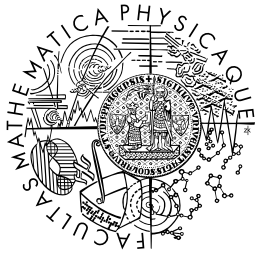


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bohuš Brečka

Efficient light transport simulation of participating media in color 3D printing

Department of Software and Computer Science Education

Supervisor of the master thesis: Tobias Rittig, B.Sc., M.Sc.

Study programme: Informatics

Study branch: Computer Graphics and Game Development

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor, Tobias Rittig, B.Sc., M.Sc., for his valuable advice and guidance throughout my work on this thesis. I would also like to thank my colleagues, RNDr. Martin Šik, Ph.D. and Mgr. Petr Vévoda, for the helpful consultations.

Název práce: Efektivní simulace šíření světla v opticky aktivních médiích pro barevný 3D tisk

Autor: Bohuš Brečka

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: Tobias Rittig, B.Sc., M.Sc., Kabinet software a výuky informatiky

Abstrakt: Monte Carlo simulácia transportu svetla je použitá v pipeline pre farebnú 3D tlač, ktorá má informáciu o šírení svetla (Elek et al. [2017], Sumin et al. [2019]), na riadenie iteratívneho optimalizačného cyklu. Jej účelom je nájsť rozloženie materiálov, ktoré vedie k najväčšej zhode so vzhľadom povrchu cieľa. Keďže simulácia transportu svetla zaberá asi 90% času, predstavuje značnú prekážku pre praktické využitie tejto technológie. Husté uloženie volumetrických textúr taktiež vyžaduje veľa pamäte. Explicitná simulácia každej interakcie svetla je obzvlášť náročná v kombinácii s vlastnosťami 3D výtlačkov kvôli heterogenite, vysokej hustote a vysokému albedu médií. V tejto práci skúmame existujúce techniky pre volumetrický rendering (Křivánek et al. [2014], Herholz et al. [2019]) a nakoniec zostrojíme estimátor prispôsobený pre naše podmienky, čím výrazne zvýšime výkon. Navyše skúmame rôzne riešenia pre ukladanie volumetrických údajov a úspešne znižujeme pamäťovú stopu. Všetky algoritmy sú k dispozícii vo forme pluginov pre Mitsuba renderer.

Klíčová slova: volumetrický path tracing, efektivní rendering, photon mapping

Title: Efficient light transport simulation of participating media in color 3D printing

Author: Bohuš Brečka

Department: Department of Software and Computer Science Education

Supervisor: Tobias Rittig, B.Sc., M.Sc., Department of Software and Computer Science Education

Abstract: A Monte Carlo light transport simulation is used in scattering-aware color 3D printing pipeline (Elek et al. [2017], Sumin et al. [2019]) to drive an iterative optimization loop. Its purpose is to find a material arrangement that yields the closest match in terms of surface appearance towards a target. As the light transport prediction takes up about 90% of the time it poses a significant bottleneck towards a practical application of this technology. The dense volumetric textures also require a lot of memory. Explicitly simulating every light interaction is particularly challenging in the setting of 3D printouts due to the heterogeneity, high density and high albedo of the media. In this thesis, we explore existing volumetric rendering techniques (Křivánek et al. [2014], Herholz et al. [2019]) and finally engineer a customized estimator for our setting, improving the performance considerably. Additionally, we investigate various storage solutions for the volumetric data and successfully reduce the memory footprint. All the algorithms are available in the form of Mitsuba renderer plugins.

Keywords: volume path tracing, efficient rendering, photon mapping

Contents

Introduction

In recent years, the use of 3D printers has become more frequent in many fields, ranging from fashion or entertainment to medicine. The printing devices are more available, they can use a wider variety of materials and vividly reproduce more colors.

The Computer Graphics Group at Charles University is conducting research aimed at the accurate reproduction of textures on the printed objects. The works of Elek et al. [2017] and Sumin et al. [2019] propose an optimization algorithm that uses an iterative feedback loop reorganizing the deposited materials to reduce the appearance degradation caused by a significant subsurface scattering in the printer materials. The loop consists of forward prediction and backward refinement. In order to provide a correct prediction, volumetric light transport needs to be simulated.

A major drawback of this approach is high computational complexity, mainly due to the prediction phase that takes up about 90% of the overall runtime. The virtual scene setup is exceptionally difficult for conventional light solvers like path tracing. The high density and mostly high albedo of the printer materials cause the rays to preserve significant throughput even in excessive depth. Such rays are either terminated too soon, increasing the image variance, or traced for many scattering events at the price of computational time.

Much work has been done in the area of volumetric rendering. Jensen and Christensen [1998] extend the original photon mapping algorithm [Jensen, 1996] to participating media. Jarosz et al. [2008] propose an improvement of the algorithm with so-called beam-queries, which explicitly gather all photons along a ray. Křivánek et al. [2014] provide a review of various volumetric estimators and combine them into a robust algorithm via multiple importance sampling. Herholz et al. [2019] present a set of volumetric path construction techniques guided by a precomputed adjoint transport solution that significantly decrease the variance.

Moreover, due to the heterogeneous nature of the working data and the fact that they are stored in dense grids in the current version of the framework that implements the optimization algorithm, there is also a problem with high memory requirements.

The aim of this thesis is to tackle both the *memory* and *performance* issues and thereby increase the practical usability of the 3D printing appearance optimization algorithm. In order to achieve that, we *employ alternative ways to store heterogeneous volumetric data*, using both a custom solution and the OpenVDB library [Museth et al., 2012]. Next, we *design a custom estimator* leveraging the assumptions given by the specific scene setup. We optimize it for the usage in scenes with trivial light conditions (constant incident radiance over the surface), simple geometry setup (single object), and volume parameters resulting in lasting high ray throughput. Since the optimization framework uses the Mitsuba renderer [Jakob, 2010], we wrap the estimator as a Mitsuba integrator plugin. It is accompanied by plugins implementing medium distance sampling, transmittance evaluation, and data storing.

In the section 1, we examine the optimization algorithm in more depth and describe the libraries used in the current implementation relevant to our work.

Section 2 contains a theoretical background review of volumetric rendering using Monte Carlo and related scientific papers. Section 3 guides through our contribution and describes the approaches that we implemented, together with measurements comparing the impact of the incremental changes. Section 4 compares the performance of our final algorithm setup to the default Mitsuba volumetric path tracer. It also contains notes on the correctness of our implementations.

1. Project background

1.1 Algorithm overview

To set our work in context and justify its contribution, we provide an overview of the optimization and previewing process proposed by Sumin et al. [2019]. Hopefully, it will clarify why we focus our effort on the rendering phase.

In the beginning, the user selects a textured 3D model for printing. The model is transferred to a voxel representation. There are multiple voxel grids, each carrying different information about the source geometry, texture or printer material distribution. The most relevant ones for us are

- *distance grid* - for each voxel, the distance to the nearest surface point is stored
- *distance gradient grid* - contains 3D gradients of the distance grid that is for each voxel in which direction is the nearest surface. This grid is developed solely for the purpose of the rendering optimization. It is not part of the original algorithm
- *linear RGB grid* - stores the float triplets representing the linear RGB color values at a voxel. It is initialized either as white with the texture colors at the surface or extruded multiple layers inwards
- *label grid* - each voxel represents a printer material ID. It is a discretized version of the previous grid obtained via halftoning. Note that unlike the other grids, this grid is stored densely and fills the whole bounding box of the underlying 3D mesh - it has a vast impact on the algorithm's memory footprint

Using the label grid and measured properties of the printer materials, we create a volume describing the physical properties at each point of the printed object to be visualized by conventional rendering techniques. There are two modes - first, previewing the object as if it was printed with the material placement described by the label grid. Second, measuring the radiance on the object surface using a special camera that shoots rays in the orthogonal direction to the surface.

The second camera's rendering results are compared to the desired radiance described by the surface texture, and the difference is used for guiding the optimization feedback loop. In each iteration, the linear RGB grid is altered, so the rendered result is closer to the texture.

Finally, after a sufficient appearance quality is reached, the label grid is converted to 3D printer instructions, which place the materials accordingly.

1.1.1 Performance implications

Due to the constant voxel size (given by the printer technical parameters), the grids scale up with the input model's growing size. Consequently, the memory and time requirements of the algorithm are rapidly increasing with the model

dimensions. For example, scaling a cube model by a factor of two increases the number of surface voxels four times and the overall number of voxels eight times.

The optimization process can be split into three parts - the initialization, including the voxelization and creating of the grids that remain the same throughout the whole run, the optimization loop, and the exporting of results. The initialization is entirely handled by functions of OpenVDB [Museth et al., 2012], so there is not much room for improvement, and the exporting is not computationally demanding. Plus, these two phases are done at most once per the entire run of the pipeline described above. Hence the most concerning is the optimization loop. It contains the most computationally demanding phases, rendering and halftoning, which are moreover run multiple times, depending on the number of iterations. Usually, tens of iterations are necessary to get the desired result.

According to the measurements, the rendering makes up about 90% of the overall time. Very little can be done about the number of rendered images or their size (which is linear to the number of surface voxels for the surface radiance estimation), so the only way is an in-depth analysis and optimization of the rendering algorithm. The main objective of this work is to come up with a more efficient rendering algorithm while also reducing the memory requirements.

1.1.2 Scene setup assumptions

The printed models may vary in shapes and textures, but the rendering scenes in the optimization pipeline have a lot in common. Since our algorithms rely on the specific scene setup, we list the most important assumptions

- *trivial light conditions* - the special camera in the optimization pipeline measures the radiance on the object's surface, for that, constant incident radiance over the surface is used
- *simple scene* - the meshes for the printed objects may be complex, but there is usually only one model rendered at a time, there are no nested or intersecting objects
- *no object re-entering* - the optimization pipeline does not consider object self-shadowing (the effect is added by real illumination of the real printouts), therefore the rays are terminated immediately after they exit the medium (note that this causes bias compared to the basic path tracer, but it is desirable)
- *high density and albedo* - the printer materials mostly possess a high density and the textures are usually bright, especially the object's core is assigned the white color which is very dense

1.2 The application

The algorithm described in the previous chapter is a result of ongoing research in the Computer Graphics Group at Charles University. A follow-up project resulted in a user-friendly application that allows users to utilize the research software.

Foreseeing the necessities to optimize various parts of the optimization pipeline, one of the project's goals was to transform the codebase into a single C++ solution holding all intermediate data in memory and calling functions on a set of linked libraries. This way, further development is way more manageable.

The research for this work was conducted using the application with the goal to incorporate the resulting algorithm there. However, we do not disclose the application, only a command line version with the necessary files is submitted.

1.3 Libraries & third-party software

The aforementioned application frequently uses third-party software, mostly inherited from the original research codebase. The majority of that is irrelevant in the context of our task, so we only address its subset. However, in the upcoming chapters, we will work closely with Mitsuba renderer [Jakob, 2010] and OpenVDB library [Museth et al., 2012] so we provide a brief overview of some of their relevant details.

1.3.1 Mitsuba renderer

The Mitsuba renderer [Jakob, 2010] official webpage defines it as a "research-oriented rendering system implementing a variety of biased and unbiased rendering techniques". It consists of a set of core libraries providing common functionality and many different plugins representing particular light source types, BSDF functions, geometry types, integrators and other components found in modern renderers.

Throughout this work, we will develop a collection of custom plugins representing volume data storages, media and integrators. The plugin system's modularity allows us to easily incorporate our ideas in the codebase simply by inheriting from core classes and implementing their virtual functions.

However, the versatility and modularity lead to an unpleasant consequence in the form of performance overhead. The rendering is usually supervised by the core libraries calling an integrator plugin, which subsequently calls other plugins, creating a deep hierarchy. The virtual calls and limited space for compiler optimisation are reflected on the overall performance, considering the plugin functions' frequency lower in the hierarchy, like phase function sampling or medium distance sampling. Moreover, to provide data for a variety of algorithms, the structures representing geometry intersections or medium scattering events carry much information that is usually not used. However, it is filled by the respective plugins.

1.3.2 OpenVDB

OpenVDB [Museth et al., 2012] is a C++ library providing data structures for storing volumetric data using a sparse representation, and functionality for manipulating it. Since we will be describing optimizations exploiting low-level properties of OpenVDB, we provide a rather detailed description of OpenVDB grids structure and related mechanisms.

The essential class is called *Grid* - it associates a tree with a transform and metadata. This class is directly used in our code.

Museth [2013] describes the *trees* as data structures similar to octrees but with branching factors varying between depth levels. The branching factors are limited to the powers of two. The tree nodes are split into three groups - *root node*, *internal nodes* and *leaf nodes*

- *root node* - contains its children in a hashmap in order to achieve potentially unlimited grid size (it would not be possible with fixed layer count and limited children count)
- *internal node* - has a fixed size, can contain either value (when whole corresponding volume contains the same value) or an array of children with direct access (meaning the worst-case lookup time is O(1))
- *leaf node* - has a fixed size as well, contains an array of values with direct access

The configuration of the tree layers and their sizes, same as the type of the values, is determined by templates. The library provides a set of predefined trees (and grids) that are most commonly used, like integer or float tree (grid).

The values in the tree are not accessed directly but using *value accessors*. They can be obtained by requesting the tree for a specific type of accessor, e.g. constant, non-constant and unsafe (not registered by the tree, in a sense that it is not notified when a tree topology changes, but it can have better performance). The accessors cache nodes along a path to requested voxels, which leads to a significant acceleration of spatially coherent accesses. The accessors require integer coordinates of voxels in index space.

There is also an option to access voxel values using world space coordinates, combined with a possibility to filter the values when a point between voxels is accessed. For this, so-called grid samplers are used.

2. Theory & related work

2.1 Volumetric light transport basics

The printer materials are highly translucent for the UV curing process to work and allow the discrete inks (CMYK + white) to mix subtractively through light scattering underneath the surface. Much research has been conducted to describe accurately and predict light transport inside translucent media. This chapter aims to describe the fundamental principles and methods used in so-called volumetric light transport. Unless stated otherwise, the source material for writing this section is Physically Based Rendering [Pharr and Humphreys, 2010].

The models describe the medium as a collection of particles that interact with photons and shape their path. The collection is so dense that the transport needs to be modeled as a probabilistic process instead of a direct simulation of interactions with individual particles. Using a stochastic formulation, the effect of media on the radiance along rays passing through the media can be evaluated.

The physical model of the radiance distribution describes three types of events that occur inside the media - absorption, emission and scattering (further divided into out-scattering and in-scattering).

Absorption - the light is converted into another form of energy (e.g. heat), so the radiance is reduced. It is described by σ_a, a probability density that light is absorbed per unit distance traveled in the medium. The relationship between the radiance arriving at the point p with direction ω, $L_i(p, -\omega)$, and the exitant radiance, $L_o(p, \omega)$, can be expressed using a differential equation as $dL_o(p, \omega) = -\sigma_a(p)L_i(p, -\omega)dt$.

Emission - luminous particles produce additional radiance at the collisions due to chemical, thermal or nuclear processes. The radiance change is $dL_o(p, \omega) = L_e(p, \omega)dt$.

Scattering - the collisions of rays with particles cause the change of the ray direction. It can be either in-scattering or out-scattering.

Out-scattering - describes a radiance loss when its part gets scattered in a different direction. Similarly to absorption, there is a coefficient σ_s determining the probability of an out-scattering event occurring per unit distance and a differential equation $dL_o(p, \omega) = -\sigma_s(p)L_i(p, -\omega)dt$.

In-scattering - describes a situation where a ray from an arbitrary direction starts heading our direction after a scattering event, increasing the radiance of our ray. The equation here is more complex, as it needs to consider all possible in-scattered ray directions

$$dL_o(p, \omega) = \sigma_s(p) \int_{S^2} p(p, \omega_i, \omega) Li(p, \omega_i) dw_i \tag{2.1}$$

where p stands for the so-called phase function, which describes the distribution of scattered radiation given by an incoming and outgoing direction (similar to the BRDF function).

The combined radiance loss due to the absorption and out-scattering can be expressed using a common coefficient σ_t called attenuation (or extinction) coefficient. The formula is $dL_o(p, \omega) = -\sigma_t L_i(p, -\omega)dt$. The coefficient is also

called density, and the ratio σ_s/σ_t is called albedo (α), describing the medium color. If $p' = p + t\omega$, then the transmittance (the fraction of radiance transmitted between two points) from p to p' is

$$T_r(p \rightarrow p') = exp(-\int_0^t \sigma_t(p + t'\omega, \omega)dt') \tag{2.2}$$

The negated expression in the exponent is called optical thickness.

2.1.1 Light transport equation

Putting the components together, we get the differential form of the transfer equation as follows

$$dL_o(p, \omega) = -\sigma_t(p)L_i(p, -\omega) + \sigma_a L_e(p, \omega) + \sigma_s(p) \int_{S^2} p(p, \omega_i, \omega_o)L_i(p, \omega_i)d\omega_i \tag{2.3}$$

or in the integral form (where $p' = p + \omega t$)

$$L_i(p, \omega) = \int_0^\infty T_r(p' \rightarrow p)[\sigma_a(p')L_e(p', -\omega) + \sigma_s(p') \int_{S^2} p(p', \omega', -\omega)L_i(p', \omega')d\omega']dt \tag{2.4}$$

The previous equations did not consider any geometry in the scene that would stop the ray. If we denote the intersection point of ray (p, ω) and the scene geometry as p_0 and t as its distance from p, then the integral equation of transfer has the following form

$$\begin{aligned} L_i(p, \omega) = {} & T_r(p_0 \rightarrow p)L_o(p_0, -\omega) + \\ & \int_0^t T_r(p' \rightarrow p)[\sigma_a(p')L_e(p', -\omega) + \sigma_s(p') \int_{S^2} p(p', \omega', -\omega)L_i(p', \omega')d\omega']dt' \end{aligned} \tag{2.5}$$

where $p' = p + t'\omega$.

2.1.2 Monte Carlo integration

Monte Carlo is a numerical integration technique that uses random sampling to estimate the value of integral. The idea is to evaluate the integrand for random samples in the domain and compute the average of results weighted by the probability density function for the respective samples. The estimate $\langle I \rangle$ of the integrated function f is

$$\langle I \rangle = \frac{1}{N} \sum_1^N \frac{f(X_i)}{p(X_i)}, \xi_i \propto p((x)) \tag{2.6}$$

The condition that $p(x) > 0$ must hold for all x such that $|f(x)| > 0$. It can be proved that the expected value of $\langle I \rangle$ is equal to the value of the integral I.

As Novák et al. [2018] suggest, the Monte Carlo estimator for volumetric rendering can be formulated as

$$\begin{aligned}\langle L(x,\omega)\rangle = {} & T_r(p_0 \to p) L_o(p_0, -\omega)/P(t) + \\ & T_r(p' \to p)[\sigma_a(p') L_e(p', -\omega) + \sigma_s(p') \int_{S^2} p(p', \omega', -\omega) L_i(p', \omega') d\omega']/p(t')\end{aligned} \tag{2.7}$$

where $p(t')$ is the probability density of distance t', and $P(t)$ is the probability of exceeding the maximum distance t. The interpretation is straightforward, the light transfer equation is split into two terms, and we evaluate either of them depending on the result of random distance sampling - if the sampled distance is lower than the maximum distance, we evaluate the first term and divide it by the probability of sampling such distance. Otherwise, we evaluate the second term divided by the probability of sampling failure (exceeding the maximum distance).

Variance derivation

Based on the Monte Carlo estimator formula, we can derive its variance as follows (F_N is so-called secondary estimator using N samples, F_{prim} is primary estimator using a single sample, $F_{prim} = \frac{f(X_i)}{p(X_i)}$

$$\begin{aligned} Var[F_N] &= Var[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}] \\ &= \frac{1}{N^2} N Var[\frac{f(X_i)}{p(X_i)}] \\ &= \frac{1}{N} Var[F_{prim}] \end{aligned} \tag{2.8}$$

This implies that the standard error for N samples is $\sqrt{N}$ smaller than the standard error for one sample. It is called the convergence rate and for Monte Carlo integration, it stays the same irrespective of the dimensionality of the integral.

2.1.3 Volumetric path tracing

Having the Monte Carlo estimator, we can outline an algorithm for its evaluation. It can be described by the following pseudocode (source: Mitsuba renderer by Jakob [2010]). It does not include emission of any kind.

```
Li = 0.0
throughput = 1.0

medium = nullptr

while (depth++ < maxDepth)
    rayIntersection(ray, its)
    // sample medium
```

```
    if (medium &&
        medium->sampleDistance(ray, its.t, out mediumRecord))

        throughput *= mediumRecord.sigmaS *
                      mediumRecord.Tr /
                      mediumRecord.pdfSuccess

        phaseRecord = phase.sample(ray.d)
        throughput *= phaseRecord.value

        ray = Ray(mediumRecord.p, phaseRecord.wo)
    else
        if (medium)
            throughput *= mediumRecord.transmittance /
                          mediumRecord.pdfFailure

        <Handle scene exit>

        bsdfRecord = sampleBSDF()
        throughput *= bsdfRecord.value

        if (its.isMediumInteraction)
            medium = its.getTargetMedium()

        ray = Ray(its.p, bsdfRecord.wo)

    <Russian roulette based on the throughput>

return Li
```

At this point, the only things left to figure out are the implementation details of the phase function and the medium distance sampling. Additionally, media usually implement a function for the evaluation of transmittance between two given points - its usage will be introduced later. These features are addressed in the following chapters.

2.1.4 Phase function

Phase functions are comparable to BSDF functions, but for media. Similarly, they implement two main functions, one for the sampling of outgoing direction based on a given incoming direction, the other for evaluating the function for two given directions. There are a variety of these functions, describing media with different properties.

We are using the so-called Henyey-Greenstein function [Henyey and Greenstein, 1940]. It has a single parameter g used to control the forwardness of the scattered rays. Its value can be in the range [-1, 1], causing the outgoing rays to head backwards for negative values and forwards for positive ones. If the g is equal to zero, the scattering is isotropic, meaning that the rays are scattered in arbitrary directions, all with the same probability.

The value for a given cosine is

$$p_{HG}(cos\theta) = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g^2 + 2g(cos\theta))^{3/2}} \tag{2.9}$$

It is possible to derive the inverse of this function analytically. It can be sampled in a way that the ratio of the sample value and PDF is equal to one.

2.1.5 Distance sampling & transmittance evaluation

With respect to the implementations of these two functions, the volumes can be divided into two groups - homogeneous and heterogeneous. In the first case, the coefficients σ_s and σ_a (or the albedo and density) are constant for the whole volume. In the second case, the parameters are spatially varying.

If the σ_t is constant, the transmittance for a given distance d is given by Beer's law [Pharr and Humphreys, 2010] as

$$T_r(p \rightarrow p') = exp(-\sigma_t d) \tag{2.10}$$

Because of the simple analytical formula it can trivially be inverted and solved for d. Consequently, the distance sampling based on a random number from the uniform distribution can be done using the following formula

$$t = -ln(1 - \xi)/\sigma_t \tag{2.11}$$

In the case of heterogeneous media, it is far more complicated. Several approaches were developed, we will describe two of them that will be frequently used throughout our work.

2.1.6 Regular tracking

When evaluating the transmittance, this method [Sutton et al., 1999] splits the ray into multiple segments where the coefficients are constant inside each segment. The overall transmittance is computed as

$$T_r(p \rightarrow p') = exp(-\sum_{i=0}^{N} \sigma_{t_i} \Delta_i) \tag{2.12}$$

where N is the number of segments along the ray from p to p' and Δ_i is the length of the i-th ray segment.

In the case of distance sampling based on a random number from the uniform distribution, the desired optical thickness is computed using the formula $\tau = ln(1 - \xi)$. The algorithm then traverses the segments along the ray and accumulates their respective thickness until the target thickness is reached.

This technique evaluates the transmittance exactly. However, due to the rather strong requirement of the piecewise constantness and relatively low performance, it is not widely used. In general media, especially in the case of black

box plugins present in production rendering, it is difficult or impossible to determine the constant parts. However, it is relatively easy when the volumes are stored in 3D grids or octrees.

Amanatides and Woo [1987] propose a simple incremental grid traversing method based on the DDA line algorithm. Museth [2014] implements hierarchical DDA in the OpenVDB library, fully exploiting its tree structure. Both these algorithms are available in the OpenVDB library, and we create our custom versions based on them. We provide their further analysis in the section 1.3.2.

2.1.7 Woodcock tracking

Unbiased method using the maximum density of the whole volume to create a virtual volume such that at each point, the sum of the real and virtual volume density is the same, equal to the maximum density. Proposed by Woodcock et al. [1965].

Consequently, the distance sampling can be implemented in the same way as in homogeneous media, with one difference - at the point corresponding to the new distance, we randomly decide whether the particle collision involves a real or a virtual particle (with the probability proportional to $\sigma_t(t_i)/\sigma_{t_{max}}$). The whole sampling procedure is repeated in a loop until an interaction with a real particle occurs or until the maximum distance is exceeded.

The most straightforward implementation of the transmittance evaluation only calls the sampling function with the maximum distance set to the length of the ray segment where the transmittance is evaluated and returns 1 in case of success and 0 otherwise.

2.2 Multiple importance sampling

In the case of the basic path tracing without volumes, the paths shaped by the sampling of surface BRDF functions often fail to find scene light sources if they are too small, hidden behind other objects, or the BSDF causes high variance of the outgoing rays. The situation is similar in participating media.

Fortunately, there are other sampling strategies that are more successful in these conditions - for example, light sampling. It creates the samples directly on the light source surface and tries to connect them to the camera. Therefore it finds even more distant and smaller lights. Contrary to the BSDF sampling, it results in noisy images when the light source is large or too close to the intersection due to the high variance of the sampled directions. When the BSDF is too directional, it assigns low values for the rays going in directions more deviated from the direction of perfect reflection/refraction, so most of the samples lead to a low gain.

Apparently, the two techniques work in opposite situations. Veach [1997] propose a robust estimator combining multiple sampling techniques to evaluate the same integral to fully leverage the advantages of each technique while hiding its disadvantages. Let N be the number of sampling techniques, n_i the number of samples generated using the i-th technique with probability density p_i, $X_{i,j}$ is the j-th sample generated using the i-th technique (the $X_{i,j}$ are independent). Each estimator has a weighting function $w_i(x)$. Then the multi-sample estimator is

$$\langle I \rangle = \sum_{i=1}^{N} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(X_{i,j}) \frac{f(X_{i,j})}{p_i(X_{i,j})} \tag{2.13}$$

Intuitively, it is a weighted sum of the estimators $\frac{f(Xi,j)}{p_i(X_{i,j})}$. The estimator is unbiased if the weights satisfy the following conditions

- the sum of weights for a given sample is 1 whenever f(x) is not 0
- weight is 0 whenever $p_i(x) = 0$

2.2.1 Weighting functions

The task of the weighting functions is to assign the weights so that the estimator variance is as low as possible. There are various functions, we describe only one of them, the Balance heuristics. It is given by the following formula

$$\omega_i(x) = \frac{n_i p_i(x)}{\sum_k n_k p_k(x)} \tag{2.14}$$

The detailed analysis of balance heuristics and its performance is provided by Kondapaneni et al. [2019].

