Karlsruher Institut für Technologie

Pin Transmittance Sampling

Master Thesis of

Sidney Hansen

At the Department of Informatics
Institut für Visualisierung und Datenanalyse,
Lehrstuhl für Computergrafik

January 20, 2025

Reviewer:	Prof. Dr.-Ing. Carsten Dachsbacher
Second reviewer:	Prof. Dr. Hartmut Prautzsch
Advisor:	Dr. Johannes Schudeiske

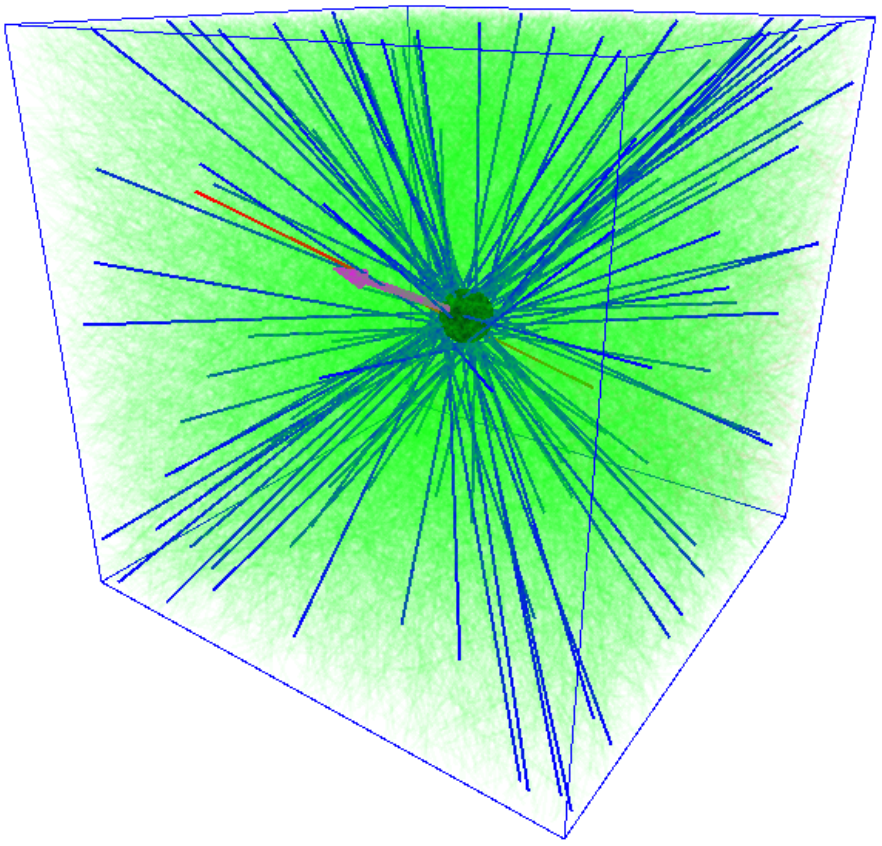


Figure 1: Pins

Contents

Zusammenfassung

Wir stellen eine neuartige Methode vor, um heterogene Volumen für Monte-Carlo Path-Tracing auf der GPU darzustellen. Anstatt das Volumen als Skalarfeld zu speichern, repräsentieren wir es als eine Menge von Linien, die durch das Volumen verlaufen. Diese „Pins“ werden durch ein niedrig aufgelöstes 5D-Gitter indiziert.

Das Tracking wird durch das Auswählen des nächstgelegenen Pins ersetzt, der dann zur Schätzung der Transmittanz oder zum Samplen der Distanz in beide Richtungen entlang der Linie verwendet wird. Beide Funktionen werden mittels bitweiser Operationen implementiert und verursachen konstante, niedrige Kosten. Durch die starke Quantisierung auf das Gitter entstehen strukturierte Artefakte. Wenn die Pins jedoch nur für die Berechnung von mehrfach gestreutem Licht verwendet werden, resultiert dies in einem Blur, der weit weniger sichtbar ist. Während der Kontrast an harten Grenzen verloren geht, bleibt der Effekt bei diffusen Volumen kaum wahrnehmbar.

Wir evaluieren unseren Algorithmus für die Berechnung von Mehrfach gestreutem Licht. Für niedrig aufgelöste diffuse Volumen erzielen wir ähnliche Ergebnisse wie beim Ray Marching. Im besten Fall, für das Rendern vieler Instanzen überlappenden Perlin-Rauschens, können wir die Gesamtlaufzeit halbieren. Hierbei verwenden wir eine Gittergröße, die einen Speicher-Overhead von nur 10% der ursprünglichen Texturgröße hinzufügt. Unsere Tests beschränken sich auf kleine Volumen ($\leq$ 0,25 GB), wir schlagen jedoch vor, die Anwendung für größere Volumen in zukünftigen Arbeiten zu untersuchen.

Abstract

We introduce a novel method to represent heterogeneous participating media for Monte Carlo path tracing on the GPU. Instead of storing the medium as a scalar field, we represent it as a set of lines passing through the volume. These 'Pins' are indexed as a low resolution 5D grid. Tracking is replaced by indexing the closest pin and using it to estimate the transmittance or to sample from a free path distribution in either direction along the line.

Both operations are implemented using bitwise operations and come at a constant low cost. Because of the heavy quantization to the grid, structured artifacts are introduced. But, when using pins only for multiple scattering, these result in a blur, which is far less visible. While contrast is lost around hard boundaries, the effect is barely visible for diffuse volumes.

We evaluate our algorithm for multiple scattering. For low resolution diffuse volumes we obtain similar results compared to ray marching. In our best case, we manage to halve the total run time, when rendering many instances of overlapping Perlin Noise. Here, we use a grid size, that adds a memory overhead of only 10% to the size of the original texture. We only evaluate small volumes ($\leq$ 0.25 GB) for our test cases, but suggest exploring the application to large volumes for future work.

1. Introduction

Physically accurate depiction of participating media is a computationally expensive task. Nonetheless, this capability is required across several domains. While it is of particular interest for offline rendering in the film industry, it has been steadily making its way into the realm of interactive applications in recent years. This progress has been facilitated by the rapid growth of GPU computation power over the last decades.

The technique introduced in this thesis applies to Monte Carlo (MC) path tracing performed on the GPU. We do not discuss other physically accurate methods of rendering participating media, such as voxel-based radiosity methods, and refer to the survey by Cerezo et al. for a more complete overview [CPP+05]. We also do not discuss other path tracking methods, such as metropolis light transport, and path guiding, as our method operates orthogonal to the path sampling process. In any case (bidirectional-) MC path tracing is widely adopted, and may be considered the standard approach to obtain highly accurate renderings.

While path tracing has long been considered too expensive to be performed in real-time, there have been promising examples in recent years. Whether or not the application runs at interactive frame rates, leveraging the GPU for path tracing can lead to significant runtime improvements. The addition of hardware-level ray tracing functionality to recent GPUs allows us to implement our method for a path tracer that runs entirely on the GPU and achieves interactive frame rates for computing 1-sample per pixel.

In this thesis we wish to contribute to the research on GPU path tracing by providing a novel method to obtain free path samples and transmittance estimates for participating media at low cost. In specific, we provide a method to sample a bitmask-representation of the collision probability along a line. We call these lines and their respective stored data 'pins' throughout this thesis. The collision probability is computed for these pins with ray marching. During path tracing, a pin is fetched to avoid stepping through the medium. The bitmask-representation permits direct free path sampling and transmittance estimation. The respective sampling algorithms use word-parallel operations that have a constant runtime.

In the first section we cover the fundamentals of volumetric light transport theory, and lay out the radiative transfer equation. We then move on to explain how to obtain numerical solutions, using the Monte Carlo method in Section 2.2. The path integral formulation gives us a simple way to write these estimates. An estimate is obtained by dividing the

radiance measurement by the probability of the associated path. By sampling paths proportional to their contribution the variance of this estimate can be eliminated. Because the exact contribution is not know in advance, we can only sample with incomplete knowledge of that value. Importance sampling (Section 2.3) is the umbrella term for a large set of techniques that permits informed sampling of paths, to minimize variance of the corresponding estimate. These techniques are essential to obtain good estimates in a reasonable amount of time. We discuss some of these techniques, that are required for further comprehension.

Our method is designed for rendering scenes that have a participating medium, such as fog or clouds. The associated cost is particularly high, if the medium has a heterogeneous density. In such a case sampling paths and evaluating contributions requires knowledge that can only be obtained by tracking through the volume and repeatedly accessing the local density. To lay the foundations for our method later, we discuss a variety of such tracking methods in Sections 2.4 and 2.5.

Our method also requires performing tracking procedures. But instead of invoking tracking on-demand, the process is undertaken separately and only the results are fetched on-demand. The purpose of decoupling this computation lies in the ability to save and reuse the results, leading to an increased overall efficiency. But this also requires us to store, access and possibly interpolate the results.

Because of the high dimensionality of the data, finding a representation that permits free path sampling has not been successfully attempted before. Therefor we develop our method from scratch. There exist several methods in literature that can be applied to individual components of our problem. Unfortunately, these solutions come with limitations that make them unviable in context of the task we are trying to accomplish. To give a frame of reference, discuss such methods in Section 2.6. Before moving on, we also explain the complications of implementing parallel algorithms and data structures for the graphics processor and name some solutions (Section 2.6.3)

In the following chapter we begin by presenting our method and dividing the overall problem in multiple subproblems. In Section 3.3 we then present an overview of our complete algorithm. In the ensuing sections we derive this algorithm and argue its validity. We start by developing a 1D distance sampling solution 3.4. Next, in Section 3.4.2, we incrementally build and validate our sampling strategy. That strategy requires deriving a discrete representation of the density along a line. This representation constitutes the core component of our algorithm. In particular, it contains a bitmask that stores binary sampling decisions. To avoid correlation between resulting distance samples, we suggest to continuously resample this mask. This leads to a stream of values, that is kept alive during path tracing. We extend this stream to a stream of pins, to further reduce correlation. In the following section, we use the same representation to estimate transmittance.

Fetching pins requires us to solve a high dimensional point location problem. We address this problem for reading (Section 3.6) and writing (Section 3.7) pins. We resolve to use a 3Dx2D nested grid. Each entry within corresponds to a range of incident angles for the range of points inside a grid cell. These entries contain the representation for the latest pin that was computed by the stream and which intersects the corresponding region of space.

In the remaining sections we describe our solutions to several problems that are required to complete the algorithm so that it can be applied in practice. These are the random generation of a bitmask with a desired bit probability (Section 3.8), the further reduction of correlation through various means (Section 3.9) and finally the practical implementation inside a GPU path tracer (Section 3.10).

In the following 2 chapters we evaluate our algorithm and discuss the results. Because we do not handle large scale volumes in our application, the evaluation is restricted to volumes with a maximum size of $\leq$ 250 MB. We find that our method works well when paired with ray marching for the primary ray and single scattered direct illumination. Our method blurs contrast when volumes have hard boundaries, but converges faster in cases where ray marching needs to perform many steps. Also, we discover that a small number of pins is sufficient to represent the volume for multiple scattering while storing large amounts of pins comes with diminishing returns.
In addition, we find that recomputing pins while rendering does not lead to any improvements when using pins only for multiple scattering.

Finally, in the future work Section 7, we discuss how we would proceed to improve our method and apply it to large volumes. Possible modifications to individual components of our algorithm provide another source for future work. We do not list all of these in the end, but mentioned them at hand in chapter 3.

2. Literature Review

2.1 Volumetric Light Transport

Simplifying assumptions: Physical accuracy implies, that we base our rendering algorithms, on a physical model of the real world. Such a model is not equivalent to the real process, but must make simplifying assumption. In cases where these assumptions do not hold, the model must be adapted. In short, we simulate light as photons, traveling along piecewise straight trajectories (rays). We do not account for wave optics, electromagnetic optics or quantum optics. Also we do not simulate photons in time, but always regard their complete trajectories. Thus, measurements are always taken from the ray density, not the photon density.

Radiometry: To express these measurements we use radiometric quantities. In the following we list such quantities, that are most important to us. In every case these quantities are evaluated per wavelength.

- **Flux / Radiant power Φ:** Energy per differential unit of time (watt)
- **Radiance $L(x, \omega)$:** Flux arriving at a differential surface x (or cross section of a particle) at a differential angle ω (watt per steradian per angle)
- **Irradiance:** $E(x)$ Flux arriving at a differential surface x (or cross section of a particle).

Radiance: To simulate an (idealized) virtual camera at a location x, we measure the radiance $L(x, \omega)$. Each pixel maps to an angular range for ω. For our model of light transport, the rate of change for $L(x, \omega)$ along a ray, is described by the radiative transfer equation (RTE).

$$\frac{\partial}{\partial \omega} L(x, \omega) = \mu_a(x) L_e(x, \omega) - (\mu_a(x) + \mu_s(x)) L(x, \omega) + \mu_s(x) \int_\Omega f_s(\omega \cdot \omega_i) L(x, \omega_i) d\omega_i \quad (2.1)$$

In the following we motivate the terms of this equation. Next we rewrite the equation to the more well known integral form. Then we extend it to account for surface interactions using the surface rendering equation [Kaj86].

The radiance value along a ray changes based on 4 phenomena.

- Increases:
  - emission: $\mu_a(x)L_e(x,\omega)$
  - in-scattering: $\mu_s(x)\int_\Omega f_s(\omega\cdot\omega_i)L(x,\omega_i)d\omega_i$
- Decrease:
  - absorption: $-\mu_a(x)L(x,\omega)$
  - out-scattering: $-\mu_s(x)L(x,\omega)$

Optical properties: The coefficients μ_a and μ_s quantify the local probability of light being scattered or absorbed by the participating medium. Instead of writing μ_a and μ_s we often use the *extinction coefficient* (or medium density) $\mu_t = \mu_a + \mu_s$ and the *scattering albedo* $\frac{\mu_s}{\mu_t}$ to describe the medium. For a *homogeneous* medium, the extinction coefficient is constant.

Scattering: A scattering event changes the direction of a photon. The phase function f_s quantifies the probability of that direction over the angle. In-scattered radiance at a location y is evaluated by integrating the product of f_s and the incident radiance over the sphere Ω.

Optical density: The integral over the extinction coefficient μ_t along a ray segment is called the *optical density* and can be expressed as:

$$\tau(x,y) = \int_0^d \mu_t(x + t\cdot\omega)dt \tag{2.2}$$

Here, d is the length and ω the direction of the segment. The optical density can be additively combine along the ray segment:

$$\tau(x,z) = \tau(x,y) + \tau(y,z)$$

Where y is a point along the segment $\overline{xz}$.

Intergral form: To obtain the value $L(x,\omega)$ the equation 2.1 needs to be integrated on both sides. We use y for $x + \omega\cdot t$. This yields:

$$L(x,w) = \int_0^\infty T(x,y)\left(\mu_a(y)L_e(y,\omega) + \mu_s(y)\int_\Omega f_s(\omega\cdot\omega_i)L(y,\omega_i)d\omega_i\right)dt \tag{2.3}$$

Transmittance: The term $T(x,y)$, which is introduced by the integration of the differential equation is called the *transmittance*. The transmittance can be written as

$$T(x,y) = \exp\{-\tau(x,y)\} \qquad (= e^{-\tau(x,y)}) \tag{2.4}$$

The process of exponential attention for light passing through a medium is also referred to as the *Beer-Lambert law*. Because of the mathematical properties of exponentiation:

$$T(x,y)\cdot T(x',y') = \exp\{-\tau(x,y)\}\cdot\exp\{-\tau(x',y')\} = \exp\{-(\tau(x,y)+\tau(x',y'))\}$$

Surface interaction: Most scenes define hard surfaces. The in-scattered radiance at the surface is expressed by the (surface) rendering equation [Kaj86].

$$L(x,w) = L_e(z,w) + \int_\Omega f_r(\omega_i,z,\omega)L(z,\omega_i)(\omega\cdot\omega_i)d\omega \tag{2.5}$$

f_r denotes the bidirectional scattering distribution function (BSDF). We do not discuss the rendering equation here. The surface interaction allows us to set an upper integration bound for the RTE.

