bounce].dVC /= cosTheta;
            ray[bounce].dVM /= cosTheta;

            if(hit.emissive)
                // Emitting material s = 0
                colour += ray.colour * GetLightRadiance();

            //Direct Illumination, s = 1
            colour += ray.colour * DirectIllumination();

            if(!Specular)
                for (int i = 0; i < hashGrid.size() ; i ++)
                    colour += ray.colour * ConnectVertices(ray, lightPath[i]);
                colour += ray.colour * mVmNormalizationr * Process(ray, lightPath);

        }
        ray[bounce+1] = sampleScattering();
        ray[bounce+1].dVCM = 1/bsdfDirPdf;
        ray[bounce+1].dVC = (cosThetaOut / bsdfDirPdf) * (ray[bounce].dVC * bsdfRevPdfW +
     ray[bounce].dVCM + mMisVmWeightFactor);
        ray[bounce+1].dVM = (cosThetaOut / bsdfDirPdf) * (ray[bounce].dVC * bsdfRevPdfW +
     ray[bounce].dVCM * mMisVcWeightFactor + 1.0);
        ContinuePath(ray);
    }
}
```

Listing 4.4: Second Stage.

To calculate the pixel colour we access the HashGrid in functions ConnectVertices and Process, see listing 4.4. In the first function we only need to access the ray i in the HashGrid to calculate the MIS weight, but in the second function we only calculate the MIS weight for the light nodes within range that is why we use a range search structure, to optimize the search for valid light nodes, as we can see in Figure 4.10.

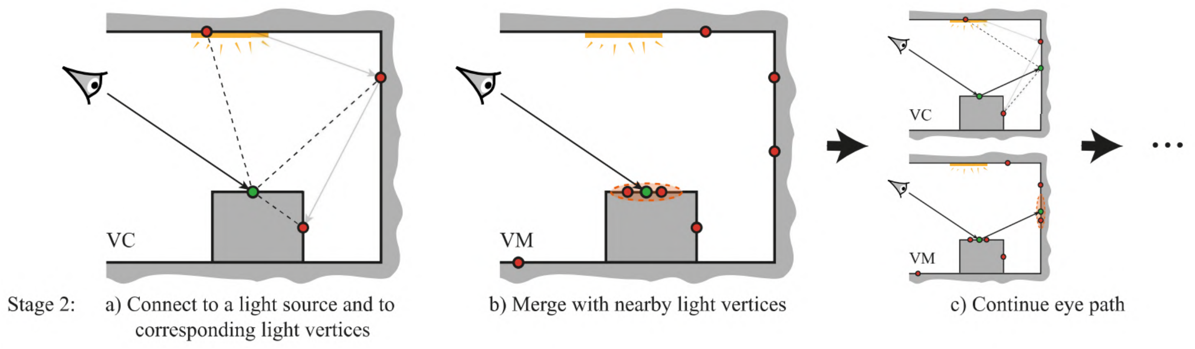

VC
VM
VC
VM
...
Stage 2:
a) Connect to a light source and to corresponding light vertices
b) Merge with nearby light vertices
c) Continue eye path

Figure 4.10: VCM Second Stage.

4.6.3 Implementation issues and alternative

During the rendering process we had some problems. The shaders in the ray tracing pipeline were unable to access the data in the HashGrid, they behaved as if it was empty resulting in images where the only contributions to the the final colour were paths that intersected a light source and DirectIllumination, this resulted in images with low quality as we can see in Figure 4.11.

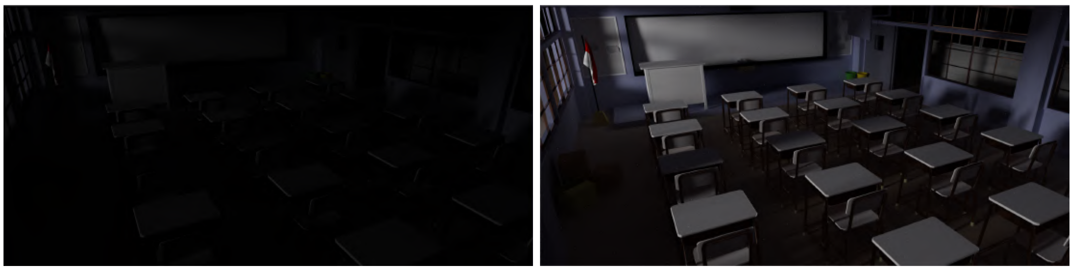


Figure 4.11: Render of the scene ”Japanese Classroom” after 4096 iterations. On the left the result using the HashGrid, on the right the result of our Modified VCM

Facing these problems and due to the time constraints we decided to develop a modified version of VCM that uses only one stage. This way we construct the light path and calculate the pixel colour in the same shader so the data is not lost. However, this leads to fewer light nodes available when calculating the colour of the pixel.

4.7 Modified VCM

Our solution takes the concepts of VCM and adapts them to a single stage algorithm. Instead of constructing the light paths beforehand, like in VCM and PM, the light paths and camera paths are initiated at the same time. During the construction of the camera path both techniques used in VCM are applied, instead of only connecting the camera and light nodes, as in BDPT, we also merge the light nodes within range of the camera node. This implementation is very similar to the BDPT, the only difference is that it uses the function Process to evaluate the light nodes in the vicinity of the camera node after connecting the camera vertex to all light nodes.

4.7.1 Light Tracing

```
void lightTracing () {

    LighPath ray[] = generateRay(Random);
    //initiate sub-path vertex data
    ray[0].dVCM = directPdfA / emissionPdf;
    ray[0].dVC = 0;
    ray[0].dVM = 0;

    for (int bounce = 0; bounce < lighPath_lenght ; bounce ++) {
        ray[bounce] = traceRay();
        if(ray.hit && !Specular){
            ray[bounce].dVCM /= cosTheta;
            ray[bounce].dVC /= cosTheta;
        }
        ray[bounce+1] = sampleScattering();
        ray[bounce+1].dVCM = 1/bsdfDirPdf;
        ray[bounce+1].dVC = (cosThetaOut / bsdfDirPdf) * (ray[bounce].dVC * bsdfRevPdfW +
    ray[bounce].dVCM + mMisVmWeightFactor);
        ray[bounce+1].dVM = (cosThetaOut / bsdfDirPdf) * (ray[bounce].dVC * bsdfRevPdfW +
    ray[bounce].dVCM * mMisVcWeightFactor + 1.0);
        ContinuePath(ray);
    }
}
```

Listing 4.5: Modified VCM Light Path

directPdfA	$p_0^{connect}$, area pdf of the light emitting a photon in that direction.
emissionPdf	p_0^{trace}, area pdf for sampling the light itself.
lightCos	$\overleftarrow{g}_0$, reverse area pdf, the value is the cosine of the angle between the light normal and the reverse ray direction.
cosTheta	$\overrightarrow{p}_1$, forward area pdf for node i, the value is the cosine of the angle between the normal at the hit point and the reverse ray direction.
bsdfDirPdf	$p_i^{connect}$, forward pdf for node i, its value is calculated by the bsdf function, because it depends on the type of the sampled event as is shown in section 4.4.2
cosThetaOut	$\overleftarrow{g}_{i-1}$, reverse pdf conversion factors from solid angle measure to area measure, the value is the cosine of the angle between the normal at intersection point and the reversed of the new ray direction.
bsdfRevPdfW	$\overleftarrow{p}_{\sigma,i-2}$, reverse solid angle pdf for node i, its value is calculated by the bsdf function just like bsdfDirPdf.

Table 4.1: Variables meaning.

The Modified VCM implementation, for each pixel in the final image, imitates by constructing the light path and storing the light nodes in a vector. We used the technique developed by Georgiev [23] to calculate the MIS weights, so when creating the light paths we also calculated the sub-path vertex data for each node. This solution facilitates the calculation of the MIS weights during the camera path loop in addition to being faster.

In this code snippet 4.5, we can see the implementation of the recursive formulation developed by Georgiev [23] for the light paths and the calculations needed to get the values of the sub-path vertex data dVCM, dVC and dVM explained in section 3.2.3. These values will be used during the camera path, the next stage of the Ray Generation shader, when calculating the pixel colour.

The images presented in Figures 4.134.13, rendered in Lift [27], using Modified VCM, show that it can render diffuse lighting easily. The bleeding effect can be seen in the first scene, ”Cornell Box”, on the side of the boxes and in the scene ”Cornell Box Dragon”, in the dragon. However, the second scene, ”Reflection Cornell Box” raises a few doubts about the efficiency of Modified VCM when rendering SDS

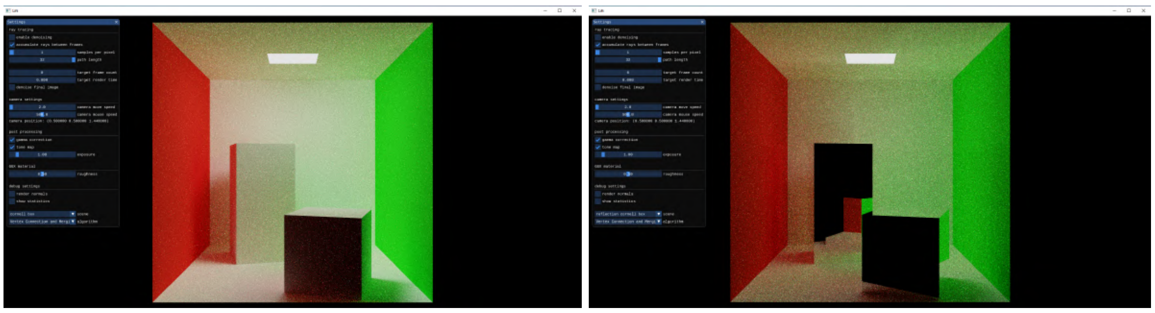


Figure 4.12: Modified VCM render of Cornell Box (left) and Reflection Cornell Box (right).

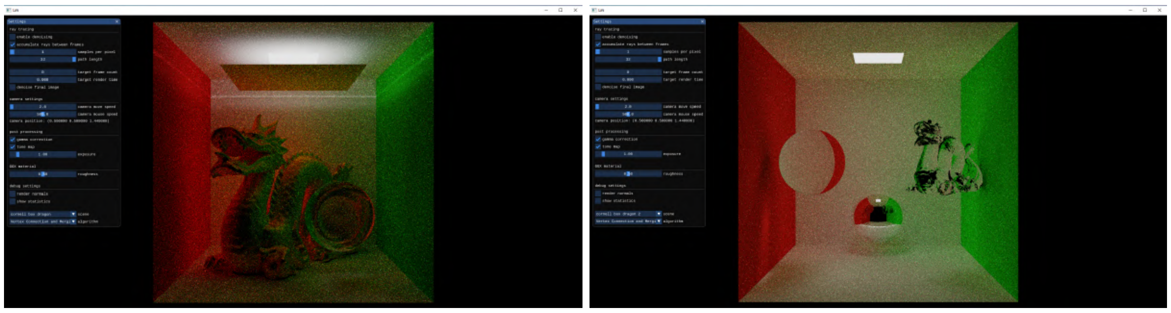


Figure 4.13: Modified VCM render of Cornell Box Dragon (left) and Cornell Box Dragon 2 (right).

paths with the few available light paths.

4.7.2 Camera Tracing

Camera path tracing has the same structure as in VCM, but we needed to change some values, like the number of light paths available, to correctly estimate the final colour of the pixel. It builds a camera path for each pixel. When building the camera path, for each node, first it calculates the sub-path vertex data and before going to next node it executes the techniques Vertex Connection and Vertex Merging to update the pixel colour.