2.3 Volume Path Guiding Based on Zero-Variance Random Walk Theory

So far, we have described methods that use only a part of the light transport equation during the decision process since they do not have prior knowledge of the incident (L) and in-scattered (L_i) radiance. The distance is sampled only based on the transmittance term and the σ_s. Similarly, the direction is sampled solely on the incoming direction and phase function, and the Russian roulette is driven by the actual throughput.

Zero variance sampling theory [Hoogenboom, 2008] provides a collection of formulas used for path guiding decisions done by distance sampling, direction sampling, and path termination that cause all traced paths to have precisely the same contribution. Consequently, the variance of the estimator is zero (hence the name). We denote the nearest surface contribution as L_s and the medium interaction contribution as L_m.

- the probability of sampling the medium from the starting point x_j and the direction ω_j is

$$P_m(x_j, \omega_j) = \frac{L_m(x_j, \omega_j)}{L_s(x_j, \omega_j) + L_m(x_j, \omega_j)} \tag{2.15}$$

- in case the volume sampling was selected (based on the previous probability), the sampled distance d_{j+1} to the next scattering position x_{j+1} is

$$p_d(d_{j+1}|x_j, \omega_j) = \frac{T_r(x_j, x_{j+1})\sigma_s(x_{j+1})L_i(x_{j+1}, \omega_j)}{L_m(x_j, \omega_j)} \tag{2.16}$$

- the scattered direction at the medium sampled point x_{j+1} is sampled based on the following PDF

$$p_\omega(\omega_{j+1}|x_{j+1}, \omega_j) = \frac{f(x_{j+1}, \omega_j, \omega_{j+1})L(x_{j+1}, \omega_{j+1})}{L_i(x_{j+1}, \omega_j)} \tag{2.17}$$

- the probability for the termination of the path at the point xj where an emitter with the radiance Le is hit is

$$p_{RR}(x_j, \omega_{j-1}) = \frac{L_e(x_j, \omega_{j-1})}{L_o(x_j, \omega_{j-1})} \tag{2.18}$$

  In case there is no emission at x_j, the p_{RR} is equal to one.

The requirement of the prior knowledge of L and L_i creates a cyclic dependency. Herholz et al. [2019] propose an approximate solution using a precomputation of these quantities. Before the actual rendering, a photon tracing preprocessing phase takes place. The photons are cached and then used to construct the directional representation of the incident radiance $\tilde{L}$ and in-scattered radiance $\tilde{L}_i$. The data are called the adjoint transport solution. The distributions are represented using parametric mixtures of von Mises-Fisher distribution [Fisher, 1953] lobes $\mathcal{V}$ (see the original paper for more details).

Next, the authors propose path guiding techniques using the approximated radiances. We will briefly go through them in the following chapters.

2.3.1 Distance sampling

The decision between the medium or surface contribution and the distance sampling in case of the former are merged into a single decision described by the following PDF

$$p_d(d|x, \omega) = \frac{T_r(x, x_d)\sigma_s(x_d)L_i(x_d, \omega)}{L(x, \omega)} \tag{2.19}$$

Two methods for sampling this product are proposed - a simple naive solution and an efficient incremental distance sampling. For brevity, we only describe the first of them and refer the reader to the original paper for the second one. It is based on the uniform sampling of the ray length and building of a discrete PDF and CDF. As the authors point out, this approach is inefficient due to the potential high ray length and steep transmittance falloff in dense media.

2.3.2 Direction sampling

Similar to the zero-variance equation, the direction is obtained by importance sampling of product mixture of the mixtures $\mathcal{V}_L$ (incoming radiance) and $\mathcal{V}_F$ (phase function). To eliminate the inaccuracies of the estimates, MIS between the mixture samples and the actual phase function samples is used.

2.3.3 Russian roulette and splitting

According to the zero variance theory, the contribution of every path to a given pixel should be equal in order to get zero variance. Based on this, the authors propose to set the survival probability for Russian Roulette based on the expected contribution of the current path. Naturally, there are cases where the path contribution is too high. Then, the splitting takes place.

The paper describes two types of the Russian Roulette and splitting. One is based on the distance sampling, the other on the direction sampling. In both, the splitting factor q is computed as $q = E[r]/I$, where the $E[r]$ is the expected path contribution, and I is the true pixel value (in practice, it is $\hat{I}$ gathered from the precomputed in-scattered radiance using a few camera rays marching through the medium). The $E[r]$ is given by the sampling version.

Distance sampling starts in the situation where we have a scattering point x_j, sampled direction ω_j and path weight a_r. The expected path contribution is $E[r]_{dist} \approx a_r \tilde{L}(x_j, \omega_j)$. Based on that, we either cut the path right at x_j or split it to multiple paths based on distance sampling along the direction ω_j.

Direction sampling takes position x_{j+1} where the path got from the point x_j with the direction ω_j. The updated weight is

$$a'_r = a_r \frac{T_r(x_j, x_{j+1})\sigma_s(x_{j+1})}{p_d(d|x_j, \omega_j)} \tag{2.20}$$

The expected path contribution is $E[r]_{dir} \approx a'_r \tilde{L}_i(x_{j+1}, \omega_j)$. The path is split into several samples directions ω_{j+1}.

Due to the high complexity of this approach and the necessity of a rather lengthy precomputation phase (comparing the usual number of samples that we use in our application), we decide not to implement and test the whole framework. We only try adaptive splitting coefficients based on the actual throughput and scattering coefficient.

2.4 Volumetric photon mapping & Beam radiance estimate

So far, we have presented only variations of unidirectional path tracing that creates paths starting from the camera. The algorithm proposed by Herholz et al. [2019] uses a precomputed structure created from particles traced from the light sources, but only as a tool for guiding the path guiding decisions. There is a family of algorithms for volumetric rendering that uses paths starting both from the camera or from the light sources directly for the radiance estimation. In this

chapter, we introduce the most basic of them all - photon mapping [Jensen and Christensen, 1998], and its optimization for volumetric rendering, Beam Radiance Estimate [Jarosz et al., 2008].

2.4.1 Photon mapping

The photon mapping method proposed by Jensen [1996] is a two-pass algorithm for enhanced global illumination rendering. It does not concern volumes. The first phase creates two photon maps (global map and caustics map) by tracing photons emitted from light sources. In the second phase, the maps are used to compute the final image via distributed ray tracing.

The caustics map stores photons that passed through specular reflections / refractions and hit a diffuse surface - the map is used for direct (using the rays coming directly from the camera) caustics rendering since this effect is costly to compute using the paths originating from the camera. It requires high density, and it is created by shooting the photons exclusively towards specular objects. The global map consists of indirect illumination photons shot in a general direction and gathered on non-specular surfaces. It is used to obtain indirect illumination instead of rays with a high number of bounces as they are costly in distributed ray tracing. Since it is visualized only indirectly, it can be less dense and therefore more efficient.

Each photon keeps the impact position, the incoming direction, and energy. The photons are stored in a balanced KD-tree [Bentley, 1975] that allows quick searches for n nearest photons to a given point x. The radiance is then estimated using a low-pass filter.

The fixed number of photons in the first phase causes the algorithm to be biased, as it does not converge to the correct result with an arbitrary number of rays shot in the second phase. Additionally, the filter does not yield a correct estimate when the photons are near face edges.

2.4.2 Volumetric photon mapping

The extension for volumes proposed by Jensen and Christensen [1998] is straightforward - the third photon map is added for storing photons created at medium scattering points. Only the photons that do not come directly from the light source are stored. The authors argue that single scattering in media is easy to evaluate via the forward tracing. Therefore the photon map does not need to be cluttered with such photons.

During the second phase, the radiance inside the volume is estimated in the same fashion as on the surface - n nearest photons is found near the given point x and the radiance is compute as

$$L_i(x,\omega) \approx \frac{1}{\sigma_s(x)} \sum_{p=1}^{N} f(x,\omega_p,\omega) \frac{\Delta\Phi_p(x,\omega_p)}{\frac{4}{3}\pi r^3} \tag{2.21}$$

where $\Delta\Phi_p$ is the flux carried by each photon p in direction ω_p. The contribution from the photons is gathered using an adaptive ray marching algorithm.

The in-scattering and emission are approximated as constant within each step, so the radiance at the point x_k is given by

$$\begin{aligned} L(x_k, \omega) = &\sigma_a(x_k) L_e(x_k, \omega) \Delta x_k \\ &+ \sigma_s(x_k) L_i(x_k, \omega) \Delta x_k \\ &+ exp(-\sigma_t(x_k) \Delta x_k) L(x_{k-1}, \omega) \end{aligned} \tag{2.22}$$

The $L(x_{k-1}, \omega)$ from the previous step is multiplied by the step attenuation and summed with the contribution of emission and in-scattering for the current step.

2.4.3 Beam radiance estimate

Jarosz et al. [2008] point out multiple shortcomings of the basic volumetric path tracing algorithm. First, it is difficult to choose a viable step size (or step distribution in general) - if the steps are too long, photons are being missed out, leading to higher variance. On the other hand, having many short steps is computationally expensive. Moreover, when the photons are evaluated in the neighborhoods of step points, one photon can be counted twice if it lies inside two such neighborhoods.

The paper proposes a method called "beam gathering" that avoids the problematic stepping through the medium by gathering all photons along the ray via a specialized search structure. The algorithm consists of the following five steps

- *Photon tracing* - works in the same way as in the basic volumetric photon mapping
- *Constructing a balanced KD-tree for the photons* - also identical with the basic method
- *Assigning a radius for each photon* - given a constant integer N, for each photon, the N-th nearest photon is found, and the distance to it is assigned to the photon radius (this N will be noted as lookup size). By this, photons have lower radii at the areas with high photon concentration and vice versa. Therefore, the areas densely filled with photons are less blurred, while the sparse ones are more blurred. The number n determines the overall blurriness of the resulting image
- *Constructing a bounding-box hierarchy over the photons with radii from the previous step* - having the balanced KD-tree from the second step, a balanced BBH is constructed in linear time in the following way. Starting from the bottom, each photon is assigned a bounding box enclosing itself and its two children's bounding boxes
- *Using the BBH to render the image* - for each single-scattering camera ray, let N be the number of photons whose bounding boxes are intersected, then the radiance estimate is

$$L(x,\omega) = \frac{1}{N}\sum_{i=1}^{N} K_i(x,\omega,s,x_i,r_i)T_r(x,x_i')\sigma_s(x_i,\omega,\omega_i)\alpha_i \tag{2.23}$$

where $x_i' = x + t_i\omega$ and it represents the projection of the i-th photon position on the camera ray, $t_i = (x_i - x).\omega$ is the vector from the starting point x to the i-th photon projected on the camera ray, p is the phase function, α is the photon weight and K_i is the kernel defined as

$$K_i(x,\omega,s,x_i,r_i) = r_i^{-2}K_2(\frac{d_i}{r_i})\ if\ d_i\ \in\ [0,r_i]\ and\ 0\ otherwise \tag{2.24}$$

where r_i is the photon radius, d_i is the distance between the ray and the photon position, $K_2(x)$ is Silverman's two-dimensional biweight kernel [Silverman, 1986] defined as $K_2(x) = 3\pi^{-1}(1-x^2)^2$. Note that the kernel is two-dimensional (the radiance is blurred in two dimensions perpendicular to the ray) instead of the three dimensions used in the basic photon mapping since the third dimension is included in the integration along the ray

Moreover, the authors outline an idea for optimizing the algorithm for heterogeneous media, where the computation of the $T_r(x,x_i')$ is too expensive to be evaluated for each photon. Instead, a LUT containing transmittance from the origin to discrete points along the ray segment is created using ray marching. After that, the individual photons intersected by the ray are evaluated, utilizing the transmittances cached in the LUT.

2.5 Unifying Points, Beams, and Paths in Volumetric Light Transport Simulation

Users often desire to render scenes containing volumes that vary greatly in density, scattering albedo, and anisotropy. Many approaches have been developed, however, each of them is usually successful with specific settings while failing with others. Křivánek et al. [2014] seek to develop a robust technique that reasonably handles a combination of volumes with different settings in one scene. First, they provide an analysis of a variety of volumetric rendering algorithms. Then they combine some of them using MIS into so-called unified points, beams, and paths (UPBP) algorithm. Since the basic printer materials all have relatively high densities, we do not need the robustness provided by the UPBP. We only use the analysis to identify the approaches fitting our use case, hence we do not describe the MIS in this chapter.

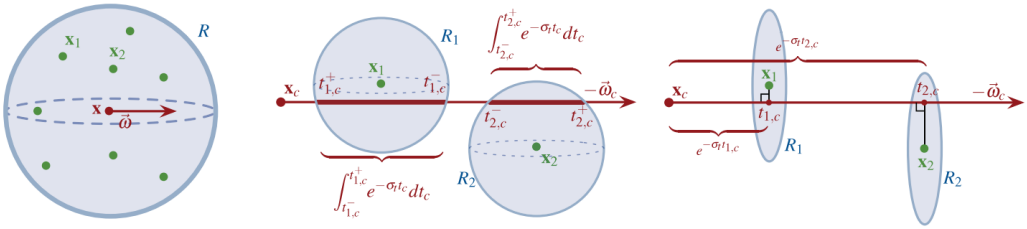
Based on the work of Jarosz et al. [2011], the paper works with a set of estimators with the following attributes

- *photons/beams* - the in-scattered radiance can be evaluated either at the query point or along a ray. This applies both for light traced radiance data and for the query type

- *long/short beams* - a beam is a cylinder with a given radius along a ray. The beam can be either the same length as the ray ("short beams"), or it can extend to the nearest surface ("long beams")

- *dimension* - refers to the blurring kernel dimension, e.g., the basic photon mapping uses 3D kernels, the technique proposed by Jarosz et al. [2008] uses 2D kernel

The individual techniques are referred to as the combination of the radiance data (photons/beams), query data (photons/beams), both with the long/short option, and the dimension of the blurring kernel. For clarification, we include the illustration from Jarosz et al. [2011] (Fig. 2.1)



(a) Point × Point (3D) (b) Beam × Point (3D) (c) Beam × Point (2D)

Figure 2.1: Illustrations of the estimation techniques from Jarosz et al. [2011].

Using a theoretic setup for variance derivation, the authors provide variance analysis of multiple estimator variants. Based on the notation used in the Fig. 2.1, the setup uses fixed orthogonal rays (a, ω_a) and (c, ω_c), and therefore fixed distances, t_a and t_c. As the blurring kernel, a d-dimensional cube is selected. The medium is assumed to be homogeneous.

For the variance derivation of the individual estimators, we refer to the original paper. Here, we only include the comparison of the variance depending on the kernel width w in the units of mean free paths (inverse of the medium density), expressed as normalized standard deviation $\sigma_\mu[\langle I \rangle_*] = \sqrt{V[\langle I \rangle_*]}/E[\langle I \rangle_*]$.

The plot in Fig. 2.2 shows that neither the points nor beams behave the best in each situation. With bigger kernels, the points are better, but as the kernel width gets close to zero, the variance rises to infinity, even for the long beams variants. In general, the long beams perform better than the short beams. Again, for further analysis, we refer to the original paper. Additionally, the paper includes a set of images rendered with different estimators and kernel widths (Fig. 2.3). It confirms the conclusions from the theoretical analysis - the P-P3D estimator is better with larger kernel sizes, while the $B_s - B_s1D$ achieves lower variance with the smaller kernels.

The beam length poses a trade-off between the variance reduction and computational time. Long beams appear to obtain a lower variance, while its evaluation is more costly. The authors decide to use long query beams and short photon beams based on testing results.

The final choice of estimators combined using the MIS is BPT, then P-P3D, $P - B_l2D$, $B_s - B_l1D$ for volumes and P-P2D for surfaces. The testing on a wide range of volumes confirms the expectations - none of the techniques dominates in

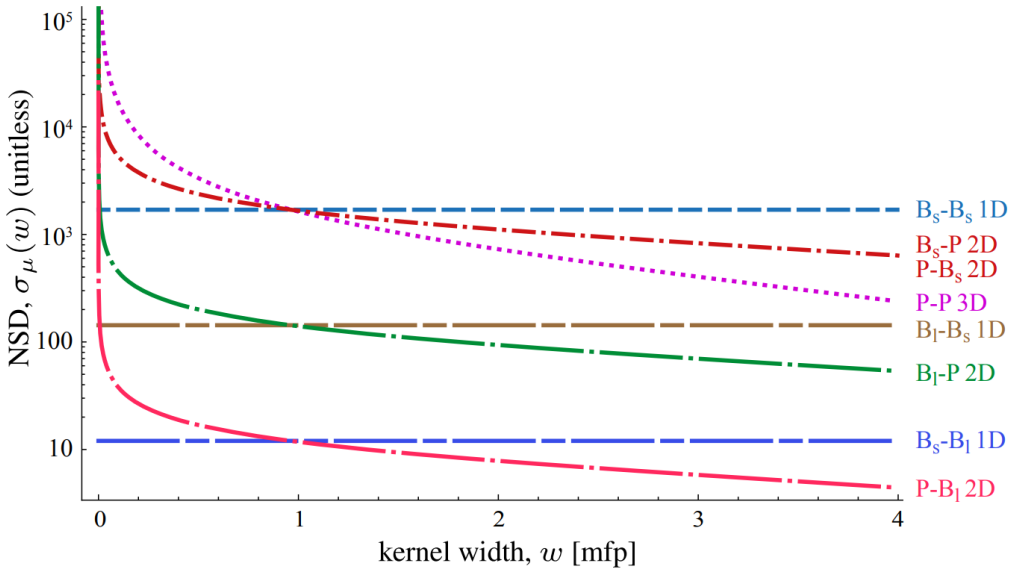


Figure 2.2: NSD as a function of the kernel witdth from Křivánek et al. [2014].

all cases. The contributions of the different estimators are visualized in the Fig. 2.4.

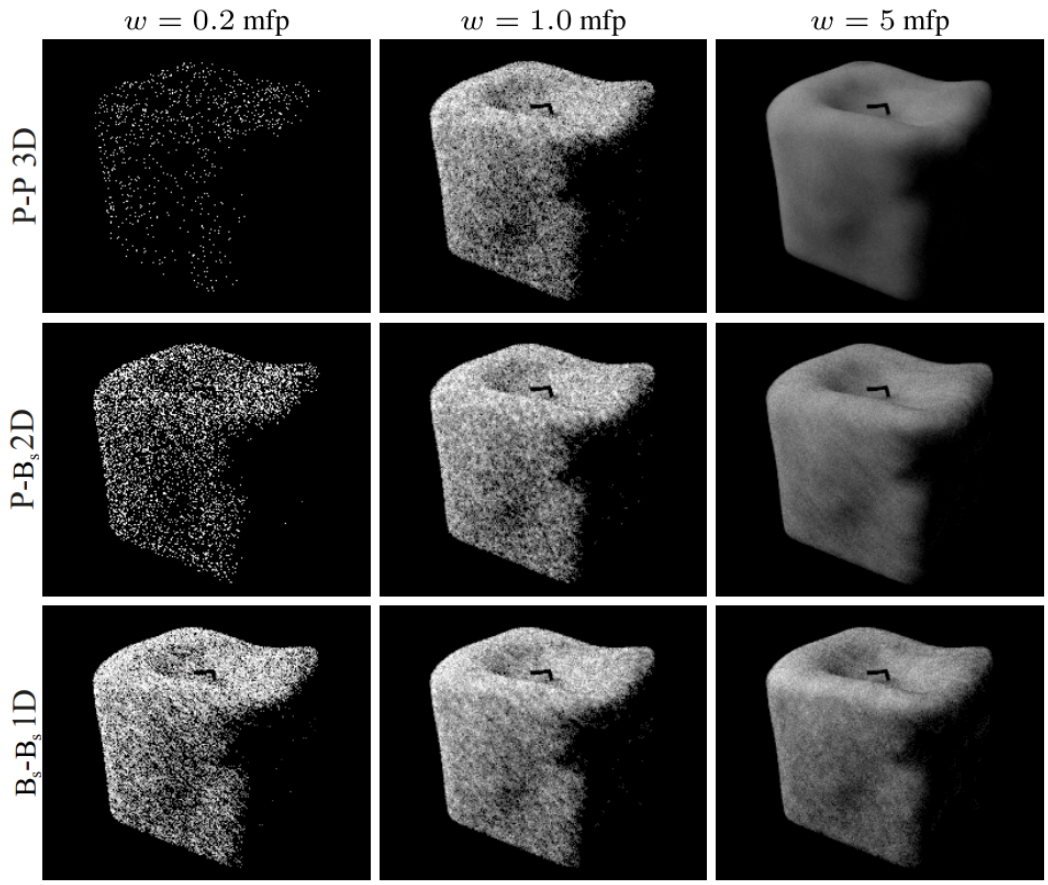


Figure 2.3: The results obtained using various estimators and kernel widths from Křivánek et al. [2014].

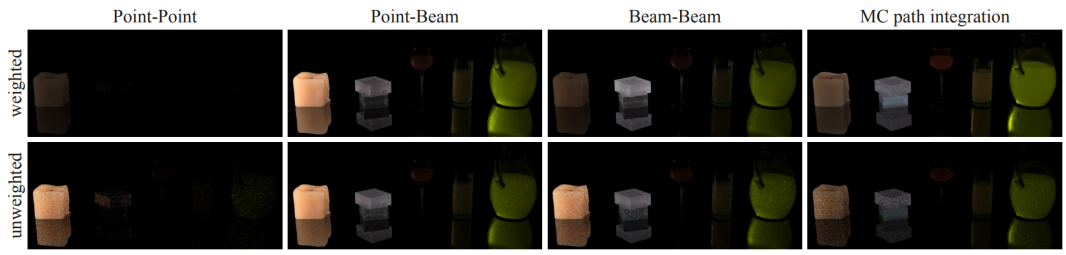


Figure 2.4: Contributions of the different estimators to the image. The bottom row shows the contributions without the MIS weighting. The image is from Křivánek et al. [2014].

3. Our work

At this point, it is clear that there are a variety of options when it comes to volumetric rendering. They possess different properties, making them suitable for different scene types and utilization. In this chapter, we describe the implementation and comparison of both path tracing and photon mapping and our custom improvements, everything in the context of usage in the 3D printing optimization. We derive our implementations from the default plugins in Mitsuba renderer [Jakob, 2010].

We focus on optimizations that are easily transferable to other renderers. Hence we do not spend much time either optimizing or describing suboptimal features of Mitsuba, like slow geometry intersection or unnecessary immense structures passed among the rendering plugins (intersection, medium scattering event and others).

First, we describe our evaluation process and testing scenes. Then we proceed with building the optimized algorithm from the bottom up - we start with the data storage and fetching, then we go through multiple implementations of distance sampling and transmittance evaluation. Finally, we design a specialized unidirectional path tracer. Additionally, we offer an evaluation of the UPBP algorithms [Křivánek et al., 2014] and implementation of the BRE approach [Jarosz et al., 2008] for heterogeneous media in Mitsuba and an assessment of its capabilities.

3.1 Results evaluation & testing set

An essential role in this thesis belongs to the system for results evaluation and comparison. It secures the ability to decide what are the strengths and shortcomings of various techniques. Based on that, we can tell which ones are worth a further investigation and participation on the final product and which are not. The main focus is to shorten the *rendering time*, decrease the *variance* and lower the *memory requirements*.

As for the *rendering time*, we are interested in the time spent on the actual rendering, not the overall runtime of the previewing or optimization processes. The main reason is that the processes involve voxelization of the 3D objects, halftoning, and other time-consuming tasks, which are not subject to optimization in this work, therefore they would distort the results. The only exceptions are the photon mapping and other two-phase algorithms, where also the photon gathering and follow-up lookup structure building is counted as the part of the rendering because it is not present in the basic unidirectional technique, making it a disadvantage of that group of algorithms.

All time measurements presented in this chapter are obtained using a laptop with AMD Ryzen 7 4800H with eight physical cores (16 logical cores) and 16GB of RAM. The rendering was allowed to use all cores, usually utilizing 97-100% of the CPU. Each test was run multiple times to average out any effects of OS scheduling and background tasks. In the section 4, we include measurements also on other machines.

The *variance* is computed from the image pixel values. As the mean value, we use images of the respective scene rendered with a high number of samples per

pixel (usually 10-50x more than the images tested for variance). We call them the ground truth images. The variance of the ground truth image introduces an error to the result variance. Therefore it needs to be as low as possible.

We also keep track of multiple other properties of the result images, such as quality and sharpness of the edges or local differences in the variance. We did not use any algorithm to quantify these characteristics, they were only evaluated visually.

The rendered image resolution is given by the surface voxel count in case of the camera that is used in the optimization loop. We include the numbers for particular testing models later. For the preview renders, the resolution is always noted for individual tests. The sample counts are also noted for the respective results - we often try to match rendering times or result variance of multiple different methods, so we use a variety of sample counts. We use the Mitsuba independent sampler [Jakob, 2010] (*independent* plugin) because the other plugins usually round the sample count (e.g., to the closest power of two). It is undesirable since we want to set a specific sample count (e.g., to match the rendering time of two methods).

The *memory consumption* is either calculated using the grid dimensions and the voxel memory footprint in the case of the dense grids or obtained from an OpenVDB grid API function.