In the rest of this thesis we use the integral form of the RTE that accounts for surface interaction. We also leave out volumetric emission, as this is always 0 for our rendered volumes.

$$\begin{aligned} L(x,w) = T(x,z)\left(L_e(z,w) + \int_\Omega f_r(\omega_i,z,\omega)L(z,\omega_i)(\omega\cdot\omega_i)d\omega\right) \\ + \int_0^d T(x,y)\mu_s(y)\int_\Omega f_s(\omega\cdot\omega_i)L(y,\omega_i)d\omega_i dt \end{aligned} \tag{2.6}$$

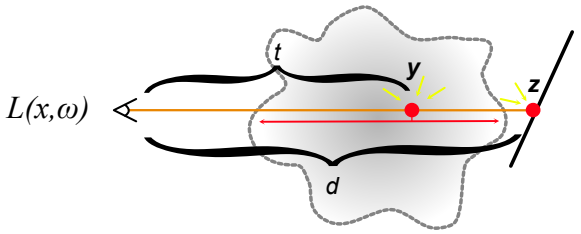


Figure 2.1: Illustration of in-scattered contributions, in context of the equation above.

2.2 Monte Carlo Path Tracing

To obtain numerical solutions to equation 2.6 there exist different methods. Monte Carlo (MC) integration is the most versatile, but also produces noisy estimates. Radiosity methods are an alternative approach, but are more restrictive in their application.

MC Estimate: We can estimate an integral by sampling the respective function at random locations x_i and averaging the result:

$$\frac{1}{N}\sum_{i=0}^{N}\frac{f(x_i)}{p(x_i)} \approx \int f(x)dx \tag{2.7}$$

$p(x)$ is the probability density function, which we use to sample x_i. As long as $p(x) > 0$ for all locations where $f(x) \neq 0$, and $x_0, x_1, ...$ are independent random variables we get:

$$E\left[\frac{f(x)}{p(x)}\right] = \int f(x)dx \tag{2.8}$$

Therefore, the MC estimate will eventually converge. We can also perform MC estimation recursively, which allows us to in turn to estimate recursive integrals.

Path Integral: In the case of the RTE we do just that. Each instance of the MC integration, therefor corresponds to a geometric construction step of a path in 3D space. The product over the probability density functions for these sampling steps determines the probability of the entire path estimate.

Instead of writing these estimates recursively we use the path integral formulation introduced in the dissertation of Veach [Vea98]. We also adopt the notation of the survey on MC methods by Novak et al. [NGHJ18].
A path $\bar{x} = (x_0, x_1, ..., x_k)$ is an element of the path space $\mathcal{P}$. The integral for the radiance arriving at a pixel j is equivalent to:

$$I_j = \int_{\mathcal{P}} f_j(\bar{x})d\bar{x} \tag{2.9}$$

The function f_j is the *measurement contribution function*. The measure space is the infinite set of paths of length k, defined by their vertices. The corresponding measure is the product vertex-area measure $d\bar{x}$. The integral must be understood as the sum over contributions of all path lengths $k \in [1, \infty)$. In this thesis we only require the MC estimate, which can be written as before:

$$\frac{1}{N}\sum_{i=0}^{N}\frac{f(\bar{x_i})}{p(\bar{x_i})} \approx \int_{\mathcal{P}} f(\bar{x})d\bar{x} \tag{2.10}$$

Where only the meaning of f and p have changed and the recursion has been eliminated. We leave out j as it is not relevant to us. Mainly, this expression allows for an easier reasoning about path tracing. For the exact definition of f via a Neumann series we refer to the survey by Novak et al.[NGHJ18].

2.3 Importance Sampling

Above we mention that MC integration produces noisy estimates. In statistics, the error of an approximation is usually quantified by giving the *mean square error* (MSE). For an estimate X we have:

$$MSE = Bias(X)^2 + Var(X) \tag{2.11}$$

The noise comes from the variance (Var) in the estimate. To decrease the noise in MC integration, following actions are equivalent:

- take 4 times the amount of samples
- half the standard deviation of a single sample

Given a fixed time frame, the former can be achieved by reducing computation time per sample. We seek to achieve this with our method in chapter 3.
The latter can be accomplished by sampling paths with a probability density p that matches the contribution f more closely. Hence, more important paths are sampled at a higher rate.
When constructing the path recursively, this can be achieved by sampling from distributions proportional to terms in the RTE. A common way to sample from a given distribution is *inverse transform sampling*. However, only performing sampling decisions based on local information creates a random walk, that is unlikely to quickly reach a light source. Thus, contributions are usually collected from light sources via *next event estimation*.

Inverse Transform Sampling: Given an analytic function $f : [a, b] \to \mathbb{R}$, inverse transform sampling allows us to transform (pseudo-) random numbers from a uniform distribution between $[0, 1)$ to a distribution over $[a, b)$ with a probability density $p \propto f$. In order to perform inverse transform sampling we need to evaluate the inverse of the cumulative distribution function (CDF) $F(x) = \int_a^x p(t)dt$. Higher dimensional functions can be sampled by marginalizing the distribution and sampling each dimension separately. The difficulty of evaluating $F(x)^{-1}$ in an efficient manner, limits the amount of distributions that can be sampled in this fashion.

Sampling only local properties, such as the BSDF or the phase function, ignores variance induced by the incident radiance over the angle, $L(x, \omega_i)$. Unfortunately, the recursive definition of $L(x, \omega)$ makes it impossible to sample the illumination term. This can be amended by splitting it into direct and indirect illumination and estimating the respective integrals separately.

Next-event estimation: Next event estimation is used to sample the direct illumination. Instead of testing for the closest hit along a sampled ray direction, NEE samples the

next path vertex directly at the light source. To compute the contribution, the visibility along the respective ray needs to be evaluated. For a participating medium this requires computing a *transmittance estimate*.

The indirect illumination is estimated by sampling the incident angle. In that case the closest hit along the ray must be found. For participating media, that location is stochastically determined by the density along the ray. Importance sampling that location requires sampling from the *free path distribution*.

In the following section we discuss algorithms to obtain these 2 estimates.

2.4 Distance Sampling

In this section and the next one, we parametrize all terms along the length t of a given ray.
Sampling free paths is achieved by sampling from the collision probability density:

$$p(t) \propto T(t)\mu_t(t) \tag{2.12}$$

A distance can be sampled from another distribution. In that case, the resulting estimate must be weighted, which requires evaluating the transmittance. Hence, weighted distance sampling introduces variance from the transmittance. This can be preferable in cases, where this allows reducing other sources of variance.
In the following we only consider free path sampling.

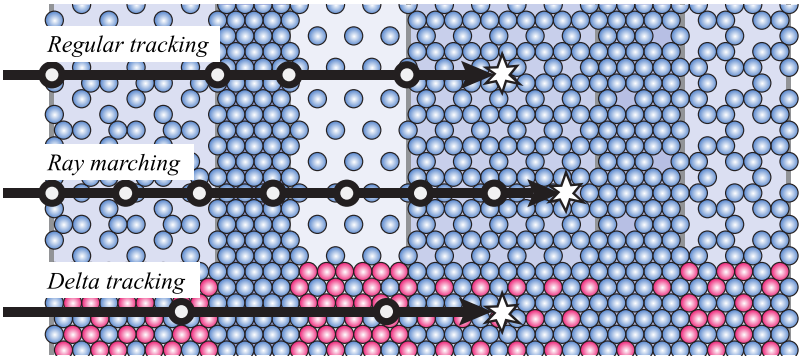


Figure 2.2: Illustration of the algorithms described in this section. (Source: [NGHJ18])

Analytic: For a homogeneous medium the free path distribution can be sampled analytically via the inverse CDF method. A distance sample is obtained by from a uniform random variable $\xi \in [0, 1)$ by computing:

$$t = \frac{\ln(1-\xi)}{\mu_t} \tag{2.13}$$

To extend this estimate to heterogeneous media there exist different algorithmic approaches.

Regular tracking: The most direct application to heterogeneous media is regular tracking. Here, we assume a constant collision coefficient μ_t over regions of space inside the medium. In practice, this approach is often combined with a hierarchical data structure, such as an octree, for storing regions with constant density. For each intersected leaf we can directly compute the optical density τ for the local ray segment.

We can obtain a free path distribution, by sampling the traversed optical density.

$$ln(1-\xi) = \hat{\tau} \tag{2.14}$$

Then, regions are traversed along the ray, until the accumulated optical density is larger than $\hat{\tau}$. Finally the exact position inside the last region is determined analytically.

For the given discretization of the medium the computed distance samples are distributed exactly according to the free path distribution. In practice such a discretization is often avoided. Instead ray marching is used.

Ray marching: Instead of discretizing the medium, ray marching makes the assumption that density values are locally similar. For sufficiently small step sizes, the density is thus assumed to be constant between sample locations. To make this assumption valid, the step size must be adapted based on the local smoothness of the collision coefficient. In the the results chapter in Figure 4.22 we show the error produced by using a too large step size. Apart from that, the distance can be sampled as with regular tracking.

Ray marching introduces a bias. This bias comes from estimating (and not computing exactly) the optical density and using that value to compute the transmittance. The transmittance is a convex function. Thus, following *Jensens inequality* [Jen06] the variance in the exponent translates to a positive bias of the transmittance estimate.

Bias is often acceptable, if it results in a slight blur. This can be achieved for ray marching by jittering the sample locations.

In some cases the small step size required for ray marching is too expensive. *Null collision methods* allow to obtain unbiased free path samples and transmittance estimates without relying on an extremely small step size.

2.4.1 Null Collision Methods

Null collision methods were original developed in physics for neutron transport. As with many other techniques, they have been later introduced to computer graphics. These methods rely on a two stage sampling process. They combine a distance sampling step, with a subsequent rejection sampling step. While the runtime is technically unbounded, the resulting distance sample is obtained from the unbiased distribution. The base algorithm is *delta-tracking* (or *Woodcock-tracking*) [WMHL65].

Delta-tracking: While there exists a purely mathematical formulation by Galtier et al., we will here discuss the more intuitive physical interpretation of the algorithm [GBC+13]. We recall that the RTE models linear light transport using the coefficients μ_s and μ_a for describing participating media. Delta tracking introduces a new coefficient μ_n, which describes the density of fictitious matter. The fictitious matter is defined to be perfectly forward scattering, such that losses and gains due to interaction with this matter cancel out.

$$-\mu_n(x)L(x,\omega) + \mu_n(x)\int_\Omega \delta(\omega-\omega_i)L(x,\omega_i)d\omega_i = 0 \tag{2.15}$$

Therefore, the field μ_n may be arbitrarily chosen, without changing the radiance obtained from evaluating the RTE. By setting $\mu_n(x) = \hat{\mu} - \mu_t$ the sum density becomes equal to the upper majorant $\hat{\mu}$ of the collision coefficient μ_t. A weaker upper bound may also be chosen if the majorant is unobtainable. The now homogeneous medium, can be sampled analytically. In return, on each collision, 3 types of interactions need to be evaluated

Delta-tracking does this, by using Russian roulette to decide between a collision with fictitious matter (μ_n), and a collision with real matter ($\mu_s + \mu_a$). The corresponding probability of a collision with real matter is.

$$P(real) = \frac{\mu_t(x)}{\hat{\mu}}$$

The fictitious matter has no effect on the overall light transport. Due to the Russian roulette sampling no additional weights are introduced. Thus, the first collision with real matter must be distributed according to the exact free flight distribution.

Majorant: Achieving a good performance with delta-tracking comes down to reducing the number of rejected collision before one is accepted. This number is lowered, when using a tight upper bound. One way to obtain a tight bound can be achieved by computing the majorant locally and storing it inside an octree. In that case delta-tracking must be performed per intersected grid cell.

Decomposition tracking: Delta-tracking can also be improved by combining it with analytic distance sampling via *decomposition tracking* [KHLN17]. As stated earlier, we can compute the transmittance over a sum of optical density values, by multiplying the respective transmittances. Correspondingly one can also sample the distance for such a decomposition, by sampling the minimum distance over all components. This corresponds to the closest hit determination step in ray tracing algorithms. Decomposition tracking extracts a homogeneous component from original density field. Distance sampling is then performed on the homogeneous and heterogeneous components using the analytic method and delta tracking respectively. The advantage gained from this, is that delta tracking can be stopped, if the current track length surpasses the track length sampled on the homogeneous component. In chapter 3 we draw inspiration from this, but propose a different decomposition of the medium density.

$$L_{scatter}(x,w) = \int_0^{t_{max}} T(t)\mu_t(t)\frac{\mu_s}{\mu_t}(t) \int_\Omega f_s(\omega \cdot \omega_i) L(t,\omega_i) d\omega_i dt \tag{2.16}$$

Figure 2.3: Free path sampling, importance samples the integral for the local collision probabillity (marked in red). Hence, these terms do not need to be evaluated thereafter.

2.5 Transmittance Estimation

When sampling distances using the free path distribution, we are spared evaluating the transmittance. If the transmittance term is not sampled, or if the probability density function (PDF) needs to be evaluated for other reasons (e.g. multiple importance sampling), the transmittance needs to be computed.

Track-length estimator: To estimate the transmittance over a ray segment, we can use all the algorithms discussed in the previous section to sample a free path length $\hat{t}$. We can use this value to estimate transmittance.

$$T(t) = \begin{cases} 1 \ if \ \ \hat{t} \leq t \\ 0 \ else \end{cases} \tag{2.17}$$

Of course such a binary estimate produces a large variance.

Expected-value estimator: For ray marching and regular tracking we accumulate optical density along the length of the ray. Naturally we can use this value to directly compute the resulting transmittance.

Ratio-tracking: When performing null-collision methods, the optical density τ is never evaluated. Therefore, the exact transmittance value is unknown. Even so, the available information can be used to obtain an estimate, that is better than the binary track length estimate. To do this, delta tracking is carried out along the ray segment, without yet performing the collision rejection sampling step. Instead, tracking is continued until the

end of the segment. For such a path, the collisions $c_0, c_1, ...$ can be rejection sampled independent of each other. Hence the expected value of the transmittance is the product of the respective rejection probabilities:

$$\begin{aligned} E[T(t)] = P(T(t) = 1) &= E[\prod_{i=0}^{k} P(c_i = declined)] \\ &= \prod_{i=0}^{k} E[P(c_i = declined)] \\ &= \prod_{i=0}^{k} \frac{\mu_n(t_i)}{\hat{\mu}} \end{aligned}$$

Therefore, the rejection sampling step never needs to be performed. Instead the product over the ratio $\frac{\mu_n(t_i)}{\hat{\mu}}$ is accumulated. In chapter 3 we obtain a transmittance estimate in a similar fashion. We also use a track-length estimator and leave out the rejection sampling step.

Residual ratio tracking: The estimate can be further improved by decomposing the medium in a homogeneous a heterogeneous part. Analogous to decomposition tracking the transmittance over the homogeneous part is computed analytically and combined with the ratio-tracking estimate over the heterogeneous component. The total transmittance estimate is the product of both values.

2.6 Data Representation

We have just seen different ways to estimate transmittance and sample free paths for heterogeneous volumes. To decouple these steps from path tracing, an intermediate representation is required. This representation needs to permit both sampling free paths and evaluating the transmittance efficiently. The transmittance between any 2 locations in 3D space is scalar value that is parameterized over 6 dimensions. The free path distribution is a function that depends on the starting location (3D) and a direction (2D). To store such high dimensional data a compact representation is required. We find different possible solutions in existing literature.