```
void pathTracing () {

    vec3 colour = Vec3(0);

    CameraPath ray[] = generateRay(Random);
    //initiate sub-path vertex data
    ray[0].dVCM = mLightSubPathCount / cameraPdfW;
    ray[0].dVC = 0;
    ray[0].dVM = 0;

    for (int bounce = 0; bounce < cameraPath_lenght ; bounce ++) {
        ray[bounce] = traceRay();
        if(ray.hit && !Specular){
            ray[bounce].dVCM /= cosTheta;
            ray[bounce].dVC /= cosTheta;
            ray[bounce].dVM /= cosTheta;

            if(hit.emissive)
                // Emitting material s = 0
                colour += ray.colour * GetLightRadiance();

            //Direct Illumination, s = 1
            colour += ray.colour * DirectIllumination();

            if(!Specular)
                for (int i = 0; bounce < lighPath_lenght ; bounce ++)
                    colour += ray.colour * lihtPath[i].colour * ConnectVertices(ray,
    lightPath[i]);
                colour += ray.colour * mVmNormalizationr * Process(ray, lightPath);

        }
        ray[bounce+1] = sampleScattering();
        ray[bounce+1].dVCM = 1/bsdfDirPdf;
        ray[bounce+1].dVC = (cosThetaOut / bsdfDirPdf) * (ray[bounce].dVC * bsdfRevPdfW +
     ray[bounce].dVCM + mMisVmWeightFactor);
        ray[bounce+1].dVM = (cosThetaOut / bsdfDirPdf) * (ray[bounce].dVC * bsdfRevPdfW +
     ray[bounce].dVCM * mMisVcWeightFactor + 1.0);
        ContinuePath(ray);
    }}
```

Listing 4.6: Modified VCM Camera Path

The important calculations are wrapped in the functions GetLightRadiance(), DirectIllumination(), ConnectVertices() and Process(). In these functions, we use the sub-path vertex data to calculate the MIS weight of each path and add it to the pixel final colour. As we can see in figure 4.14, the functions DirectIllumination, ConnectVertices and Process are the main contributions to the final pixel colour. The first function works as next-event-estimation. At every new camera vertex it evaluates if the light source is occluded and if it is not it adds its component. The second function connects the camera vertex to all light vertices not occluded, creating new paths that add the light contribution to the final colour of the pixel. The final function used evaluates all light nodes and add the contribution of the ones within the merge radius.

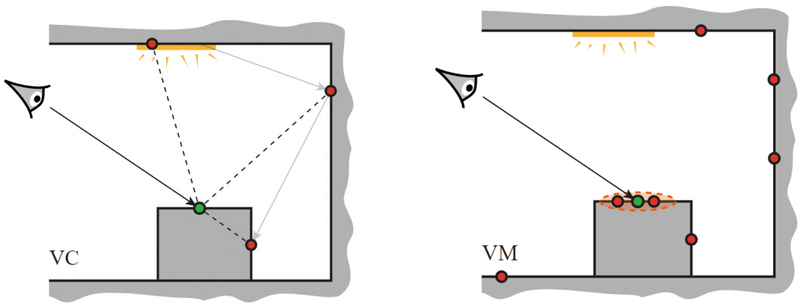


Figure 4.14: Representation of the Vertex Connection (on the left) and the Vertex Merging (on the righ) techniques.

Chapter 5

Evaluation

This master thesis goal is to implement and integrate Vertex Connection and Merging [19] in Lift [1] and to achieve photo-realistic results while maintaining a real-time performance. We implemented a modified VCM that uses the same principles as VCM, but a different structure. For these reasons, we are going to evaluate the quality of the output and its performance.

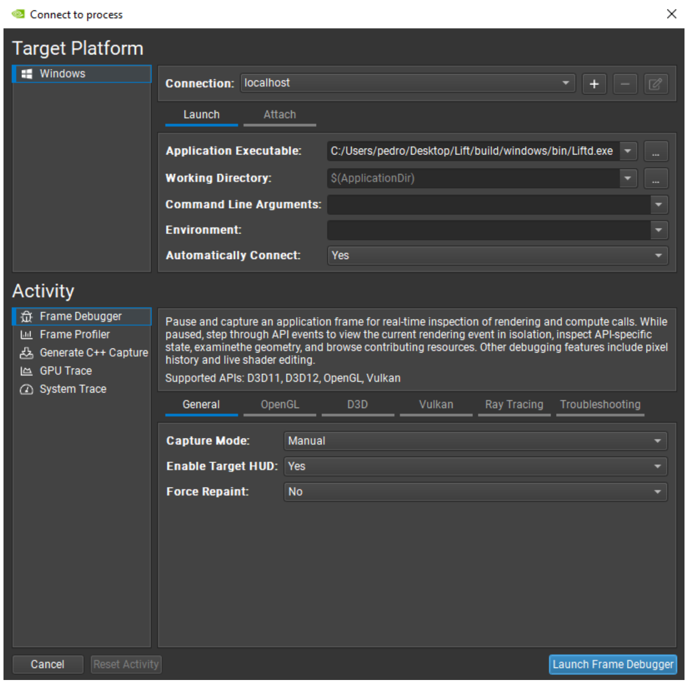
5.1 Evaluation Methodology

To evaluate the algorithms implemented, Modified VCM and Our BDPT, and compare them with the algorithms already developed and integrated in Lift, PT and BDPT, we use a set of scenes with different characteristics such as reflected caustics, low light and difficult to reach light sources.

All the results were obtained in the same hardware: a machine containing an Intel(R) Core(TM) i7-4770K with a base clock speed of 3.5 GHz, 24GB of RAM and with a GPU ASUS NVIDIA Geforce RTX 2080 graphics card. To evaluate the performance of the algorithms we used NVIDIA Nsight Graphics.

5.2 NVIDIA Nsight Graphics

NVIDIA Nsight Graphics is a powerful tool used to debug, profile and export frames from applications that uses the Vulkan API, among others. It is a complex tool with many configurations: Frame Debugger, Frame Profiler, Generate C++ Capture, GPU Trace and System trace (Figure 5.1).


Connect to process

Target Platform

Windows

Connection: localhost

Launch | Attach

Application Executable:	C:/Users/pedro/Desktop/Lift/build/windows/bin/Liftd.exe
Working Directory:	$(ApplicationDir)
Command Line Arguments:	
Environment:	
Automatically Connect:	Yes

Activity

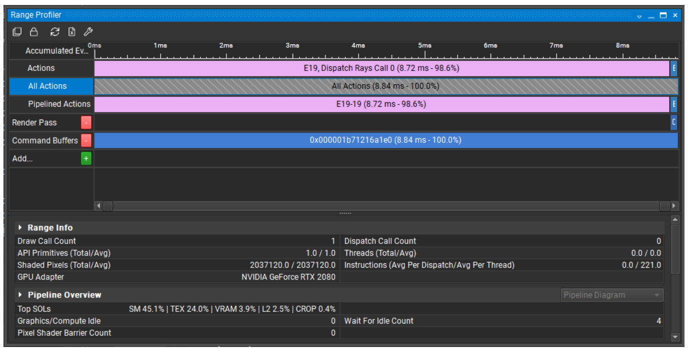
Pause and capture an application frame for real-time inspection of rendering and compute calls. While paused, step through API events to view the current rendering event in isolation, inspect API-specific state, examinethe geometry, and browse contributing resources. Other debugging features include pixel history and live shader editing.

Supported APIs: D3D11, D3D12, OpenGL, Vulkan

Capture Mode:	Manual
Enable Target HUD:	Yes
Force Repaint:	No

Figure 5.1: Nsight Graphics launch menu.