The measurements were performed on a set of testing scenarios designed to vary in the properties that significantly influence the performance and the memory footprint of the rendering. These are the most significant ones

- *overall object size* - it influences the memory footprint of the respective voxel grid, as well as the resolution of the image rendered for the optimization (see the section 1.1)

- *object thickness* - allows longer light paths. In thin object, the rays resurface quickly cutting the rendering time

- *density of the material* - the density of the printer materials for various colors and color channels are different. For example, the red channel of magenta material has a density of $2.5mm^{-1}$, while the blue channel of white material has $24mm^{-1}$. Also, the transparent material has an approximate density of 0, but it was not included in our testing scenes. The higher density usually leads to higher rendering times since the algorithms sample the steps in the medium based on the density. Therefore the higher density leads to shorter steps. Hence, it takes more scattering events to travel the same distance in a dense medium

- *albedo* - the path tracing algorithm uses Russian Roulette to terminate rays with low throughput. In darker media, the rays lose throughput faster, and therefore they are terminated sooner, leading to shorter rendering times

- *shape of the objects* - rays shot towards thin parts of the objects tend to return to the surface sooner than rays penetrating thick parts. The basic unidirectional path tracer terminates the rays when they leave the object and head towards the environment. It spends less time tracing a particular ray when it leaves the object sooner

- *variance of the materials* - this concerns the algorithms that tend to degenerate the edges because their results generally look better when there are smooth color transitions or vast areas with a constant color

The testing set consists of the models in the Table 3.1. The voxel resolution provides the grid's actual size that is created by the optimization pipeline, with the printer voxel size 600 x 300 x 900 dpi. There is transparent padding around the grid with a thickness equal to 0.12mm, and the object is extruded a few layers during the voxelization. The listed sizes are the sizes of the underlying obj triangle meshes. The camera resolution is listed because it depends on the surface voxel count. The models are displayed in the Fig. 3.1.

Table 3.1: Testing models overview.

Model	*Size (mm)*	*Voxel res.*	*Vertices*	*Camera res.*
Box_white_10	10x10x10	246x124x372	8	410x410
Box_greek_10	10x10x10	246x124x372	8	410x410
Box_white_20	20x20x20	482x242x726	8	818x818
Box_greek_20	20x20x20	482x242x726	8	818x818
Animal_white	4.9x11.9x13.1	126x147x483	5850	248x248
Animal_spotted	4.9x11.9x13.1	126x147x483	5850	248x248
Bone	42x42x34.5	1002x502x1242	50k	920x920

(a) Box_x_white (b) Box_x_greek (c) Animal_white

(d) Animal_spotted (e) Bone

Figure 3.1: Preview of the testing models.

We created all the models ourselves, except the bone model, it is created by Artec Group inc. The greek and the spotted textures are from the websites https://es.123rf.com and https://www.dreamstime.com respectively.

Finally, we describe the default testing setup - the settings are used when not stated otherwise.

- *camera*: orthogonal
- *image resolution:* given by the model
- *sample count:* 100spp
- *texture extrusion:* 1mm
- *Mitsuba integrator:* volpath_simple

3.2 Memory optimization

A heterogeneous volume is usually described by a 3D grid containing values for particular quantities in each point of such a grid, called voxels. In our case, the quantities are albedo and density (see the section 2.1 for clarification). Unlike general-purpose solutions for storing heterogeneous volumes, our voxels only represent one of a finite set of materials. Therefore, we only have a finite set of albedo-density value combinations. Moreover, the set only consists of seven materials. This introduces excellent possibilities for memory saving.

3.2.1 Original solution

The research code utilizes the basic Mitsuba heterogeneous volume plugin [Jakob, 2010] (*gridvolume*) for both albedo and density. It stores the grids using dense representation. The default heterogeneous media plugin (*heterogeneous*) asserts that albedo is three-channel while density is one-channel. Both the media are expectedly using float values. The following calculation shows how many bytes such media occupy in memory.

Let v be the number of voxels (keep in mind that it is a large number since it is the product of voxel counts in each of the three dimensions), then the number of bytes B is

$$B = v * 3 * 4 + v * 1 * 4 = v * 16 \tag{3.1}$$

where the number four represents the usual amount of bytes used by a single-precision floating point number.

First, we include the memory necessary to store the grids describing the volume physical properties (Table 3.2). Note that it does not include all the other grids used by the optimization algorithm.

We also append the time measurements as we want to evaluate the rendering time differences among the various approaches (Table 3.3). Due to the time complexity, we do not provide measurements for all scenes in each set. We select a subset that covers all significant cases regarding the models' shapes and textures.

Table 3.2: Memory footprint in MB.

Model	*Memory*
Box_white_10	173
Box_greek_10	173
Box_white_20	1 292
Box_greek_20	1 292
Animal_white	137
Animal_spotted	137
Bone	9533

Table 3.3: Time measurements - original. The times are in seconds. We use the settings described in the section 3.1.

Model	*R*	*G*	*B*
Box_white_10	19.5	19.5	19.7
Box_greek_10	7.9	6.1	5.2
Box_greek_20	32.9	25.2	20.7
Animal_white	5.5	5.8	8.0
Animal_spotted	10.9	11.5	12.3

3.2.2 Optimization using material IDs

Since the volume can be fully described by the materials' IDs, we can create a grid where each voxel contains only an 8-bit ID. This way, the memory footprint of a grid with v voxels is only v bytes - it is 16-times less than the original. The material's physical properties can then be obtained using a lookup table for both albedo and density. The size of the table is negligible as we only have seven materials.

Apart from the apparent memory usage reduction, there is also another advantage - more voxel values fit the cache. Therefore fewer RAM accesses are necessary when the rendering algorithm does grid lookups close to each other.

On the other hand, an extra indirection occurred because of the lookup table. Fortunately, the lookup table is small and fits the cache easily. Furthermore, since the number of different material sets is fairly small and the grid lookup time is critical for the rendering's performance, we hardcode the table into the Mitsuba plugin, making the indirection even faster. In our implementation, the same binary of the plugin is used for each color channel. Therefore there is a switch deciding between values for red, green, and blue. It is another element that slows down the algorithm compared to the original solution.

Table 3.4: Memory footprint - original/IDs in MB.

Model	*Original*	*IDs*	*Improvement*
Box_white_10	173.1	10.8	x16.0
Box_greek_10	173.1	10.8	x16.0
Box_white_20	1292.2	80.8	x16.0
Box_greek_20	1292.2	80.8	x16.0
Animal_white	136.5	8.5	x16.0
Animal_spotted	136.5	8.5	x16.0
Bone	9532.6	596	x16.0

Table 3.5: Time measurements - original/dense+IDs. The times are in seconds. We use the settings described in the section 3.1.

	Original			*dense+IDs*			*Diff*		
Model	*R*	*G*	*B*	*R*	*G*	*B*	*R*	*G*	*B*
Box_white_10	19.5	19.5	19.7	20.0	20.0	20.1	+0.5	+0.5	+0.4
Box_greek_10	7.9	6.1	5.2	8.3	6.3	5.4	+0.4	-0.7	+0.2
Box_greek_20	32.9	25.2	20.7	34.2	26.5	21.8	+1.3	+1.3	+1.1
Animal_white	10.9	11.5	12.3	11.4	12.1	12.9	+0.5	+0.6	+0.6
Animal_spotted	5.5	5.8	8.0	5.7	6.1	8.3	+0.2	+0.3	+0.3

The memory footprint is expectedly reduced 16-times (Table 3.4). The difference between the sets of values is approximately 5% (Table 3.5), which is a great trade-off considering the 16-times lower memory consumption of the grids. We attribute the lower performance to the additional indirection, and conditional jump caused by the switch explained above.

3.2.3 Optimization using OpenVDB

The next stage of the optimization is based on the fact that the voxel grids tend to contain large segments with constant color. This is an implication of the fact that the volume is colored only up to a certain depth, and the rest of the interior is white. Also, there are cases when the texture contains areas with the same color. However, it is not that frequent, especially when the materials need to be constant between the layers, not only within a particular layer.

To store such grids, the sparse OpenVDB library [Museth et al., 2012] is a logical choice. It keeps the constant segments in larger, higher-level nodes of the tree, efficiently saving memory (see the section 1.3.2). The library allows defining custom configurations of the trees - with a varying number of layers and node sizes. Smaller nodes are better with volumes that contain smaller constant areas. On the other hand, they create too many nodes in volumes with larger constant areas. Similarly, there is a trade-off when it comes to the number of layers - more layers allow smaller nodes and better sparsity while slowing down the lookup since more nodes need to be traversed to get to the bottom of the tree.

In each case, the data stored in OpenVDB have a smaller memory footprint (Table 3.6). The most significant factors are the overall object size (the constant

Table 3.6: Memory footprint - original/OpenVDB in MB.

Model	*Original*	*OpenVDB*	*Improvement*
Box_white_10	173.1	3.6	x48.0
Box_greek_10	173.1	9.0	x19.2
Box_white_20	1292.2	18.7	x69.1
Box_greek_20	1292.2	39.7	x32.5
Animal_white	136.5	3.9	x35.0
Animal_spotted	136.5	4.7	x29.0
Bone	9532.6	91.4	x104.3

areas are more extensive, fitting more higher-level tree nodes) and the ratio of the bounding box filled by the mesh (the portion of the bounding box outside the mesh contains constant air).

Table 3.7: Time measurements - original/OpenVDB. The times are in seconds. We use the settings described in the section 3.1.

	Original			*OpenVDB*			*Diff*		
Model	*R*	*G*	*B*	*R*	*G*	*B*	*R*	*G*	*B*
Box_white_10	19.5	19.5	19.7	21.6	20.8	21.6	+2.1	+1.3	+1.9
Box_greek_10	7.9	6.1	5.2	10.8	7.7	7.0	+2.9	+1.6	+1.8
Box_greek_20	32.9	25.2	20.7	47.8	34.0	28.4	+14.9	+8.8	+7.7
Animal_white	10.9	11.5	12.3	10.5	12.4	13.2	-0.4	+0.9	+0.9
Animal_spotted	5.5	5.8	8.0	5.1	6.2	7.3	-0.4	+0.4	-0.7

The times are generally worse compared to the previous approach, especially in the second testing case (Table 3.7). However, the following chapters gradually improve the lookup performance of OpenVDB grids.

3.3 Grid lookup time optimization

The previous chapter lined up that the grid lookup is frequent and, therefore, a performance-critical operation. The use of a rather complex library, OpenVDB [Museth et al., 2012], gives us plenty of space for optimization. It concerns simplification of the world to grid transformation, which can also be also used in the dense 3D grid and OpenVDB-specific improvements like tweaking the grid layout.

3.3.1 Thread local accessors

Having the two options to obtain values from OpenVDB grids (accessors and grid samplers), we picked accessors since we implement the transformation from world space to grid space as well as nearest neighbor interpolation ourselves, in a more specialized and efficient manner. The grid samplers use the library's general implementations of these operations.

Initially, we used a single accessor stored as a member variable in the plugin class created in the constructor. It is important to note that Mitsuba splits

the rendered image into blocks (the default size is 32x32) and assigns them to rendering threads [Jakob, 2010]. It creates considerable distances between the pixels rendered at the same time by different threads. As a result, the caching of traversed nodes in a single accessor is ineffective.

We decided to improve this by having a dedicated accessor for each thread, using thread local storage. The results suggest that it helps, as the rendering times are comparable to those measured for the dense grid (Table 3.8). The most significant improvement is in the test with bigger and more colorful objects. It makes sense in terms of caches - bigger objects allowed lookups farther apart, and more detailed textures resulted in more smaller tree nodes, both causing the inefficiency of the caching in the single accessor. On the other hand, there is an overhead of the function that ensures that the accessor is created in the current thread - it has to be executed on each medium function call, since we do not employ optimizations further up the plugin structure so far.

Table 3.8: Time measurements - original/OpenVDB TL. The times are in seconds. We use the settings described in the section 3.1.

	Original			*OpenVDB TL*			*Diff*		
Model	*R*	*G*	*B*	*R*	*G*	*B*	*R*	*G*	*B*
Box_white_10	19.5	19.5	19.7	20.1	20.3	20.8	+0.6	+0.8	+1.1
Box_greek_10	7.9	6.1	5.2	8.4	6.3	5.5	+0.5	+0.2	+0.3
Box_greek_20	32.9	25.2	20.7	35.1	26.6	21.9	+2.2	+1.4	+1.2
Animal_white	10.9	11.5	12.3	11.2	12.3	13.0	+0.3	+0.8	+0.7
Animal_spotted	5.5	5.8	8.0	5.6	6.2	8.2	+0.1	+0.4	+0.2

3.3.2 World to grid transformation

The rendering uses the world coordinate system to describe object placement, distance computing as well as shooting and tracing of rays. When the volume tracing algorithm needs the density or albedo value at a certain point of the volume, the position of the point needs to be transformed from the world coordinates to the grid coordinates to determine the respective voxel or voxels and ask them for their value.

Usually, the grid transformation is represented by a transformation matrix used to multiply the point in world space to obtain its grid space coordinates. Here, we exploit one of the assumptions given by the circumstances of our usage. Even though the rotation is supported, it is baked into the object coordinates, and the world to grid transformation consists only of the translation and scale. These two operations can be achieved by simple addition and multiplication of the point coordinates vector. The point is a 3D vector. Therefore the two operations can be computed using SSE instructions, making it much faster than the matrix multiplication.

In order to achieve smooth transitions between the values in heterogeneous grids, the result value is usually computed by interpolating between the nearby voxels of the point in the grid. In our case, though, we use the nearest neighbor search. This saves us from inquiring values of multiple voxels and interpolating

between them. Usually, the nearest voxel is found by rounding each coordinate of the voxel in grid space. From our measurements, the round operation is quite expensive, so we replace it with a much cheaper integer cast in combination with moving the grid coordinate 0.5 voxels in each direction before the cast takes place. The operation can be baked into the aforementioned translation vector.

To clarify this further, we provide pseudocode for the initialization of the SSE vectors and their usage

```
void initialize() {
    Vector3 extents = gridAABB.size(); // World units
    Vector3i resolution = grid.resolution(); // Grid voxels

    __m128 scale = _mm_set_ps(resolution / extents);

    // Move in world space
    __m128 translation = _mm_set_ps(gridAABB.min);

    // Scale world -> grid space
    translation = _mm_mul_ps(translation, scale);

    // Move in grid space
    translation = _mm_add_ps(translation, 0.5);
}

uint8_t lookupLabel(Point p) {
    __m128 p128 = _mm_set_ps(p);
    p128 = _mm_mul_ps(p128, scale);
    p128 = _mm_add_ps(p128, translation);
    Point3i pCoord(static_cast<int>(p128));
    return grid.lookup(pCoord);
}
```

Table 3.9: Time measurements - original/dense+SSE. The times are in seconds. We use the settings described in the section 3.1.

	Original			*dense+SSE*			*Diff*		
Model	*R*	*G*	*B*	*R*	*G*	*B*	*R*	*G*	*B*
Box_white_10	19.5	19.5	19.7	19.2	19.3	20.4	-0.3	-0.2	+0.7
Box_greek_10	7.9	6.1	5.2	8.1	6.3	5.4	+0.2	+0.2	+0.2
Box_greek_20	32.9	25.2	20.7	33.0	25.5	20.9	+0.1	+0.3	+0.2
Animal_white	10.9	11.5	12.3	11.0	12.2	12.8	+0.1	+0.7	+0.5
Animal_spotted	5.5	5.8	8.0	5.6	6.2	8.2	+0.1	+0.4	+0.2

Table 3.10: Time measurements - original/OpenVDB TL+SSE. The times are in seconds. We use the settings described in the section 3.1.

	Original			*OpenVDB TL+SSE*			*Diff*		
Model	*R*	*G*	*B*	*R*	*G*	*B*	*R*	*G*	*B*
Box_white_10	19.5	19.5	19.7	19.6	19.9	20.0	+0.1	+0.4	+0.3
Box_greek_10	7.9	6.1	5.2	8.1	6.2	5.3	+0.2	+0.1	+0.1
Box_greek_20	32.9	25.2	20.7	33.9	26.3	22.2	+1.0	+1.1	+1.5
Animal_white	10.9	11.5	12.3	10.9	12.0	12.7	0.0	+0.5	+0.4
Animal_spotted	5.5	5.8	8.0	5.6	6.2	8.1	+0.1	+0.4	+0.1

Unsurprisingly, this optimization results in success (Tables 3.9 and 3.10). However, even if the actual transformation is much faster, the overall speedup is not that big because the portion of time the program spends on transformations is minor.

3.3.3 Finding the best tree layout

OpenVDB [Museth et al., 2012] allows to use either predefined or custom grids to store data. The primary option is a four-layer tree with one root, two internal, and one leaf layer. The node sizes are marked as a sequence of numbers (for four-layer trees, it is a triplet, e.g., $< 5, 4, 3 >$), where a particular number n means that the node contains an n-th power-of-two child nodes in each dimension. The first number corresponds to the first internal layer, the second number to the second internal layer, and the third number to the leaf node size. The root stores the child nodes in a hash map, so there is no number corresponding to the root node.

With the layout $< 5, 4, 3 >$, the leaf size is 2^3, the lowest internal layer contains 2^4 leaves in each dimension, therefore its size is $2^4 * 2^3 = 128$. The second internal layer's node size is similarly $2^5 * 128 = 4096$ in each dimension.

The default configuration is $< 5, 4, 3 >$. We tried multiple custom configurations to see which behaves the best with the object of sizes commonly used in our application. We assumed that the default config is such that it works reasonably well with most use cases, from small volumes to huge ones used in movie production. Therefore, if we create a layout specifically optimized for our sizes, it will work better.

We did not consider grids with a depth of more than four since our objects are relatively small, and an additional layer would slow down the traversal.

The crucial information is the sizes of the rendered grids, so we note them for each model (section 3.1). For simplicity, we provide only the results for the two models at the opposite sides of the model complexity and thickness spectrum - the miniature animal model and the cube with the size 20mm x 20mm x 20mm (Table 3.11).

Table 3.11: Time measurements - various OpenVDB grid layouts. The times are in seconds. We use the settings described in the section 3.1.

	box_20_greek			*animal_spotted*		
Layout	*R*	*G*	*B*	*R*	*G*	*B*
$< 3, 3 >$	34.8	26.6	24.5	5.9	6.4	8.1
$< 4, 3 >$	34.9	26.5	24.4	5.9	6.4	8.2
$< 5, 3 >$	34.0	26.5	24.0	5.7	6.2	8.0
$< 5, 4, 3 >$	33.9	26.3	22.2	5.6	6.2	8.1
$< 4, 3, 2 >$	35.0	26.4	24.3	6.0	6.3	8.4
$< 3, 2, 2 >$	34.8	26.7	24.4	6.0	6.4	8.3

The first three configurations have only three layers. We assumed that it would shorten the traversal time. On the other hand, the nodes in the topmost internal layer are considerably smaller than in the default case. It results in more elements in the hash table in the root node.

The size of the topmost internal node (in one dimension) in the first configuration is 64 (computed as 3 + 3 to the power of two), in the second configuration, it is 128, and in the third, it is 256. The number of elements in the root node cache is quite big for the cube case (we divide the cube grid dimensions with 64), and it is getting smaller for the other two cases. We see that the time is declining with the growing size of the nodes.

The other three configurations follow the same pattern. The configurations number three and four lead to an empty/almost empty cache in the root level node. The top-level internal nodes in the fourth configuration have a size of 4096 in each dimension, which is a lot bigger than our objects, so we assume it is not necessary to try any configurations with bigger nodes.

In conclusion, the best performing configuration is the $< 5, 4, 3 >$, while it is also the default configuration of the OpenVDB grids. We suspect that it was selected not only because it offers the best trade-off between the node sizes and the number of elements in the cache for most volumes but also because it exploits the constants in the cache hierarchy of modern computers.

At this point, the two approaches for storing the data - the dense grid and OpenVDB grid - reach similar rendering times, the OpenVDB grid is still slightly slower. However, its memory footprint can be multiple times smaller, so we opt for using that instead of the dense grid.

3.4 Using homogeneous media

Reading the grid values presents one of the significant problems of heterogeneous volume rendering. It overloads the memory with an immense number of accesses, plus in the case of non-trivial data structures like OpenVDB grids, it brings extra work tied with traversing those structures. We present various approaches to reduce the overhead, but it does not vanish completely.

On the other hand, the rendering of homogeneous media, by its very nature, does not have this problem. It uses the same values for albedo and density throughout the whole medium. Let us see what the difference between the rendering times achieved using the two types of media is. Note that we use single-

material volumes, so the two algorithms render the same thing. We use the white and cyan material properties, which are noted in the results table. This test is just for illustration, so we use only one model.

Table 3.12: Time measurements - homogeneous/heterogeneous. The model is box with the size 20x20x20. The times are in seconds. We use the settings described in the section 3.1.

	Homogeneous			*Heterogeneous*			*Diff homo/hetero*		
Model	*R*	*G*	*B*	*R*	*G*	*B*	*R*	*G*	*B*
Box_20 (white)	69.7	70.1	73.3	80.2	81.1	85.6	0.87x	0.87x	0.87x
Box_20 (cyan)	17.4	22.5	62.4	18.9	26.4	93.8	0.92x	0.85x	0.67x

We can see that there is a measurable advantage to using homogeneous media. Unfortunately, our application is focused on the optimization of textured objects' appearance. Therefore the use of heterogeneous media is crucial. However, the inside of the objects is usually white, and the varying colors are only occupying a few dozens of surface layers. Based on this fact, we present multiple approaches that approximate the white core of the object using homogeneous volume.

3.4.1 Inner geometry

OpenVDB [Museth et al., 2012] provides features that allow removing multiple layers of volume from the surface and then create a mesh based on the shrunk volume. This mesh can be placed inside the original mesh but with a homogeneous volume. The thickness of the heterogeneous layer can be controlled by the number of layers that are removed. We use only a single model, the white cube with the size 20mm x 20mm x 20mm, since it is enough to demonstrate the capabilities of the approach (Table 3.13).

Table 3.13: Time measurements - inner geometry. The times are in seconds. We use the settings described in the section 3.1, except the sample count is 4.

Approach	*R*	*G*	*B*	*Vertices*
Heterogeneous (OpenVDB)	3.2	3.2	3.3	8
Homogeneous	2.8	2.9	3.0	8
Dual (-1.2)	15.1	15.2	15.3	258 754
Dual (-0.6)	11.3	15.2	11.7	296 594
Dual (-0.3)	7.7	7.7	7.6	314 648
Dual (-0.1)	6.7	6.8	6.6	325 736
Only inner	68.2	69.5	63.7	333 234

We intentionally included numbers of vertices of the generated core objects since it is the primary factor influencing the results - the number of vertices is huge (consider that the original cube model has only eight vertices).

Moreover, the numbers also grow with the increasing size of the core (a thinner heterogeneous layer leads to a bigger inner core). This is a critical scaling issue because for the same model shape, either an unnecessarily detailed core model

is created when it is larger, or a too coarse model is created when it is smaller. It is a consequence of the fact that the model is created based on the voxel grid (that has a resolution dependent on the object size), not on the original triangle model.

To better understand the poor performance, we also measured the time for only the generated model filled with the white homogeneous medium. Its rendering time is multiple times higher than the reference times. It could be improved using a faster intersection search system (like Embree), but it is clear that this is not a correct approach.

The measured times decrease with a thinner heterogeneous layer, which is what we expected because the ray spends less time in the slower heterogeneous medium. However, we suspect that there is also another reason - the difference between references for the homogeneous and heterogeneous medium is about 20%. However, the difference between the thickest and the thinnest layers is almost twofold. It is probably an outcome of differences between the intersection searches in the two configurations, but we are not aware of the Mitsuba intersection algorithm details [Jakob, 2010].

There are also other potential problems, apart from poor performance. The first one is that if there are more colorful layers than the heterogeneous layer thickness threshold, they are not rendered. In this case, the optimization process would fail since it usually adds colors to lower layers to get more saturated colors. It would have to be either computed dynamically, based on the colors in the grid or set to a very conservative value. The second issue is that the approach relies on the correct reconstruction of the shrunk geometry in all possible use cases - it is a strong assumption regarding the fact that the optimization pipeline could be used in production.

3.4.2 Approximation using distance field and gradients

After the failure of the method using additional geometry, we provide a more lightweight solution using two additional grids - *distance grid* saving the distance to the closest surface for each voxel and *distance gradient grid*, which saves the gradients of the distance grid. The idea is that based on the distance to the surface, we decide whether the current scattering event is far enough to be in the homogeneous white core. The gradients are used to decide whether the sampled direction goes towards the surface or deeper inside the object to help the decision process and reduce bias. Unfortunately, even in the homogeneous core, we have to use the heterogeneous version of the transmittance evaluating function because it always reaches the surface layers - it is used for the direct illumination evaluation. In our case, the (environment) light is always outside the object, so it crosses the medium interface.

Due to the high frequency of the grid lookups, we inlined the OpenVDB grids into the medium plugin (even though the difference is negligible). For the measurements, we do the same in the Woodcock tracking plugin, so the comparison is fair.

We present two versions of the algorithm - the first is based solely on the distance to the surface, there is a hard threshold between the homogeneous and the heterogeneous medium. The advantage is that it only requires one additional

grid lookup compared to the heterogeneous algorithm. The disadvantage is the possible bias - first, there can be colorful voxels below the threshold.

Second, the sampled ray can point towards the surface, so even if it starts in the homogeneous part, the sampled point can be in the heterogeneous part. Therefore a fraction of the distance in the heterogeneous part is incorrectly considered homogeneous. For this reason, we add a maximum expected step length to the threshold. Of course, the step length is potentially infinite, but we determined the threshold in a way that is safe in 95% of cases - for the minimum white density, it is 0.25. For the depth threshold, we use the values 1.0 and 1.5, corresponding to 12 and 18 layers, respectively. The depth at dark areas and edges can be up to 20 layers, so the depth of 1.5 is still not enough, while the depth of 1.0 could be only used for bright textures with mild edges. We intentionally set the depth less conservatively so that the algorithm can compete with the purely heterogeneous medium.

To avoid the extension of the threshold, we employ the distance gradient grid, which tells us where the ray is headed. The most straightforward option would be first to check the threshold and then compare the gradient dot product and the ray direction with zero. If it is below zero, it goes inside the object. Otherwise, it goes back to the surface, and the volume has to be treated as heterogeneous. However, this way, we would throw away an opportunity to use homogeneous volumes with the rays going sideways and slightly to the surface. Therefore we use the following steps

```
cosine = cos(gradient, ray.d)
directDistanceToBorder = THRESHOLD - distance
realDistanceToBorder = directDistanceToBorder / cosine
isHomogeneous = distance < THRESHOLD &&
                (cosine <= 0.0 && realDistanceToBorder > MAX_STEP)
```

where the gradient and distance values are obtained from the grids, THRESHOLD, and MAX_STEP are constants described above.

This version requires additional lookups in two grids plus a few arithmetic instructions. To reduce the overhead, we merge the distance and the distance gradient grids into one grid containing four-element vectors (three for gradient, one for distance).

Unfortunately, using the distance gradient grid does not eradicate the bias as well. The problem is with curved surfaces, where the sideways going ray can enter the heterogeneous zone sooner than in the case of a flat surface. The bias there would be locally dependent, which is more dangerous for the optimization than the bias that is constant over the whole surface because it would incorrectly compensate for the dark/bright spots.