2.6.1 Transmittance

To store transmittance along a ray we find 2 general approaches in existing literature.

Arrays: Depth shadow maps store the exponential shape of the transmittance function via piecewise linear segments of variable lengths [LV23]. To reduce the memory size of the representation, Locovik et al. run a compression algorithm which reduces the number of segments by allowing for a certain amount of error to occur.
Salvi et al. build on this to develop adaptive volumetric shadow maps (AVSM), which uses a fixed size compression scheme allowing for an efficient gpu implementation [SVL$^+$18].

Moments: Peters et al. first suggest the use of moments, to encode multiple occluder-depths in a single texel of a shadow map [PK15]. Münstermann et al. use moments, to store and reconstruct the depth dependent transmittance from the camera perspective [MKKP18]. For the exploration of large volumetric datasets, Rapp et al. decouple the ray marching step from the transfer function evaluation by storing the densities along the viewing ray using bounded trigonometric moments [RPD22].

Both representation would allow us to store the transmittance along a ray. But, also neither permits direct free path sampling, without performing a linear/binary search on the representation.

2.6.2 Storing High Dimensional Data

In the cases above the transmittance is stored in a 2D texture. In our case we need a representation that can handle higher dimensional data. We find an interesting solution to this in *precomputed radiance transfer* [SKS23]. If we store transmittance along a pin we need to access the closest pin for a ray location. We find that *beam photon mapping* solves a similar problem [JNSJ11, JNT+11].

Precomputed Radiance Transfer: The transfer function expresses the effect of outgoing radiance at one location B on the reflected radiance at another location A. It can be computed to consider any number of bounces in between. In the case where no indirect paths are considered it can be computed directly by evaluating the distance and visibility between A and B and multiplying with the local phase function or BSDF at A. To include indirect paths it can be estimated stochastically using MC path tracing.

In its most general form the transfer function is parameterized over 10 dimensions. 3 each for the respective position A and B and 2 each for the respective angles at these locations.

Nonetheless, precomputed radiance transfer (PRT) manages to store the transfer function for participating media inside a 3D grid. Dimensions are eliminated from the original transfer function through various means.

- only store $T_{transfer}$ for directional illumination from infinite distance (eliminate B)
- restrict to diffuse reflection at A (eliminate angle at A)
- use *spherical harmonics* to compress function over the angle to a set of scalar coefficients (eliminate angle at B)

The resulting function representation can be stored and interpolated in 3D space.

In our case such a representation could be used to encode the transmittance over the angle for positions inside a 3D volume. Unfortunately, the total transmittance value does not allow us to sample free paths.

Beam Photon Mapping: One way to reduce the cost per MC sample is to create an approximation of the radiance field $L(x, \omega)$, over the visible part of the scene. This is handled in different ways by different techniques. Photon mapping performs a light tracing step in advance, and stores the traced photons in a spatial data structure [Jen96]. The result is a local representation of the global illumination. Beam photon mapping stores the trajectories of photons as they intersect the volume of the participating medium. Contribution is gathered from these beams by intersecting them with beams along the view rays. To obtain these intersections, photon beams are segmented and stored inside a BVH. This approach can be extended to the GPU, by splatting the beams using the rasterization hardware.

In chapter 3 we discuss using such methods to store and retrieve the beam transmittance along a ray. We discard this approach, because it requires evaluating too many beams for obtaining a single transmittance / distance sample.

Discussion: In computer graphics larger than 3D data structures are usually avoided, because of their exponentially increasing size per dimension. In our case we do not find any way to use a low dimensional data structure. Instead we manage the total size by minimizing the size per entry and storing them at a coarse resolution. In chapter 4 we see that a low resolution can be sufficient for computing multiple scattered illumination.

2.6.3 GPU Efficiency

In theoretical literature, the efficiency of algorithms is often expressed in asymptotic runtime. For randomized algorithms, such as Monte Carlo integration, bias and variance must also be considered. In the recent decades another aspect has gained a lot of significance. The growth rate in the overall performance of streaming processor such as the GPU, significantly outmatches the growth rate of CPU performance (Huang's law). Thus, moving computationally heavy steps to the GPU becomes a major point of concern. The architecture of the GPU allows for massive parallel execution of arithmetic logical instructions but can easily be stalled by incoherent memory access patterns. Additionally, the need to synchronize data structure write-access across many threads can result in poor performance. This makes it hard to use data structures with adaptive topologies such as kd-trees and BVHs on the GPU.

Kd-trees and BHVs: Nonetheless, there exist some examples of efficient GPU algorithms, for such purposes. Zhou et al. propose an algorithm for GPU kd-tree construction and achieve interactive frame rates for dynamic photon mapping [ZHWG08]. Since the insertion of hardware ray tracing capabilities, BVH construction and traversal can be performed on the GPU using modern Graphics APIs such as Vulkan and DirectX12. Khlebnikov et al. implement a parallel irradiance cache construction on the GPU [KVS+14]. They do not only construct, but also progressively updates a kd-tree entirely on the GPU.

Octrees: For sparse structured 3D data, OpenVDB is commonly used across the industry [MLJ+13]. The derivative *NanoVDB*, provides real-time support for the GPU [Mus21]. The format stores data inside an octree. The topology of the octree is encoded compactly via bitmasks. While the data can be edited dynamically on the GPU, the topology of the octree is static.

Discussion: Our method runs on the GPU, but makes use of none of the described techniques in this section. In Section 3.6 we argue our case against kd-trees and BVHs. We implement splatting as an experiment, but also are not able to use it effectively. In Section 3.10 we suggest the usage of the VDB format for future work. We now proceed with introducing our method.

3. Method

In this chapter we present a novel method for acquiring approximate transmittance estimates and free path samples at any point and in any direction inside a heterogeneous participating medium. As we already mentioned in the introduction, we provide a method that represents transmittance along lines that intersect the volume. We call these lines and their respective data pins. This data permits us to sample free paths in the approximate direction of these lines. It can also be used to estimate the transmittance. We later propose updating the stored data continuously while accumulating samples for the final Monte Carlo estimate. During that process, all computationally expensive steps are performed on the GPU.

3.1 Pins

We recall the definition of transmittance and optical density:

$$T(x,y) = \exp\{-\tau(x,y)\}$$
$$\tau(x,y) = \int_0^d \mu_t(x + t \cdot \omega)dt$$

Free path sampling is the sampling of distances with a distribution $p \propto T(x,y)\mu_t(y)$. Precomputing transmittance as a data point in 6D space does not help us with sampling distances. Further, achieving a sufficient sampling rate is hard due to the curse of dimensionality. Therefore,we reformulate the transmittance.

$$T_{(AB)}(\alpha,\beta) = T(lerp(A,B,\alpha), lerp(A,B,\beta))$$

$T_{(AB)}$ is the transmittance along the straight passing through distinct points in 3D space A and B. We suggest precomputing $T_{(AB)}$ for a large number of randomly generated pairs of points on the convex hull of the volume. We call these representations "pins". For a pin ψ passing through A, B we write the transmittance T_ψ. A pin ψ is an element of the 4D space of pins Ψ_V. for a bounded volume V. Instead of using a convex hull we can also sample points A and B on the surface of an enclosing sphere as illustrated in figure below.

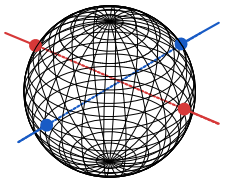


Figure 3.1: *Illustration with two pins (red/blue). Each intersection point per pins can be parametrized by a set of polar coordinates.*

3.2 Problem Statement

Finding a representation: In the last chapter we have seen different methods of storing the transmittance and depth dependent visibility. Such methods use moments or arrays to store the transmittance. We also need such a representation. Compared to the previous work we specifically focus on a representation that can be sampled efficiently.

Finding a data structure: Besides that we also need to find a data structure to store this representation in. We have seen that beam photon mapping uses a BVH to store its photon trajectories. In contrast to beam photon mapping we are not interested in accumulating beams over the hemisphere. When tracking through the volume we are only interested in finding one pin that passes close to our actual trajectory.

First approach: Both these problems seem disconnected from each other. For our first approach this is true. We describe this approach in Section 3.4.1. We use an array to store the prefix sum over the optical density along a pin. This permits us to calculate the transmittance directly and perform distance sampling via a binary search. We find that this approach is too slow when implemented on the GPU.

Second approach: Our second approach is more complex, but requires far less memory and can be sampled in constant time. We develop this approach for distance sampling in Section 3.4. In the following section we extend it to transmittance estimation. The representation relies on sampling from local collision probabilities in advance, and storing the results in a bitmask. Reusing the same bitmask over and over introduces correlation between samples. Therefore, we suggest recomputing pins between frames in Section 3.4.4

This means that we need to be able to continuously update the values inside our data structure on the GPU. With this in mind we discuss different options to store pins in Section 3.6. In Section 3.7 we discuss how to compute pins on the GPU. Finally we resolve to use a 5D nested grid to index pins.

The bitmasks are not our only source of correlation. Fetching pins via accessing the grid creates hard edges and box-like artifacts that can be seen in the results in Chapter 4. We discuss how to deal with these artifacts in Section 3.9. We jitter the access location to the pin grid to blur these lines and make the error less visible. Because the resolution at which we store pins is very coarse we also suggest using other tracking methods such as ray marching for the primary view ray and single scattered illumination.

In the following we show an overview of the full algorithm.

3.3 Algorithm Overview

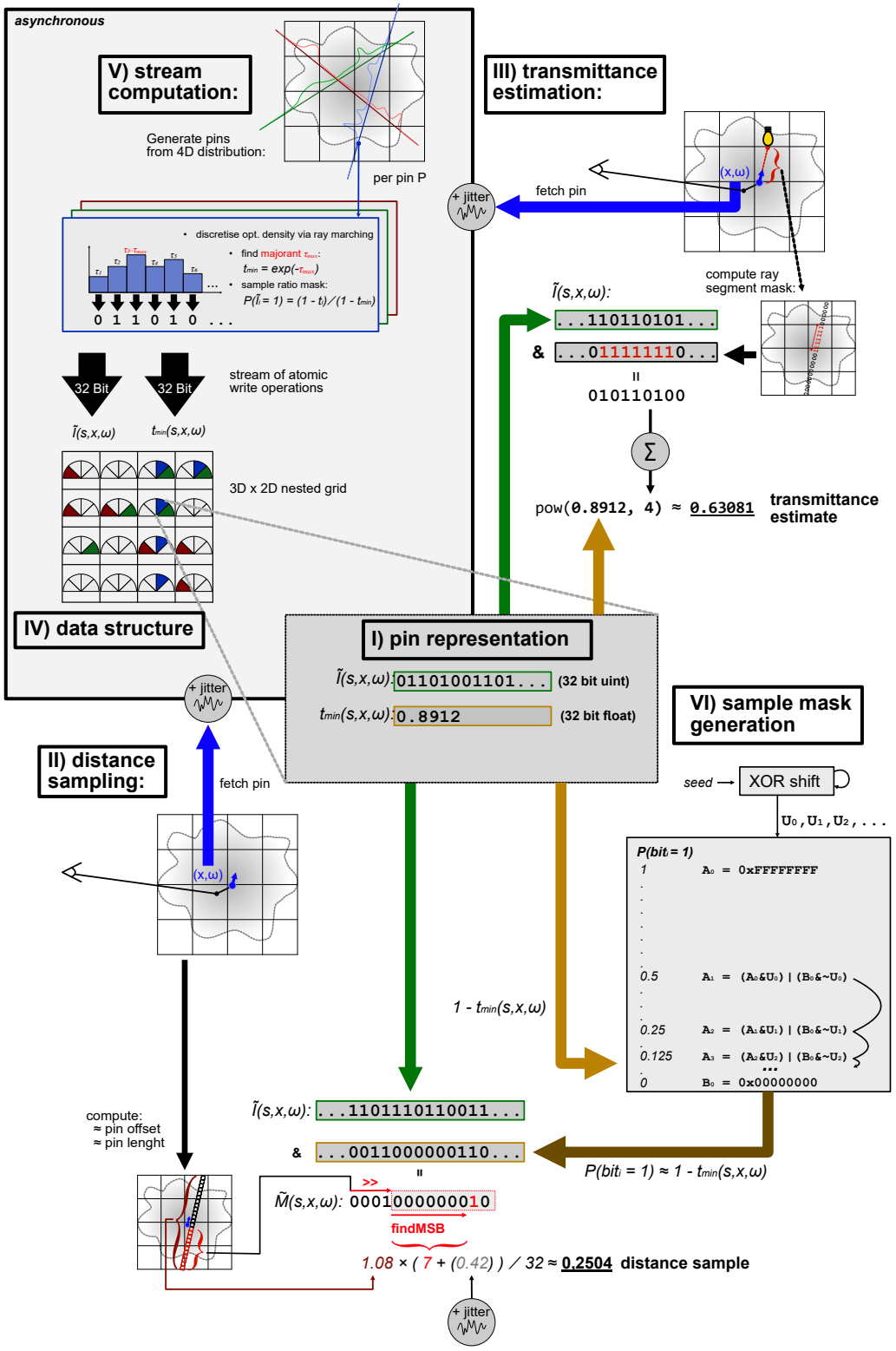


Figure 3.2: Pin sampling: Full algorithm (some simplification). We discuss components in respective sections. (I, II) Distance sampling and representation in Sections 3.4, (III) Transmittance estimation in Section 3.5. (IV) Data structure in Section 3.6. (V) Stream computation in Section 3.7. (VI) Sample mask generation in Section 3.8. We also provide code samples for II, III, V and VI in the appendix.

Table 3.1: Notation for pins

ψ	pin
$T_\psi(\alpha, \beta)$	exact pin transmittance
$\tilde{T}_\psi(\alpha, \beta)$	our approximation
C_i	collision probability at interval i
M	collision matrix
$\tilde{M}$	collision bitmask
I	ratio matrix
$\tilde{I}$	ratio bitmask
H	majorant matrix
t_i	transmittance for pin interval i
t_{min}	minimum over all t_i
$1 - t_{min}$	majorant
s, x, ω	data structure state, pos, direction
$\tilde{M}(s, x, \omega)$	collision bitmask stream
$\tilde{I}(s, x, \omega)$	ratio bitmask stream
$t_{min}(s, x, \omega)$	minimum transmittance stream

3.4 Distance Sampling

We recall that analytic free path sampling can be achieved for homogeneous media via the inverse CDF method. Extending this to heterogeneous media is particularly difficult. This difficulty arises from the fact that the optical density along a ray is in the exponent of the desired distribution $p \propto T(x, y)\mu_t(y)$. Due to the mathematical complexity of integrating and then inverting said distribution, practical analytic free path sampling is restricted to homogeneous media. Rejection sampling offers another way to accurately sample a distribution. But this may require a large number of samples. Also, it still requires a representation of the transmittance along the pin.

In light of the error that is already produced by selecting a pin, we deem it reasonable to only approximately sample the distance. Consequently we use *discrete CDF* sampling as a starting point to develop our sampling strategy.

3.4.1 Discrete CDF sampling

A straightforward way to sample complicated distributions, is to discretize the sampling domain. For a photon traveling along the length of a pin ψ we can evaluate the probability of a collision occurring in a fixed interval. Such an estimator is referred to as a *collision estimator* in literature. We can obtain an estimate of this probability for a discretization into intervals of equal length, by ray marching the optical density $\tau_i = \tau(\alpha_i, \beta_i)$ per interval and computing the transmittance t_i:

$$C_i = (1 - t_i) \cdot \prod_{k=0}^{i-1} t_k$$

$$t_i = \exp(-\tau_i)$$

To sample a free path distance we generate a random value $\xi \in [0, 1)$ and find the first index i such that

$$\xi \leq \sum_{k \leq i} C_k \tag{3.1}$$

If no such value exists, the photon does not collide.