We used the options Frame Profiler and GPU Trace to analyze the performance of our implementation and compare it to other algorithms in Lift. Frame Profiler gives the option to analyze the behaviour of our implementation and of the hardware when rendering one frame. As we can see from Figure 5.2, Lift is very efficient and after ending the Render Pass, it was rendering the next frame in less than one millisecond. Since we had the opportunity, we also used the option to profile the shaders and evaluate the time distribution between them when rendering one frame. From Figure 5.3, we can confirm what we have discussed in Section 4.2. When rendering a frame, 49% of the time is spent in the Ray Generation shader. This was expected as the complex calculations to find the pixel final colour are all done in this shader. We also did this evaluation with Modified VCM to understand the impact of its higher complexity, and as we can see in Modified VCM, the Ray Generation shader takes even a bigger percentage of the time, 66% shown in Figure 5.4.


Range Profiler

Actions	E19, Dispatch Rays Call 0 (8.72 ms - 98.6%)
All Actions	All Actions (8.84 ms - 100.0%)
Pipelined Actions	E19-19 (8.72 ms - 98.6%)
Render Pass	
Command Buffers	0x000001b71216a1e0 (8.84 ms - 100.0%)
Add...	

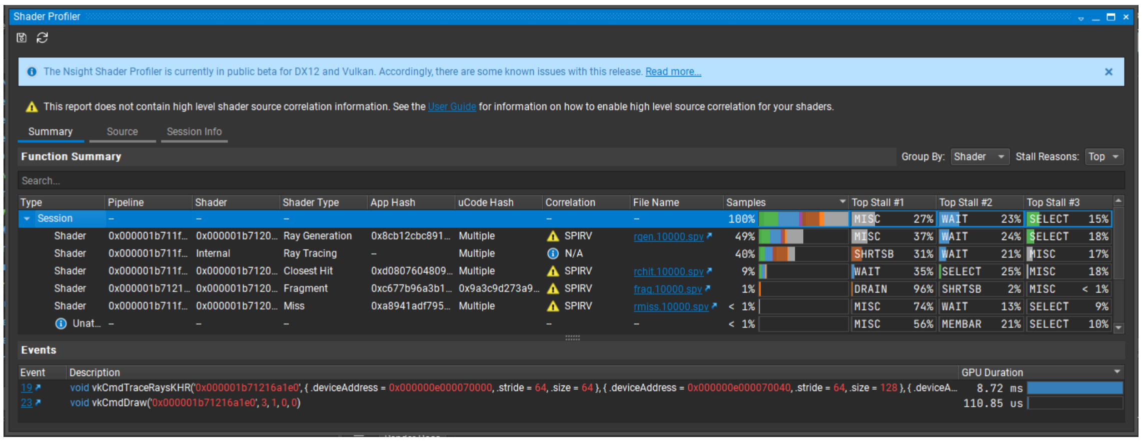
Range Info

Draw Call Count	1	Dispatch Call Count	0
API Primitives (Total/Avg)	1.0 / 1.0	Threads (Total/Avg)	0.0 / 0.0
Shaded Pixels (Total/Avg)	2037120.0 / 2037120.0	Instructions (Avg Per Dispatch/Avg Per Thread)	0.0 / 221.0
GPU Adapter	NVIDIA GeForce RTX 2080		

Pipeline Overview

Top SOLs	SM 45.1% | TEX 24.0% | VRAM 3.9% | L2 2.5% | CROP 0.4%		
Graphics/Compute Idle	0	Wait For Idle Count	4
Pixel Shader Barrier Count	0		

Figure 5.2: Nsight Graphics Frame Profiler.



Shader Profiler

The Nsight Shader Profiler is currently in public beta for DX12 and Vulkan. Accordingly, there are some known issues with this release. Read more...

This report does not contain high level shader source correlation information. See the User Guide for information on how to enable high level source correlation for your shaders.

Summary | Source | Session Info

Function Summary

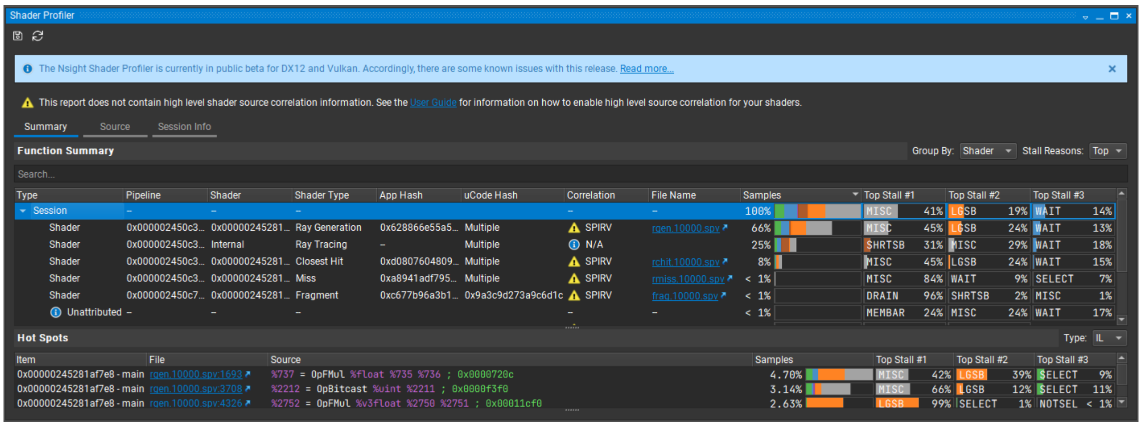
Group By: Shader | Stall Reasons: Top

Type	Pipeline	Shader	Shader Type	App Hash	uCode Hash	Correlation	File Name	Samples	Top Stall #1		Top Stall #2		Top Stall #3	
Session	–	–	–			–	–	100%	MISC	27%	WAIT	23%	SELECT	15%
Shader	0x000001b711f...	0x000001b7120...	Ray Generation	0x8cb12cbc891...	Multiple	SPIRV	rgen.10000.spv	49%	MISC	37%	WAIT	24%	SELECT	18%
Shader	0x000001b711f...	Internal	Ray Tracing	–	Multiple	N/A		40%	SHRTSB	31%	WAIT	21%	MISC	17%
Shader	0x000001b711f...	0x000001b7120...	Closest Hit	0xd0807604809...	Multiple	SPIRV	rchit.10000.spv	9%	WAIT	35%	SELECT	25%	MISC	18%
Shader	0x000001b7121...	0x000001b7120...	Fragment	0xc677b96a3b1...	0x9a3c9d273a9...	SPIRV	frag.10000.spv	1%	DRAIN	96%	SHRTSB	2%	MISC	< 1%
Shader	0x000001b711f...	0x000001b7120...	Miss	0xa8941adf795...	Multiple	SPIRV	rmiss.10000.spv	< 1%	MISC	74%	WAIT	13%	SELECT	9%
Unat...	–	–	–			–	–	< 1%	MISC	56%	MEMBAR	21%	SELECT	10%

Events

Event	Description	GPU Duration
19	void vkCmdTraceRaysKHR('0x000001b71216a1e0', { .deviceAddress = 0x000000e000070000, .stride = 64, .size = 64 }, { .deviceAddress = 0x000000e000070040, .stride = 64, .size = 128 }, { .deviceA...	8.72 ms
23	void vkCmdDraw('0x000001b71216a1e0', 3, 1, 0, 0)	110.85 us

Figure 5.3: Nsight Graphics Shaders Profiler, running Path Tracer.



Shader Profiler

The Nsight Shader Profiler is currently in public beta for DX12 and Vulkan. Accordingly, there are some known issues with this release. Read more...

This report does not contain high level shader source correlation information. See the User Guide for information on how to enable high level source correlation for your shaders.

Summary | Source | Session Info

Function Summary

Group By: Shader | Stall Reasons: Top

Type	Pipeline	Shader	Shader Type	App Hash	uCode Hash	Correlation	File Name	Samples	Top Stall #1		Top Stall #2		Top Stall #3	
Session	–	–	–			–	–	100%	MISC	41%	LGSB	19%	WAIT	14%
Shader	0x000002450c3...	0x00000245281...	Ray Generation	0x628866e55a5...	Multiple	SPIRV	rgen.10000.spv	66%	MISC	45%	LGSB	24%	WAIT	13%
Shader	0x000002450c3...	Internal	Ray Tracing	–	Multiple	N/A		25%	SHRTSB	31%	MISC	29%	WAIT	18%
Shader	0x000002450c3...	0x00000245281...	Closest Hit	0xd0807604809...	Multiple	SPIRV	rchit.10000.spv	8%	MISC	45%	LGSB	24%	WAIT	15%
Shader	0x000002450c3...	0x00000245281...	Miss	0xa8941adf795...	Multiple	SPIRV	rmiss.10000.spv	< 1%	MISC	84%	WAIT	9%	SELECT	7%
Shader	0x000002450c7...	0x00000245281...	Fragment	0xc677b96a3b1...	0x9a3c9d273a9c6d1c	SPIRV	frag.10000.spv	< 1%	DRAIN	96%	SHRTSB	2%	MISC	1%
Unattributed	–	–	–			–	–	< 1%	MEMBAR	24%	MISC	24%	WAIT	17%

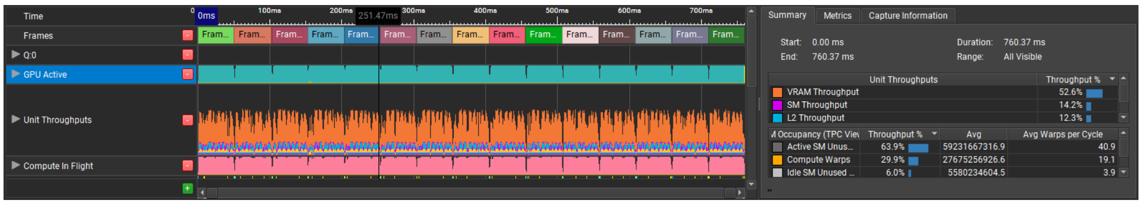
Hot Spots

Type: IL

Item	File	Source	Samples	Top Stall #1		Top Stall #2		Top Stall #3	
0x00000245281af7e8 - main	rgen.10000.spv:1693	`%737 = OpFMul %float %735 %736 ; 0x0000720c`	4.70%	MISC	42%	LGSB	39%	SELECT	9%
0x00000245281af7e8 - main	rgen.10000.spv:3708	`%2212 = OpBitcast %uint %2211 ; 0x0000f3f0`	3.14%	MISC	66%	LGSB	12%	SELECT	11%