In the tests (Table 3.14), we use the white texture to match the white homogeneous core to get results that mirror the improvement/overhead of the method. If there were a colorful surface, the incorrect estimate of the homogeneous-heterogeneous interface would also distort the time measurements due to the lower albedo of colorful materials. Again, we use only the two contrasting models for simplicity.

Table 3.14: Time measurements - homogeneous core approximation. The ratio is homogeneous/heterogeneous calls. The approach configuration is noted as ($THRESHOLD$, MAX_DEPTH). We use the settings described in the section 3.1.

Model	*Approach*	*Time*			*Diff with base*			*Ratio*
		R	*G*	*B*	*R*	*G*	*B*	
Box_20_white	Heterogeneous	80.0	80.8	85.4	-	-	-	-
	Simple (1.0, 0.25)	83.1	83.6	83.7	+3.1	+2.8	-1.7	7.9
	Simple (1.5, 0.25)	83.4	83.5	83.8	+3.4	+2.7	-1.6	25.0
	Complex (1.0, 0.25)	84.0	83.6	84.6	+4.0	+2.8	-0.8	5.0
	Complex (1.5, 0.25)	86.1	85.9	85.2	+6.1	+5.1	-0.2	15.5
Animal_white	Heterogeneous	10.8	12.0	12.6	-	-	-	-
	Simple (1.0, 0.25)	11.2	12.0	13.2	+0.4	0.0	+0.6	46.0
	Simple (1.5, 0.25)	11.0	12.0	13.0	+0.2	0.0	+0.4	670.0
	Complex (1.0, 0.25)	11.4	12.6	13.2	+0.6	+0.6	+0.6	26.0
	Complex (1.5, 0.25)	11.5	12.7	13.4	+0.7	+0.7	+0.8	230.5

We include the ratios of the homogeneous and heterogeneous calls. Expectedly, the depth threshold 1.0 leads to more homogeneous calls than 1.5. Similarly, the more sophisticated version of the algorithm increases the portion of such calls.

However, none of the algorithms matched the reference times set by the pure heterogeneous medium algorithm. The gain from the occasional homogeneous version calls does not make up for the overhead brought by the additional grid lookups and arithmetic operations. The necessary heterogeneous layer is too thick - the best achieved ratio is $1 : 5$, so the homogeneous calls occur only in less than 20% of cases.

3.5 Regular tracking

Regular tracking is usually in the shadow of the modern heterogeneous medium tracking algorithms due to the properties mentioned in the section 2.1.6 - it is considerably slower, and it has higher requirements on the medium (piecewise constantness) and on the API of the medium implementation (it needs a function that can query the piecewise constant areas along the ray).

However, we have a considerable advantage. Having the OpenVDB grids under complete control, we can assume that the volume is piecewise constant (the grid has a finite resolution). Therefore, we can determine the constant parts of the volume, which can be either voxels or nodes in the tree. Therefore, it allows us to use the works of Amanatides and Woo [1987], and Museth [2014], who propose methods for simple and hierarchical grids traversal, respectively. We plan to take advantage of the fact that OpenVDB trees can contain constant values higher up in the hierarchy [Museth et al., 2012], therefore for the whole area respective to the constant node, only one transmittance computation is necessary instead of one for each voxel in the node.

This algorithm has one property that other mainstream algorithms lack - it computes the transmittance precisely (unlike, e.g., ray marching or woodcock tracking). It could help us reduce the variance. This property is particularly

interesting for evaluating the transmittance between two points in the medium because the basic Woodcock algorithm provides only a binary answer - the transmittance is either 0 or 1, depending on whether a scattering event occurs during the tracking on the ray segment or not. Ratio Tracking [Novák et al., 2014] is an improvement, but it is under a patent.

It will become clear from the measurements that bigger constant tiles are not very frequent. The additional overhead of dealing with the varying size of constant areas in the volume obliterates the occasional gain from constant tiles. However, we value the ability to evaluate the transmittance of a ray segment, so we also implement the simple version of the traversal algorithm proposed by Amanatides and Woo [1987].

3.5.1 DDA & HDDA in OpenVDB

Both the simple [Amanatides and Woo, 1987] and the hierarchical [Museth, 2014] algorithms are implemented in the OpenVDB library [Museth et al., 2012] - they are marked as DDA and HDDA, respectively. However, the HDDA algorithm is provided only as an algorithm for finding ray segments inside the active part of the volume. We need to distinguish not only active and inactive voxels but also the density of the active voxels. Therefore we only use this implementation as an inspiration, while the DDA is used without major changes. Now, let us describe these algorithms in more detail.

DDA

DDA can be split into two parts - initialization and step. For better understanding, we provide pseudo-code for both.

```
init(ray)
{
    t = 0
    voxel = castToVector3i(ray.origin)
    for (axis in x,y,z)
    {
        if (ray.dir[axis] > 0)
        {
            next[axis] = (voxel[axis] + 1 - ray.origin[axis]) /
                         ray.dir[axis]
            step[axis] = 1
        }
        else
        {
            next[axis] = (voxel[axis] - ray.origin[axis]) /
                         ray.dir[axis]
            step[axis] = -1
        }
        delta[axis] = step[axis] / ray.dir[axis]
    }
}
```

The *voxel* variable holds the current voxel in grid coordinates. Note that the grid points are treated as the voxel centers, not the corners of the voxel cube. The *step* vector holds the voxel index change in the case of a movement in a particular axis. The *delta* vector represents the portion of the ray (in t units) traversed when the algorithm moves one voxel in the respective axis. The *next* vector holds the distance (in t units) that the vector needs to traverse to reach the closest voxel wall in the respective axis (e.g., in the z-axis, the closest upper wall needs to be reached).

```
step()
{
    stepAxis = minIndex(next)
    tOld = t
    next[stepAxis] += delta[stepAxis]
    voxel[stepAxis] += step[stepAxis]

    return t - tOld
}
```

The *step* function is straightforward. It first selects the *stepAxis* as the *next* vector element, which has the lowest value. It means that the ray starting at the position *ray(t)* and direction *ray.dir* has the shortest distance to the voxel boundary in the selected axis. Then the *next* vector and *voxel* are updated according to the selected axis. The function returns the distance traveled in the units of the ray direction length.

The *step* function is lightweight, and it provides high performance since it only consists of a few arithmetic operations.

HDDA

OpenVDB [Museth et al., 2012] provides functions to search for a tree node of a given depth level containing a particular voxel. The HDDA uses the function to check whether such a node exists in the tree. If it does, the node is not constant, and the HDDA needs to continue on a lower tree level. Otherwise, DDA is called on the current level, meaning that it has a voxel size corresponding to the current level's node size.

The algorithm's details are not relevant to us (since it is used only for active voxel lookup). It is also quite complicated (the calls to lower levels are done using a recursive call to an instance templated to the next lower level of the tree), so we are not providing pseudo-code here.

3.5.2 Our implementation

Each Mitsuba medium has to offer two crucial functions - *evalTransmittance* for evaluation of transmittance on a given ray segment and *sampleDistance* for sampling a scattering event along the given ray direction [Jakob, 2010]. As previously mentioned, we offer two versions of the regular tracking algorithm, the simple version using the basic DDA and the hierarchical version.

Simple version

Both the functions create a DDA instance with the given ray origin and direction and then call the *step* function in a loop until one of the finishing criteria is met. Having transformed the (normalized) ray from the world to grid coordinates without follow-up normalization ensures that the distance in t units returned from the *step* function corresponds to the real distance in the world coordinates. Therefore, the ray segment's transmittance corresponding to the step can be obtained as an exponent of the distance and the voxel density.

The *evaluateTransmittance* function accumulates the transmittance over the whole ray segment, so the loop is stopped only when the DDA oversteps the ray *maximum* t value.

The *sampleDistance* function first randomly selects a desired transmittance according to the exponential distribution. Then it tries to locate the point such that the transmittance between the ray origin and this point equals to the desired transmittance. Again, the DDA is used, but with one difference - there are two stopping criteria (either the desired transmittance is met, or we find out that the ray is not long enough to gather such transmittance).

Hierarchical version

Same as the OpenVDB HDDA algorithm [Museth et al., 2012], we also take advantage of the function for finding a node on a specific layer of the tree containing a given voxel. However, we start from the leaf layer and move only to the lowest internal node layer in case of failure. When the searches in the two lowest layers do not find any node, we claim that the current constant area has only the size of one unit voxel. There are multiple reasons for this. The most important one is that the recursive multi-layer approach is not worth the overhead it brings since the nodes of the higher layers are rare in our volumes. For example, in the classic $< 5, 4, 3 >$ configuration, the size of the second smallest internal node is 128x128x128. Even if there were such a large constant node in the volume, the *sampleDistance* function would not traverse the whole node since the sampled distance usually covers less than ten voxels (the large constant areas are mostly white, and the white material has very high density). Although this might save considerable time in the *evalTransmittance* function, we eventually optimize-out the calls of this function for larger distances in the path tracer.

Hence the tree probing has only three possible outcomes

- the probe function finds a node on the leaf layer, meaning that the portion of the volume respective to this node is not constant, and it will be traversed using the basic DDA
- leaf node is not found while the node in the next higher layer is found, meaning that there is a constant chunk of the volume with the size of leaf node - its transmittance can be therefore computed in closed form knowing the ray segment length that we obtain using a single-step version of the DDA algorithm
- both the nodes are not found, leading us to the constant area with unit voxel size where we also can compute the transmittance in the same way as above

Apart from this, the algorithm is similar to the simple version.

3.5.3 Measurements

Unlike the previous chapters, here we also measure the variance of the images along with the rendering times. In theory, regular tracking should perform better than the conventional algorithms because the distance sampling is exactly proportional to the transmittance and the transmittance evaluation is precise [Novák et al., 2018]. In order to have a clear view of the regular tracking performance, we provide analysis of the call frequency ratio between the *sampleDistance* and *evalTransmittance* methods and measurements of the average distance traversed in the functions. We use multiple OpenVDB layouts as well.

We use the cube with the size 20x20x20 millimeters, a rather thick model with little detail and no thin parts other than corners, so the rays are getting deeper inside the volume. Second, we employ the small animal model, which has much thin geometry, and the ray cannot get deep inside the medium. To observe the effect of density and albedo, we use entirely white and colored textures for both models.

Function call frequency

The two functions, *sampleDistance* and *evalTransmittance*, are used for different purposes. Measuring their call frequency, we see the influence they have on the result rendering times. We use two of the basic volumetric path tracing algorithms provided by Mitsuba [Jakob, 2010], *volpath*, and *volpath_simple*.

In the case of *volpath_simple*, the call ratio of evalTransmittance and sampleDistance is roughly 1 to 5M (box_20_white) and 1 to 100k (animal_spotted). In the case of *volpath*, the calls are roughly 1 to 1.

The *sampleDistance* function is used to determine how far is the next scattering event. This is the basic building block of volumetric path tracing. Hence it is frequently used in both implementations.

The *evalTransmittance* function is used in the next event estimation (NEE) algorithms to compute the transmittance from the current scattering point to the light source. The volpath algorithm uses the NEE as a part of the MIS. We can see that both the functions are called roughly with the same frequency. On the other hand, the *volpath_simple* algorithm does not implement MIS, and the *evalTransmittance* is called only on exceptional occasions.

Mean ray segment length

The distance traversed in the medium and the consequent number of voxels has arguably a massive influence on the overall runtime. To better understand the measured rendering time results, we start with the analysis of this distance. Based on the measurements from the previous chapter, we only use the *volpath* plugin - the *volpath_simple* uses the *evalTransmittance* function only occasionally, so we would not get realistic results.

Table 3.15: Mean traversed distance in mm (Red channel). We use the settings described in the section 3.1.

Model	*evalTransmittance*	*sampleDistance*
Box_white_20	8.63	0.15
Box_greek_20	10.82	0.20
Animal_white	1.9	0.14
Animal_spotted	2.0	0.15

The results are somewhat intuitive (Table 3.15). The sampled distance is proportional to the transmittance. Therefore the values are relatively low because of the high material density. On the other hand, the *evalTransmittance* function is called from arbitrary points in the medium, so the distance to the surface may be huge. We can also see that the distance measured for this function is much higher for the box model. As we already mentioned, the reason is that the model is thicker, allowing the rays to go further from the surface. Another interesting observation is that the distances are always higher with colorful textures. It is expected for the *sampleDistance* function simply because the white material has a higher density than the other materials. At first glance, it is not clear why the distances are higher for the colorful textures even for the other function - the distance there depends on the distance to the surface, it is not determined by the density in any way. We believe that the reason is that in a less dense medium, the ray will get deeper inside the medium sooner, increasing the average distance to the surface.

Now let us consider Beer's law. It states that in a homogeneous medium, the transmittance between two points equals $e^{-\sigma_t d}$. The density of the white material for the R, G, and B channels is 6.0, 9.0, and 24.0 (mm^{-1}), respectively. Even for the least dense channel (red), with the average evaluated distance of the *evalTransmittance* function being roughly 2.0, the transmittance is approximately 6.10^{-6}. It is clear that the contribution of the function is negligible while the computation time for such a long distance is vast. Therefore we decided to measure the rendering times using only the *volpath_simple* plugin, which practically does not call this function. Of course, we could engineer a simple heuristic that would decide whether it is worth calling this function based on the distance. However, we rather postpone this to the chapter about the optimization of the path tracing algorithm. There we deal with the optimal usage of this function with regard to its contribution to the overall result. The measurements comparing both the regular tracking approaches combined with a path tracer that frequently calls *evalTransmittance* are in the section 4.4.

Variance levels measurement

In this chapter, we present results of variance level measurements comparing the regular tracking with the Woodcock tracking. The variance levels are the same regardless of the regular tracking algorithm (simple or hierarchical), so we provide only one set of results. The ground truth image was rendered using 1000spp.

Table 3.16: Image variance (Red channel). We use the settings described in the section 3.1.

Model	*Regular tracking*	*Woodcock tracking*
Box_white_20	0.0318510	0.0319084
Box_greek_20	0.0005042	0.0005042
Animal_white	0.0141845	0.0154023
Animal_spotted	0.0008501	0.0008542

The results show that the regular tracking is slightly better in the case of colored textures (Table 3.16). It is expected since the regular tracking precisely captures the colors of all voxels along the ray. In completely white volume, the algorithms achieve similar results. Note that only the behavior of *sampleDistance* function was observed here. The *evalTransmittance* is not used as we do not have a path tracing algorithm that would use it in an optimal way yet. We will provide a comparison of this function's performance later (see the section 4).

Time measurements

Finally, we provide the measurements of regular tracking rendering times and their comparison to those achieved using Woodcock tracking (Tables 3.17 and 3.18). We measure both the simple and the hierarchical implementation of the regular tracking algorithm. For the hierarchical implementation, we try various OpenVDB grid layouts. To better understand the results, we also provide a frequency observation of the three tree probing function outcomes (see the Hierarchical version chapter). The third case is rare (rounded to 0%), so we do not include it in the table.

Same as in the section 3.4.2, here we also inline the OpenVDB grid into the medium plugins.

Table 3.17: Different OpenVDB grid layouts - hierarchical regular tracking. The times are in seconds. The last two columns are the percentages of the tree probing function outcomes (see the section 3.5.2). We use the settings described in the section 3.1.

		Time			*Tree probing case*	
Model	*Layout*	*R*	*G*	*B*	*DDA*	*Tile*
Box_white_20	$< 5, 4, 4 >$	103.2	105.1	110.7	60.0%	40.0%
	$< 5, 4, 3 >$	99.5	102.2	104.5	28.0%	72.0%
	$< 5, 4, 2 >$	96.3	101.7	101.4	8.0%	92.0%
Box_greek_20	$< 5, 4, 4 >$	44.3	34.3	28.9	85.0%	15.0%
	$< 5, 4, 3 >$	47.9	37.2	31.5	84.0%	16.0%
	$< 5, 4, 2 >$	50.6	39.1	32.0	84.0%	16.0%
Animal_white	$< 5, 4, 4 >$	13.1	14.1	15.4	80.0%	20.0%
	$< 5, 4, 3 >$	12.7	13.9	15.1	54.0%	46.0%
	$< 5, 4, 2 >$	13.0	13.7	14.9	41.0%	59.0%
Animal_spotted	$< 5, 4, 4 >$	6.2	6.8	9.1	99.9%	0.1%
	$< 5, 4, 3 >$	6.3	6.9	8.9	99.5%	0.5%
	$< 5, 4, 2 >$	6.1	6.8	8.9	76.0%	24.0%

Table 3.18: Time measurements - Regular tracking/Simple regular tracking/ Woodcock tracking. The times are in seconds. We use the settings described in the section 3.1.

		Time			*Diff with base*		
Model	*Algorithm*	*R*	*G*	*B*	*R*	*G*	*B*
Box_20_white	Woodcock	80.0	80.8	85.4	-	-	-
	Regular simple	87.6	90.4	88.0	+7.6	+9.6	+2.6
	Regular	99.5	102.2	104.5	+19.5	+21.4	+19.1
Box_20_greek	Woodcock	33.7	26.1	22.0	-	-	-
	Regular simple	37.6	28.8	22.8	+3.9	+2.7	+0.8
	Regular	47.9	37.2	31.5	+14.2	+11.1	+9.5
Animal_white	Woodcock	10.8	12.0	12.6	-	-	-
	Regular simple	11.8	13.0	14.2	+1.0	+1.0	+1.6
	Regular	12.7	13.9	15.1	+1.9	+1.9	+2.5
Animal_spotted	Woodcock	5.5	6.1	8.1	-	-	-
	Regular simple	6.2	6.7	8.2	+0.7	+0.6	+0.1
	Regular	6.3	6.9	8.9	+0.8	+0.8	+0.8

The frequencies of constant vs. non-constant node occurrences in the tree probing function behave as expected. Smaller leaf nodes have a bigger chance to be constant. Also, less detailed or single-color textures lead to more constant nodes. Similarly, a grid filling a less complicated object tends to contain more constant nodes. The animal model with the spotted texture allowed more constant nodes when their size got smaller since the texture has quite large plain areas. However, the greek vase texture is not friendly to constant nodes of any size because of the high frequency of its details.

In all layout tests, there are only a few cases where neither the leaf node nor the lowest internal node is found, meaning that there are no higher-level nodes and very few edge cases. We suppose that the OpenVDB tree pruning function [Museth et al., 2012] does not create higher-level nodes. We verified it using the $< 5, 2, 2 >$ configuration, where we added a branch inquiring about the level 2 node, in case the level 0 and level 1 nodes were not found - the level 2 node search never succeeded (the results for the $< 5, 2, 2 >$ config are the same as for the $< 5, 4, 2 >$ config, so we do not include them in the table). We decided not to pursue this further since the algorithm cannot compete with the simple DDA algorithm regardless.

Regarding the time measurements, there is no outright winner among the grid layouts. In general, more constant nodes improve the performance since multiple voxels can be skipped, and the transmittance of the ray segment inside the whole node can be computed analytically. On the other hand, the smaller the constant nodes are, the fewer voxels are skipped. Based on the measurements, we can conclude that these two implications go against each other. For example, the $< 5, 4, 2 >$ layout with leaf nodes of the size 4x4x4 performs better with the animal model with the white texture because of the higher constant node ratio (compared to the default $< 5, 4, 3 >$ layout). However, it also performs worse with the cube model with the detailed texture because the constant node rate does not improve, while the performance gain brought by traversing a single constant node is reduced. In conclusion, unless we know the exact use case in

terms of texture and geometry details, we cannot pick one of these layouts.

Compared to the simple regular tracking implementation, all of the layouts perform worse. The difference is more significant when the colored texture is used, as there is not much gain from the constant nodes while the overhead of the whole mechanism is still present. Similarly, both the regular tracking implementations are slower than Woodcock tracking.

3.6 Path tracing optimization

General-purpose renderers, including Mitsuba [Jakob, 2010], implement path tracing in a way that it renders correctly basically any scene, and it is optimized in a way that most of the common cases are reasonably fast. The consequences are that the code needs to handle multiple exceptional cases. There are various techniques where not all of them are efficient in particular use cases. The configuration constants are selected in such a way that they work well with most scenes, but they are not optimal in any of them. On the other hand, we can afford multiple strong assumptions about the rendered scene (see the section 1.1.2.

Based on these, we develop a Mitsuba plugin focused primarily on the rendering of scenes that comply with these assumptions. We will describe the particular improvements step by step while proving their contribution using various measurements. The target is cutting the rendering time, suppressing the variance, and increasing the overall image quality.

3.6.1 General simplifications and improvements

We base our plugin on the Mitsuba plugin *volpath_simple* [Jakob, 2010], a simple unidirectional path tracer without the MIS. The first step is to remove the unnecessary versatility that Mitsuba offers and use assumptions provided by our single-object scenes. Combining this with a few other simplifications, we get a more compact code where we implement more complex optimization ideas. In this chapter, we describe the changes incrementally and then provide the pseudocode of the final algorithm as well as the performance comparison with the original plugin.

Removing unnecessary branching and radiance query versatility

The plugin offers a system of enums that allow customizing the radiance query by turning on/off the direct and indirect illumination or emission. We do not need this functionality, so we remove it and save a few conditional jumps in the code. Furthermore, we get rid of the complicated system of depth computation - according to the code commentary, it is engineered to match the output of other integrators. It is not necessary for us since we use very high maximum depths (> 100). Finally, we remove the direct illumination computation entirely until we find an efficient heuristic for transmittance evaluation in such dense media.

No reentering object specialization

The path of a ray traced with no reentering can be split into two parts

- *outside the object* - single path segment because there are no obstacles between the camera and the object. Of course, not all rays hit the object. In case a ray misses the object, the tracing finishes right away.
- *inside the object* - usually multiple segments, the tracing ends with the ray hitting (the inner side of) the geometry and refracting outside. When the ray reflects on the object interface, the tracing continues.

Consequently, we know exactly when we hit the inner and when the outer side of the object without testing. It is also unnecessary to compute the reflected ray on the outer side and the refracted ray on the inner side because these cases result in the ray termination.

Note that this modification slightly changes the results, more in the section 4.4.

BSDF and phase function inlining

Mitsuba BSDF plugins [Jakob, 2010] provide the *sample* function that takes intersection data (and the random sampler). Simultaneously, the output is the number of variables, among which is the output ray direction and the corresponding pdf. In our conditions, the (smooth) *dielectric* plugin is the best choice. It computes the Fresnel reflectance. Based on that, it randomly decides between reflection and refraction, computing the outgoing ray using a simple geometric formula as the surface is perfectly smooth.

It seems that it cannot get easier than this. However, we split this and first compute the Fresnel reflectance. Then we decide whether the ray should continue (as described in the previous chapter), and only if it does, we compute the outgoing direction. For this, we need to violate the strict plugin structure of Mitsuba and directly use the functions for Fresnel evaluation and reflection/refraction computation. As a side effect, we dodge the filling of the Mitsuba BSDF record with unnecessary data and the virtual plugin call.

Similarly, we inline the functions for the Heyney-Greenstein phase function evaluation. Besides the advantages applying to the BSDF case, the sampling function always returns 1, so we can avoid the throughput multiplication.

Coordinates transformation

The phase function sampling needs to build the orthonormal basis to transform the sampled direction from the local to the world coordinate system. Profiling the CPU usage during the rendering, we learned that this operation is quite costly. We replaced the basic formula for creating the orthonormal basis with the improved version of the Frisvald formula proposed by Duff et al. [2017].

The pseudocode of the modified path tracing algorithm looks as follows.

```
if (!rayIntersect(ray, its))
    return irradiance // missed the object

<compute Fresnel F for the incoming ray>
if (nextRandom() < F)
    return irradiance // reflect back outside
```

```
ray = refract(ray)
medium = its.getInsideMedium()

while (depth++ < maxDepth)
    rayIntersect(ray, its)
    ray.maxt = its.t // sample only up to the inner surface hit

    // sample medium
    if (medium->sampleDistance(ray, mediumRecord)
        throughput *= mediumRecord.sigmaS *
                      mediumRecord.Tr /
                      mediumRecord.pdfSuccess

        phaseRecord = phase.sample(ray.d)
        ray = Ray(mediumRecord.p, phaseRecord.wo)
    else // hit inner surface
        throughput *= mediumRecord.transmittance /
                      mediumRecord.pdfFailure

        <compute Fresnel F>
        if (nextRandom() > F)
            Li += throughput * irradiance // exit the object

        ray = reflect(ray)

    <Russian roulette based on the throughput>

return Li
```

Note that it yields the same results as the original plugin (up to some differences listed in the section 4.4, which do not influence the variance level), so it is unnecessary to compare variance.

Table 3.19: Time measurements - Original/optimized path tracing. The times are in seconds. Settings - from the section 3.1, except we use the custom integrator, Woodcock tracking with the inline OpenVDB grid as the medium plugin.

	Original			*Optimized*			*Speedup*		
Model	*R*	*G*	*B*	*R*	*G*	*B*	*R*	*G*	*B*
Box_white_20	80.0	80.8	85.4	44.0	43.1	43.9	x1.81	x1.87	x1.95
Box_greek_20	33.7	26.1	22.0	18.3	12.9	8.9	x1.84	x2.02	x2.47
Box_white_10	19.4	19.7	20.0	10.4	10.4	11.0	x1.87	x1.89	x1.82
Box_greek_10	8.0	6.2	5.3	4.4	3.2	2.3	x1.82	x1.94	x2.30
Animal_white	10.8	12.0	12.6	5.7	6.0	6.7	x1.89	x2.00	x1.88
Animal_spotted	5.6	6.1	8.1	2.8	3.0	3.9	x2.00	x2.03	x2.08

The performance of the specialized path tracer easily outperforms the original one in each testing setup, irrespective of geometry or textures (Table 3.19).

Note that this is the first time the optimizations put restrictions on the scene or plugins. Up to this point, the optimized plugins could render any valid Mitsuba scene [Jakob, 2010] correctly. Here, we can only render scenes with a limited geometry content with a smooth dielectric surface and a medium with the Hayney-Greenstein phase function. Some of these restrictions can be lifted while keeping the performance benefits of the other improvements (e.g., enabling rough surfaces). The others (like allowing nested geometry or supporting self-shadowing) do not make sense in the kind of rendering that we do.

3.6.2 Russian roulette

The algorithm presented in the previous chapter gets the radiance contribution only in two cases - when the ray reflects at the first intersection or when it reaches the surface from the inner side and refracts outside of the object. The first case's occurrence is very low, considering that the Fresnel term for perpendicular rays (the rendering used for the optimization shoots rays orthogonal to the surface) is only about 0.04.