In the current form, a linear search still needs to be performed on the collision probabilities $C_0, C_1, \ldots$. Compared to the tracking algorithms reviewed earlier, the only advantage this offers is the localization of memory access. We discuss in short some modifications to the algorithm.

Arbitrary starting location: Using discrete CDF sampling with a collision estimator for each interval, enables us to sample distances only from the start of the pin ($p(\beta) \propto \tilde{T}_\psi(0,\beta)\mu_t(\beta)$). We can allow distance sampling from any starting location α along the pin, by storing local optical density τ_i instead and computing C_i on the fly. For indices at the beginning and the end of ray segments we write i_α, i_β, with $i_\alpha \leq i_\beta$.

Prefix Sum: In fact, do not need to compute the collision coefficients C_i directly. Instead, we can sample the amount of optical mass that is traversed

$$ln(1-\xi) = \int_\alpha^t \mu_t(s)ds$$

and find the first index i:

$$ln(1-\xi) \leq \sum_{i_\alpha < k \leq i} \tau_k$$

This is effectively regular tracking in 1D. In contrast to the 3D variant, we can replace linear tracking, via a binary search. To enable this, we store the prefix sum

$$\tau_0,\ \tau_0+\tau_1,\ \tau_0+\tau_1+\tau_2, \ldots$$

which lets us evaluate $\sum_{i_\alpha < k \leq i} \tau_k$ in place. We illustrate this process in figure 3.3.
While the binary search reduces asymptotic runtime, it may in practice increase runtime for small arrays on the GPU. This is caused by the incoherent memory access pattern and branching execution paths of the binary search.
Using an array A to store the prefix sum we can also utilize this representation to directly evaluate the transmittance, by computing the total optical density over a segment.

$$\tau(\alpha,\beta) \approx lerp(A[i_\alpha], A[i_\alpha+1], \gamma_\alpha) \ - \ lerp(A[i_\beta], A[i_\beta+1], \gamma_\beta) \tag{3.2}$$

Here, $\gamma_\alpha, \gamma_\beta$ are the weights which used for linear interpolation ($lerp$).

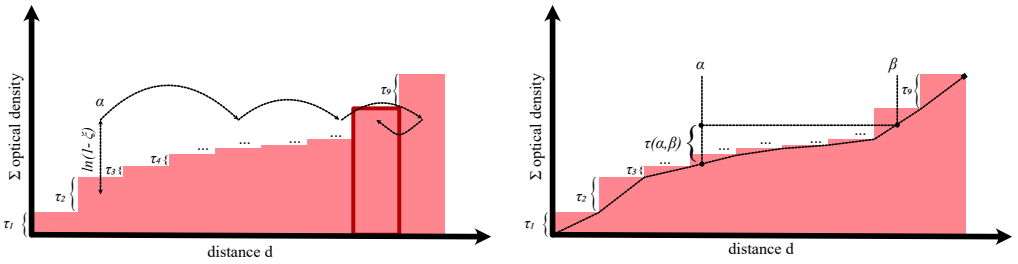


Figure 3.3: *Left: Distance sampling via binary search. Right: Optical density evaluation*

Discussion: We experiment with this method for transmittance estimation at first. Unfortunately, this quickly proves unviable. Distance sampling performs even worse. While the error due to the quantization to pins is expected, we achieve no runtime improvement in return. Therefore, we drop this idea, and instead develop a new approach. We still discretize the density along the pins, but do not store that value in the data structure.

3.4.2 MSB Sampling

The methods discussed so far use one sampling decision to determine how much optical matter is traversed before a collision occurs. Free paths can also be sampled by decomposing the medium density in additive components, as with decomposition-tracking, and sampling track-lengths on each component. These estimates can then be combined by picking the minimum sampled distance. This provides another way to perform discrete CDF sampling. We test if a collision occurred for each interval and select the closest collision, if one occurred at all. Of course, this creates additional work, as it requires far more random sampling decisions compared to the method above. The only benefit, is that sampling can be performed on each interval, independent of the others. In addition, sampling collisions is decoupled from the closest hit determination. As such, it does not require knowledge of the starting location or even the direction of the ray.

This makes us consider presampling these decisions in a Matrix $M \in \{0,1\}^{n \times i_{max}}$ for each pin. Because we only test IF a collision occurs on an interval, M only contains the entries 1 and 0.

Matrix Layout: So, instead of storing the local collision probability of an interval, we directly store samples of the corresponding binary distribution. For the i^{th} interval, the i^{th} column of M contains independent samples $M_{1,i}, ..., M_{n,i}$, with probabilities

$$P(M_{k,i} = 1) = 1 - t_i \; \forall k \in \{1, ..., n\}$$

Given such a matrix, we can obtain a sampling decision for each interval by selecting a row M_j.

Closest-Bit determination: To obtain a distance sample, we traverse the row M_j until we find an entry that is 1 or reach the end of the row. This walk can be performed in both directions. For simplicity we only consider the positive direction here. The probability of sampling the first collision at index i is:

$$\begin{aligned} C_i &= P(M_{ji} = 1) \cdot \prod_{k=0}^{i-1} P(M_{jk} = 0) \\ C_i &= t_i \cdot \prod_{k=0}^{i-1} (1 - t_k) \end{aligned} \tag{3.3}$$

This allows us to sample path lengths equivalent to discrete CDF sampling. By offsetting / inverting the row, we can handle any starting location / walk direction along the pin.

Word Parallelism: Obtaining the offset of the first 1-bit for a 32-bit integer is part of the instruction set of most processors. For CUDA this maps to the 'count leading zeros' (clz) instruction. Hence, by setting $i_{max} = 32$ and storing M_j in a single integer we can perform closest hit determination in a single hardware instruction by computing the most significant 1-bit (MSB). Other instructions are required to offset and potentially reverse the bit order. These also map directly to hardware instructions.

Discussion: At the end of Section 3.4.1, we mention the difficulty of achieving a sufficient sampling rate while keeping the memory requirements in check. The method above allows for constant time free path sampling. But this implies that multiple samples generated for the same pin produce independent distances. This can be approximated by computing a large number of rows for M and randomly selecting one for each sample. The problem is that the number of rows may have to be exceedingly high to avoid correlation bias. This bias, unfortunately, appears in the form of plainly visible structured artifacts.

Consider a low volume density with a minimum transmittance t_{min} of 0.99 per segment. To obtain a probability of $p > \frac{1}{2}$ that for a given interval i at least one row contains a value $M_{ji} = 1$ at least $\frac{-\log(0,99)}{\log(2)} \approx 69$ rows are necessary. Now consider that the volume is also homogeneous and the density can be described accurately using one single 16-bit half float. Clearly, in this case the matrix is a terribly inefficient way of storing the volume density. While this is a special case, we note that from an information theory perspective, the quantity of information stored in the matrix depends on the entropy of the stored data. In the case we just described, probably almost all values are zero, thus the information density is very low.

In the light of this reflection, we propose a more efficient manner to represent the volume. We decompose the collision probability into a (heterogeneous) ratio component and a (homogeneous) majorant component. The ratio is sampled in advance, giving us the desired compression of 1-bit per interval. The majorant is sampled on the fly, allowing us to decorrelate subsequent samples. The decomposition bears resemblance to delta-tracking. It also can be extended in a similar fashion like ratio-tracking, to produce a transmittance estimate.

3.4.3 Extension: Majorant Decomposition

Instead of presampling the matrix M, we suggest following decomposition:

$$M = I \odot H \tag{3.4}$$

where $\odot$ is the component wise matrix product. $I, H \in \{0,1\}^{n \times i_{max}}$ are matrices representing the (inhomogeneous) ratio and (homogeneous) majorant, as discussed above. All entry are computed from independent random variables with following probabilities:

$$\begin{aligned} P(I_{ji} = 1) &= \frac{1 - t_i}{1 - t_{min}} \\ P(H_{ji} = 1) &= 1 - t_{min} \end{aligned} \tag{3.5}$$

Because $t_i, t_{min} \in [0, 1)$ and $1 - t_i \leq 1 - t_{min}$, the probabilities are valid in both cases. Hence for entries in M we have:

$$\begin{aligned} P(M_{ji} = 1) &= P(I_{ji} = 1, H_{ji} = 1) \\ &= P(I_{ji} = 1) \cdot P(H_{ji} = 1) \\ &= \frac{1 - t_i}{1 - t_{min}} \cdot (1 - t_{min}) \\ &= 1 - t_i \end{aligned} \tag{3.6}$$

This is the local collision probability over the given interval. Because all entries in I, H are independent, also all entries in M must be independent. Thus we can use M for sampling free path lengths as described in the last section.

Discussion: Expressing M as the component wise product of I and H, permits separate computation of the respective matrices. Of course, this doubles the number of random bits that need to be computed and stored. But, at closer look, we see that we do not need to presample H. While computing I requires knowledge of the optical density for each interval, generating H requires only the majorant $1 - t_{min}$ of the local collision probability. This value can be stored in a single 32-bit float, or even 16-bit half float value. Hence, we can compute the desired row M_j by reading I_j and t_{min} from memory and generating H_j on the fly, using a (pseudo-) random number generator. We recall that we use bit vectors to store the matrix rows. For combining two bits, the logical AND operation is equivalent

to multiplication. This allows us to perform component wise multiplication by combining pairs of rows (from I and H) using bitwise AND operations.

By storing one instance of I and t_{min}, we are able to obtain a new matrix $M(seed)$ for each distance sample along the pin. Of course, these matrices and thus the sampled distances are still correlated due to the reuse of I. But this correlation is far less than when reusing the same matrix M.

Effectively, instead of having fixed distances for different starting locations in each column, we get distributions of distances. These distributions individually are not free path distributions. But, by selecting one distribution at random and then sampling from that distribution, we approximate the desired free path distribution.

Storing many rows of I may still be required, to reduce correlation between samples to an acceptable level. But, the decomposition greatly lowers this number, especially for volumes with low density and a tight majorant.

We demonstrate this technique in figure 3.4, by comparing the unmodified and decomposed variant in an isolated experiment. We use a fixed number of rows in M and I respectively and accumulate 100 samples per row. For the studied case, the decomposed variant produces a far lower error.

Unfortunately, even with the proposed optimization, storing enough rows for I is difficult, considering our tight memory constraints. An exception to this the homogeneous medium, in which case $1 - t_i = 1 - t_{min}$ and thus all values of I are 1. Hence, storing more than one row is redundant. But of course, in that case we do not need pins in the first place. For small values of $\frac{1-t_i}{1-t_{min}}$ we need an increasing amount of rows to obtain uncorrelated samples. This leads to a dynamic storage requirement, which is problematic for a GPU implementation. In the next section we discuss a method, to store only one row for I. This row in return is continuously resampled. We then explain how this approach can be extended to pins.

3.4.4 Extension: Stream

Randomly selecting a row in I, decorrelates the resulting distance samples. Of course, a better strategy would be to select a different row for each ensuing distance sample, leading to no correlation at all. This can be implemented by managing a stream of rows, such that subsequent distance samples access subsequent elements of that stream. Another advantage is, that the entirety of I does not need to be stored in memory at any time. Instead the bitmask for each row can be computed on demand. We refer to the single row/bitmask stored for a pin as $\tilde{I}$.

Unfortunately, this would defeat the whole purpose of precomputing pins in the first place. We recall that our goal was to gain efficiency by packaging and reusing information, gained from tracking procedures in order to eliminate on-demand computation.

Luckily, the stream approach can be modified to achieve just that. The solution is simple, instead of computing a new element of the stream on demand, we recompute the entry periodically and reuse it for all distance samples in between. If the entries are reused in between distinct (light-) paths, the temporal correlation only reduces convergence rate but does not further bias the estimate. For instance, if one where to randomly shuffle all computed light paths over time that use certain pin, this would effectively mirror choosing a random matrix row M_j locally in time in the original version of the algorithm. Because all samples are averaged in the overarching MC estimate shuffling is unnecessary. We can

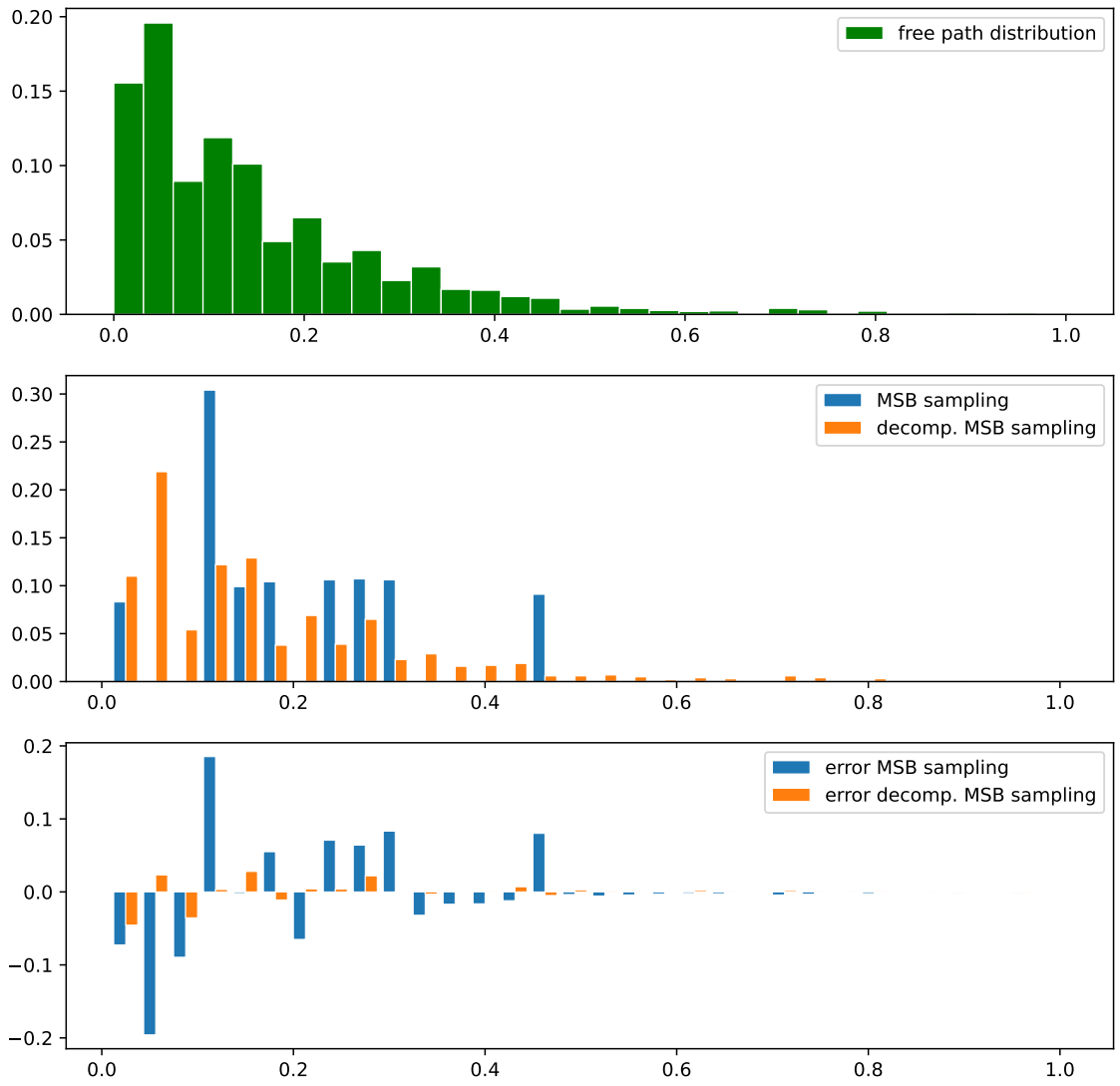


Figure 3.4: *Sampled free path distribution on 32 intervals. Optical density per interval is chosen uniformly at random between 0 and 0.3. Green: True free path distribution. Blue: Distribution obtained with MSB sampling for $M \in \{0,1\}^{10\times 32}$. Orange: Distribution obtained with decomposed MSB sampling with $I \in \{0,1\}^{10\times 32}$. In each case, 100 samples are generated per matrix row, resulting 1000 samples in total.*

write this estimate as follows:

$$\int_{\mathcal{P}} f(\bar{x})d\bar{x} \approx \frac{1}{K \cdot N} \sum_{j=1}^{K} \sum_{i=1}^{N} \frac{f(x_{ij}, s_j)}{p(x_{ij})}$$

Here, K is how often we recompute the values s_j of the stream (entries of $\tilde{I} \in \{0,1\}^{32}$) while N is the number of samples per pixel, computed for the same values of $\tilde{I}$. Note that the pdf p does not depend on our method because we do not sample measurement contribution. The only case, where the correlation becomes problematic, is if the same region of the pin bitmask is accessed multiple times along the length of a light path. Fortunately, this is very unlikely for most phase functions, and a reasonable number of pins.