0x00000245281af7e8 - main	rgen.10000.spv:4326	`%2752 = OpFMul %v3float %2750 %2751 ; 0x00011cf0`	2.63%	LGSB	99%	SELECT	1%	NOTSEL	< 1%

Figure 5.4: Nsight Graphics Shaders Profiler running Modified VCM.

The last configuration we used was GPU Trace and from here we were able to calculate the average time it takes to render a frame by each algorithm and evaluate their performance by calculating the average frame per second.



Summary | Metrics | Capture Information

Start: 0.00 ms | End: 760.37 ms | Duration: 760.37 ms | Range: All Visible

	Unit Throughputs	Throughput %
🟧	VRAM Throughput	52.6%
🟪	SM Throughput	14.2%
🟦	L2 Throughput	12.3%

	M Occupancy (TPC View	Throughput %	Avg	Avg Warps per Cycle
⬛	Active SM Unus...	63.9%	59231667316.9	40.9
🟨	Compute Warps	29.9%	27675256926.6	19.1
⬜	Idle SM Unused ...	6.0%	5580234604.5	3.9

Figure 5.5: Nsight Graphics GPU Trace running Modified VCM.

In this view, we can also evaluate the usage of the GPU when running Lift and, as we can see in Figure 5.5, the GPU usage is always at maximum capacity during the render pass. We can confirm that our implementation is taking full capacity of the hardware, meaning that if we improved our hardware setup, a bump in the performance could be expected.

5.3 Metrics

Before analysing the final results, we will go over the quality and performance metrics used in this thesis.

5.3.1 Quality Metrics

Structural Dissimilarity

Structure Dissimilarity Index Metric (DSSIM) is an image quality metric that compares the differences between two images. It returns a positive value and the closer the images look alike, the closer its value is to 0, and if it returns 0, it means that there are no differences in the two images. DSSIM is calculated using SSIM, $DSSIM = 1/SSIM - 1$.

Structure Similarity Index Metric (SSIM) is an image quality metric that compares the similarity between a distorted image and a reference image [33]. It gives a value between 0 and 1, where 1 means that the images are completely equal. This is a good metric because it not only compares the difference between each pixel but also compares luminance and contrast from different areas of the image.

Peak Signal-To-Noise Ratio

Image quality can be subjective, and so Peak Signal-To-Noise Ratio (PSNR) works as an approximation to the human perception of reconstruction quality. PSNR is used to measure the noise in a distorted image when compared to the reference image. It can be used to evaluate compression or rendering algorithms. The higher the value it returns, the closer the distorted image is to the reference one. For an 8-bit image, values closer to 50db mean that there is almost no noise in the rendered image.

5.3.2 Performance Metrics

Frames per Second

This metric evaluates the number of frames rendered per second. We will calculate the average and calculate how much time it took to render each frame. This metric is important to evaluate the responsiveness of our implementation and if it can keep a real-time performance perceived as a minimum of 30 frames per second.

Time to Converge

This metric will be used in a static scene to measure how the quality of the output of each algorithm evolves and how long it takes to converge to the reference image. We will also evaluate how the quality of the image evolves per frame rendered. This way we can evaluate the algorithms that need less iterations and the ones that need less time.

This metric will be used and shown along the results of the quality metrics, because it evaluates the quality of the outcome over a period of time. We will also evaluate the quality of the outcome over the number of iterations.

5.4 Test Scenes

To test the performance and the quality of our solution we have chosen scenes with different levels of complexity and different characteristics.

The ”Reflection Cornell Box” is a simple scene with few primitives but perfect to evaluate how well the algorithms find SDS paths. These are paths that have a high impact on the final image but that are very hard to find.

The ”Veach Ajar” scene is a more complex scene where the light is outside the room and enters it through a semi closed door. This scene is good to evaluate how the algorithms perform in scenes with difficult to reach light sources. The ”Dining Room” has the same characteristics as ”Veach Ajar” but it has multiple different geometries, which increases the complexity of the scene. Finally, we will also use the ”Japanese Classroom”, a scene that is frequently used to evaluate and benchmark global illumination algorithms and ray tracing applications.

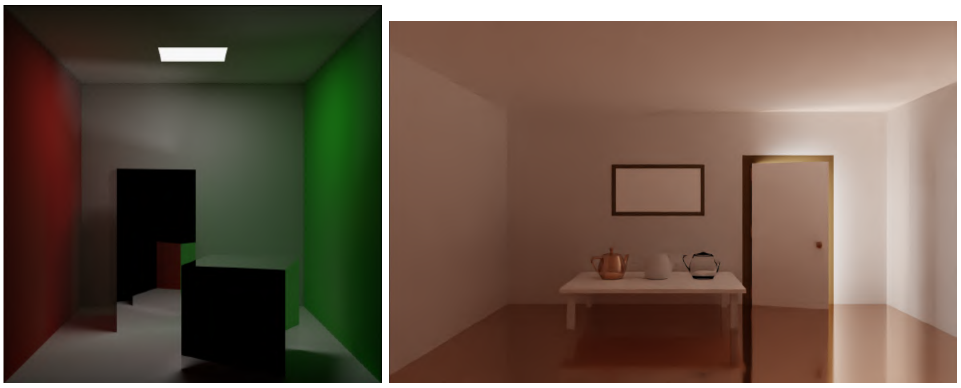


Figure 5.6: Reflection Cornell Box (left). Veach Ajar (right)

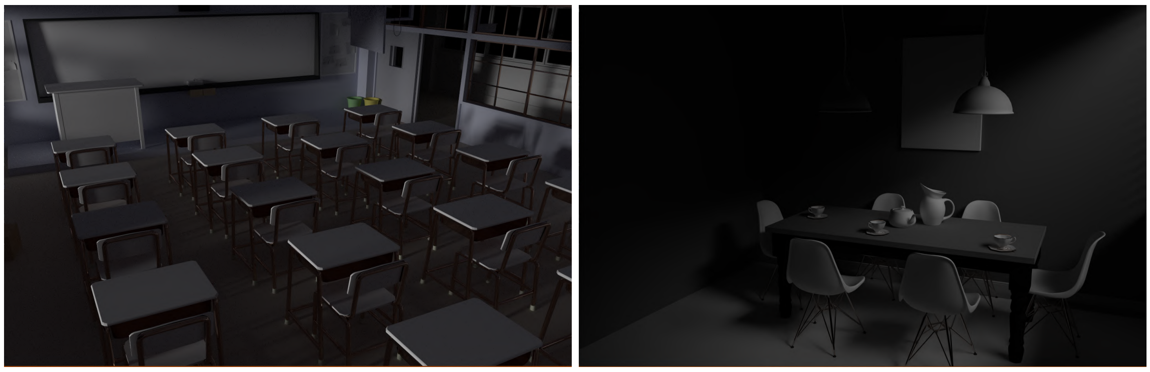


Figure 5.7: Japanese Classroom (left). Dining Room (right)