Let us look at the average length of the path that reached the surface using the default Russian roulette setup, which computes the survival probability as the current throughput clamped to [0.0, 0.95]. We measure the average depth of the exiting path and the path reaching the surface for the first time - the ray may reflect back inside without gaining any radiance, so the second numbers are lower (Table 3.20).

Table 3.20: Average path length (number of scattering events) - medium exit or first inner surface hit. Settings - from the section 3.1, except we use the custom integrator, Woodcock tracking with the inline OpenVDB grid as the medium plugin.

Model	*Medium exit*	*First inner surface hit*
Box_white_20	11.27	7.60
Box_greek_20	4.98	3.74
Animal_white	10.41	6.81
Animal_spotted	5.96	4.30

Note that a ray with a substantial throughput ($1 <$) in the white cube has only a 68% ($0.95^{7.6}$) chance of reaching the surface and only a 56% ($0.95^{11.27}$, not considering the inner reflection probabilities) chance of bringing a contribution because of the Russian roulette clipping. Based on this knowledge, we test multiple configurations of the Russian roulette with the target to minimize the terminating of high-throughput rays before they manage to yield a contribution (Tables 3.21 and 3.22).

Table 3.21: Time measurements - various Russina roulette configs. The times are in seconds. Settings - from the section 3.1, except we use the custom integrator, Woodcock tracking with the inline OpenVDB grid as the medium plugin.

		Time			*Slowdown*		
Model	*Config*	*R*	*G*	*B*	*R*	*G*	*B*
Box_white_20	0.95	44.0	43.1	43.9	-	-	-
	0.99	99.6	101.2	102.8	x2.26	x2.35	x2.34
	0.99, min 10	110.5	112.4	115.2	x2.51	x2.61	x2.62
Box_greek_20	0.95	18.3	12.9	8.9	-	-	-
	0.99	24.8	14.8	9.2	x1.36	x1.15	x1.03
	0.99, min 10	37.1	19.9	13.2	x2.03	x1.54	1.48
Animal_white	0.95	5.7	6.0	6.7	-	-	-
	0.99	10.8	12.8	15.3	x1.89	x2.13	x2.28
	0.99, min 10	12.3	14.2	16.9	x2.16	x2.37	x2.52
Animal_spotted	0.95	2.8	3.0	3.9	-	-	-
	0.99	3.2	3.8	6.3	x1.14	x1.27	x1.62
	0.99, min 10	4.6	5.7	9.5	x1.64	x1.90	x2.44

Table 3.22: Variance levels - various Russian roulette configs (red channel). The variance improvement is for the equal rendering time. Settings - from the section 3.1, except we use the custom integrator, Woodcock tracking with the inline OpenVDB grid as the medium plugin.

Config	*Model*	*Variance*	*Variance improvement*
Box_white_20	0.95	0.0319084	-
	0.99	0.0013460	x10.49
	0.99, min 10	0.0011269	x11.28
Box_greek_20	0.95	0.0005042	-
	0.99	0.0003204	x1.15
	0.99, min 10	0.0002612	x0.95
Animal_white	0.95	0.0154023	-
	0.99	0.0010715	x7.61
	0.99, min 10	0.0008714	x8.18
Animal_spotted	0.95	0.0008542	-
	0.99	0.0005238	x1.43
	0.99, min 10	0.0004512	x1.15

In most cases, the algorithms with longer paths achieve better results than the original configuration. The difference is more significant in the cases where the average path length is longer. It is expected as the probability of reaching the surface is lower, and the less strict Russian roulette has more room for improvement. Furthermore, we observe that the configuration which does not apply the Russian roulette at all in the first ten iterations is slightly worse than the one with the same maximum probability without that, in case of colored textures. We assume that it is because while the $< 0.99 >$ configuration sets only the upper limit for the probability, effectively terminating rays with low throughput, the $< 0.99,\ min\ 10 >$ configuration lets them continue for the first ten iterations even with low throughput, leading to the lower effectivity.

We will compare the visual quality of the images in the section 4 since there are more improvements in the following chapters that reach the same variance level using fewer samples per pixel. There may be a concern that using fewer samples could lower image quality even if the variance stays the same.

3.6.3 Fresnel term

We can take the path prolonging even further with the following idea. From the average depth measurements in the previous chapter, it is evident that even when a ray reaches the surface, it does not always acquire a contribution because it reflects back inside. However, instead of randomly reflecting or refracting, we can do both and weigh the contributions of both options by the Fresnel term. Note that this does not lead to an actual branching as the refracted ray is immediately terminated. In practice, this means that the throughput of the ray reflecting back inside is multiplied by the Fresnel term F.

The performance penalty of this approach is that the paths are longer, and it takes more time to trace them. Nevertheless, the Russian roulette is applied right after, so if there is a ray with a low throughput due to the multiplication by a low F, it is probably terminated.

On the other hand, the average depth where the rays obtain a contribution is lower, and the resulting image should be less noisy since we removed an element of randomness. Moreover, the ray which hit the surface from the inner side probably continues near the surface and hits it again. So it should be more worthy of being traced than a ray further from the surface.

A similar idea can be applied at the first geometry intersection, where the ray is either reflected in the environment or refracted inside the object. Since we immediately know the contribution of reflected rays (1, the color of the environment), we are interested only in the refracted ray. We can always refract the rays and multiply their throughput by 1 - F while add F * environment color to compensate for the reflected rays.

The rendering time measurements are in the Table 3.23.

Table 3.23: Time measurements - without/with the Fresnel optimization. The times are in seconds. Settings - from the section 3.1, except we use the custom integrator, Woodcock tracking with the inline OpenVDB grid as the medium plugin.

	Without			*With*			*Diff*		
Model	*R*	*G*	*B*	*R*	*G*	*B*	*R*	*G*	*B*
Box_white_20	99.6	101.2	102.8	106.8	108.0	110.3	+7.2	+6.8	+7.5
Box_greek_20	24.8	14.8	9.2	25.3	15.7	9.4	+0.5	+0.9	+0.2
Animal_white	10.8	12.8	15.3	11.4	13.5	16.1	+0.6	+0.7	+0.8
Animal_spotted	3.2	3.8	6.3	3.4	3.9	6.4	+0.2	+0.1	+0.1

Table 3.24: Variance levels - without/with the Fresnel optimization. The variance improvement is for the equal rendering time. Settings - from the section 3.1, except we use the custom integrator, Woodcock tracking with the inline OpenVDB grid as the medium plugin.

Model	*Varianxe without*	*Varianxe with*	*Variance improvement*
Box_white_20	0.0013460	0.0012285	x1.17
Box_greek_20	0.0003204	0.0001945	x1.68
Animal_white	0.0010715	0.0010925	x1.04
Animal_spotted	0.0005238	0.0004192	x1.33

We present the variance improvement adjusted for the rendering time difference (Table 3.24). In the case of the white texture, the variance is very similar. The colorful textures bring more apparent differences. This contradicts the previous chapter results, where longer paths lead to more significant improvement, especially with the white texture.

The fact that the rendering times do not differ that much can be explained by two facts. First, the Fresnel reflectance at the surface hit from the outside is minor (0.04) because the rays are shot in the perpendicular direction. Second, the rays reflected from the surface usually do not get far enough to hit the surface again since the multiplication of their throughput by the Fresnel reflectance leads to their early termination by the Russian roulette.

3.6.4 Switching geometry intersection and medium sampling

The PBRT book [Pharr and Humphreys, 2010] suggests: "In scenes with very dense scattering media, the effort spent on first finding surface intersections will often be wasted, as Medium::Sample() will usually generate a medium interaction instead. For such scenes, a more efficient implementation would be to first sample a medium interaction, updating the ray's tMax value accordingly before intersecting the ray with primitives in the scene. In turn, surface intersection tests would be much more efficient, as the ray to be tested would often be fairly short."

Our medium density is considered especially dense, so we take advantage of this idea. It is simple in terms of implementation complexity. Moreover, it does not change the result, only the rendering time, so we do not have to inspect the variance levels.

As expected, the difference gets more significant with the growing vertex count of the geometry. Ranging from 10% for the simple cube model to 300% rendering time cut for a complicated bone model, this seems to be a worthy improvement (Table 3.25).

We did not inspect the details of the Mitsuba ray intersection system [Jakob, 2010], and we did not try to optimize it in any way since the changes would probably not be applicable elsewhere. There are production-ready solutions like Embree, which would be used instead.

Table 3.25: Time measurements - without/with the medium sampling optimization. The times are in seconds. Settings - from the section 3.1, except we use the custom integrator, Woodcock tracking with the inline OpenVDB grid as the medium plugin.

	Without			*With*			*Speedup*		
Model	*R*	*G*	*B*	*R*	*G*	*B*	*R*	*G*	*B*
Box_white_20	99.6	101.2	102.8	74.2	75.6	77.8	x1.34	x1.34	x1.32
Box_greek_20	24.8	14.8	9.2	21.2	13.2	8.4	x1.17	x1.12	x1.10
Animal_white	10.8	12.8	15.3	5.3	5.5	6.4	x2.04	x2.33	x2.39
Animal_spotted	3.2	3.8	6.3	2.3	2.3	3.4	x1.39	x1.65	x1.85
Bone	267.2	114.7	46.5	83.7	38.2	19.2	x3.19	x3.00	x2.42

3.6.5 Ray branching

The idea of ray branching at intersections or scattering events is not frequently used in path tracing because it leads to an exponential growth of the ray count when it is done at each of these events. Sure, it can be done only at a subset of events, but picking such events is a complex problem.

In our case, though, it is possible to pinpoint events where it is worth branching the ray. In general, we want to focus on the medium layers closer to the surface as they contain the majority of the colorful voxels and details. Therefore, it is more profitable to place the branching near the surface. It also makes sense to branch rays with high throughput. Putting together these two criteria, the best candidates are the scattering events as close to the surface as possible. There are three available options how to do it

- *Directly at the surface intersection, sampling the direction using the BRDF function* - considering the smooth dielectric BRDF, which only decides between the perfect geometric reflection and refraction, the sampled directions would boil down to the refracted ray since the reflected rays are immediately terminated
- *At the first medium scattering event, sampling the direction using the phase function* - the Hayney-Greenstein function with the mean cosine parameter G equal to 0.4 produces a wide range of directions, and the first scattering event is relatively close to the surface. This approach can be repeated multiple times at the first few scattering events
- *Sampling the distance in the direction of the refracted ray* - compared with the previous case, the sampled rays also vary in the origin, not only the direction. The price is that the medium sampling function is called per each branch, not only once

Naturally, the following question arises here. Since we are trying to branch the rays as close to the surface as possible, why not just shoot more rays from the camera and omit the branching? As Pharr and Humphreys [2010] suggests, when fewer rays are necessary for a good pixel aliasing than for the following shading, the branching can save multiple rays shot from the camera while achieving comparable results. In our case, it means that we save multiple geometry intersection

lookups that would be computed for rays shot from the camera. Apart from the overall variance level, we also monitor the aliasing and edge quality issues that might occur when the branching coefficient gets too high (and consequently, the number of primary rays decreases to achieve the same rendering time). The tests are in the 4 chapter.

Herholz et al. [2019] propose to decide between the branching and Russian roulette at any medium interaction based on the current value of a metric (see the section 2.3 where we describe this approach in more detail). As we explained, we are only interested in branching at the early stages of the ray path. Also, we do not have the means to compute such a metric. However, we can take inspiration from the paper and determine the number of branched rays dynamically based on the throughput value. It is easy, and it makes perfect sense to trace fewer rays when the throughput (and therefore expected contribution) is lower.

Results

We present multiple versions of the algorithm and compare them with the default version without branching. For the versions with the fixed splitting coefficient, we use a relatively high number to highlight the differences. The versions are following

- branching at the first scattering event, as described in the second option above, with a fixed branching coefficient of 8 (B2)
- branching using the distance sampling from the surface intersection, as described in the third option above, same branching coefficient (B3)
- branching using the phase function sampling at the first few scattering events, specifically splitting the rays in two at the first three events (reaching the final number of 8 rays) (B4)
- branching at the first scattering event, this time with a variable number of branched rays determined as *min(branching coefficient, int(throughput.max * branching coefficient))*, the constant is also 8 (B2d)

The results are in the Tables 3.26 and 3.27.

Table 3.26: Time measurements - various branching configs. The times are in seconds. Settings - from the section 3.1, except we use the custom integrator, Woodcock tracking with the inline OpenVDB grid as the medium plugin.

		Time			*Slowdown*		
Config	*Model*	*R*	*G*	*B*	*R*	*G*	*B*
Box_white_20	Default	74.9	75.2	80.1	-	-	-
	B2	574.4	576.1	592.7	x7.67	x7.66	x7.40
	B3	590.2	598.6	601.5	x7.88	x7.96	x7.73
	B4	572.3	580.5	619.1	x7.71	x7.72	x7.58
	B2d	577.4	539.0	555.0	x7.20	x7.17	x6.93
Box_greek_20	Default	21.4	13.2	8.4	-	-	-
	B2	141.7	77.0	42.4	x6.62	x5.83	x5.05
	B3	142.8	81.4	47.6	x6.67	x6.17	x5.67
	B4	133.8	78.3	49.6	x6.25	x5.93	x5.90
	B2d	109.6	54.3	23.0	x5.12	x4.11	x2.74
Animal_white	Default	5.3	5.5	6.4	-	-	-
	B2	40.8	41.8	48.9	x7.70	x7.60	x7.64
	B3	41.8	43.6	50.2	x7.89	x7.93	x7.84
	B4	37.2	39.5	46.4	x7.02	x7.18	x7.25
	B2d	36.6	37.3	45.0	x6.91	x6.78	x7.03
Animal_spotted	Default	2.3	2.3	3.4	-	-	-
	B2	14.2	14.6	23.8	x6.17	x6.35	x7.00
	B3	15.6	15.4	24.3	x6.33	x6.70	x7.13
	B4	12.5	13.8	22.8	x5.44	x6.00	x6.70
	B2d	11.4	12.8	20.9	x4.94	x5.55	x6.13

Table 3.27: Variance levels - various branching configs. The variance improvement is for the equal rendering time. Settings - from the section 3.1, except we use the custom integrator, Woodcock tracking with the inline OpenVDB grid as the medium plugin.

Config	*Model*	*Variance*	*Variance improvement*
Box_white_20	Default	0.0012285	-
	B2	0.0001755	x0.91
	B3	0.0001731	x0.90
	B4	0.0002008	x0.79
	B2d	0.0001829	x0.93
Box_greek_20	Default	0.0001945	-
	B2	2.803e-05	x1.05
	B3	2.543e-05	x1.15
	B4	4.286e-05	x0.73
	B2d	4.147e-05	x0.92
Animal_white	Default	0.0010925	-
	B2	0.0001451	x0.97
	B3	0.0001401	x0.99
	B4	0.0001836	x0.84
	B2d	0.0001604	x0.99
Animal_spotted	Default	0.0004192	-
	B2	8.182e-05	x0.83
	B3	4.404e-05	x1.50
	B4	9.314e-05	x0.83
	B2d	7.253e-05	x1.16

The rendering time is getting higher the sooner the ray is split, which is expected. Regarding the variance level, better results are achieved by earlier splitting, even after we account for the higher run time. Also, the branching algorithm's speedup (meaning the difference between the default time multiplied by 8) is more significant when darker textures are used. The albedo also changes across the three channels, so we always match the red channel's rendering times, where we also compare the variance.

The variable number of branches based on the throughput only reduced the rendering time by approximately 20%. The additional samples we could use did not balance the increased variance value caused by using fewer branches. This suggests that even the dark areas require a substantial number of samples. Based on this observation, we do not try the dynamic branching on the other approaches since there it is more challenging to implement. For example, in the distance sampling case, we have the same throughput unless we sample the media and compute individual throughputs for the sampled points. Terminating the rays there would waste the costly calls of the media distance sampling function. Another disadvantage of the dynamic approach is that the compiler might lose some opportunities to simplify the code as the number of branches is no longer a constant expression.

In conclusion, branching using a constant coefficient helps reduce the variance. There is a tradeoff between the variance reduction and the edge quality - an extreme case would be to use only one sample per pixel and branch the ray into a large number of branches leading to a faster rendering and a poor texture

appearance. Therefore, the ultimate setup uses only a conservative coefficient, 4.

3.6.6 MIS

The default volumetric path tracer in Mitsuba supports multiple importance sampling [Jakob, 2010], combining the phase function sampling with light sampling. However, with zero knowledge of the object shape, it is usually a case that the ray towards the light sample traverses a large part of the medium, leading to a negligible contribution because of the high density and costly evaluation. We implement an algorithm that uses additional OpenVDB grids to guide rays towards the surface and decide whether they are worth tracking based on their maximum expected yield.

The algorithm uses MIS to combine two sampling techniques, the same as the default Mitsuba solution. However, instead of sampling a position on the light source and sending a ray in its direction, it determines the ray direction by sampling the distance gradient grid to obtain the direction to the closest surface point. This direction cannot be used straight away because it is generated using the delta distribution, and therefore it could not be used in combination with another technique via the MIS. On the other hand, path tracing based solely on this sampling is not valid as it cannot cover all possible paths. We distort this direction using various sampling techniques, which generate random directions around the original direction. It gives us a random direction and its respective PDF leading to a straightforward combination with the phase function sampling.

On the other hand, this method has a considerable overhead caused by the lookups in the distance and distance gradient grids, transmittance evaluation, and additional geometry intersections. Even if we know the distance to the closest surface point, we need to know the distance for the randomized direction as well.

Transmittance evaluation

We use the simple regular tracking evaluation because it has zero error, unlike the approach based on Woodcock tracking, which only yields 0 or 1. For longer ray segments, the evaluation is costly as it has to traverse all voxels and read their density values. However, we are employing a set of optimizations to reduce this problem. They are discussed later. The comparison of the variance between Woodcock and regular tracking with respect to the *evalTransmittance* function implementation is in the section 4.2.

Randomization distribution

We considered two options - hemisphere cosine sampling and van Mises-Fisher distribution [Fisher, 1953] sampling. The latter works better since it has a parameter to adjust the variance of the sampled directions. Sampling the directions closer to the original gradient direction leads to faster computation and higher contribution of the rays (the distance to the surface is closer to the optimal distance). It also reduces variance as the variance of the directions is lower (verified by variance measurements).

Optimization

Rays oriented in the gradient direction travel shorter distances until they reach the surface than the rays with a random direction. However, the distance can still be too long when their origin is deep inside a thick object. We use a simple decision system to avoid the evaluation of such rays.

First, we use the OpenVDB distance grid that stores the distance between the current voxel and the closest surface point. Using the knowledge about the lowest density of the materials, we compute the highest possible transmittance of the ray heading straight to the surface and consider whether it is worth evaluating. Let us call it the decisive value.

After we obtain the randomized direction and compute the geometry intersection, we compute the highest possible transmittance again, using the distance to the surface intersection. The phase function value is also included in the decision process, so we can avoid the evaluation of rays heading backwards, because the forward-oriented phase function (Heyney-Greenstein with g equal to 0.4) would significantly reduce the contribution. Here, the decisive value is the product of the transmittance and the phase function.

If the contribution evaluation were skipped based on comparing the decisive value with a fixed threshold, the result would be biased. We use the Russian roulette driven by the decisive value multiplied by a constant that gives the option to adjust the number of skipped evaluations. A set of measurements was conducted to prove that this approach is unbiased.

Note that this is based on the lowest density of the non-transparent materials. When there are bigger chunks of transparent material in the medium, the approximate highest possible transmittance is no longer valid. However, due to the use of Russian roulette instead of hard thresholding, the approach stays unbiased.

Additionally, early termination of the transmittance evaluation is used in the regular tracking function, based on the current transmittance, checked in each voxel. Contrastingly, the Russian roulette in the regular tracking loop would lead to a considerable overhead, so we accept a bias here and terminate the computation if the transmittance drops below a threshold. It needs to be set relatively low to avoid a higher bias.

Results

First, we compare the rendering time and variance values with the default algorithm that uses only the phase function sampling (Tables 3.18 and 3.29).

Table 3.28: Time measurements - without/with MIS. The times are in seconds. Settings - from the section 3.1, except we use the custom integrator, simple regular tracking with the inline OpenVDB grid as the medium plugin.

	Without			*With*			*Slowdown*		
Model	*R*	*G*	*B*	*R*	*G*	*B*	*R*	*G*	*B*
Box_white_20	74.9	75.2	80.1	276.9	258.9	225.3	x3.70	x3.44	x2.81
Box_greek_20	21.2	13.2	8.4	67.8	33.8	15.4	x3.20	x2.56	x1.83
Animal_white	5.3	5.5	6.4	17.9	19.5	20.0	x3.38	x3.55	x3.13

Table 3.29: Variance levels - without/with MIS (same sample counts). Settings - from the section 3.1, except we use the custom integrator, simple regular tracking with the inline OpenVDB grid as the medium plugin.

Model	*Varianxe without*	*Varianxe with*	*Variance improvement*
Box_white_20	0.0012285	0.0017554	x0.70
Box_greek_20	0.0001945	6.596e-05	x2.95
Animal_white	0.0010925	0.0016211	x0.67

The rendering times increase about 3-times in each test. The performance of the MIS in comparison to the default approach, on the other hand, strongly depends on the texture type. In the entirely white texture, the MIS reaches the same or even higher variance levels. When darker colors are used, the MIS outperforms the default approach by far.

The measurements in the Tables 3.30 and 3.31 were obtained using the algorithm's version without the pruning in the regular tracking loop, so the variance compared to the default solution can be compared (the bias introduced by the hard threshold pruning changes the result). Apart from the usual time and variance measurements, we also include

- the ratio of the skipped evaluations - first/second condition (Skipped)
- the ratio between the NEE contribution and the total gathered illumination (NEE/T)
- the ratio between the NEE contribution not divided by the success probability and the NEE contribution without the pruning, it is computed because the previous metric is the same for all the Russian roulette settings due to the division by the success probability - it does not provide the information about how the NEE contribution changes across the Russian roulette configurations (P/NP)

For simplicity of the table, we only perform the measurements for the red color channel. Also, we use only three models - the 20mm x 20mm x 20mm cube with both the white and the colored textures and the animal model with white texture. We believe that they are sufficient for illustration of the effects of the settings.

Table 3.30: Various MIS configurations - the configs are (first decisive value, second decisive value) from the section 3.6.6. The times are in seconds. Settings - from the section 3.1, except we use the custom integrator, simple regular tracking with the inline OpenVDB grid as the medium plugin.

Model	*Config*	*Time*	*Variance*	*Skipped*(%)	*NEE/T*(%)	*P/NP*(%)
Box_white_20	1.0/1.0	174.9	0.0034596	31.3/6.6	78.2	12.1
	0.5/2.0	156.2	0.0035854	15.7/5.5	77.6	12.0
	1.5/5.0	156.5	0.0016192	44.2/27.5	76.5	61.5
	3.0/12.0	204.2	0.0014728	68.9/55.2	86.8	64.2
Box_greek_20	1.0/1.0	29.6	0.0001961	25.4/4.7	55.0	10.1
	0.5/2.0	28.0	0.0002725	12.5/3.7	55.9	9.2
	1.5/5.0	35.6	8.547e-05	35.5/18.2	54.5	48.5
	3.0/12.0	40.9	6.698e-05	46.9/38.8	55.3	78.4
Animal_white	1.0/1.0	10.5	0.0036133	54.8/8.5	63.9	15.2
	0.5/2.0	9.2	0.0034672	27.9/8.4	64.1	14.7
	1.5/5.0	13.6	0.0018077	68.4/41.1	61.9	61.0
	3.0/12.0	16.4	0.0016876	79.7/71.3	62.1	84.2

Table 3.31: Performance comparison with the version without MIS. The times are in seconds. The variance improvement is for the equal rendering time. Settings - from the section 3.1, except we use the custom integrator, simple regular tracking with the inline OpenVDB grid as the medium plugin.

Model	*Config*	*Time*	*Variance*	*Variance improvement*
Box_white_20	No MIS	74.9	0.0012285	-
	1.0/1.0	174.9	0.0034596	x0.15
	0.5/2.0	156.2	0.0035854	x0.16
	1.5/5.0	156.5	0.0016192	x0.36
	3.0/12.0	204.2	0.0014728	x0.31
	unlim	276.9	0.0017554	x0.19
Box_greek_20	No MIS	21.2	0.0001945	-
	1.0/1.0	29.6	0.0001961	x0.71
	0.5/2.0	28.0	0.0002725	x0.54
	1.5/5.0	35.6	8.547e-05	x1.36
	3.0/12.0	40.9	6.698e-05	x1.51
	unlim	67.8	6.596e-05	x0.92
Animal_white	No MIS	5.3	0.0010925	-
	1.0/1.0	10.5	0.0036133	x0.15
	0.5/2.0	9.2	0.0034672	x0.18
	1.5/5.0	13.6	0.0018077	x0.24
	3.0/12.0	16.4	0.0016876	x0.21
	unlim	17.9	0.0016211	x0.20

In all test configurations, the pruning appears to skip most evaluations when the white cube model is used. It is expected, as the pruning is based on the evaluated distance, and in smaller models, the distances to the surface are shorter on average. The texture also seems to affect the pruning, however, in a somewhat counter-intuitive way. In less dense material, the throughputs are usually higher, so one could expect that more rays pass the pruning. The opposite effect could

be caused by the fact that rays in a sparser medium get deeper inside the object more quickly.

Using the two Russian roulette settings, we tried to optimize the pruning, so it skips the evaluations that are costly and have a minor contribution. In terms of our metrics, it means minimizing the skipped branches ratio while maximizing the NEE contribution ratio. The target is to achieve a similar variance as in the results without any pruning with a much lower rendering time.

In theory, it is desirable not to terminate too many of the evaluations that already passed the first condition, since before the second condition, the geometry intersection is computed, which is a heavy computation on its own. However, the measurements using the first two setups ($< 1, 1 >$ and $< 0.5, 2.0 >$) do not confirm that. Even when the first condition is much more efficient (passing fewer rays) in the second setup and the rays that pass it have a higher chance to pass the second condition as well, the variance and the rendering times results are comparable.

The first two configurations also show that too aggressive pruning increases the variance significantly, even exceeding the levels of the default solution. It can be caused by the additional randomness introduced by Russian roulette. Having this knowledge, we tried settings that skip fewer evaluations. The rendering time of the last configuration is only about 20% higher than the time of the first configuration, while the NEE contribution and, consequently, the variance is close to the ones of the algorithm without any pruning. Simultaneously, the last configuration applied on the box models is still about twice as fast as the algorithm without any pruning. This means that a large part of the NEE rays are too expensive to evaluate while carrying a negligible contribution, and they are successfully detected and skipped by the Russian roulette.