In summary, the stream extension to our sampling method changes the mental model from precomputing an approximations $\tilde{T}_\psi \approx T_\psi$ to decoupling the sampling rate of the volume

density along the pins, from the overall path sampling procedure. This still accomplish our overall goal, which is the reduction of the combined cost of distance sampling for MC path tracing on the GPU (and transmittance estimation as we see in the next section).

In Chapter 4, we see that the error of pin sampling is mostly caused by quantizing tracking procedures to be performed along the pins, and less by our representation $\tilde{T}_\psi$. For that reason, we extend the stream approach introduced above, to not only update the values of $\tilde{I}$, but the set of pins entirely.

Pin Stream: Hence, in our final version, s_j represents the current set of pins (not only their respective bitmasks $\tilde{I}$). We update this set by continuously resampling pins for our data structure. For practical reasons we do not update all pins at once. Instead, we update them at random to obtain a new set s_{j+1}. Because $s_j \cap s_{j+1} \neq \emptyset$, correlation is introduced between subsequent estimates. Again, this does not break the convergence of the overall MC estimate, but is another instance of decoupling sampling frequencies.

In the illustration 3.2 and for the rest of this thesis we write:

$$\tilde{I}(s, x, \omega),\ t_{min}(s, x, \omega)$$

for the values of $\tilde{I}$ and t_{min} assigned to a region of 5D space $\mathcal{V} \times \Omega$ by our data structure for a given state of the pin stream s. We continue to use $\tilde{T}_\psi$ for the transmittance value along a fixed pin ψ.

In the next chapter we discuss how to use the same representation for transmittance estimation. This changes nothing of the overall approach and enables us to use our method in combination with NEE.

3.5 Transmittance Estimation

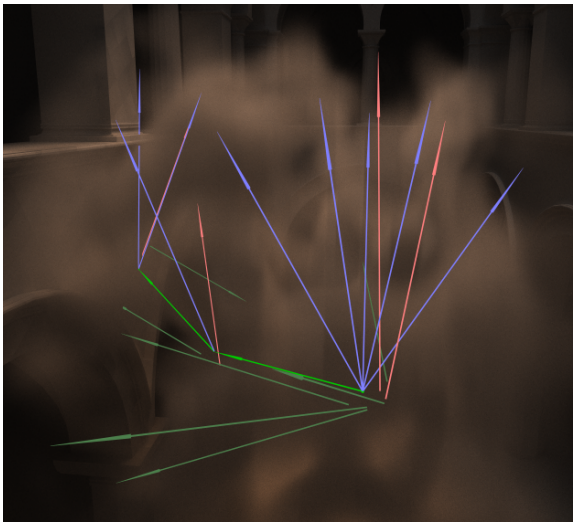


Figure 3.5: *At each path vertex we sample direct illumination with NEE. This requires us to evaluate transmittance inside the participating medium (blue arrows). Hence, a lot of efficiency can be gained by using pins (red arrows) for this task.*

In the last chapter, we explained how transmittance can be estimated along a ray segment by using a track length estimator. This produces a binary estimate of 1 or 0, depending on if the sampled path length surpasses the length of the ray segment or not. Later, we explain how ratio-tracking builds on the delta-tracking algorithm to obtain a lower variance estimate. We build on the decomposed MSB sampling algorithm in a similar fashion. Just

as ratio-tracking, we obtain a better transmittance estimate by removing randomness from the distance sampling algorithm. We do this by omitting the computation of $M = I \odot H$ and instead operate directly on the matrix I. I already contains information about the collision distribution along the pin. In contrast to distance sampling we do not want to sample from that distribution. Multiplying with H only causes us to lose information. We still need to combine I with t_{min}, but we can do this in a way that does not increase the variance by adding another random sampling step. Precisely we compute:

$$\tilde{T}_{s=(i_\alpha,i_\beta)} = t_{min}^{\Sigma(I_{ji}:i\in\{i_\alpha,\dots,i_\beta\})} \tag{3.7}$$

Here, s is the discretized segment along the pin over which we estimate the transmittance. i_α, i_β are the respective indices for the first and last interval of s. We continue to use $i_\alpha \leq i_\beta$ w.l.o.g.. $\Sigma(I_{ji} : i \in \{i_\alpha, ..., i_\beta\})$ is the sum over the row j in I and can be expressed in programming terms as $bitcount(I_j \;\&\; mask_{i_\alpha,\dots,i_\beta})$. Here, $mask_{i_\alpha,\dots,i_\beta}$ is a bitmask that is set to 1 only for the bits $i_\alpha, ..., i_\beta$ and *bitcount* returns the number of 1-bits. This is visualized in the algorithm overview in figure 3.2.

While this formula may seem counterintuitive at first, its correctness can be easily understood for a homogeneous medium. The notation for $\tilde{T}_{i_\alpha,i_\beta}$ restricted to a homogeneous medium is $\tilde{T}^h_{i_\alpha,i_\beta}$. For the general case we show the correctness by proving for the expected value that:

$$E[\tilde{T}_{i_\alpha,i_\beta}] = \prod_{i=i_\alpha}^{i_\beta} t_i \tag{3.8}$$

Homogenous case: For a homogeneous medium we have $1 - t_i = 1 - t_{min}$. Thus, $P(I_{ji} = 1) = 1$. This allows us to simplify the expression above to:

$$\tilde{T}^h_{i_\alpha,i_\beta} = t_{min}^{i_\alpha - i_\beta + 1}$$

t_{min} is the (constant) transmittance per segment and can be expressed as $t_{min} = \exp\{-\mu_t \cdot \Delta\}$. Here, Δ is the length of an interval. Hence we can write $\tilde{T}^h_{i_\alpha,i_\beta}$ as:

$$\begin{aligned}
\tilde{T}^h_{i_\alpha,i_\beta} &= \prod_{i=i_\alpha}^{i_\beta} \exp\{-\mu_t \cdot \Delta\} \\
\tilde{T}^h_{i_\alpha,i_\beta} &= \exp\{-\mu_t \cdot \Delta \cdot (i_\alpha - i_\beta + 1)\} \\
\tilde{T}^h_{i_\alpha,i_\beta} &= \exp\{-\mu_t \cdot \|s\|\}
\end{aligned} \tag{3.9}$$

where $\|s\|$ is the total length of the segment s. For a discretization of the transmittance to the interval and omitting interpolation this is the correct transmittance.

General case: *Proof:* Let $i_\alpha, ..., i_\beta$ be the indices of the intervals over which we estimate the transmittance. Let I_j be the selected row in the matrix I. We recall that all entries in I are generated using independent random variables and that

$$P(I_{ji} = 1) = \frac{1 - t_i}{1 - t_{min}}$$

We start by showing an equivalence. For $b \in \{0, 1\}$, following expressions is true:

$$1 - (1 - t_{min}) \cdot b = t_{min}^b \tag{3.10}$$

(Case $b = 0$) $1 - (1 - t_{min}) \cdot 0 = 1 = t_{min}^0$

(Case $b = 1$) $1 - (1 - t_{min}) \cdot 1 = 1 - 1 + t_{min} = t_{min}^1$

We prove 3.8, by starting at the left side.

$$\begin{aligned}\prod_{i=i_\alpha}^{i_\beta} t_i &= \prod_{i=i_\alpha}^{i_\beta} 1-(1-t_i) \\ ... &= \prod_{i=i_\alpha}^{i_\beta} 1-(1-t_{min})\cdot\frac{1-t_i}{1-t_{min}} \\ ... &= \prod_{i=i_\alpha}^{i_\beta} 1-(1-t_{min})\cdot P(I_{ji}=1) \\ ... &= \prod_{i=i_\alpha}^{i_\beta} 1-(1-t_{min})\cdot E[I_{ji}] \\ ... &= \prod_{i=i_\alpha}^{i_\beta} E[1-(1-t_{min})\cdot I_{ji}]\end{aligned}$$

using the equivalence from above:

$$\prod_{i=i_\alpha}^{i_\beta} t_i = \prod_{i=i_\alpha}^{i_\beta} E[t_{min}^{I_{ji}}]$$

All I_{ji} are independent random variables, thus all $t_{min}^{I_{ji}}$ are independent too. Hence, we can pull out the expected value from the product.

$$\begin{aligned}\prod_{i=i_\alpha}^{i_\beta} t_i &= E[\prod_{i=i_\alpha}^{i_\beta} t_{min}^{I_{ji}}] \\ ... &= E[t_{min}^{\sum_{i=i_\alpha}^{i_\beta} I_{ji}}]\end{aligned}$$

And thus:

$$\prod_{i=i_\alpha}^{i_\beta} t_i = E[t_{min}^{\Sigma(I_{ji}:i\in\{i_\alpha,...,i_\beta\})}]$$

$$\prod_{i=i_\alpha}^{i_\beta} t_i = E[\tilde{T}_{i_\alpha,i_\beta}]$$

■

We have proved that $\tilde{T}_s$ is an expected value estimator for the discretized transmittance along the pin. To be exact, this only holds if each estimate uses a different row I_j. But we already discussed this matter in the last section.

We have now established our pin representation and sampling algorithms. In the next section we discuss how to store and access pins while path tracing.

3.6 Pin Storage

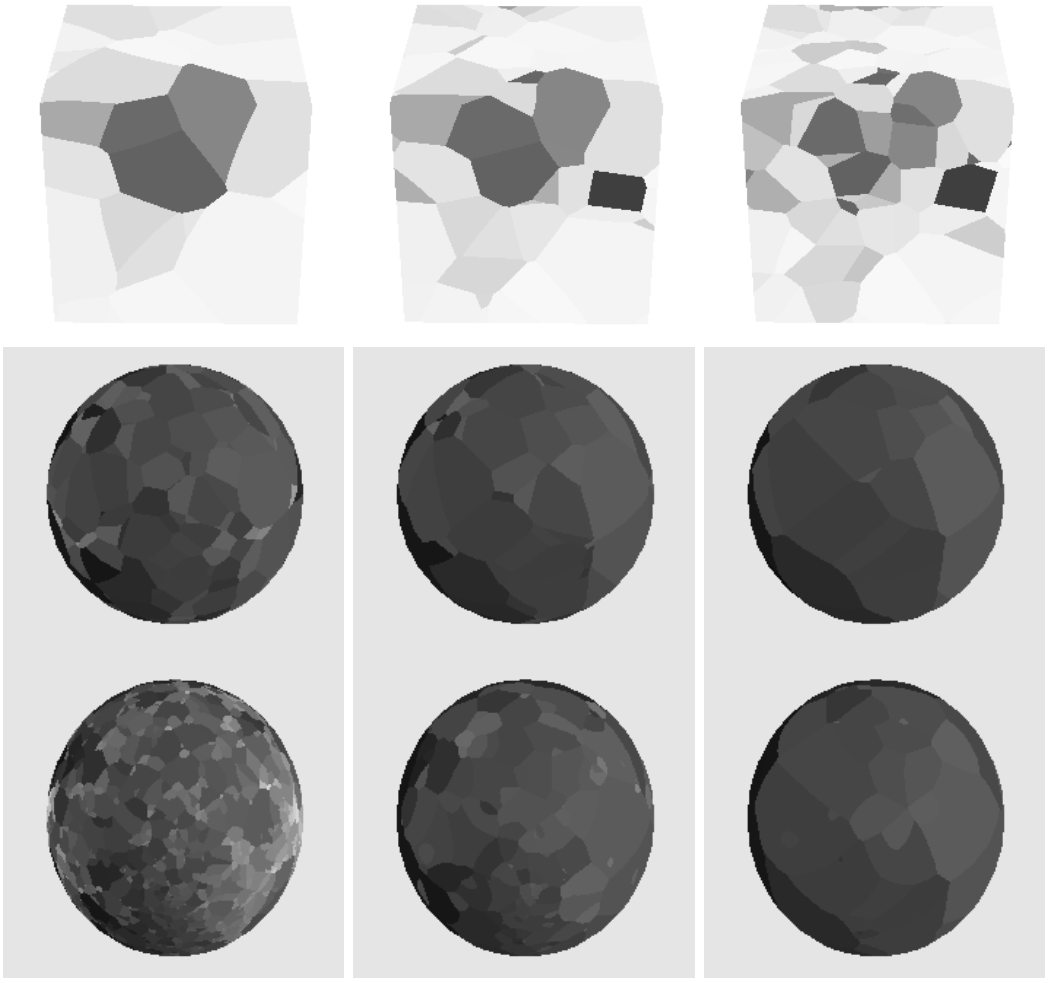


Figure 3.6: *How to select the best pin? In the trial above we experiment with different distance metrics. In the top row we compute the transmittance along the primary ray for a fixed set of pins. At the surface of the bounding volume, we query the (globally) optimal pin for the current metric. In the bottom 2 rows we evaluate the transmittance over the hemisphere for a location inside the medium. For visualization, we project the obtained value to the surface of a sphere. In the top 2 rows we use a distance metric which weighs angular and positional distance. We increase the importance of angular proximity from the right to the left. In the last row we replace angular distance with the positional distance to the end of the ray segment. The shapes visible on the surfaces are projections of the 4D Voronoi cells, implicitly defined by our distance metric. Unfortunately, we are unaware of any data structure that allows us to query pins for such a metric. For this trial we use rasterization and implement an argmin operation with depth testing.*

Finding a mechanism to efficiently locate spatial data is a fundamental problem in computer science. In computer graphics two general approaches stand out. In real-time application rasterization is often used to solve the point location problem. Deferred shading, for example, processes local entities, such as (local) light sources and decals, by rasterizing the corresponding bounding volume. These volumes provide a conservative approximation of the zone of influence for the corresponding entity. Consequently, a set of interactions is generated per pixel. This stream of interactions can then be efficiently processed in parallel on the GPU. The results are written to a framebuffer and can be combined using

a set of fixed functions. This method comes with some limitations. Above all, it requires computing parameters for the point location queries (normal, position...) over a large set of pixels, forming a g-buffer. This approach is not applicable to a recursive path tracer, except for intersections of the primary ray. Photon (beam) splatting, which we discuss in the last chapter, uses this to collect contributions only along the view ray. Nevertheless, we test this approach for primary transmittance estimation, as it allows us to find the globally optimal pin for an arbitrary distance metric (see Figure 3.6).