All reference scenes were rendered using the Lift framework running the Path Tracer algorithm with no denoise. To achieve this images we defined a high number of paths per pixel and a high number of maximum number of bounces per path. With these specification PT is able to converge to the solution more easily. We then let Lift accumulate frames for more than a hour per scene to get best results possible. We did not use SmallVCM to render the reference images because the material model used in this render is very simple and was not capable of rendering the more advanced scenes ”Japanese Classroom”, ”Dining Room” and ”Veach Ajar”.

5.5 Results

Now we analyse and explain the results obtained, compare them to Path Tracing and Bidirectional Path Tracing and evaluate if Modified VCM can keep a real-time performance while achieving good quality results.

5.5.1 Quality Metrics

In this section, we take the images rendered by the algorithms and compute their DSSIM values to see how the quality of the final results evolve over time. We will also evaluate the evolution of the quality over frames. This way we may understand which algorithm needs less iterations to achieve a good result.

Japanese Classroom

Figure 5.9 shows the outcome of 4096 iterations of each algorithm. We can see that, from the three algorithms, the one that seems closer to the reference image, Figure 5.8 shown above, is Modified VCM, see Figures 5.105.11. BDPT and PT seem to struggle to find light carrying paths in this scene, where the only light entering the room comes through the windows on the left side of the classroom.

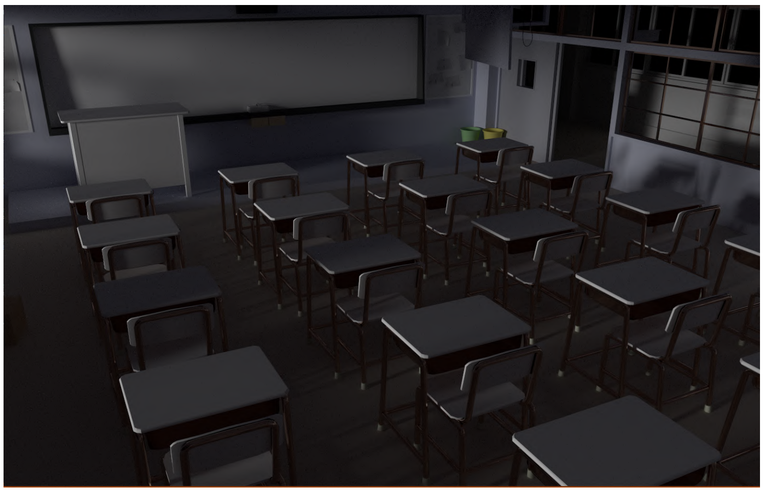


Figure 5.8: Japanese Classroom reference image.

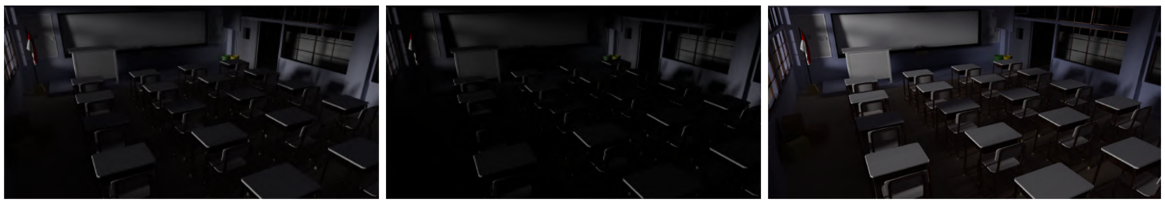


Figure 5.9: PT (left), BDPT (middle) and Modified VCM (right) ”Japanese Classroom” render after 4096 frames.

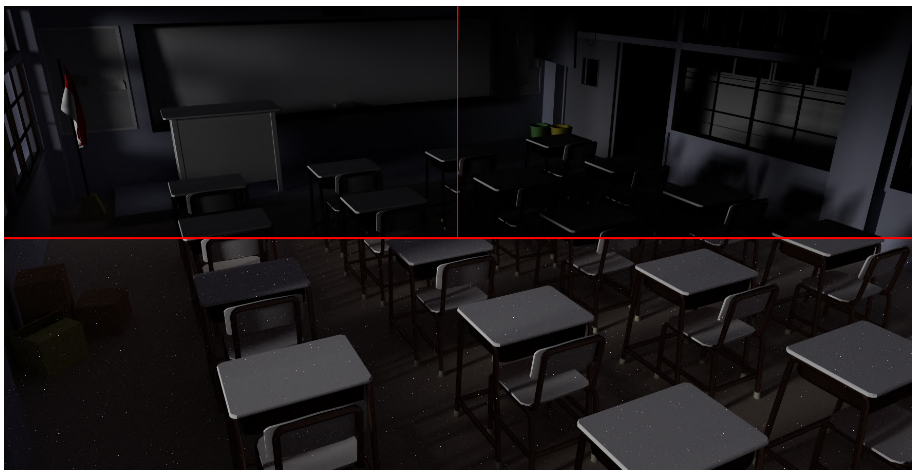


Figure 5.10: ”Japanese Classroom” - Side to Side comparison rendered in 4096 frames. PT (top left), BDPT (top right) and Modified VCM (bottom).

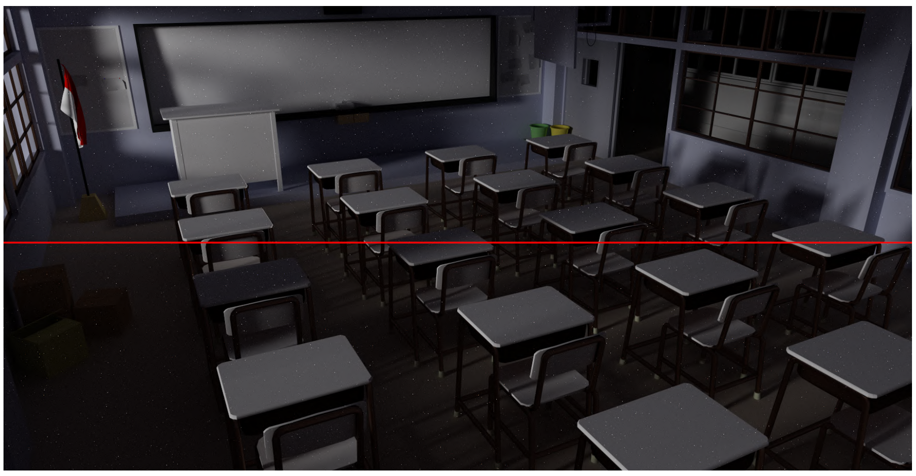


Figure 5.11: ”Japanese Classroom” - Comparison rendered in 4096 frames. Our BDPT (top), Modified VCM (bottom).

From Figure 5.11 we can see that the VM component in Modified VCM does not have any impact in the final image when compared to our implementation of BDPT. However, from the graphic in Figure 5.12 we can conclude that it also does not have any impact in the performance.

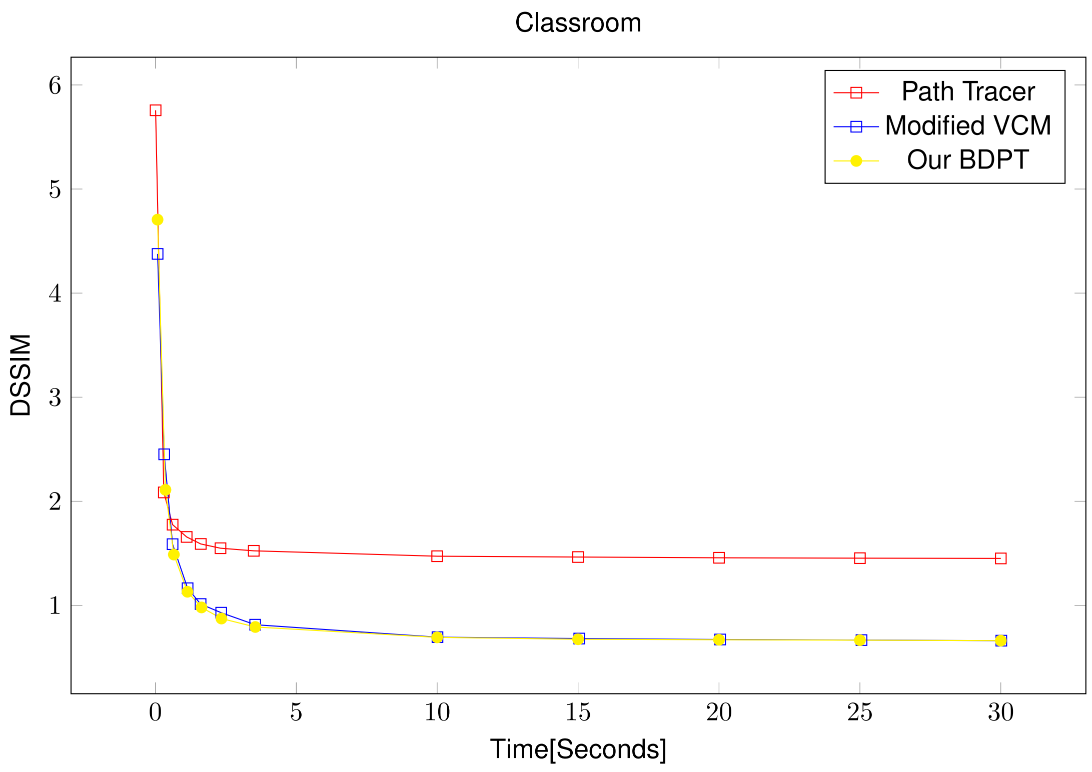


Figure 5.12: Japanese Classroom DSSIM over time

As we can see from the images rendered and the graphic, Modified VCM needs less time and less frames to converge to the solution. In Figure 5.12, we did not evaluate BDPT because in his thesis, Gonçalo [1] had already concluded that PT takes less time to converge to the solution. Even though we reduced the intensity of the light source, we can see from figures 5.9 and **??** that it still applies.

We also concluded, from the graphic shown in Figure 5.12, that even though VM adds complexity to Modified VCM in scenes with no caustics and where the low number of light paths leads to a reduced impact by VM, Modified VCM still keeps up with our BDPT, getting the same DSSIM over time.

These results confirm how efficiently Modified VCM deals with diffuse lighting. In spite of the fact that BDPT is an algorithm that also deals deals well with diffuse lighting, it is clear that the implementation integrated in Lift does not reach the expected results, achieving worse results than Our BDPT.

Dining Room

Figures 5.14 and 5.15 are the outcome of 4096 iterations of each algorithm. This scene is composed almost exclusively of diffuse materials and the only light in the scene is coming from a semi closed window. The outcome produced by Modified VCM seems to have more clarity than the ones produced by PT and BDPT. However the ones produced by BDPT and PT seem closer to the reference image, Figure 5.16.

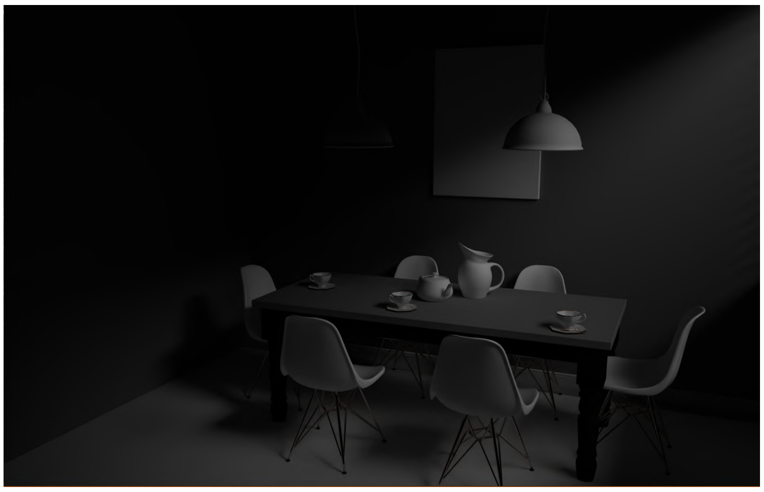


Figure 5.13: Dining Room reference image.

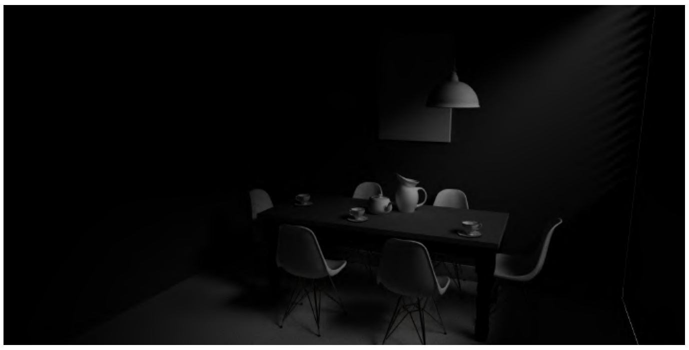


Figure 5.14: PT ”Dining Room” render after 4096 frames.

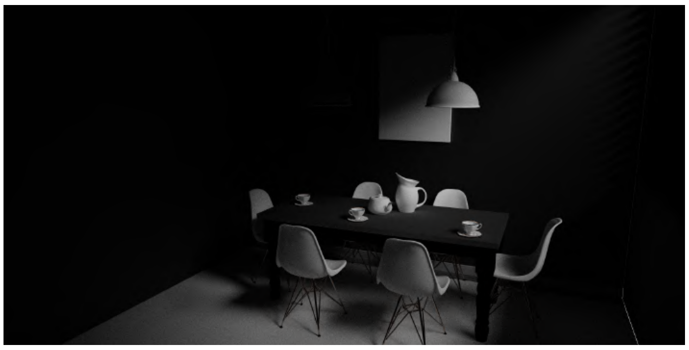


Figure 5.15: Modified VCM ”Dining Room” render after 4096 frames.

As we have seen Modified VCM is able to find difficult to reach light carrying paths easier than PT. However, in this scene PT is still able to outperform BDPT and Modified VCM in time and iterations needed to converge to the solution. Although Modified VCM cannot keep up with PT, it still performs better than BDPT needing less time and iterations to converge to the solution. Again, as seen in the previous scene, Modified VCM has results very close to Our BDPT, see Table 5.1. The added complexity of the VM component does not seem to have a big impact in Modified VCM performance.

Table 5.1: DSSIM evolution over frames rendered.

Implementation	Time[Seconds]	Frames	DSSIM	PSNR
PT	0.015	1	1.477	17.34
PT	0.059	8	1.597	18.35
PT	0.116	16	1.46	19.46
PT	0.380	64	1.220	22.55
PT	1.355	256	0.669	27.43
PT	9.940	1024	0.286	32.22