Now, let us see how the rendering times change when the regular tracking optimization is employed combined with the last configuration of the Russian roulette settings. The threshold is $T_r = 0.01$ (Table 3.32).

Table 3.32: Early termination of regular tracking performance effect. The times are in seconds. Settings - from the section 3.1, except we use the custom integrator, simple regular tracking with the inline OpenVDB grid as the medium plugin.

Model	*Time*	*Time without*	*Speedup*
Box_white_20	206.3	207.2	x1.0043
Box_greek_20	49.8	49.6	x1.0040
Animal_white	18.0	17.9	x1.0042

Unfortunately, the performance improvement is negligible, probably due to the additional branching in the regular tracking loop and the fact that the long rays usually do not make it to the transmittance evaluation. Note that the exponential function is not evaluated in the condition. Only the integrated density is compared to the precomputed ln(0.01) value.

Finally, we investigate why the variance does not improve when the white texture is used. We compare images obtained using both the default algorithm using only phase function sampling and the MIS algorithm with the path tracking depth limited to 2, 10, and 100. When the maximum depth is 2, the default

algorithm results are much noisier. It is due to the fact that only the rays that immediately return to the surface result in a non-zero contribution. Considering the Heyney-Greenstein phase function with the mean cosine equal to 0.4, such paths are improbable, hence the variance. On the other hand, the MIS weights down such rays because the phase function PDF is low, and the other method's (sampling based on the gradient to the surface) PDF is higher (as the direction is heading to the surface). Instead, the short and coherent rays heading to the surface obtain higher weights. However, after more bounces, the MIS algorithm results start picking up variance, both from the phase function sampling contributions and the less coherent NEE rays. Hence we discover why the white texture performs worse with the MIS - because the paths are on average longer, and there is more room to create the variance.

Finally, we note that for simplicity, the MIS algorithm tested there was implemented in the non-branching version of the path tracer. It is also implemented in the branched version, and the overall performance will be discussed in the section 4.

Conclusion

The focus of our application is to optimize the texture appearance of the printed models. It is reasonable to assume that the models with colorful texture and lower albedo will be more frequent than pure white models. Therefore we consider the MIS as an improvement because of its significant performance gain observed in the tests with colorful textures.

3.7 Points, beams and paths

Křivánek et al. [2014] combine multiple estimators, each of them focused on different effects, achieving a robust estimator that handles all situations reasonably well. We utilize their analysis and provide implementation to evaluate the behavior of these methods when they are applied to our type of media and scene setup. We choose one of them, the point-beam estimator, and reimplement it as a Mitsuba plugin. We conduct various measurements and decide whether they can compete with our version of path tracing presented in the previous chapter.

3.7.1 Estimators survey

A brief overview of the methods and their behavior in various scene types is in the sections 2.4 and 2.5. It contains information from the paper, where the methods were evaluated using different medium densities and light conditions that are usually found in production scenes. Here we focus exclusively on dense media and white environment light.

We use the SmallUPBP renderer that provides implementations of all the estimators from the paper along with many more. It allows making arbitrary combinations of the estimators using MIS, changing their parameters, and running them on a set of predefined scenes. However, it supports only homogeneous media, so we will not include the evaluation of image quality and texture details in this survey, only the measurements of rendering time and variance level.

Testing setup

Let us start with the description of the testing scene and parameters

- *Geometry* - a simple box with sizes 10x10x10 and 20x20x20. We used two different sizes to demonstrate the difference between the images rendered with the same number of light samples and different object sizes. All effects and shortcomings of the estimators can be illustrated using such a simple model, so we do not run the tests on more complex ones.
- *Medium* - two different types are used, a white medium with albedo 0.999 and density 6.0, a grey medium with albedo 0.5, and a density of 3.0
- *Light* - an environmental light with an intensity of 1.0
- *Camera* - contrary to the tests in the previous chapters, we use a basic perspective camera. The camera shooting the orthogonal rays used in the optimization pipeline could not be easily reimplemented in the UPBP because it is tied with Mitsuba [Jakob, 2010] and OpenVDB [Museth et al., 2012]. However, the perspective camera yields images that can be easily visually interpreted.
- *Resolution* - we render images with the size 200x200 in tests. The reason for this relatively small resolution is that the UPBP allocates an array with the size image width * image height * maximum depth for path segments. Larger images caused a memory overflow on our testing machine with 16GB RAM.
- *Maximum depth* - 20, which unfortunately introduces a slight bias, but it cannot be set higher for the same reason that we described in the previous point
- *Number of samples* - UPBP renders in so-called "iterations", in one iteration, one ray per image pixel is traced. Algorithms using lightpaths trace an arbitrary number of them at the beginning of each iteration. We use one light path per two camera paths. There is an important consequence - the structures for photon mapping and similar estimators are rebuilt in each iteration. We use the time limit of 20 seconds in each test. We also state the number of iterations that each algorithm managed to do in the time limit to demonstrate their speed.
- *Algorithms* - UPBP provides implementations for several estimators, we use
  - unidirectional path tracer (PT) to set a base for all other algorithms
  - bidirectional path tracer (BDPT), note that in our scene, the BDPT and VCM boil down to the same setup, so their results are also identical - therefore, we include only BDPT
  - point-point 3D algorithm with two radius options, 0.1 to demonstrate how the larger radii influence the image variance, and 0.02 (based on the shortest edge of the printer voxel) with the shrinking constant alpha (the radius is multiplied by the constant in each iteration, leading

to its steady decrease over the iterations) equal to 0.97 providing a good tradeoff between the result image quality and bias

- point-beam 2D algorithm with the initial radius 0.02 and alpha 0.97 both short and long query beams, short photon beams
- beam-beam 1D algorithm with the initial radius 0.02 and alpha 0.97 with short photon beams and both short and long query beams

Results

The performance measurements are in the Table 3.33. Considering the major differences between the estimators, we do not need to measure the results' variance. Instead, we provide the actual rendered images where we explain the particular features of the respective approaches (Figures 3.2, 3.3 and 3.4).

Table 3.33: Comparison of rendering times for various estimators (Number of iterations in 20 seconds.)

Algorithm	*White cube* 10^3	*Grey cube* 10^3	*White cube* 20^3
PT	1029	1095	975
BDPT	186	184	176
PP3D (α1 r0.1)	403	364	338
PP3D (α0.97 r0.02)	309	317	275
PB2D (α0.97 r0.02 qL pS)	160	182	226
PB2D (α0.97 r0.02 qS pS)	241	302	293
BB1D (α0.97 r0.02 qL pS)	48	35	64
BB1D (α0.97 r0.02 qS pS)	169	119	224

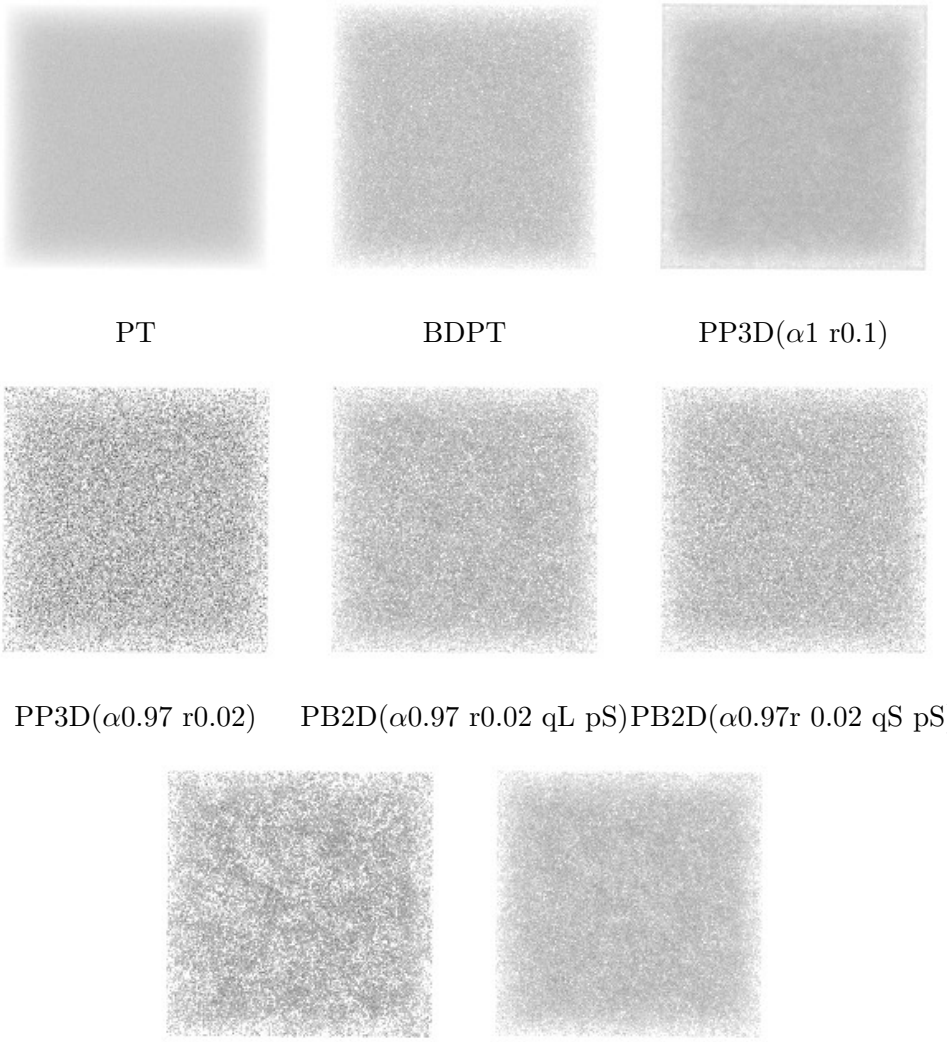


Figure 3.2: Rendering results comparison of various estimators (white cube 10 x 10 x 10).

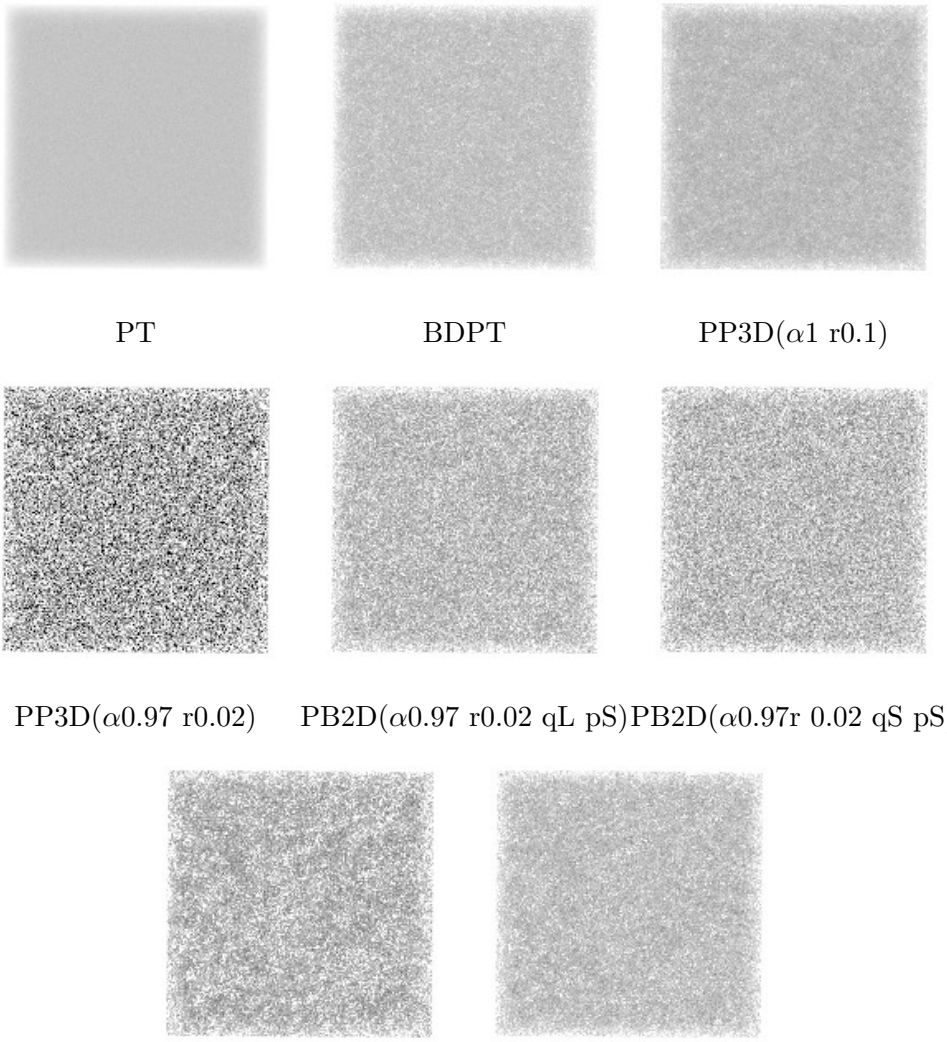


Figure 3.3: Rendering results comparison of various estimators (white cube 20 x 20 x 20).

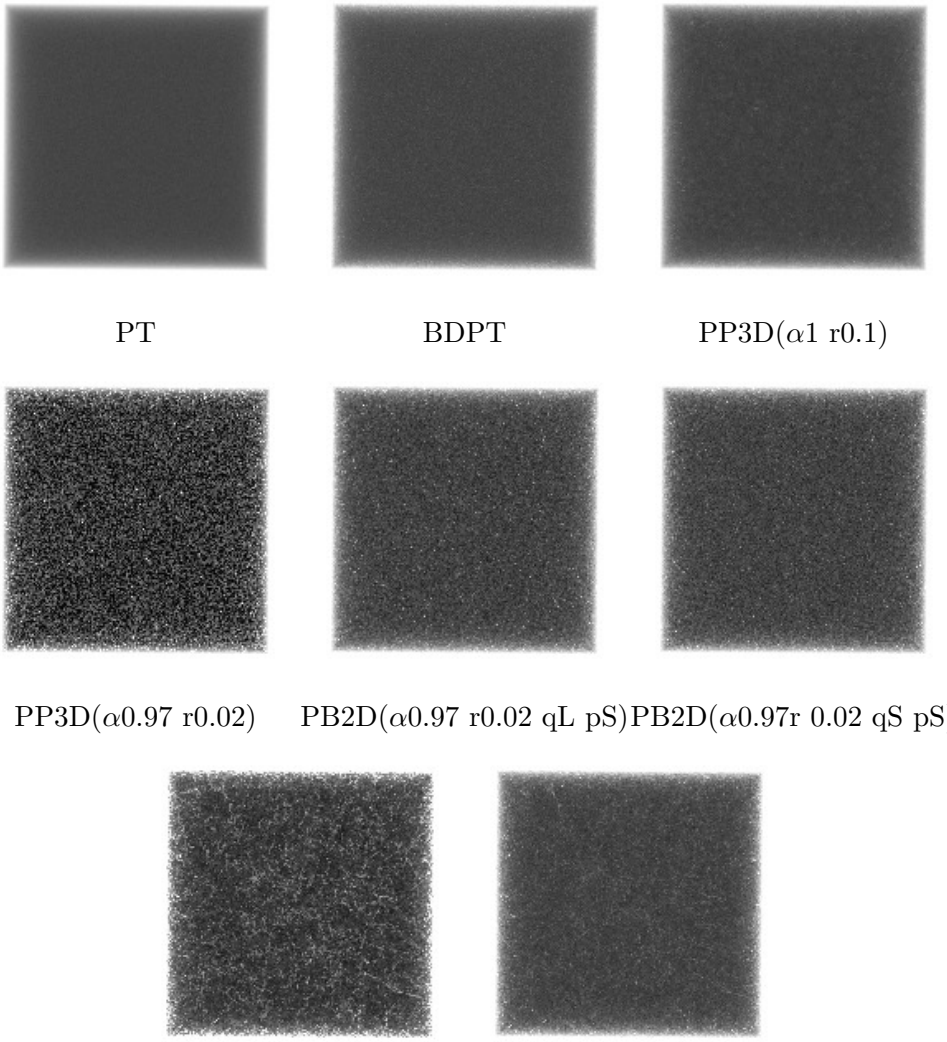


Figure 3.4: Rendering results comparison of various estimators (grey cube 10 x 10 x 10).

Evaluation

The first two algorithms provide acceptable results varying only in the variance level. The PT manages to render more than 5-times more iterations which reflects on the result. In theory, BDPT behaves well in scenes containing light sources hard to find by the camera rays. The constant incident radiance over the surface is the exact opposite. Hence the BDPT algorithm does not get to use its advantage. On the other hand, it is left with the inherent additional overhead that prevents it from finishing more iterations. The object size does not affect the results in both cases. In our application, the media with lower albedo achieved faster rendering times because the Russian roulette terminated the rays sooner as they lost throughput because of the low albedo. It is not the case in UPBP because the Russian roulette is driven by a constant set in the medium constructor. It does not take the current ray throughput into account.

The results of the PP3D estimator show multiple shortcomings. Although

the radius of 0.1 leads to almost noise-free results, there are two issues. First, the photon spreads across 2-4 voxels, depending on the axis - that would result in blurry images with a lack of detail if there were a texture. Second, there is a dark stripe along the cube edges. It is a consequence of the photon density computation using a fixed spherical filter - the contribution of the photons closer to the requested point than the radius is summed and weighted by the filter. When an edge is closer to the surface than the radius, fewer photons are summed as the medium is cut by the surface intersection, but the spherical filter stays the same.

The second configuration eliminates both these problems using the shrinking photon radius. In theory, it converges into an unbiased solution. However, a few hundred iterations were not enough to achieve a passable solution. Apart from the performance, which is far worse than the previous estimators, it carries additional parameters and bad scalability - with the same number of light samples and the same radius, the bigger cube is noisier as the photon density is lower.

Despite the lower number of iterations, the PB2D brings much better results. This is more or less in line with Křivánek et al. [2014]. They claim that the point estimators perform better in dense media but also that beam estimators are more efficient with smaller kernels. In our case, the small kernel criterium is obviously stronger. The paper also suggests using long query beams (and short photon beams - this is only applicable in the BB1D estimator, not here). They lower the variance also in our case, again, despite the fewer iterations.

The BB1D estimator performs well with the short query beams, confirming the paper's claim that for small kernels, it is the best of all three approaches. However, the long query beams are computationally demanding, leading to a really low number of iterations and high variance. It is debatable why this contradicts the paper and our measurements for PB2D. The reason may be that the cube volume is so big and dense that the beams traveling to the surface have very low transmittance leading to the negligible contribution, while the computation is still demanding. Both in our results for BB1D and in Figure 7 of the paper, one can spot a low-frequency noise that is not that apparent in the PB2D results. This is a problem, as it may confuse the optimization.

In summary, none of the methods can compete with the basic unidirectional path tracing in any of the test cases and with any of the settings. From the last three estimators, the worst one is definitely the PP3D. The best one is either PB2D or BB1D, depending on whether the low-frequency noise is an issue.

3.7.2 Beam radiance estimate

Mitsuba renderer [Jakob, 2010] provides an implementation of Beam radiance estimate method. In terms of the naming used by Křivánek et al. [2014], it is a single scattering version of the PB2D estimator. Same as the SmallUPBP, it also supports only homogeneous media. Contrastingly, it only builds the photon lookup structure once - while in SmallUPBP, we would eventually get an unbiased image by computing a sufficient number of iteration, in Mitsuba, the image stays biased irrespective of the number of camera rays shot. The bias can be partially removed using a higher number of photons, but it is limited by available memory, making it a problem for larger objects.

Despite the poor performance compared to the unidirectional path tracing demonstrated in the previous chapter and the inability to repeat the photon tracing phase multiple times per one rendering, we create our optimized version of this plugin with the support for heterogeneous media. We intend to observe the effects of different settings on the quality of the texture details, which is, to our knowledge, a novel contribution of our work.

Implementation

Our implementation is based on the *photonmapper* Mitsuba plugin and a set of core classes that it uses. The most radical changes are in the photonmapper itself, the photon tracing algorithm, and the BRE query function.

The default photon tracing algorithm is replaced by the optimized version of the path tracing algorithm without some of its parts, like the ray splitting and the refraction branching. Additionally, we remove the options to store surface photons and leave only the storing at medium interactions.

The main photon mapping radiance estimating function works as follows. Same as the path tracer, it starts with testing the geometry for an intersection against the input ray. If the object is missed, the environment irradiance is returned. If not, we both account for the reflection and multiply the throughput by (1 - F) as if the ray refracted (as described in the section 3.6.3). Now being inside the geometry, the refracted ray is again tested for intersection. The photons are queried along the ray segment that is inside the object. Finally, the overall transmission of the ray segment is evaluated, and the result is multiplied by the environment light irradiance, simulating the event where the ray traverses the medium, refracts outside, and hits the environment light.

The BRE query function has one significant change. It evaluates the transmittance between the ray origin and a photon using the medium interface function instead of the hardcoded Beer formula, hence gaining the support for heterogeneous media.

We incorporate two major optimizations of the heterogeneous transmittance evaluation. First, we implement the idea proposed by Jarosz et al. [2008]. Before running the BRE query, we use a regular tracking based function to compute a vector of ray segments with a constant density. Each segment contains a starting t, the transmittance at the point, and the segment's density. Later, when the query is evaluated, the transmittance between the ray origin and the photon is computed by traversing the vector and finding the respective segment for the photon distance, and computing the transmittance inside the segment using the cached density. We limit the vector length so it can be allocated on the stack. As a consequence, we can only evaluate a limited distance using the vector. However, we assume that the transmittance at the end of the vector is negligible. The transmittance is computed precisely thanks to the regular tracking, unlike the original Jarosz et al. [2008] proposal using ray marching.

Second, we cut off photons whose ray segment possesses transmittance of less than 1%. It can be implemented in both versions with and without the vector optimization. In the latter version, we compute the distance based on the lowest material density we work with. In the former version, we terminate the vector filling when the transmittance drops under 1%. The distance is then used as the maximum distance of the query ray. Note that the limited length of the

vector and the fixed distance in the unoptimized version would not work if the medium contained an excessive amount of transparent material voxels. However, these improvements made the algorithm at least comparable with path tracing performance-wise.

Results

Apart from the pixel sample count affecting the performance and quality of results, BRE has two additional parameters - the overall number of photons and the nearest neighbors lookup size (see the section 2.4.3 for further explanation). These two parameters indirectly determine the photon radius. We test multiple values for each of them and observe the average photon radius and run times of particular phases of the algorithm.

The measured phases are

- Photon gathering
- KD-tree building in the photon map
- Photon radius computation based on the local photon density and the lookup size parameter
- Building of the lookup hierarchy
- Rendering

The measurements are obtained incorporating the second optimization as the times without that are enormous (Table 3.34). We include times obtained both with and without the first optimization in order to see its influence on the result (Table 3.35). The rendering times are provided only for the red channel for simplicity.

Table 3.34: Various (photon count, lookup size) configurations and time measurements. The rendering times are including the second optimization. The times are in seconds. We include the average photon radius (in mm) to observe the effect of the two parameters.

Model	*Config*	*PG*	*KD*	*Rad*	*Hier*	*Rend*	*Avg. radius*
Box_white_20	8M/10	4.1	8.8	4.2	0.3	182.0	0.1
	8M/4	3.7	10.7	3.5	0.3	66.2	0.05
	8M/40	4.0	10.6	5.2	0.3	981.0	0.15
	24M/4	11.1	39.8	12.2	0.9	97.8	0.02
	24M/10	11.1	40.9	14.9	0.9	290.5	0.04
	2M/4	0.9	1.9	0.7	0.08	37.1	0.07
Box_greek_20	8M/10	2.3	10.6	4.2	0.3	194.1	0.1
	8M/4	2.3	10.7	3.5	0.3	73.5	0.05
	8M/40	2.3	10.6	5.2	0.3	1091.0	0.15
	24M/4	6.7	39.1	12.3	0.9	103.8	0.02
	24M/10	6.7	39.1	14.9	0.9	115.4	0.04
	2M/4	0.6	1.9	0.7	0.08	44.4	0.07
Animal_spotted	8M/10	-	-	-	-	-	-
	8M/4	-	-	-	-	-	-
	8M/40	-	-	-	-	-	-
	24M/4	5.4	48.6	12.1	0.9	32.7	0.01
	24M/10	-	-	-	-	-	-
	2M/4	0.4	2.3	0.8	0.08	3.1	0.35

Table 3.35: Comparison of the performance without and with the first optimization (the second optimization is already included in both). The times are in seconds.

Model	*Config*	*Default*	*Optimized*	*Speedup*
Box_white_20	8M/10	182.0	37.6	x4.84
	8M/4	66.2	16.1	x4.11
	8M/40	981.0	98.7	x9.94
	24M/4	97.8	21.5	x4.55
	24M/10	290.5	40.9	x7.10
	2M/4	37.1	11.8	x3.14
Box_greek_20	8M/10	194.1	29.6	x6.56
	8M/4	73.5	20.1	x3.66
	8M/40	1091.0	128.7	x8.48
	24M/4	103.8	27.3	x3.80
	24M/10	115.4	49.2	x2.35
	2M/4	44.4	13.2	x3.36
Animal_spotted	8M/10	-	-	-
	8M/4	-	-	-
	8M/40	-	-	-
	24M/4	32.7	6.2	x5.27
	24M/10	-	-	-
	2M/4	3.1	3.0	x1.03

Multiple patterns can be spotted immediately. The photon tracing phase behaves the same as path tracing, achieving faster rendering when a darker texture

is used. The time grows more or less linearly with the number of photons, which is also expected.

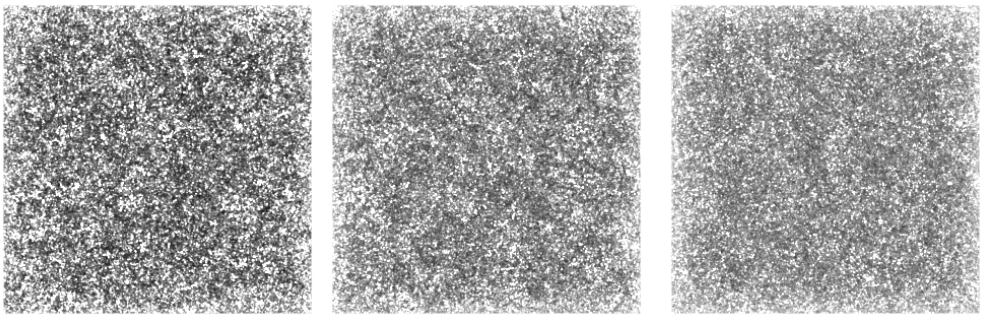
The radius computation, BRE hierarchy, and KD-tree building are not influenced by the texture nor model used. The only factor is the photon count. The major influence on the overall time among the non-rendering (and hence not influenced by the pixel sample count) phases have the KD-tree building.