The other option is the use of a spatial subdivision data structure. These are plentiful and have various trade-offs. Some examples are kd-trees and Voronoi diagrams. Voronoi diagrams are expensive to compute for high dimensions. In our case, the data that needs to be accessed is in 4D. We consider a 4D kd-tree. Olsson uses a 4D kd-tree to perform ray tracing in time continuous data [Ols07]. This requires defining a distance metric. So far, we have not yet defined such a metric. Under the assumption of local smoothness of density values such a metric should express the distance to the pin along the ray trajectory. We do not derive such a metric in this thesis and leave this point for future work. In any case we judge it to be prohibitively difficult to construct a kd-tree for an "optimal" distance metric. As mentioned above we experiment with selecting a pin via a custom distance metric in figure 3.6. But this requires us to consider every pin for each sample location and is clearly not a viable solution. Instead, we chose to solve the easier problem of finding the set of pins which pass close to the sample location and then select one of them, by evaluating their respective angular deviation.

Finding pins in the vicinity of the sampling location bears resemblance to collecting photon beams. Beam photon mapping uses a BVH as storage method. While photon mapping collects contributions over all ray directions, we can only make use of a single pin. To find such a pin would require us to iterate over all pins found via the BVH, and select the best one. This seems too expensive to rival the cost of performing the tracking procedure instead.

Therefore, we suggest using nested data structures. Each entry in the top-level data structure corresponds to a region of 3D space and contains a secondary data structure. The secondary data structure stores pins that intersect the respective region of space and organizes them over the 2D hemisphere.

We recall at this point that for the stream, the data structure needs to be continuously updated during rendering. The total render time thus heavily depends on the efficiency of computing the pins and updating the contents of the data structure. To further narrow down our choices for the data structure, we must first discuss how to compute pins. Therefore, we defer further considerations to the end of the following section.

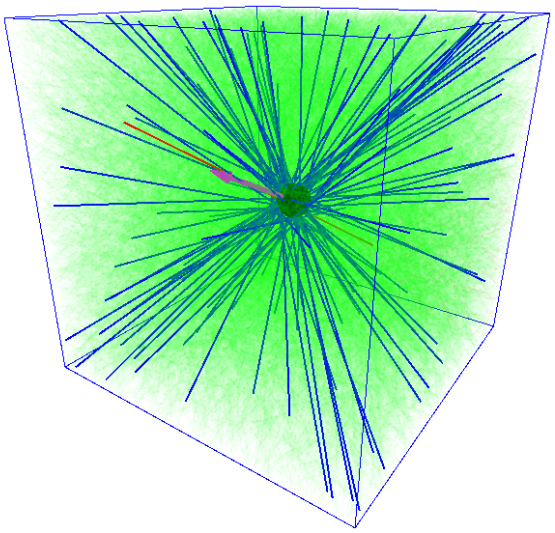


Figure 3.7: *Instead of selecting the globally optimal pin, we first select a set pins that intersects a region of space (blue lines). Then we chose one of these that also has a small angular deviation (red line).*

3.7 Updating Pins

Computing the pin representation is fairly simple and can be performed inside a compute shader on the GPU.

First, we ray march the volume density for each of the 32 intervals of our representation. Then we compute the local coefficient $\frac{1-t_i}{1-t_{min}}$. Next, we use this value to sample the corresponding bit per interval, using a (pseudo-) random number generator.

To avoid having an array of 32 floats in local storage, we compute the coefficients on the fly and only store the resulting bit. Because this requires us to know the value of t_{min} ahead, we have to compute it in advance. We also use ray marching to do this. Unfortunately, this doubles the main cost of pin computation and we suggest investigating any possible optimizations to this process.
Following thereafter we have to store the bitmask and the associated value t_{min} for each pin.

When managing shared write access to a data structure on the GPU there are two general approaches.
The *gather* approach manages access by invoking shader programs per write location. While a single program may write to multiple locations, all addresses of write operations across all shader invocations are predetermined such that they do not conflict.
The *scatter* approach computes the write locations on the fly. For that reason write-conflicts can occur and need to be considered.

For this reason, we first regard the gather approach. We can implement this by running a shader program per node in our top-level data structure. Then we have 2 options.
For one, we can directly ray march a set of pins that intersect the region. These pins are then stored only in the local bottom level data structure, even though they intersect the regions of space corresponding to other nodes. Because our data structure has 5 dimensions and pins are 4-dimensional, this leads to the computation of many redundant pins. This approach is used to initially populate the data structure. The overhead is quite significant.

The other option is to compute pins first and run a second program to insert them into the data structure. In that case all pins need to be tested for intersection with the local

region of space. This may be accelerated using a hierarchical approach. We judge that this could be archived efficiently on the GPU but would introduce significant programming complexity.

Therefore, we suggest a simpler solution. For inserting pins in the data structure, we use a scatter approach. We ray march one pin per shader invocation and directly write the resulting representation to the data structure. To achieve this, we intersect the pin with the region of space corresponding to each node in the top level data structure. In our implementation we introduce some randomness to this process to control the amount of write operations.

Datastructure: To fetch and update pins quickly, we recommend using a top level data structure that can be indexed quickly, such as a structured grid. We also require the top and bottom level data structure to have a fixed topology, such that the write locations for a pin do not depend on other pins inside the data structure. We then compute the address per intersected node, and write the pin to global memory without any considerations for write-conflicts.

Because single value write operations are atomic in nature, the worst that can happen is the combination of the t_{min} value of one pin and the bitmask of another. In that case we consider both pins to be similar enough, that the error produced because of this conflict is negligible.

5D Grid: In our implementation, we index pins using a 5D uniform grid. This matches the constraints discussed throughout the last 2 sections and creates no additional complexity. We believe thought that other options should be explored. The 5D grid should be understood as a temporal solution that allows us to focus on solving the other problems first.

3.8 Sample Mask Generation

We recall that majorant decomposition decouples sampling of the ratio :

$$P(I_{ji} = 1) = \frac{1 - t_i}{1 - t_{min}}$$

and sampling of the majorant:

$$P(H_{ji} = 1) = 1 - t_{min}$$

The bitmask for a pin only contains a row I_j. Therefore, we need a way to generate a random bitmask with corresponding bit probability to obtain M_j for distance sampling. Instead of using a (pseudo-) random float value to generate each bit, one at a time, we iteratively combine bitmasks until we reach the desired bit probability. The algorithm we use to generate uniform random float values in $[0, 1)$ generates a pseudo random integer via an XOR shift generator and interprets this as a mantissa. For a uniform distribution of integer values this gives us a bit-probability of $P(U_i = 1) = \frac{1}{2}$. We can use such a mask U to combine 2 masks A,B with $P(A_i = 1) = a$, $P(B_i = 1) = b$ by computing.

$$C = (A \textbf{ AND } U) \textbf{ OR } (B \textbf{ AND } (\overline{U})) \tag{3.11}$$

AND and **OR** are the respective bitwise operations and $\overline{U}$ the bitwise inverse of U. This gives us:

$$P(C_i = 1) = P(A_i = 1) \cdot P(U_i = 1) + P(B_i = 1) \cdot P(U_i = 0)$$
$$P(C_i = 1) = \frac{a + b}{2}$$

We start with bitmasks $P(A_0 = 1) = 1$ and $P(B_0 = 1) = 0$ and iteratively combine them after the manner of a binary search to approximate the desired bit probability. We illustrated this process in Figure 3.2.

3.9 Structured Artifacts

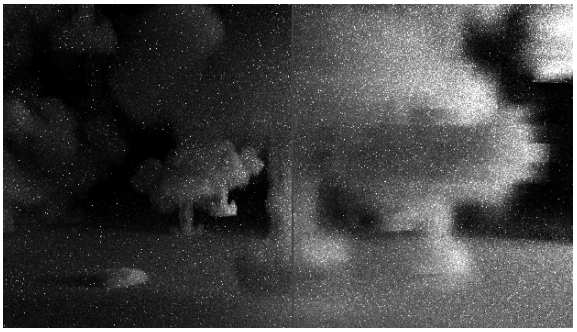


Figure 3.8: *Left: Ray marching. Right: Pins (no jittering). The correlation artifacts are most visible for pins along the borders of our 5D space partition. Below we describe how to decrease the perceived error.*

Jittering: Because of the discrete bitmasks that we store for our pins, we cannot interpolate the representation. Interpolating the resulting transmittance may be an option, but would also multiply the cost of the estimate. We have not explored this yet. Consequently, the space partition of our data structure is visible in many cases. We see this in Figure 3.8. Some of these artifacts become less pronounced over time as the pins are continuously resampled, but do not disappear entirely. Due to the eye's enhanced perception of contrast, the sharp edges caused by accessing pins at the borders of the 5D space partition, are quite noticeable. Adding a slight blur can help a lot in reducing the perceived error. We achieve this by jittering the spatio-directional coordinates used to access the data structure. In light of the blurring caused by using pins in the first place, allowing the additional blur seems well justified.

Because we only sample discrete distances we also jitter the sampled distance within the length of a single interval. The jittered offset added to the distance is distributed uniformly in our implementation. Using an exponential distribution would be more correct, but also require information about the local density. Consequently, for a dense medium we recommend sampling the local extinction coefficient and jittering the distance using an exponential distribution.

Transmittance sample location: While distance is sampled along a (5D) ray, transmittance is computed for a (6D) ray segment. Thus, we can justify estimating the transmittance by sampling a pin from any location along the segment. Because of the divergence of non-parallel lines in 3D space (Figure 3.7), the error produced by the approximation is expected to be minimal at the sampling location. Where to place the sampling location to minimize the overall error we leave to future work. We also suggest using this additional dimension to perform jittering.

Multiple Scattering: In photon mapping, final gathering delays access to the photon map until the second path vertex. The accumulation of contributions over the hemisphere of the first path vertex blurs artifacts caused by under-sampling the radiance field. We suggest using as similar approach to face our own under-sampling problem. Therefore, we do not use pin distance sampling for the primary ray. Also we do not use pin transmittance estimates for single scattering, which is computed from direct illumination at the first path vertex. In both cases we rely on ray marching. Hence, for all our current viable use cases we use pins only to compute multiple scattered illumination. We evaluate using pins for all rays in our ablation study, in section 4.1.

3.10 Implementation

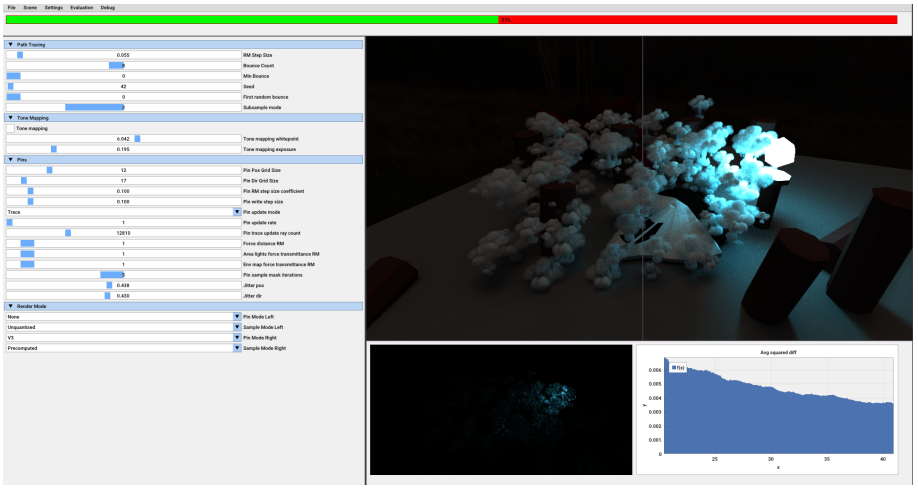


Figure 3.9: *Screenshot of the interactive application we use to develop our method.*

To develop and test our method we implement our own MC path tracker. Path tracing and pin computation both run entirely on the GPU. Relevant implementation details are discussed in this section.

Pins: We store all pins inside a storage buffer in device (GPU) local memory. The buffer holds a 5D nested array. The first 3 dimensions represent a cartesian grid over the bounding box of the volume. The 4th and 5th dimensions form a regular grid over the polar coordinates. Indices to the grid can be directly computed by discretizing the sample location. For future work we suggest experimenting with an octree for the first 3 dimensions. For example, the VDB format may be used for this. For the angular dimensions, we also recommend a different parameterization of the (hemi-)sphere, which does not have singularities at the poles. An example of such a parametrization is octahedral mapping [ED08]. Our current choices are made for simplicity and to match the restraints imposed in Sections 3.6 and 3.7. To generate pins, we sample a uniform random position, and a uniform random direction. The pin is defined by the straight passing through the point in the given direction. Hence, the sign of the direction does not matter. Pins can be evaluated in both directions by flipping the respective bitmask. For this reason, our bottom-level data structure only needs to cover the hemisphere.

We recommend exploring other ways to generate pins. Our current algorithm does not account for the volume density or illumination when choosing where to generate the pin. This information could be used to update pins more frequently in highly inhomogeneous regions. In Figure 3.10 we visualize the sum measurement contribution that is transported using the transmittance estimate of the respective pin for NEE. Because our method itself does not sample measurement contribution, the PDF used for the pin generation does not need to be accounted for during path tracing. In between grid cells, the pin PDF only determines the local update rate of the stream $\tilde{I}(s, x, \omega)$ and $t_{min}(s, x, \omega)$. For the range of pins, that map within one respective 5D grid cell, this is not true and further examination is required. For the moment we suggest using a locally smooth PDF, to achieve an approximately uniform distribution of pins inside each individual grid cell.

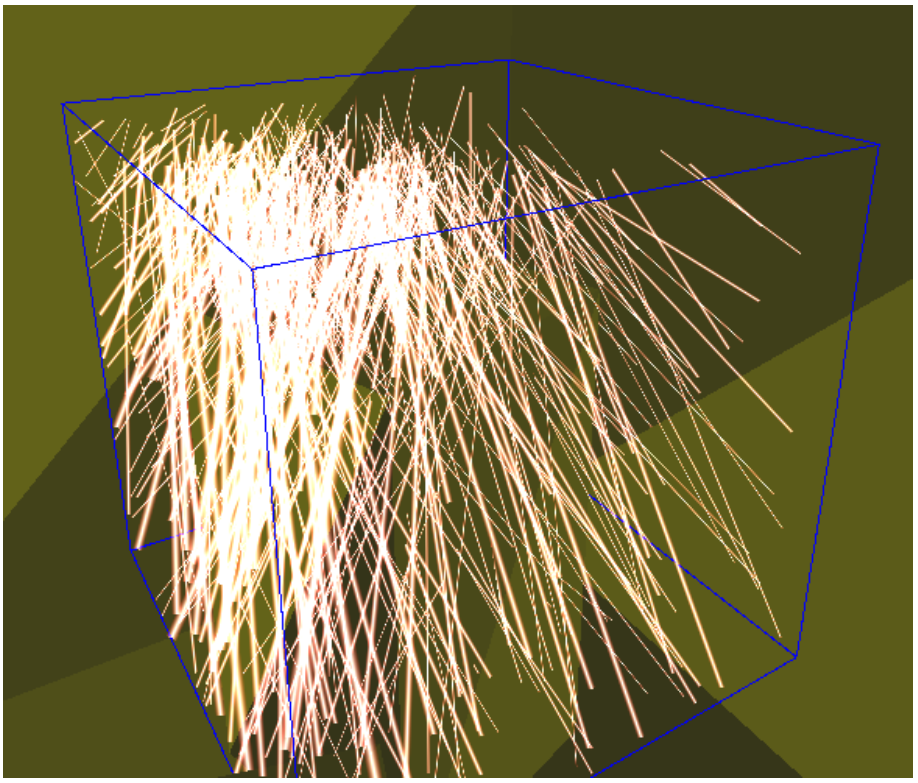


Figure 3.10: *Visualization of the measurement contribution, which is computed, using the transmittance estimate of the respective pin. The contribution was collected when viewing the Cornell box from the front (not from the current perspective). At the ceiling of the box an area light source is placed.*

Path tracer: To develop pins, we build an interactive application (see figure 3.9). For evaluation we run this application in an offline mode from the command line. In both cases we compute 1 sample per pixel and accumulate samples over repeated shader invocations.