PT	21.229	4096	0.083	38.68
PT	84.724	16384	0.049	41.78
BDPT	0.09	1	1.425	17.45
BDPT	0.140	8	1.425	18.62
BDPT	1.011	64	0.777	21.57
BDPT	4.009	256	0.594	22.33
BDPT	15.937	1024	0.514	22.6
BDPT	63.753	4096	0.474	22.7
BDPT	127.748	8192	0.466	22.72
Modified VCM	0.03	1	1.342	17.77
Modified VCM	0.27	8	1.296	17.79
Modified VCM	1.873	64	1.206	21.94
Modified VCM	7.482	256	0.675	24.56
Modified VCM	29.635	1024	0.366	25.9
Modified VCM	120.033	4096	0.232	26.49
Our BDPT	0.029	1	1.340	17.76
Our BDPT	0.249	8	1.293	17.8
Our BDPT	1803	64	1.199	21.95
Our BDPT	7.196	256	0.666	24.6
Our BDPT	28.862	1024	0.367	25.89
Our BDPT	115.506	4096	0.232	26.49

Veach Ajar

To evaluate how Modified VCM and PT compare in a scene with a difficult to reach light source but with specular materials capable of reflecting light, we also evaluated their outcomes rendering the Veach Jar scene, see Figure 5.16. In this scene, the floor is made of a Cook-Torrance material with the roughness of 0.31.

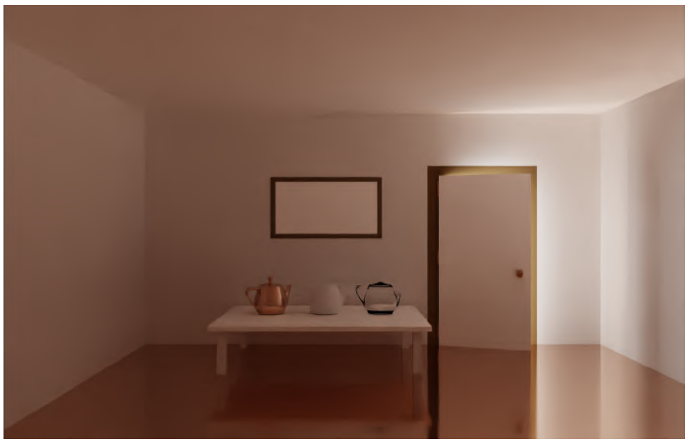


Figure 5.16: Dining Room reference image.

Figures 5.17 and 5.19 are the outcome of 4096 iterations of each algorithm. And contrary to the Dining Room scene, now the outcome of Modified VCM seems to be the one closer to the reference image, with PT finding very few paths that carry light.

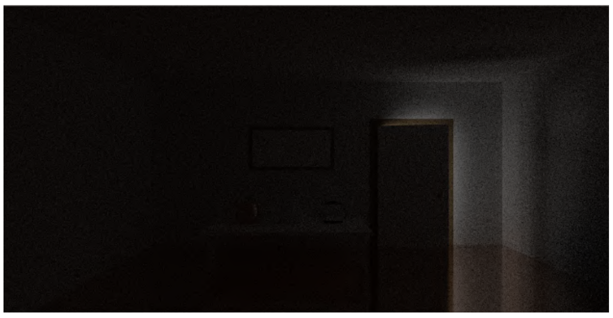


Figure 5.17: PT ”Veach Ajar” render after 4096 frames.

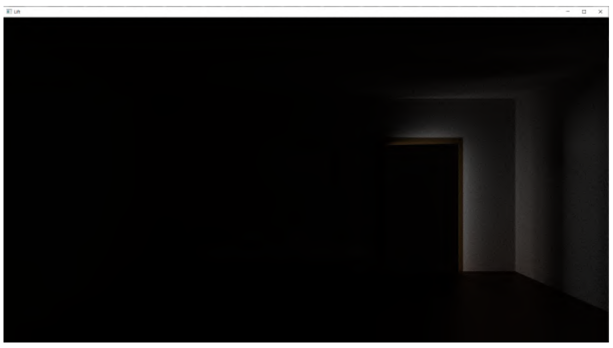


Figure 5.18: BDPT ”Veach Ajar” render after 4096 frames.

And as we can see from graph 5.21 and from table 5.2, Modified VCM clearly outperforms PT, achieving a better result with less iterations and time from the 14s onward. Notwithstanding, this scene is clearly very difficult to render and although Modified VCM does reach better quality results than PT, both algorithms need a very long time to converge to the solution.

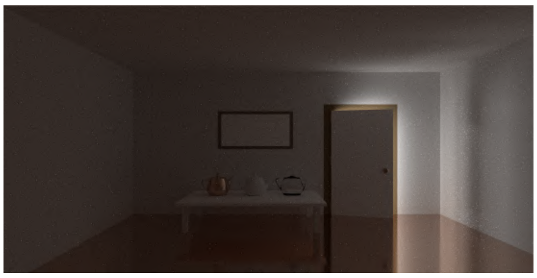


Figure 5.19: Modified VCM ”Veach Ajar” render after 4096 frames.

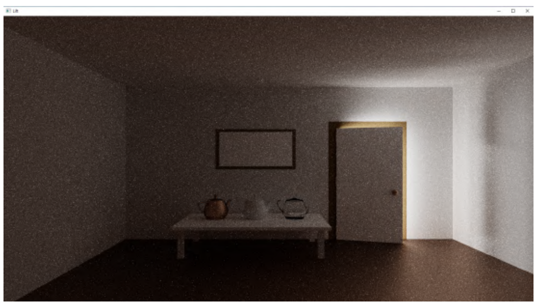


Figure 5.20: Our BDPT ”Veach Ajar” render after 4096 frames.

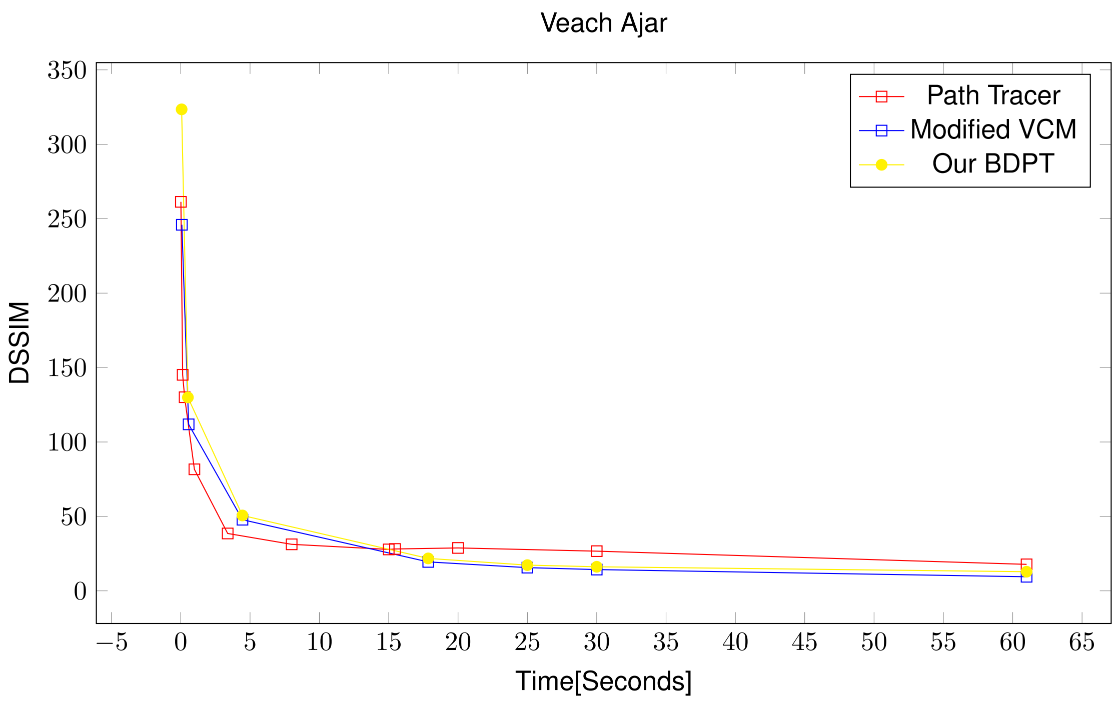


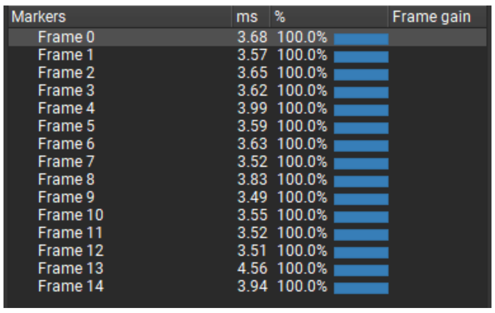
Figure 5.21: Veach Ajar over time

Table 5.2: DSSIM evolution over frames rendered.

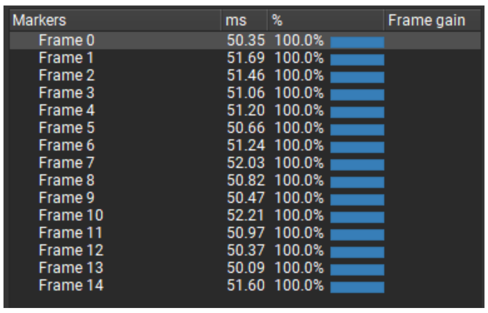
Implementation	Time[s]	Frames	DSSIM	PSNR
PT	0.013	1	261.261	2.964
PT	0.14	8	146.362	3.393
PT	0.283	16	129.276	3.652
PT	0.966	64	80.900	4.009
PT	3.898	256	39.683	3.842
PT	15.517	1024	27.918	3.542
PT	273.912	16384	14.067	3.103
Modified VCM	0.082	1	245.914	3.103
Modified VCM	0.571	8	111.854	5.07
Modified VCM	4.452	64	47.780	5.264
Modified VCM	17.851	256	19.429	5.821
Modified VCM	71.651	1024	8.852	6.079
Modified VCM	273.554	4096	4.176	6.208
Our BDPT	0.066	1	323.465	3.072
Our BDPT	0.489	8	132.067	4.048
Our BDPT	3.588	64	53.348	5.813
Our BDPT	15.637	256	24.484	6.567
Our BDPT	63.365	1024	13.031	7.023
Our BDPT	255.990	4096	7.177	7.284

5.5.2 Performance Metrics

To get the performance results, we used NVIDIA Nsight Graphics and its GPU Trace configuration that allows us to record Lift performance to a maximum of 15 frames. We did this to the four main algorithms integrated in Lift, PT, BDPT, Our BDPT and Modified VCM.



Markers	ms	%	Frame gain
Frame 0	3.68	100.0%	
Frame 1	3.57	100.0%	
Frame 2	3.65	100.0%	
Frame 3	3.62	100.0%	
Frame 4	3.99	100.0%	
Frame 5	3.59	100.0%	
Frame 6	3.63	100.0%	
Frame 7	3.52	100.0%	
Frame 8	3.83	100.0%	
Frame 9	3.49	100.0%	
Frame 10	3.55	100.0%	
Frame 11	3.52	100.0%	
Frame 12	3.51	100.0%	
Frame 13	4.56	100.0%	
Frame 14	3.94	100.0%	



Markers	ms	%	Frame gain
Frame 0	50.35	100.0%	
Frame 1	51.69	100.0%	
Frame 2	51.46	100.0%	
Frame 3	51.06	100.0%	
Frame 4	51.20	100.0%	
Frame 5	50.66	100.0%	
Frame 6	51.24	100.0%	
Frame 7	52.03	100.0%	
Frame 8	50.82	100.0%	
Frame 9	50.47	100.0%	
Frame 10	52.21	100.0%	
Frame 11	50.97	100.0%	
Frame 12	50.37	100.0%	
Frame 13	50.09	100.0%	
Frame 14	51.60	100.0%	

Figure 5.22: GPU Trace result for PT (left) and Modified VCM (right) in Reflection Cornell Box scene.

Table 5.3: Performance results for PT and Modified VCM.

	Path Tracing		Modified VCM	
	Frame Duration[ms]	Frames Per Second	Frame Duration[ms]	Frames Per Second
Reflection Cornell Box	3.7	269.9	51.1	19.6
Japanese Classroom	12.7	78.8	75.7	13.2
Dining Room	4.8	206.5	29.7	33.7
Veach Ajar	14.2	70.3	68.4	14.6

Table 5.4: Performance results for BDPT and Our BDPT.

	BDPT		Our BDPT	
	Frame Duration[ms]	Frames Per Second	Frame Duration[ms]	Frames Per Second
Reflection Cornell Box	10.7	93.2	49.2	20.3
Japanese Classroom	33.3	30.1	72.9	13.7
Dining Room	14.4	69.5	28.9	34.5
Veach Ajar	29.8	33.6	63.9	15.6

As we can see, Modified VCM is the most complex algorithm of the four. But the difference between Our BDPT is negligible, which shows that from the two, Modified VCM is the best algorithm, getting better results over time and not compromising in performance. However both algorithms are only capable of achieving a real-time performance in the simpler scene ”Dining Room”.