The lookup size significantly influences the rendering times. A higher number leads to a bigger radius. Therefore more photons are intersected by the query ray, increasing the overall rendering time. Luckily, we are interested in smaller photon radii because we want to avoid blurriness.

For some configurations, the times measured using both the optimization are even comparable with the path tracer. However, this algorithm only computes single scattering.

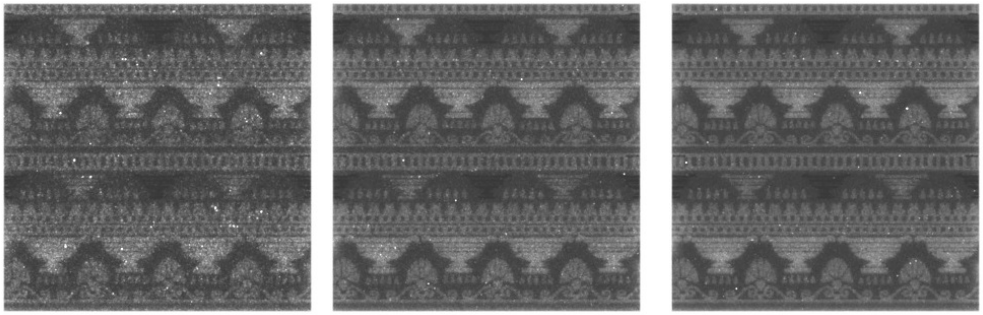
The measurements of the average photon radius present a critical issue. The size of the model dramatically influences the radius. Hence the optimal number of photons is not constant across the test cases. As we mentioned, keeping the photon radius reasonably low is crucial. To be able to use this in production, exposing this parameter for users would burden them with the task of estimating a reasonable value for each of their models. Alternatively, we would need to develop an algorithm that would estimate the number from the object size and shape. Also, note that with the voxel size (0.028, 0.056, 0.084), we can only consider the configurations $< 8M, 4 >$, $< 24M, 4 >$ and $< 24M, 10 >$, so that the photon radius is not too big compared to the voxel size.

Furthermore, we present rendering results obtained by different settings of the parameters above to manifest their influence.



(a) 8M 4 (b) 24M 4 (c) 48M 4

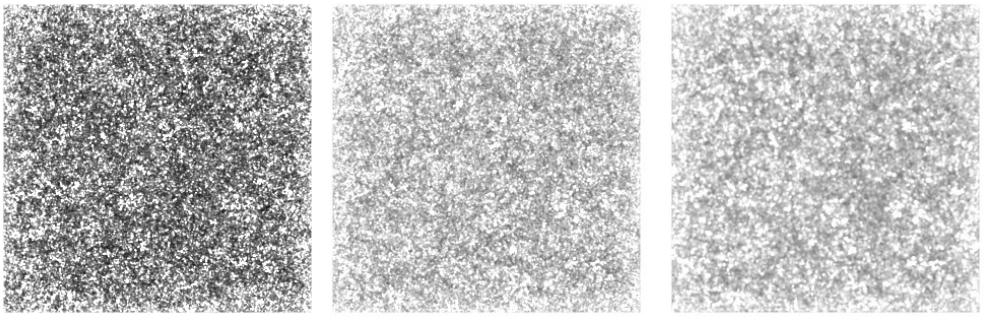
Figure 3.5: Effect of different photon count (white cube 20x20x20).



(a) 8M 4 (b) 24M 4 (c) 48M 4

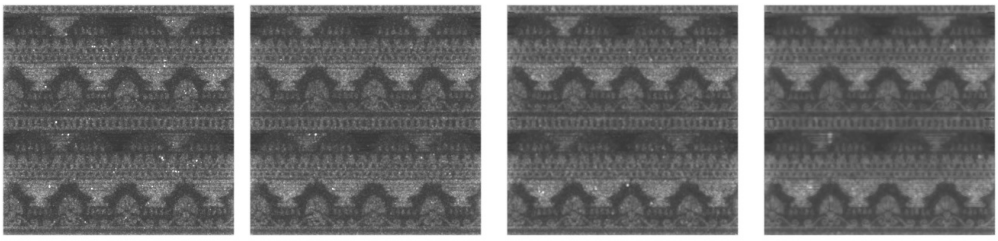
Figure 3.6: Effect of different photon count (textured cube 20x20x20).

There is an apparent connection between the volume density and the resulting quality (see the results for R, G, B channels using the white texture, which has the same albedo and varying density across the channels). Photons in denser volumes tend to create clusters which result in uneven distribution of the pixel radiance. We confirmed this fact by observing the deviation of photon radius - in the less dense media, the radius range was much smaller. We are not sure what is the reason behind this phenomenon. Otherwise, the image quality improves with the photon count as expected.



(a) 8M 4 (b) 8M 10 (c) 8M 40

Figure 3.7: Effect of different lookup size (white cube 20x20x20).

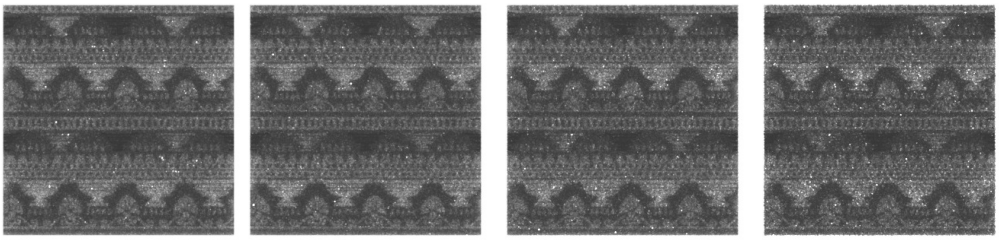


(a) 8M 4 (b) 8M 10 (c) 8M 40 (d) 8M 120

Figure 3.8: Effect of different lookup size (textured cube 20x20x20).

The effect of a higher lookup size and consequently higher photon radius is that the image is more smooth, and the fireflies are less bright. The images of textured

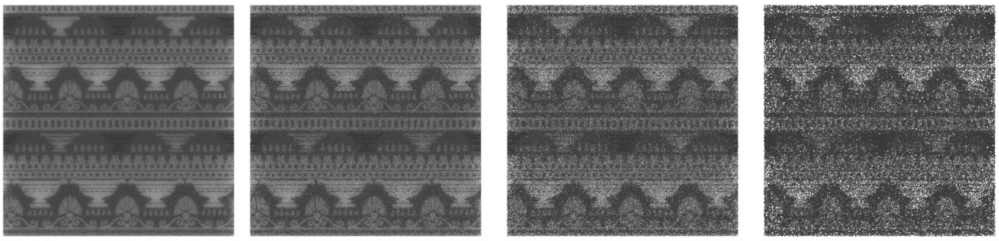
objects show increasing blurriness, which is highly undesirable because one of the optimization pipeline targets is to sharpen the printouts' texture appearance.



(a) 8M 4 - 64spp (b) 8M 4 - 16spp (c) 8M 4 - 4spp (d) 8M 4 - 1spp

Figure 3.9: Effect of different sample count (textured cube 20x20x20).

After a certain number of samples per pixel, the image quality improvement halts. For example, the difference between 16 and 64 samples is negligible. Therefore considerably fewer samples are necessary to make the best of the given photon distribution than we use with path tracing.



(a) 64spp (b) 16spp (c) 4spp (d) 1spp

Figure 3.10: Path tracing results for different sample counts for comparison.

Path tracing and BRE have two different issues when a lower camera sample count is used - path tracing suffers from white noise, while BRE results have fireflies (which usually span through multiple pixels). Even though there are post-processing algorithms that can suppress both these imperfections, we believe that the fireflies are a more significant problem since there is considerable overlap between larger fireflies and smaller texture details, which might be removed together by some algorithm. The BRE results do not have a problem with the high-frequency noise (instead of the path tracing results) because the photon radius behaves as a simple averaging filter.

Conclusion

The measurements obtained using our implementation of the BRE algorithm confirmed the conclusions from the section 3.7.1. The algorithm cannot compete with the performance and result quality of the unidirectional path tracer and its simplicity in terms of required parameters.

However, we do not say that the algorithms based on photon mapping are not usable for volumetric rendering in general or even for the rendering of volumes with parameters similar to the printer materials. These algorithms have proven themselves to be efficient in the rendering of particular effects such as caustics.

What makes unidirectional path tracing so efficient is the fact that we use the constant incident radiance over the surface which is easy to access for the camera rays.

Even if we had a way to build the lookup structure and perform the photon BRE queries in a negligible time and the expense of photon mapping would boil down to the photon tracing, it still would not outperform unidirectional methods since the environment light is easily accessible. Moreover, simple unidirectional methods do not have to store any intermediate information in memory.

4. Results

Although we provided measurements and evaluations for each of the algorithms in the previous chapter, in some cases, we could not decide whether one approach was better than the other. The reason is that regarding the high number of algorithm versions, we could not conduct a more thorough analysis using more testing scenarios. Also, given the order in which we gradually added the improvements, some algorithms may benefit from the changes that were added later, despite our effort to avoid such situations. In order to get a better view of the compared algorithms, we provide measurements obtained using a broader set of objects and textures and running the final version of all the plugins varying only in the compared algorithms. Finally, we select the very best combination of the algorithm versions and compare it with the original version in terms of rendering time, variance, and overall image quality to quantify the improvement.

4.1 Volume data storage

The performance measurements using the dense grids and the OpenVDB grids [Museth et al., 2012] provided in the section 3.3 did not reveal any significant difference (around a 5% difference in the overall time). Moreover, the algorithm using OpenVDB grids thread-local accessors needed to assure that the accessors are initialized on every medium function call. Our optimized path tracing algorithm allows us to put this call at the beginning of the radiance evaluation function called only once per camera sample. The following results are measured using the version of the algorithm, including the path tracing optimization from the 3.6, including the branching (with coefficient 4). In both versions, the volume storage is used as a separate plugin, not inlined into the medium plugin.

However, we do not use the version with MIS as it needs the distance and distance gradient grids which we store exclusively as OpenVDB grids. The reason is that, unlike the label grid, these two are created directly from the geometry during the voxelization and not from a (dense) internal representation, so constructing the dense representation of these grids would require additional memory and time.

Table 4.1: Dense vs. OpenVDB volume representation comparison. The times are in seconds. Both the medium plugins use external volume sources (as opposed to inlined). The setup is described in this chapter, the rest is in the section 3.1.

	Dense			*OpenVDB*			*Diff*		
Approach	*R*	*G*	*B*	*R*	*G*	*B*	*R*	*G*	*B*
Box_20_white	279.9	283.1	287.5	279.2	282.6	287.7	-0.7	-0.5	+0.2
Box_20_greek	74.5	44.1	26.9	75.5	44.8	27.6	+1.0	+0.7	+0.7
Box_10_white	76.9	76.3	83.9	78.0	77.4	84.9	+1.1	+1.1	+1.0
Box_10_greek	15.9	13.6	12.9	16.1	13.8	12.5	+0.2	+0.2	-0.4
Animal_white	19.6	22.8	25.8	20.6	23.0	26.5	+1.0	+0.2	+0.7
Animal_spotted	6.3	7.2	11.5	6.7	7.4	11.7	+0.4	+0.2	+0.2

The measurements in the Table 4.1 suggest that the performance is the same for the test with white texture. However, there is up to a 10% difference in the tests with colorful textures. That may be a consequence of the different number of distance sampling calls due to varying path lengths and distances between the scattering events.

An argument in favor of using the OpenVDB grids is that we use them regardless to store the distance and distance gradient grids. Therefore it makes sense to use OpenVDB grids everywhere. Also, the measurements provided in the section 3.2.3 show a considerable reduction of the memory footprint when the OpenVDB grid is used. In conclusion, we decide to use the OpenVDB representation. We believe that it is worth trading a minor slow down to unify the grid storage and the memory saving.

4.2 Distance sampling and transmittance evaluation algorithm

Once we have ruled out all the unsuccessful experiments with homogeneous medium approximations and OpenVDB grid hierarchies, we are left with the Woodcock tracking and regular tracking. Concerning the distance sampling, Woodcock tracking is faster, but regular tracking has a slightly lower variance (see the section 3.5.3). On the other hand, for transmittance evaluation, regular tracking provides much more precise estimates.

To decide between these approaches, we run an extensive set of tests also using some artificial materials to target specific properties of the algorithms.

4.2.1 Distance sampling

First, we determine the best algorithm for distance sampling. To isolate the distance sampling calls from the transmittance evaluation calls, we run the test with the MIS disabled. We already ruled out the hierarchical regular tracking as the distance sampling method in the section 3.5, so here we use only the simple version. The integrator setup is the same as in the previous chapter.

Regular tracking traverses all voxels from the ray origin to the point where the requested transmittance is integrated. Lower transmittance leads to a longer ray segment and hence more traversed voxels. On the other hand, Woodcock tracking has a lower probability of scattering when the voxel density is lower compared to the maximum density, so more iterations of the loop are performed to finally reach the scattering. To see how the two algorithms behave when the density changes, we set up a series of tests with white albedo and significantly changing density. After that, we conduct the tests from the previous chapter. Both versions use the OpenVDB representation of media.

Note that unlike the previous chapter, here, the OpenVDB grids are inlined in the medium plugins to minimize the overhead of grid lookups, which are pretty frequent in the regular tracking.

The measurements are in the Table 4.2.

Table 4.2: Comparison of regular and Woodcock tracking performance. Both the medium plugins use inlined volume sources. The times are in seconds. The setup is described in this chapter, the rest is in the section 3.1.

	Regular simple			*Woodcock*			*Regular/Woodcock*		
Medium	*R*	*G*	*B*	*R*	*G*	*B*	*R*	*G*	*B*
σ_t 10.0, α 1.0	72.1	-	-	54.1	-	-	x1.33	-	-
σ_t 1.0, α 1.0	60.4	-	-	44.8	-	-	x1.35	-	-
σ_t 0.4, α 1.0	46.2	-	-	35.8	-	-	x1.29	-	-
Box_20_white	345.9	346.3	350.4	279.0	281.5	287.2	x1.24	x1.23	x1.22
Box_20_greek	94.4	55.4	29.6	75.2	44.5	27.4	x1.26	x1.25	x1.08
Box_10_white	99.4	98.7	98.2	77.6	77.1	84.6	x1.28	x1.28	x1.16
Box_10_greek	19.2	17.0	15.4	16.0	13.6	12.4	x1.20	x1.25	x1.24
Animal_white	26.2	29.3	30.7	20.4	22.8	26.3	x1.28	x1.29	x1.17
Animal_spotted	8.0	8.4	12.7	6.6	7.3	11.5	x1.21	x1.15	x1.10

According to the variance level measurements in the section 3.5.3, regular tracking has only a negligible improvement, and it cannot compensate for the higher computation time. The tests with different densities show that the ratio of the times is roughly the same across all three tests, so we can conclude that it does not influence the performance difference between the two algorithms.

In summary, it is clear that for distance sampling, the better option is the Woodcock tracking as it is significantly faster and has a comparable variance level as regular tracking.

4.2.2 Transmittance evaluation

In this chapter, we keep the Woodcock tracking as the algorithm for distance sampling and try the options for transmittance evaluation - Woodcock tracking and regular tracking (both simple and hierarchical version). Here we use the path tracer with MIS, which actually calls the transmittance evaluation. Otherwise, the path tracer setup is the same as in the previous chapter.

The measurements are in the Table 4.3.

Table 4.3: Time measurements - transmittance evaluation. The variance improvement is for the equal rendering time. The times are in sections. VI = variance improvement, WT = Woodcock tracking, RTs = regular tracking simple, RTh = regular tracking hierarchical. The setup is described in this chapter, the rest is in the section 3.1.

		Time				
Model	*Config*	*R*	*G*	*B*	*Noise*	*VI*
Box_white_20	no MIS	279.0	281.5	287.2	0.0003097	-
	MIS (2.5/9.0) WT	614.0	635.4	690.3	0.0004325	x0.33
	MIS (2.5/9.0) RTs	833.5	841.3	839.0	0.0003761	x0.28
	MIS (2.5/9.0) RTh	812.3	802.4	786.9	0.0003761	x0.28
Box_greek_20	no MIS	75.2	44.5	27.4	4.874e-05	-
	MIS (2.5/9.0) WT	130.7	86.2	57.1	2.478e-05	x1.13
	MIS (2.5/9.0) RTs	162.3	95.4	58.9	1.891e-05	x1.19
	MIS (2.5/9.0) RTh	168.2	98.5	60.1	1.891e-05	x1.15
Box_white_10	no MIS	77.6	77.1	84.6	0.0003097	-
	MIS (2.5/9.0) WT	143.1	157.3	18.4	0.0004665	x0.36
	MIS (2.5/9.0) RTs	171.7	190.4	211.6	0.0004141	x0.34
	MIS (2.5/9.0) RTh	168.5	188.1	255.6	0.0004141	x0.34
Box_greek_10	no MIS	16.0	13.6	12.4	5.669e-05	-
	MIS (2.5/9.0) WT	30.5	19.6	13.9	3.123e-05	x0.95
	MIS (2.5/9.0) RTs	34.3	22.6	14.9	2.257e-05	x1.17
	MIS (2.5/9.0) RTh	37.8	24.4	15.9	2.257e-05	x1.06
Animal_white	no MIS	20.4	22.8	26.3	0.0002733	-
	MIS (2.5/9.0) WT	48.3	54.3	65.9	0.0004976	x0.23
	MIS (2.5/9.0) RTs	61.1	70.3	96.1	0.0004367	x0.21
	MIS (2.5/9.0) RTh	61.4	69.8	96.8	0.0004367	x0.21
Animal_spotted	no MIS	6.6	7.3	11.5	0.0001084	-
	MIS (2.5/9.0) WT	14.7	17.1	27.0	8.975e-05	x0.54
	MIS (2.5/9.0) RTs	17.3	20.8	33.3	7.438e-05	x0.56
	MIS (2.5/9.0) RTh	19.0	22.9	36.1	7.438e-05	x0.51

The evaluation of light rays using the Woodcock tracking is faster than using regular tracking (same as in the case of distance sampling). The variance levels are lower when regular tracking is used as expected.

Simple regular tracking is more efficient than the hierarchical one in all cases except the white cube. It is due to the fact that single-material simple shapes contain larger constant areas where the hierarchical algorithm can benefit from traversing higher-level nodes in the grid. However, we still prefer the simple variant, since textured models are our area of interest.

Woodcock tracking is slightly better for the white texture than the simple regular tracking, simply because it uses more samples per pixel. Since we focus primarily on colored textures, we consider regular tracking our transmittance evaluation algorithm of choice. The medium plugin is then a hybrid that uses Woodcock tracking for distance sampling and regular tracking for transmittance evaluation.

4.3 Comparison with the default path tracer

At this, we can establish the ultimate configurations which are most suitable for different setups. Their components are as follows

- *Grid representation* - OpenVDB grids are used to represent all grids, the label, distance, and distance gradient grids
- *Medium* - Woodcock tracking for distance sampling and regular tracking for transmittance evaluation
- *Integrator* - unidirectional path tracer with Russian roulette maximum threshold set to 0.99, medium sampling before geometry intersection
  - *custom1 woNEE* - no branching, no MIS
  - *custom1 NEE* - no branching, MIS with Russian roulette decisive values set to 2.5 and 9 respectively (see the section 3.6.6)
  - *custom3 woNEE* - distance-based branching with the coefficient 4, no MIS
  - *custom3 NEE* - distance-based branching with the coefficient 4, MIS with Russian roulette decisive values set to 2.5 and 9 respectively (see the section 3.6.6)

We provide time measurements comparing our algorithm to the default Mitsuba path tracer [Jakob, 2010], variance measurements, and visual analysis of the result image quality (Tables 4.4, 4.5 and Figure 4.1).

Table 4.4: Comparison with the default. The times are in seconds. The setup is described in this chapter, the rest is in the section 3.1.

Model	*Config*	*Time*			*Noise*
		R	*G*	*B*	
Box_white_20	simple	74.2	75.6	77.8	0.0191329
	custom 1 woNEE	74.9	75.2	80.1	0.0012285
	custom 1 NEE	189.1	186.8	191.4	0.0015992
	custom 3 woNEE	279.0	281.5	287.2	0.0003097
	custom 3 NEE	833.5	841.3	839.0	0.0003761
Box_greek_20	simple	34.2	27.0	21.5	0.0005042
	custom 1 woNEE	21.2	13.2	8.4	0.0001945
	custom 1 NEE	45.7	26.0	15.4	6.752e-05
	custom 3 woNEE	75.2	44.5	27.4	4.874e-05
	custom 3 NEE	162.3	95.4	58.9	1.891e-05
Box_white_10	simple	19.6	19.6	20.7	0.0192956
	custom 1 woNEE	21.7	21.2	22.8	0.0011934
	custom 1 NEE	53.4	56.5	62.7	0.0013540
	custom 3 woNEE	77.6	77.1	84.6	0.0003097
	custom 3 NEE	171.7	190.4	211.6	0.0004141
Box_greek_10	simple	7.8	5.8	5.1	0.0005314
	custom 1 woNEE	4.6	3.8	3.5	0.0002334
	custom 1 NEE	9.6	5.9	4.0	0.0001024
	custom 3 woNEE	16.0	13.6	12.4	5.669e-05
	custom 3 NEE	34.3	22.6	14.9	2.257e-05
Animal_white	simple	11.7	12.9	13.8	0.0154023
	custom 1 woNEE	5.3	5.5	6.4	0.0010925
	custom 1 NEE	14.5	15.5	17.5	0.0016708
	custom 3 woNEE	20.4	22.8	26.3	0.0002733
	custom 3 NEE	61.1	70.3	96.1	0.0004367
Animal_spotted	simple	5.3	5.8	7.8	0.0008542
	custom 1 woNEE	2.3	2.3	3.4	0.0004192
	custom 1 NEE	5.2	5.6	8.8	0.0002662
	custom 3 woNEE	6.6	7.3	11.5	0.0001084
	custom 3 NEE	17.3	20.8	33.3	7.438e-05
Bone	simple	227.6	148.4	89.4	0.0080607
	custom 1 woNEE	83.7	38.2	19.2	0.0009503
	custom 1 NEE	230.4	109.0	42.0	0.0008726
	custom 3 woNEE	304.0	132.1	57.3	0.0002457
	custom 3 NEE	873.0	386.9	147.6	0.0002316

Table 4.5: Comparison with the default, red channel.

Model	*Algorithm*	*Time overhead (R)*	*Equal variance speedup*
Box_white_20	custom 1, wo NEE	x1.05	x14.83
	custom 1, NEE	x2.54	x4.71
	custom 3, wo NEE	x4.41	x14.01
	custom 3, NEE	x11.0	x4.62
Box_greek_20	custom 1, wo NEE	x0.59	x4.18
	custom 1, NEE	x1.34	x5.57
	custom 3, wo NEE	x2.20	x4.70
	custom 3, NEE	x4.75	x5.61
Box_white_10	custom 1, wo NEE	x1.11	x14.57
	custom 1, NEE	x2.72	x5.24
	custom 3, wo NEE	x3.96	x15.735
	custom 3, NEE	x8.76	x5.96
Box_greek_10	custom 1, wo NEE	x0.54	x4.21
	custom 1, NEE	x1.23	x4.22
	custom 3, wo NEE	x2.05	x4.57
	custom 3, NEE	x4.40	x5.35
Animal_white	custom 1, wo NEE	x0.45	x31.33
	custom 1, NEE	x1.24	x7.43
	custom 3, wo NEE	x1.74	x32.39
	custom 3, NEE	x5.22	x6.76
Animal_spotted	custom 1, wo NEE	x0.43	x4.74
	custom 1, NEE	x0.98	x3.27
	custom 3, wo NEE	x1.25	x6.30
	custom 3, NEE	x3.26	x3.52
Bone	custom 1, wo NEE	x0.37	x22.93
	custom 1, NEE	x1.01	x9.15
	custom 3, wo NEE	x1.34	x24.48
	custom 3, NEE	x3.84	x9.06

The crucial knowledge from the measurements is that none of the methods is best for all use cases. In general, the configurations with branching are better, as the comparison in the section 3.6.5 indicated. However, the configurations with MIS perform better in the case of the colored textures (except the bone model, where the overhead the gain is small compared to the overhead), while the configurations without MIS are radically better with textures containing much white color. To get optimal performance, a heuristic deciding between the two configurations based on the nature of texture could be beneficial.

Secondly, the effect of switching the medium sampling and geometry intersection gets more evident with the growing model complexity. In production, models containing thousands of vertices are common, so this optimization is valuable.

In summary, we managed to speed up the rendering multiple times when the target is the same variance level. It is impossible to quantify the improvement by a single number since it strongly depends on the given geometry, texture, and algorithm configuration. We could achieve even bigger speedup if thick complex geometry combined with white textures were used.

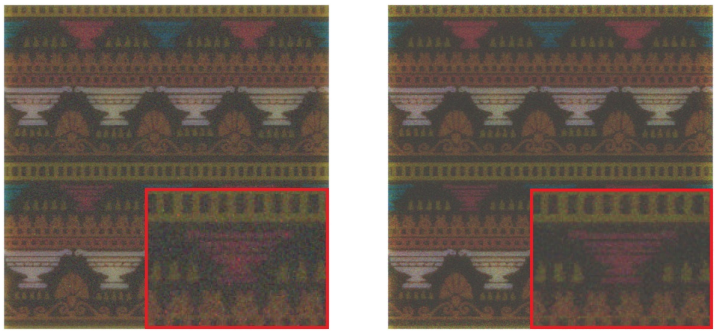
We also include Table 4.6 with the measurements using a different machine. It is an older desktop with Intel i7-4770 with 4 physical cores (8 logical cores) and 8GB of RAM. The only relevant information is the time overheads - they are roughly matching the numbers measured using the primary machine. In some

cases, the overheads are even lower.