Our MC path tracer runs in alternation with the pin update routine, in cases where we continuously recompute pins. Experimenting with running these routines concurrently we leave to future work. Currently we only support a lambert diffuse material for opaque surfaces. For the participating medium we implement isotropic and Greenstein scattering. We use NEE to sample two types of light sources. These are (triangular) area lights and light from an importance sampled environment map. For each type of light source, we compute one sample to estimate the direct illumination. Ray tracing is performed using the RT cores on our GPU. Because the entire path tracer runs inside a compute shader, we use ray queries to access the ray tracing functionality.

We support instanced rendering of surfaces and volumes. Surface intersections and volume intersections are computed separately. For surface intersections the default closest-hit determination is used. For volume intersection we implement a custom closest hit determination. Here we implement pin distance sampling.

When intersecting the bounding box of the volume, we use pins to sample the distance inside. If a collision is generated, we commit this for the global closest hit determination. As we have seen with decomposition tracking, this is a valid method of obtaining free path samples for additive overlapping density fields. For transmittance estimation we evaluate the transmittance for each intersected volume and accumulate the total transmittance via multiplication. This is also valid for overlapping volumes. Therefor, we can render such additive overlapping volumes correctly.

Ray marching is integrated as a second form of distance sampling and transmittance estimation and shares the same interface with pin sampling. We switch between ray marching and pin sampling when performing the comparison. Whenever possible we use compile time macros and specialization constants to reduce dynamic branching inside the shader at runtime.

We also minimize unnecessary distance sample / transmittance estimation steps, by always first evaluating the closest hit / visibility for opaque surfaces. In the following 2 chapter we perform the evaluation of our method.

4. Results

In this chapter we show results obtained with our method. These are then discussed in the next chapter. We structure the evaluation in four categories.

1) Ablation Study: First, we evaluate the effects of changing different parameters and components for our algorithm. As there are a lot of parameters we cannot cover all combinations. We therefore evaluate the effects of changing the parameters we deem the most important, while leaving other parameters constant.

2) Pin Grid: We show the effect of varying parameters and the total size of our 5D grid which we use to index pins. In addition we provide an overview of memory sizes and parameters used for runtime and error evaluation.

3) Runtime: Here we provide the single-sample runtimes of the conducted experiments. These measurements are averaged over all samples taken. We also show the effect of rendering large numbers of overlapping instances on the runtime.

4) Error over time: Finally we evaluate the error over time compared to ray marching. The results are computed in equal-time. To obtain the MSE we compute a reference image with ray marching at a small step size and a high sample count.

Hardware: The results are obtained on a machine with a RTX 2070 Super NVIDIA GPU. We always compute the reference image first as to achieve a heat equilibrium and thus reduce the effect of varying GPU clock rates when performing the equal time comparison. We only count the GPU runtime which we query via the graphics API. Computing pins is included in that time.

General default parameters: For all rendered scenes we use isotropic or Henyey-Greenstein (forward) scattering. We write (HG) if we use Henyey-Greenstein scattering. We always render at *512x512p* resolution. The maximum path length is set to 8. This length is also used for the reference image. We always use a scattering albedo of 1 to maximize the effect of multiple scattering. When using pins only for multiple scattering we label the image with 'PinMS' in sections 4.1 and 4.2. In sections 4.3 and 4.4 we always restrict the use of pins to multiple scattering.

Runtime and error measurement: *The following only applies to all cases where we measure runtime or error (Sections 4.3 and 4.4).* To store pins we quantize the hemisphere in $4 \cdot 16 = 64$ segments. Each segments contains a pin entry which requires ($2 \cdot 4 = 8$ Byte). We adapt the resolution of the positional grid to attain the desired relative memory size

of $\approx \times 0.1$ the size of the 3D texture. Pins are only computed initially and left constant thereafter. We find this has little effect for multiple scattering, especially when using jittering in addition. For computing pins we use a step size of 0.01 relative to the border length of the volume bounding box. The relative step size for ray marching while path tracing is provided for each case (Step = X). Because of technical issues, the equal time plots do not show the first 3 samples for ray marching and the respective number of samples taken in equal time for our method. For the rest of the curve, this has no effect.

Images: We use the FLIP error metric to visualize the perceptual error in many cases [ANSA21]. The rendered images are gamma corrected and use Reinhard tone mapping. This is of course applied after all metrics have been computed.

Asset list: To evaluate our method we use assets from the Open-Scivis dataset [Kla17]. The assets used are:

- CSAFE Heptane Gas
- Boston Teapot
- Tooth
- Head (Visible Male) (only for figure 4.11)

In some cases we combine the scalar field with subtractive Perlin noise when loading the asset in the 3D texture. For the last 2 test cases we generate 3D Perlin noise which is then stored to a texture. To store the volume density we always use a 3D texture with half precision float values (16 bit). We provide details about the memory sizes in section 4.2.

4.1 Ablation Study

4.1.1 Update Rate

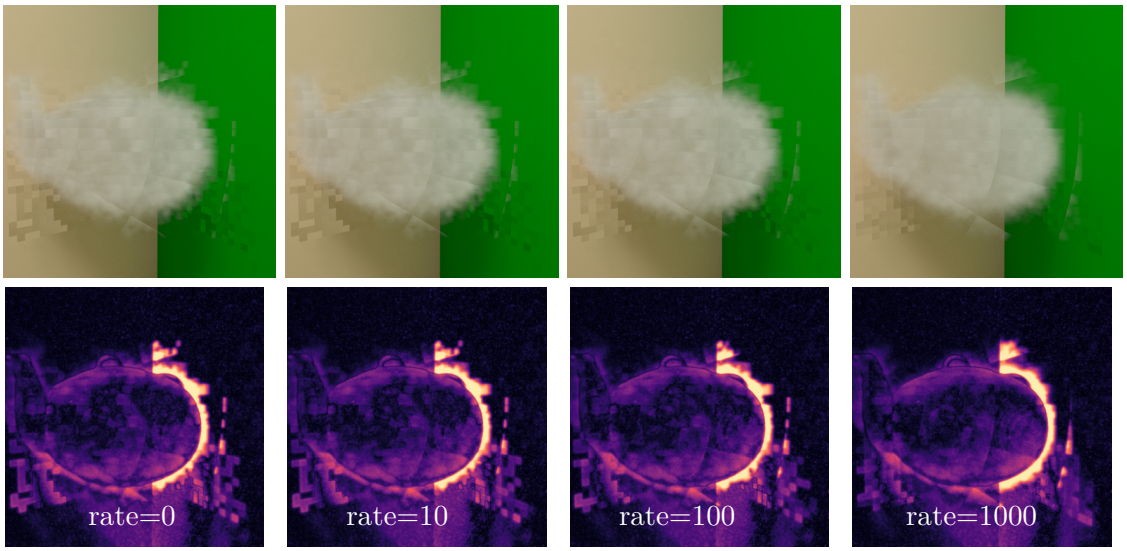


Figure 4.1: *Here, we compute a continuous stream of pins. These are computed between single samples. From left to right we increase the number of pins we compute for an equal-time render (10 sec). Each pin is added to the intersecting grid cells. When increasing the update rate, we see the greatest improvements at the shadows cast on the wall. The box-like artifacts appear on surfaces inside the volume, when evaluating the transmittance. Increasing the update rate reduces the visibility of these artifacts, but does not succeed in completely hiding them.*

4.1.2 Sample Mask Generation

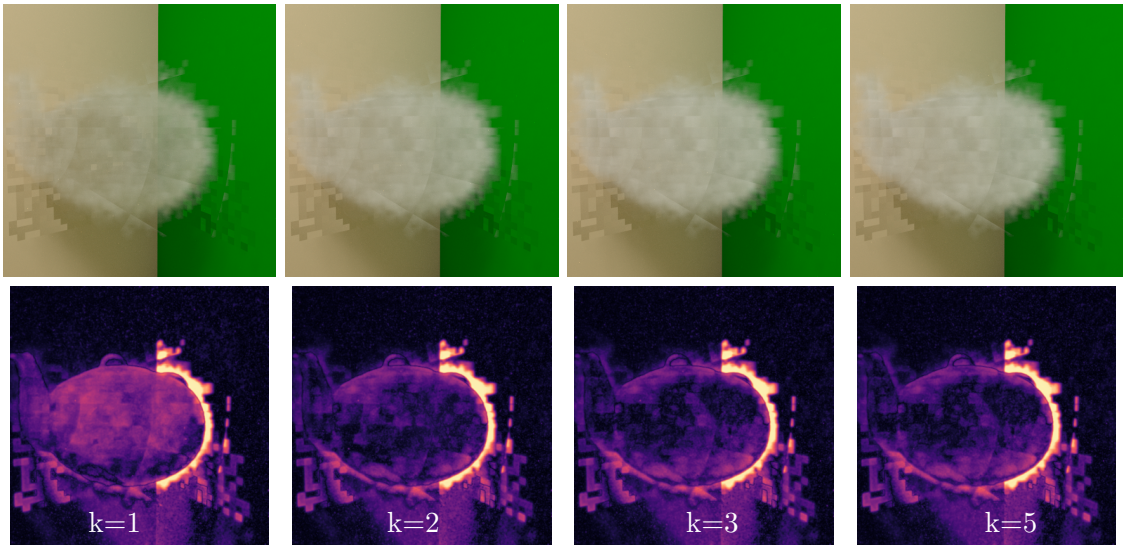


Figure 4.2: *In Chapter 3.4.2 we explain the algorithm used to generate masks with a desired approximate bit probability of* $1 - t_{min}$*. Here, we show the results of using this algorithm at different iteration counts k. Each iteration increases the accuracy of the bit probability. We find that using more than 3 iterations has little effect. We use 5 iterations in our implementation, as it also has little to no effect on the total runtime.*

4.1.3 Higher Order Scattering

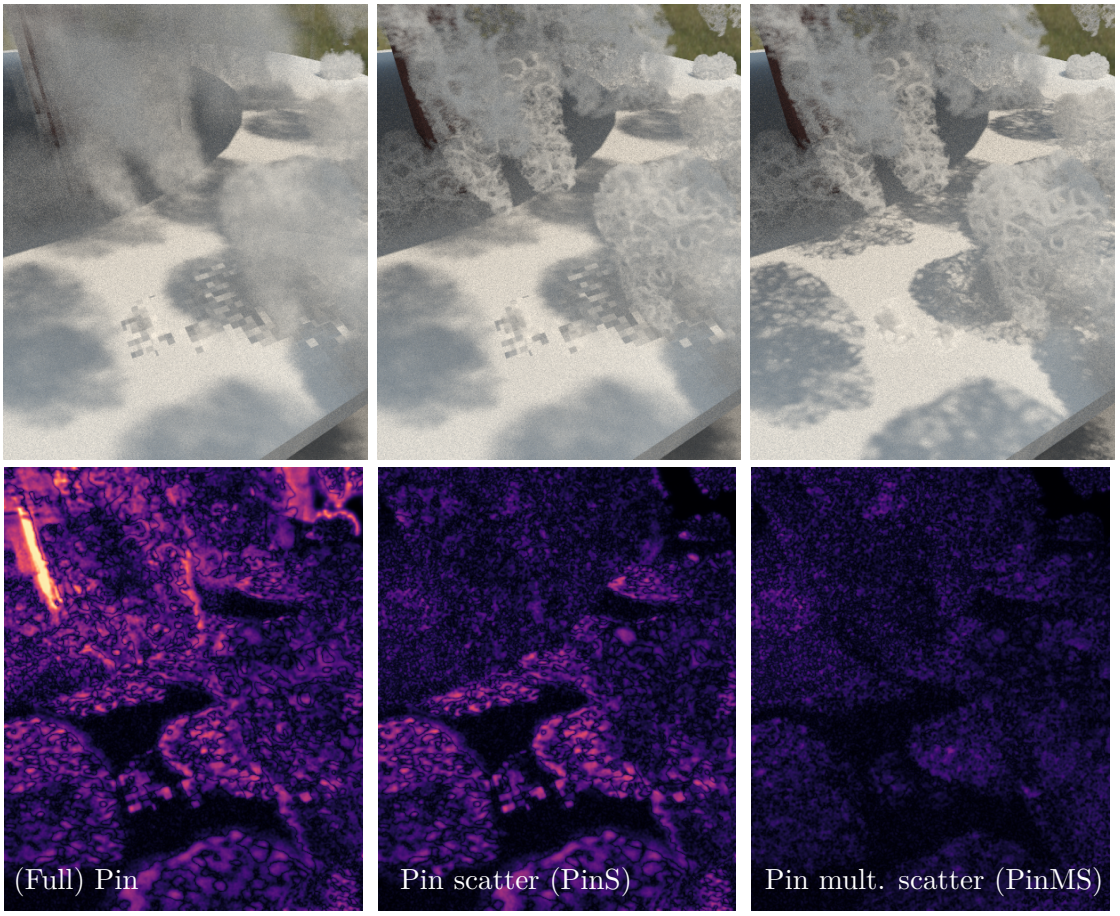


Figure 4.3: *Artifacts are heavily visible when using pins for the view ray (left). Scattering induces a blur which helps to reduce grid-like artifacts. For direct illumination at the first bounce the correlation is often still visible, especially for shadows cast by highly directional illumination (middle). We obtain good results when using pins only for higher order bounces (right). For the primary ray and the transmittance at the first bounce we use ray marching. We use this setting for error and runtime evaluation.*

4.1.4 Jitter

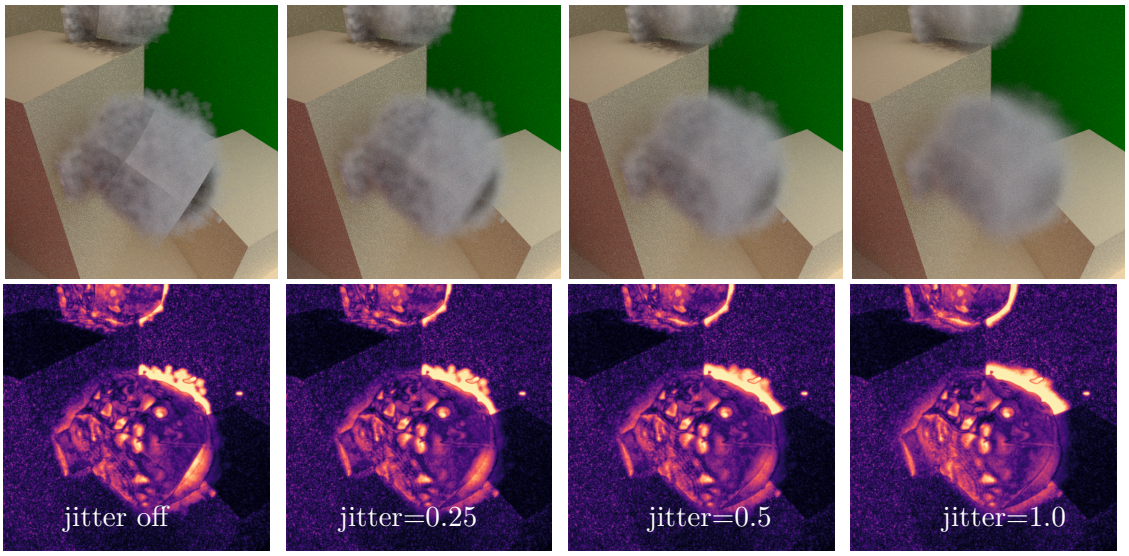


Figure 4.4: *Jittering the access location to the grid blurs the hard lines induced by our space partition. We increase the strength of jittering from left to right in equal size steps. The coefficient expresses the jitter strength relative to the grid cell size. Jittering is equally applied to Cartesian and spherical coordinates and scaled to their respective grid cell sizes. We use the highest setting, in addition to the last modifications, to further decrease visual artifacts in our evaluation.*