It is interesting to see that BDPT, unlike Modified VCM, is able to keep a real-time performance in all scenes while achieving photo-realistic results. However, if we compare the quality results it achieves with PT while also taking into account their performance, we will conclude that PT is the best solution of the two, achieving better quality results in the most complex scene, ”Japanese Classroom”. Besides, PT is also the fastest algorithm, as we can conclude from the recorded Frames Per Second in all four analysed scenes.

Finally, from these data we can better understand the difference in quality results between BDPT and our implementation of the same algorithm. It is true that Our BDPT gets better quality results than BDPT, but the performance is worse. The biggest difference is achieved in the scene ”Reflection Cornell Box”, where Our BDPT takes more 38.5ms than BDPT to render a frame, which is a difference of 459.8%.

Chapter 6

Conclusions

With the results obtained during this thesis we can conclude that for more complex scenes Modified VCM will achieve better results than Path Tracing and Bidirectional Path Tracing. However, those are the same scenes where it cannot keep a real-time performance. It would be interesting to evaluate this implementation with the new NVIDIA Geforce RTX 3080 to see if this implementation is capable of keeping a real-time performance with the best hardware available today.

However, using this hardware, the best solution to achieve the best results while maintaining a real-time performance seems to be Path Tracing with the use of a denoiser as Gonçalo [1] concluded in his thesis. The value added by a good AI-Denoiser surpasses the temporal overhead it introduces.

We can also conclude that although Modified VCM is more complex than Our BDPT it can still achieve better quality results while maintaining a similar performance. And that Our BDPT achieves far better results then the incomplete implementation integrated in Lift by Gonçalo [1].

Although we were not able to implement Vertex Connection and Merging we believe that our implementation of the Modified VCM is viable solution that achieves better results than PT and BDPT and if we evaluate the quality and performance results we conclude that Modified VCM is a better solution when the goal is to achieve the final image with the better quality.

6.1 Achievements

The goal of this thesis, which was to develop an implementation of Vertex Connection and Merging and integrate it in the Lift Framework, developed by Gonçalo Soares [1], capable of rendering photo-realistic results while maintaining a real-time performance, was not achieved. However, we developed a new Modified VCM that achieves better results than the previously implemented algorithms PT and BDPT and we integrated in Lift a complete implementation of BDPT capable of achieving more accurate results than the previous implementation.

6.2 Future Work

When developing this thesis we aimed at implementing the new Variance-aware Multiple Importance Sampling [13]. However, we were not able because of the VKHRay pipeline. Nonetheless, it would be interesting to evaluate if it is possible to implement the new Optimal MIS developed by Ivo Kondapaneni [34]. It differs from the original implementation [12] by assuming weights with negative values.

Furthermore, it would also be important in the future to try and reformulate the Framework to be able to construct the light paths before starting the rendering process to properly implement VCM. It would reduce the rendering time of BDPT, Modified VCM and Our BDPT. This way, all light vertices would be available during the camera tracing process, which would lead to better final results. It would also be necessary to use a hash grid, as a range search structure, to identify the light vertices that are within range of the camera vertex. With this improvement, it would be easy to implement a Progressive Photon Mapping [18], which would add even more value to Lift.

Bibliography

[1] G. Soares. Interactive physics-based rendering with ai-accelerated denoiser. Master's thesis, Instituto Superior Técnico, 2020.

[2] B. T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18:311 – 317, 1975.

[3] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1: 7–24, 1982.

[4] M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016. ISBN 0128006455.

[5] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980. ISSN 0001-0782. doi: 10.1145/358876.358882. URL `https://doi.org/10.1145/358876.358882`.

[6] J. T. Kajiya. The rendering equation. *ACM Siggraph Computer Graphics*, 20(4):143–150, Aug. 1986. doi:10.1145/15886.15902.

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

[8] T. Akenine-Moller, E. Haines, and N. Hoffman. *Real-Time Renderin*. A K Peters/CRC Press, 2008.