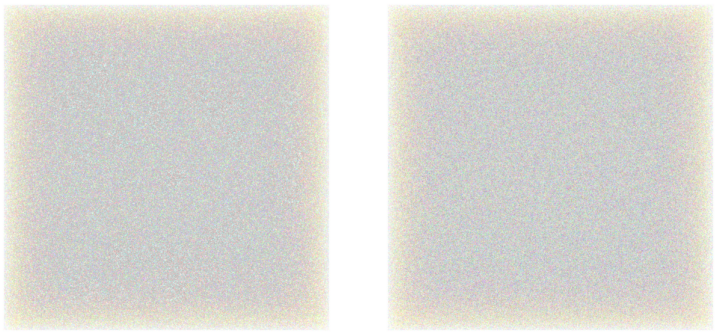
Table 4.6: Comparison with the default (Intel i7-4770, 8GB RAM). The times are in seconds. The setup is described in this chapter, the rest is in the section 3.1.

Model	*Config*	*Time*			*Time overhead (R)*
		R	*G*	*B*	
Box_white_20	simple	290.9	292.9	255.8	-
	custom 3 woNEE	848.3	807.0	811.8	x2.92
	custom 3 NEE	2496.0	2600.8	2388.7	x8.58
Box_greek_20	simple	101.7	78.0	64.8	-
	custom 3 woNEE	194.8	113.3	69.7	x1.92
	custom 3 NEE	386.9	229.2	164.5	x3.79
Box_white_10	simple	58.9	57.6	57.7	-
	custom 3 woNEE	194.8	190.8	20.0	x3.31
	custom 3 NEE	587.9	612.8	650.7	x9.98
Box_greek_10	simple	20.0	19.5	14.5	-
	custom 3 woNEE	43.6	27.2	18.0	x2.18
	custom 3 NEE	101.2	62.8	37.7	x5.02
Animal_white	simple	30.3	33.2	35.2	-
	custom 3 woNEE	49.8	52.4	63.4	x1.64
	custom 3 NEE	145.0	166.4	200.9	x4.78
Animal_spotted	simple	15.8	17.3	22.0	-
	custom 3 woNEE	16.3	17.5	28.4	x1.03
	custom 3 NEE	41.9	46.9	67.3	x2.65
Bone	simple	608.7	406.3	265.3	-
	custom 3 woNEE	873.5	390.6	195.8	x1.44
	custom 3 NEE	2502.3	1125.1	489.4	x4.11

Now let us compare the actual quality of images with the same variance level achieved by different numbers of samples (Figure 4.1). Naturally, we used the results of the perspective camera since they are more comprehensible.

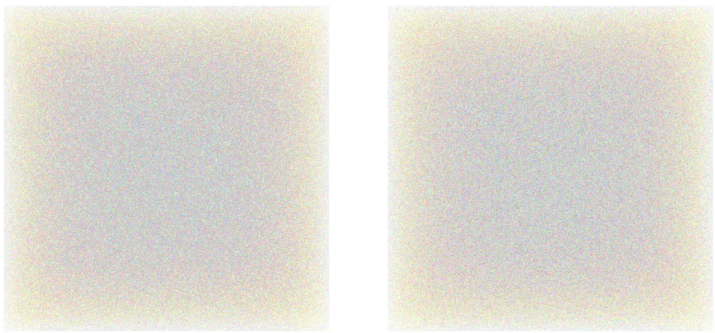


(a) Default setup - 104 spp (b) Our setup - 4 spp



(c) Default setup - 250spp (d) Our setup - 4 spp

(e) Default setup - 92 spp (f) Our setup - 4 spp



(g) Default setup - 250 spp (h) Our setup - 4 spp

Figure 4.1: Comparison of our results with the default Mitsuba setup. Samples per pixel = spp. (Part 1)

(i) Default setup - 224 spp

(j) Our setup - 4 spp

(k) Default setup - 32 spp

(l) Our setup - 4 spp

(m) Default setup - 130 spp

(n) Our setup - 4 spp

Figure 4.1: Comparison of our results with the default Mitsuba setup. Samples per pixel = spp. (Part 2)

The images in pairs have similar variance levels. The comparisons reveal that our approach leads to a more evenly distributed noise with fewer fireflies (see the zoomed areas in the first two images). The edge quality is comparable or slightly better on our side since they are often distorted by the fireflies in the default algorithm's results. The fact that we used much less samples per pixel does not seem to be a disadvantage. The animal model might have some geometry issues causing the black areas.

4.4 Correctness

All presented algorithms are proven to render correct results. First, we compare the results to the default path tracer. The only major change necessary in the default path tracer in order to match the result is terminating the ray after it exits the medium and matching the depth computation (this is a basic property of the custom path tracers, not a bug). The modified version of the default path tracer is included in our codebase as an additional plugin.

Otherwise, there is only a minor difference due to a different process of decision between reflection and refraction - we explicitly decide between reflection and refraction, compute the ray in local coordinates and transfer it to world coordinates via an intersection method that uses *shading normal*. The default path tracer computes either reflection or refraction in the BRDF plugin (and transfers it to world coordinates using the same intersection method) and then decides whether the ray is pointing in or out based on the *geometric normal*. We do not include the additional test because we would lose the advantage of not computing the reflected ray when it is not necessary. In combination with curvy geometry, this may cause a difference up to 0.2%, which is evenly distributed across the image, not concentrated at specific features.

Second, our algorithms pass the white furnace test - a white medium under a white environment light needs to be rendered as completely white. The high density of the printing materials and the albedo being equal to one in the white furnace test cause that the paths have a high number of scattering events until they exit the object, so it is pretty tricky to pass the test. First, there has to be no limit for path length because even if the limit is set to hundreds of bounces, there is a considerable loss of energy. Second, to stabilize the results, many samples need to be used - the large amount of bounces combined with the Russian roulette leads to a low probability that the ray survives to reach the surface. On the other hand, if such a ray hits the surface, it may have a really high throughput caused by the divisions by the Russian roulette probabilities. Thus the results suffer from high variance. We relax the condition to obtain an entirely white image and allow images that have an average intensity equal to 1. Still, we need thousands of samples per pixel to achieve a stable result. In general, denser and larger volumes combined with the lower maximum threshold for the Russian roulette converge slower.

In order to get results without extensive fireflies in a reasonable time, we limit the ray depth to 100. It leads to a certain bias, but the lost energy would be present in the image only in the form of fireflies, which is not desirable. Moreover, the bias is less distinct when materials with lower albedo are on the surface, which is our usual use case. The most significant energy loss is present when dense, white, and thick objects are rendered because the rays have a low probability of reaching the surface within the depth limit. We acknowledge this as a problem, but not only of our solution but of the path tracing in general.

Regarding the Russian roulette probabilities, we tested that different settings have no impact on the result average radiance. Therefore the fact that we changed the maximum probability from 95 to 99% does not cause a difference between our results and the results of the default Mitsuba path tracer [Jakob, 2010].

Future work

4.5 More experiments with Russian roulette

In our experiments, we determined the probability for the Russian roulette only based on the current throughput that was clamped between 0 and a number a little lower than one to avoid getting stuck. Pruning the paths in a more adaptive way could be a tool allowing us to prioritize the rays that are believed to bring a higher contribution. The decisions could be guided by prior knowledge about the geometry shape accessible via the distance and distance gradient grids.

4.6 Optimization for the transparent material

As a shortcoming of our research, we consider the lack of attention to the transparent material. Its properties are in direct contradiction with the high density of the other five basic materials. Fortunately, our algorithm yields correct results even for the media containing the transparent material, only the performance would not be optimal as our heuristics (e.g., for using the light rays or doing the medium sampling before the intersection testing) and the overall decision process was driven by the high density of materials. Furthermore, since the target of our research is texture appearance optimization, we believe that the non-transparent materials will make up most of the volume voxels.

4.7 Rendering multi channel density

Currently, achieving the final image requires the rendering of three separate images, one per channel, and combining them. The reason for this is that Mitsuba supports only monochromatic density of the media.

The problem can be formulated as a spectral rendering with three fixed wavelengths, where one of them is picked, and the path is sampled based on that. However, the probability of sampling a given distance differs across the wavelengths, in our case, across the three channels with different densities. Wilkie et al. [2014] propose to use multiple importance sampling to overcome this problem and construct an estimator that allows to trace multiple wavelengths at once correctly.

The MIS formula contains weights and probabilities for the traced wavelengths, which require evaluation of transmittance - it poses a problem for Woodcock tracking, and the authors propose to use a numerical integration. In our case, the implementation of regular tracking could be used to achieve unbiased results.

4.8 Building a dedicated rendering system

As we outlined in the section 1.3.1, Mitsuba renderer provides a broad set of information during the rendering, so it is easy to implement new plugins without

doing any changes in the core. Our plugins are very lightweight, using only a tiny fraction of the information stored about the geometry intersection or scattering events. Hence there is an ineligible amount of work wasted. Creating a system that computes only necessary information would be much more efficient both in terms of time and memory.

Moreover, the final configuration of the algorithm does not benefit from the modular plugin system since we only use one type of light source, geometry, and other components. Removing the virtual calls would save time and allow more aggressive optimizations.

Finally, the geometry intersection system could be replaced by Embree, which has no competition in this field and would increase the performance significantly. Profiling of the code indicates that more than 50% of the rendering time is consumed by intersection lookups.

Conclusion

We implemented and evaluated multiple algorithms for volumetric rendering, leveraging the specific conditions of appearance optimization pipeline for 3D printers to either reduce the rendering time or variance. A number of the approaches turned out to be ineffective or too complicated. However, we managed to assemble a solution consisting of several custom Mitsuba plugins providing a significant improvement in terms of *performance*. Similarly, our system of of storing material IDs inside a sparse OpenVDB tree reduces the *memory requirements* significantly.

It became apparent that more simple alternatives lead to better results in a broader set of use cases than more complex ones. As a basis of our solution, we selected the unidirectional path tracing as it achieved the best results compared to the other modern volumetric rendering algorithms. On top of that, we added improvements like multiple importance sampling and ray branching. We avoided any necessity to cache paths, medium interactions, and building search structures over them, which resulted in a significant performance drop in the case of bidirectional path tracing or photon mapping like approaches.

The emphasis was placed on the development of algorithms that are easily portable to other renderers. Only a marginal effort was spent on the optimization of Mitsuba-specific code. If we did so, it was in order to eliminate the overhead that would distort the measurements and bias the decisions between particular options. Similarly, most of the improvements would be valid even if some of the assumptions were relaxed, for example, if there was a necessity to render rough surfaces or materials with a different phase function. However, we rely on the fact that the strongest assumptions are inherently implied by the purpose of our solution, and they would not change, like the simplicity of scene and light conditions.

The presented algorithms are easily accessible via a simple command line application as well as a GUI application. However, for this thesis, we only disclose the command line version.

Bibliography

John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87*, pages 3–10, 1987.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975. ISSN 0001-0782. doi: 10.1145/361002.361007. URL `https://doi.org/10.1145/361002.361007`.

Intel Corporation. Embree. `https://embree.github.io/`, v3.13.

Microsoft Corporation. vcpkg. `https://https://github.com/microsoft/vcpkg/`, 2016.

Tom Duff, James Burgess, Per Christensen, Christophe Hery, Andrew Kensler, Max Liani, and Ryusuke Villemin. Building an orthonormal basis, revisited. *Journal of Computer Graphics Techniques (JCGT)*, 6(1):1–8, March 2017. ISSN 2331-7418. URL `http://jcgt.org/published/0006/01/01/`.

Oskar Elek, Denis Sumin, Ran Zhang, Tim Weyrich, Karol Myszkowski, Bernd Bickel, Alexander Wilkie, and Jaroslav Křivánek. Scattering-aware texture reproduction for 3D printing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 36(6):241:1–241:15, 2017.

Ronald Aylmer Fisher. Dispersion on a sphere. *Proceedings of the Royal Society of London*, 1953. doi: 10.1098/rspa.1953.0064.

L.G. Henyey and J.L. Greenstein. Diffuse radiation in the galaxy. *Annales d'Astrophysique*, 3:117, 1940.

Sebastian Herholz, Yangyang Zhao, Oskar Elek, Derek Nowrouzezahrai, Hendrik P. A. Lensch, and Jaroslav Křivánek. Volume path guiding based on zero-variance random walk theory. *ACM Trans. Graph.*, 38(3):25:1–25:19, June 2019. ISSN 0730-0301. doi: 10.1145/3230635. URL `http://doi.acm.org/10.1145/3230635`.

J. Eduard Hoogenboom. Zero-variance monte carlo schemes revisited. *Nuclear Science and Engineering*, 160(1):1–22, 2008. doi: 10.13182/NSE160-01. URL `https://doi.org/10.13182/NSE160-01`.

Artec Group inc. Spinal bone. `http://www.artec3d.com/`.

Wenzel Jakob. Mitsuba renderer, 2010. http://www.mitsuba-renderer.org.

Wojciech Jarosz, Matthias Zwicker, and Henrik Wann Jensen. The beam radiance estimate for volumetric photon mapping. *Computer Graphics Forum (Proceedings of Eurographics)*, 27(2):557–566, April 2008. doi: 10/bjsfsx.

Wojciech Jarosz, Derek Nowrouzezahrai, Iman Sadeghi, and Henrik Wann Jensen. A comprehensive theory of volumetric radiance estimation using photon points and beams. *ACM Trans. Graph.*, 30(1), February 2011. ISSN 0730-0301.

doi: 10.1145/1899404.1899409. URL https://doi.org/10.1145/1899404.1899409.

Henrik Wann Jensen. Global illumination using photon maps. In Xavier Pueyo and Peter Schröder, editors, *Rendering Techniques '96*, pages 21–30, Vienna, 1996. Springer Vienna. ISBN 978-3-7091-7484-5.

Henrik Wann Jensen and Per H. Christensen. Efficient simulation of light transport in scenes with participating media using photon maps. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, page 311–320, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919998. doi: 10.1145/280814.280925. URL https://doi.org/10.1145/280814.280925.

Jaroslav Křivánek, Iliyan Georgiev, Toshiya Hachisuka, Petr Vévoda, Martin Šik, Derek Nowrouzezahrai, and Wojciech Jarosz. Unifying points, beams, and paths in volumetric light transport simulation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 33(4), July 2014. doi: 10/f6cz72.

Ivo Kondapaneni, Petr Vévoda, Pascal Grittmann, Tomáš Skřivan, Philipp Slusallek, and Jaroslav 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.

K Martin and B Hoffman. Mastering cmake: A cross-platform build system. 2010.

Ken Museth. Vdb: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph.*, 32(3), July 2013. ISSN 0730-0301. doi: 10.1145/2487228.2487235. URL https://doi.org/10.1145/2487228.2487235.

Ken Museth. Hierarchical digital differential analyzer for efficient ray-marching in openvdb. *ACM SIGGRAPH 2014 Talks*, 2014.

Ken Museth, Peter Cucka, Mihai Alden, and David Hill. Openvdb, 2012. https://www.openvdb.org.

nataleana. Arte tribal vintage etnico de patrones sin fisuras de garabatos. https://es.123rf.com/.

Jan Novák, Iliyan Georgiev, Johannes Hanika, Jaroslav Křivánek, and Wojciech Jarosz. Monte carlo methods for physically based volume rendering. In *ACM SIGGRAPH 2018 Courses*, SIGGRAPH '18, pages 14:1–14:1, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5809-5. doi: 10.1145/3214834.3214880. URL http://doi.acm.org/10.1145/3214834.3214880.

Jan Novák, Andrew Selle, and Wojciech Jarosz. Residual ratio tracking for estimating attenuation in participating media. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 33(6), November 2014. doi: 10/f6r2nq.

Jan Novák, Iliyan Georgiev, Johannes Hanika, and Wojciech Jarosz. Monte Carlo methods for volumetric light transport simulation. *Computer Graphics Forum*

(Proceedings of Eurographics - State of the Art Reports), 37(2), May 2018. doi: 10/gd2jqq.

M. Pharr and G. Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., 2010.

B.W Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, New York, NY, USA, 1986. doi: 10.1201/9781315140919.

Denis Sumin, Tobias Rittig, Vahid Babaei, Tim Weyrich, Thomas Nindel, Piotr Didyk, Bernd Bickel, Jaroslav Křivánek, Alexander Wilkie, and Karol Myszkowski. Geometry-aware scattering compensation for 3d printing. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 2019.

T. M. Sutton, F. B. Brown, F. G. Bischo, D. B. MacMillan, C. L. Ellis, J. T. Ward, C. T. Ballinger, D. J. Kelly, and L. Schindler. The physical models and statistical procedures used in the racer monte carlo code, 1999.

SVsunny. Leopard print seamless background pattern. black and white. `https://www.dreamstime.com/`.

Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1997.

A. Wilkie, S. Nawaz, Marc Droske, A. Weidlich, and J. Hanika. Hero wavelength spectral sampling. *Computer Graphics Forum*, 33, 07 2014. doi: 10.1111/cgf.12419.

E. R. Woodcock, T. Murphy, P. J. Hemmings, and T. C. Longworth. Techniques used in the gem code for monte carlo neutronics calculations in reactors and other systems of complex geometry. *Applications of Computing Methods to Reactor Problems*, 1965.

List of Figures

List of Tables

A. Attachments

A.1 User documentation

The code for this work is shipped only as a command line application, since we do not want to disclose the full GUI application. The package consists of the following

- complete source code of Mitsuba renderer with our plugins (more details in the Development documentation) and complete dependencies (./mitsuba)
- Mitsuba scene files - root scene and files for particular scene elements that are included in the root scene file (./scenes)
- Testing data - multiple independent scenes containing all necessary data to run the rendering, including geometry and a set of volume files. Note that it is not possible to create a new testing scene due to the fact that we do not provide the application that was used to create the data (it does voxelization, halftoning and creating of the OpenVDB or dense grids) (subfolders of ./scenes)
- scripts for making the Mitsuba solution, installing the binaries into the run directory and executing the rendering with various parameters (root folder)
- directory with prebuilt binaries necessary to run the renderer (./rundir)

A.1.1 Requirements

- 64-bit Windows 10
- Python, minimum version 3.9
- Microsoft Visual C++ 2015-2019 Redistributable (x64)
- CPU with the support for AVX2 (due to the prebuilt Mitsuba dependencies)
- Microsoft Visual Studio 2019 (only for building)
- CMake, minimum version 2.8.3 (only for creating the solution)

A.1.2 How to use it

The rendering can be run immediately without any building using the prebuilt binaries running the command `python renderer.py run <options>` from the root directory. The options will be listed and explained in the following chapter. When the rendering finishes, the result images are in the root folder.

The binaries can be also built from the attached Mitsuba sources. For that, the following steps are necessary

- in the root folder, run `python renderer.py cmake` - it creates the Mitsuba solution in ./mitsuba/build

- open the solution and build everything in Release mode, the architecture is set to x64
- in the root folder, run `python renderer.py install` - it creates ./rundir subfolder in the main folder and copies necessary binaries into it
- now, the rendering can be run as `python renderer.py run <options>`, as we noted above

There is a help available via `python renderer.py help`.

A.1.3 Run script parameters

The options always start with "–" followed by one of the these keywords

- *preset* - we defined a number of presets to make the first steps easy, the presets are
  - *original* - runs the version of the algorithm that we used as a baseline for our testing (default Mitsuba volumetric path tracer and dense medium)
  - *best* - runs the version of the algorithm that we considered to be the most efficient (custom path tracer with branching and hybrid heterogeneous medium)
  - *photonmap* - runs the photon mapping algorithm

  Note that the preset can be used in combination with other options, they will override the options defined by the preset.
- *scene* - one of the scene files, the following scenes are available
  - *animal_spotted*
  - *animal_white*
  - *bone*
  - *box_10_greek*
  - *box_10_white*
  - *box_20_greek*
  - *box_20_white*
- *channel* - color channel, it can be RED/GREEN/BLUE or ALL to render all three channels at once. In the last case, the results are automatically combined to a single colorful image
- *integratorPlugin* - Mitsuba integrator plugin, the supported integrators are
  - *volpath_simple_custom* - default Mitsuba volumetric path tracer with minor tweaks described in the section 4.4
  - *volpath_custom1* - custom volumetric path tracer

  - *volpath_custom2_branched* - custom volumetric path tracer with the branching via direction sampling at the first medium scattering event
  - *volpath_custom3_branched* - custom volumetric path tracer with the branching via distance sampling from the surface intersection point
  - *volpath_custom4_branched* - custom volumetric path tracer with configurable multiple-level branching
  - *photonmapper_hetero* - photon mapper with the support for heterogeneous media
- *maxDepth* - maximum number of scattering events for each path, default is 100
- *mediumPlugin* - Mitsuba medium plugin, the supported plugins are
  - *heterogeneous_label* - heterogeneous medium that uses material IDs and LUT to obtain medium properties, uses Woodcock tracking
  - *heterogeneous_homoinside* - heterogeneous medium with the approximation using homogeneous medium core, uses Woodcock tracking
  - *heterogeneous_regular* - heterogeneous medium using hierarchical regular tracking
  - *heterogeneous_regular_simple* - heterogeneous medium using simple, non-hierarchical regular tracking
  - *heterogeneous_hybrid* - heterogeneous medium using simple regular tracking for transmittance evaluation and Woodcock tracking for distance sampling
- *volumePlugin* - Mitsuba volume plugin, the supported plugins are
  - *gridvolume_dense* - grid stored using the dense representation
  - *gridvolume_openvdb* - grid stored using OpenVDB
- *useOrthogonalCamera* - use the special camera used in the optimization pipeline, if false, use basic perspective camera, default false
- *samplesPerPixel* - number of samples per pixel, default 100
- *resolutionX* - image width, default 640
- *resolutionY* - image height, default 640
- *volumePhotons* - number of photons used in photon mapping, default 1 000 000
- *lookupSize* - number of neighboring photons used to compute photon radius in the BRE algorithm in the photon mapping, default 20

Unfortunately, not all plugin combinations are supported. The limitations are the following

- can be used only with volpath_simple_custom, because it does not support distance and distance gradient lookups. On the other hand, volpath_simpl_custom can be used with all media plugins
- photonmapper_hetero requires exactly the heterogeneous_regular_simple medium plugin

A.1.4 Common issues

- *CMake command fails* - The CMake [Martin and Hoffman, 2010] command removes the old ./build folder if it exists, creates a new one and puts the respective data there. If the solution is open, it cannot remove the folder. Therefore, we advise to close the solution first.

A.2 Developer documentation

The codebase of this work can be split into two parts

- a set of Python scripts for creating Microsoft Visual Studio 2019 solution using CMake [Martin and Hoffman, 2010], installing binaries and running the rendering
- Mitsuba renderer with dependency libraries and multiple custom plugins

In the following chapters we describe it in more detail and justify major decisions that we made.

A.2.1 Python scripts

Python is our language of choice mainly because it offers sufficient functionality for manipulation with files, strings and command line arguments. Also, the code is compact and fast to write. On the other hand, the code is not performance-critical.

To improve readability, we split the code into multiple files and functions. The files are

- *renderer.py* - root file, running help/cmake/install/run based on the command line arguments. The functionality for install and run options is in other files, the code for help and cmake options is inlined in this file as it is short.
- *install.py* - makes sure that the necessary source directories are present, the destination directory exists and is empty (recursively deletes it if it already exists), then copies the binaries
- *run.py* - parses the remaining arguments, reads the scene configuration file, asserts correctness of both the command line arguments and the scene configuration. Then it composes the arguments for *mitsuba.exe* and runs it. To be able to split the arguments resolution and configuration parsing into separate functions, we use *TypedDict* in a similar fashion as *struct* in C++

Assertion of correct rendering setup

The *run.py* script makes sure there are no missing/incorrect values both in the command line arguments and in the scene configuration file. Also it asserts that no incorrect combination of plugins is used (see the section A.1.3). However, it would be tedious to check if all referenced files exist and are correct manually, so we transfer this responsibility to Mitsuba/OpenVDB. The libraries provide comprehensible messages in cases like missing geometry file, incorrect or missing volume file.

A.2.2 Mitsuba & custom plugins

The research codebase as well as the application mentioned in the section 1.2 use Mitsuba renderer, so it is natural to use it as a base for our research. It allows easy adding and testing of new algorithms. We fork Mitsuba code from the commit $c25a40b68ccb91af3ae50bc368c054fbe095126e$.

Changes in Mitsuba

The Mitsuba code is almost in the original state, except

- hints to find dependency libraries in the CMake scripts
- a set of changes necessary to build the code in Microsoft Visual Studio 2019
- medium.h & volume.h - added API functions for obtaining material IDs (labels), distances to surface and other things necessary for the custom plugins
- custom plugins, will be discussed later

Dependencies

We include the basic Mitsuba renderer dependencies package available at $https://github.com/mitsuba-renderer/dependencies\_win64/$. Additionally, we add OpenVDB and its dependencies obtained using vcpkg.

For simplicity, we do not include source code for the dependencies, only necessary includes.

Custom plugins

The additional plugins are

- *gridvolume_dense (volume)* - dense representation of 3D grid
- *gridvolume_openvdb (volume)* - OpenVDB representation of 3D grid
- *surfacevoxels (sensor)* - special camera used in the optimization pipeline that is shooting rays in the orthogonal direction to the surface
- *heterogeneous_label (medium)* - implements Woodcock tracking that reads the medium physical properties as material IDs translated using a LUT (this is used also in the other medium plugins)

- *heterogeneous_homoinside (medium)* - heterogeneous medium that approximates its inner core using white homogeneous medium
- *heterogeneous_regular (medium)* - implements hierarchical regular tracking
- *heterogeneous_regular_simple (medium)* - implements simple (non-hierarchical) regular tracking
- *heterogeneous_hybrid (medium)* - medium plugin that uses Woodcock tracking for distance sampling and simple regular tracking for
- *volpath_custom1 (integrator)* - custom path tracer without the branching
- *volpath_custom2_branched (integrator)* - custom path tracer with branching using phase function sampling at the first scattering event, supports also dynamic branching
- *volpath_custom3_branched (integrator)* - custom path tracer with branching using distance sampling at the surface intersection
- *volpath_custom4_branched (integrator)* - custom path tracer with the multiple-layer configurable branching using phase function sampling
- *volpath_simple_custom (integrator)* - modified basic Mitsuba path tracer matching the appearance of the custom plugins
- *photonmapper_hetero (integrator)* - BRE-based integrator that supports heterogeneous media

We minimized the code duplication by separating the common functionality to files that are shared among the plugins. However, we believe it is better to duplicate small parts of the code here and there to improve the readability of the individual plugins.

Most of the plugin parameters are runtime (e.g. sample count, resolution), but some of them are compile time (using compiler directives) to maximize the performance.

A.3 Electronic attachment contents

The electronic attachment has the following contents

- Python scripts in the root folder for creating of the solution, installing built binaries and running the rendering
- mitsuba/ - Mitsuba renderer sources with necessary prebuilt dependencies
- rundir/ - contains prebuilt version of the renderer with all dependencies, can be run immediately without any building
- scenes/ - testing scenes