[9] J. Bikker and J. Schijndel. The brigade renderer: A path tracer for real-time games. *International Journal of Computer Games Technology*, 2013, 03 2013. doi: 10.1155/2013/578269.

[10] C. Schied and A. Panteleev. Real-time path tracing and denoising in quake ii, 2019. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/ presentation/s91046-real-time-path-tracing-and-denoising-in-quake-2.pdf.

[11] Ph. Dutré, E. P. Lafortune, and Y. D. Willems. Monte carlo light tracing with direct computation of pixel intensities. In *3rd International Conference on Computational Graphics and Visualisation Techniques*, pages 128–137, Alvor, Portugal, December 1993.

[12] E. Veach and L. J. Guibas. Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, page 419–428, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917014. doi: 10.1145/218380.218498. URL `https://doi.org/10.1145/218380.218498`.

[13] P. Grittmann, I. Georgiev, P. Slusallek, and J. Křivánek. Variance-aware multiple importance sampling. *ACM Trans. Graph.*, 38(6), Nov. 2019. ISSN 0730-0301. doi: 10.1145/3355089.3356515. URL `https://doi.org/10.1145/3355089.3356515`.

[14] E. P. Lafortune and Y. D. Willems. Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, pages 145–153, Alvor, Portugal, December 1993.

[15] E. Veach and L. Guibas. Bidirectional estimators for light transport. In G. Sakas, S. Müller, and P. Shirley, editors, *Photorealistic Rendering Techniques*, pages 145–167, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-642-87825-1.

[16] H. W. Jensen. Global illumination using photon maps. In X. Pueyo and P. Schröder, editors, *Rendering Techniques '96*, pages 21–30, Vienna, 1996. Springer Vienna. ISBN 978-3-7091-7484-5.

[17] H. W. Jensen. A practical guide to global illumination using ray tracing and photon mapping. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, page 20–es, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 9781450378017. doi: 10.1145/1103900.1103920. URL `https://doi.org/10.1145/1103900.1103920`.

[18] T. Hachisuka, S. Ogaki, and H. W. Jensen. Progressive photon mapping. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781450318310. doi: 10.1145/1457515.1409083. URL `https://doi.org/10.1145/1457515.1409083`.

[19] I. Georgiev, J. Křivánek, T. Davidovič, and P. Slusallek. Light transport simulation with vertex connection and merging. *ACM Trans. Graph.*, 31(6), Nov. 2012. ISSN 0730-0301. doi: 10.1145/2366145.2366211. URL `https://doi.org/10.1145/2366145.2366211`.

[20] E. Veach and L. Guibas. Metropolis light transport. *Computer Graphics (SIGGRAPH '97 Proceedings)*, 31, 02 1970. doi: 10.1145/258734.258775.

[21] T. Hachisuka, A. S. Kaplanyan, and C. Dachsbacher. Multiplexed metropolis light transport. *ACM Trans. Graph.*, 33(4), July 2014. ISSN 0730-0301. doi: 10.1145/2601097.2601138. URL `https://doi.org/10.1145/2601097.2601138`.

[22] M. Šik, H. Otsu, T. Hachisuka, and J. Křivánek. Robust light transport simulation via metropolised bidirectional estimators. *ACM Trans. Graph.*, 35(6), Nov. 2016. ISSN 0730-0301. doi: 10.1145/2980179.2982411. URL `https://doi.org/10.1145/2980179.2982411`.

[23] I. Georgiev. Implementing vertex connection and merging. Technical report, Saarland University, 2012. URL `http://www.iliyan.com/publications/ImplementingVCM`.

[24] J. C. M. L. Manuel Montoto González. Hands-on study on vulkan and the hardware ray-tracing extensions. Bachelor thesis, Universitat Autònoma de Barcelona, 2021.

[25] N. Subtil. Introduction to real-time ray tracing with vulkan, 2018. https://developer.nvidia.com/blog/vulkan-raytracing/.

[26] G. Soares. https://github.com/GoncaloFDS/Lift.

[27] G. Soares and J. M. Pereira. Lift: An educational interactive stochastic ray tracing framework with ai-accelerated denoiser. *WSCG 2021: full papers proceedings: 29*, pages 325–334, 2021.

[28] P. Rodrigues. https://github.com/pedro-miguel-rodrigues/Lift.

[29] B. Walter, S. R. Marschner, H. Li, and K. E. Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR'07, page 195–206, Goslar, DEU, 2007. Eurographics Association. ISBN 9783905673524.

[30] C. Asmail. Bidirectional scattering distribution function (bsdf): A systematized bibliography. *National Institute of Standards and Technology Journal of Research*, 96:215–223, 03 1991. doi: 10.6028/jres.096.010.

[31] J. de Vries. https://learnopengl.com/pbr/theory.

[32] C. Schlick. An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246, 1994. ISSN 1467-8659. doi: 10.1111/1467-8659.1330233.

[33] Z. Wang, E. Simoncelli, and A. Bovik. Multiscale structural similarity for image quality assessment. volume 2, pages 1398 – 1402 Vol.2, 12 2003. ISBN 0-7803-8104-1. doi: 10.1109/ACSSC.2003.1292216.

[34] I. Kondapaneni, P. Vévoda, P. Grittmann, T. Skřivan, P. Slusallek, and J. Křivánek. Optimal multiple importance sampling. *ACM Trans. Graph.*, 38(4), 2019. doi: 10.1145/3306346.3323009. URL `http://doi.acm.org/10.1145/3306346.3323009`.