4.1.5 Pin Representation

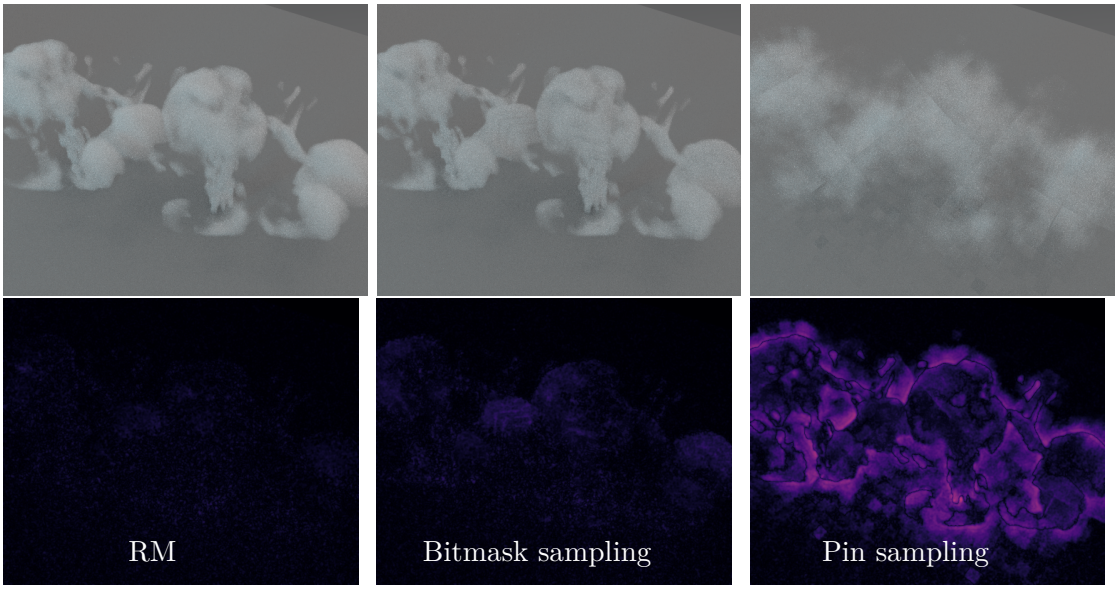


Figure 4.5: *We assess the error produced by sampling our bitmask representation compared with the error produced by selecting the closest pin. The left image uses ray marching. The middle image uses our bitmask with majorant decomposition. The respective pins are computed on-demand here. The right image uses pins obtained from the grid. We find that quantization to the closest pin (right) is our primary source of error.*

4.1.6 Stream

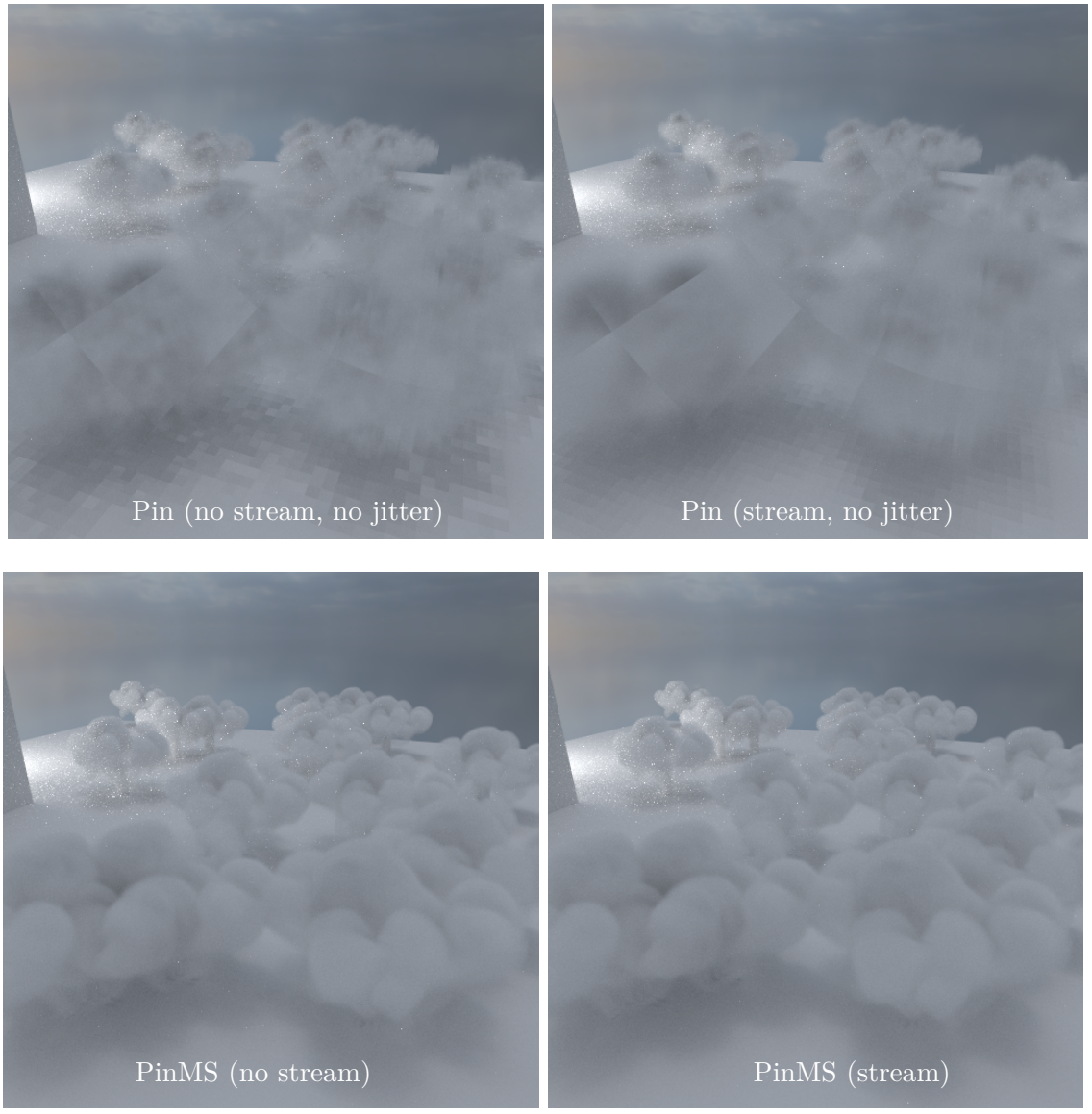


Figure 4.6: *For the left column, we use a fixed set of precomputed pins. For the right column, we update the grid with the stream approach. In the bottom row we only use pins for multiple scattering and use jittering. In the top row, we see the largest effect of using the pin stream when evaluating transmittance on surfaces inside the volume. While the corresponding soft shadows still have blocky artifacts, these are less pronounced than when not updating pins at all. When using pins for multiple scattering only, the effect of using the stream approach is not visible.*

4.2 Pin Grid

4.2.1 Direction vs. Angle

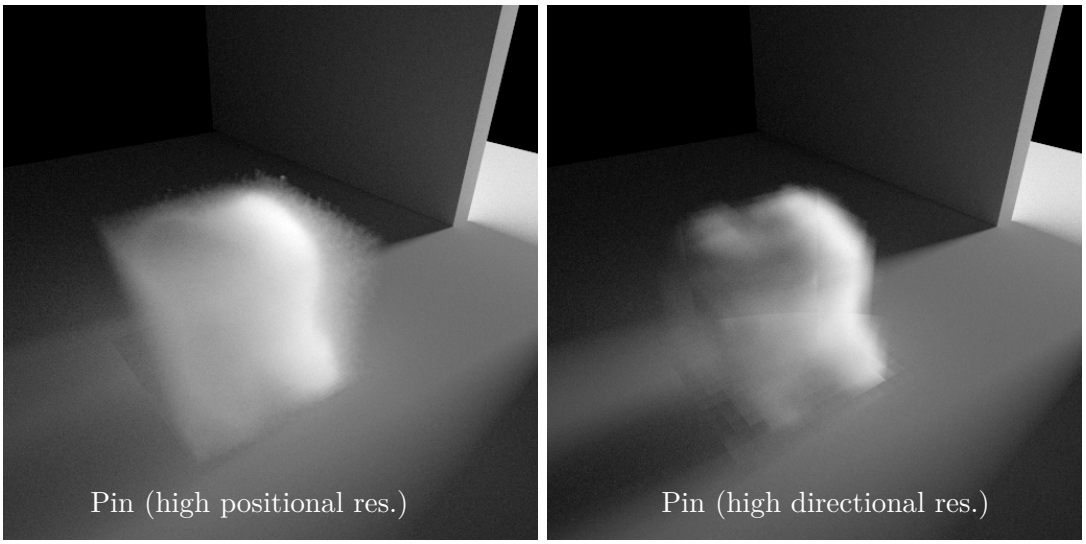


Figure 4.7: *In the figure above, we use pin sampling to completely replace ray marching for all rays. We use a pin grid that is equal in memory size compared to the original scalar field in both cases. The left image is obtained by using a low directional resolution and high positional resolution. For the right one the proportions are inverse. The left grid blurs the volume boundaries and casts almost no shadow. The right grid creates box-like artifacts.*

4.2.2 Increasing Resolution

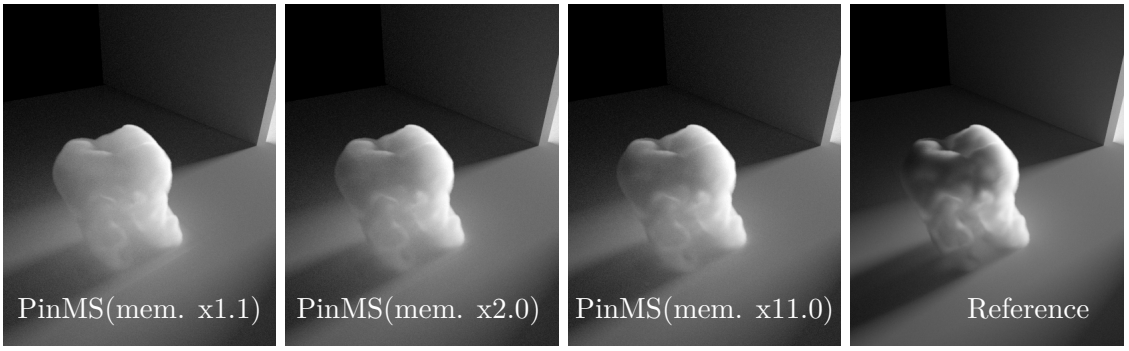
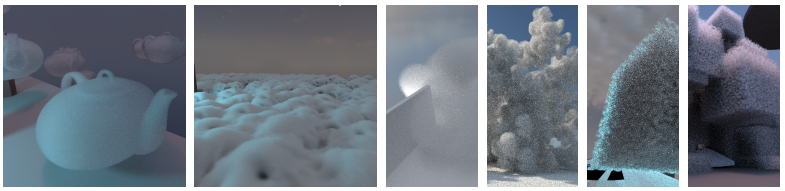


Figure 4.8: *Here, we use pins together with ray marching. Ray marching is used for the primary view ray and single scattered direct illumination. From left to right, we increase the positional resolution to achieve a relative memory size of (0.1, 1.0, 10.0) compared to the scalar field. This memory requirement is additive when using pins for multiple scattering. Therefore, we show the total memory size above. We find that increasing the pin grid resolution enhances surface contrast. But this improvement comes at a steep price in memory size.*

4.2.3 Size Table

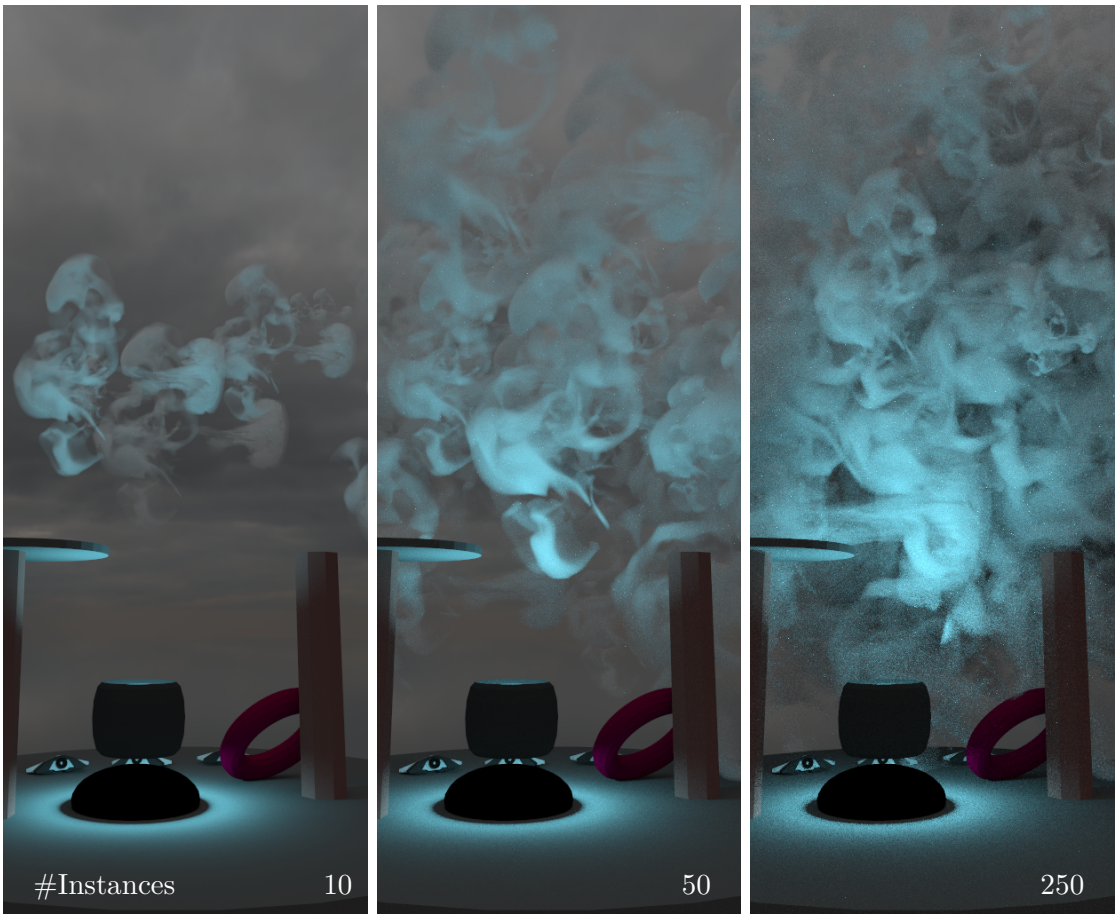


Data size	Teapot	Snow	Cloud, Many Clouds	Cube, Many Cubes
Asset	Boston Teapot	Tooth - Perlin Noise	CSAFE Heptane Gas	Perlin Noise
Tex Dim	256x256x178	103x94x161	302x302x302	512x512x512
Tex Size	23,3 MB	3,1 MB	55 MB	268,4 MB
Grid Dim	16x16x16 x 16x4	8x8x8 x 16x4	22x22x22 x 16x4	37x37x37 x 16x4
Grid Size	2,1 MB	0,26 MB	5,4 MB	25,9 MB
Total Size	25,4 MB	3,3 MB	60,5 MB	294,3 MB

Figure 4.9: *Here, we provide memory sizes for the respective volumes used in the next 2 sections. We also give the pin grid resolution for each case. To reduce the amount of varying parameters between the test cases, we stick to a fixed directional grid resolution. This parameter could be fine-tuned to individual cases which may provide a small improvement for pins.*

4.3 Runtime Measurements

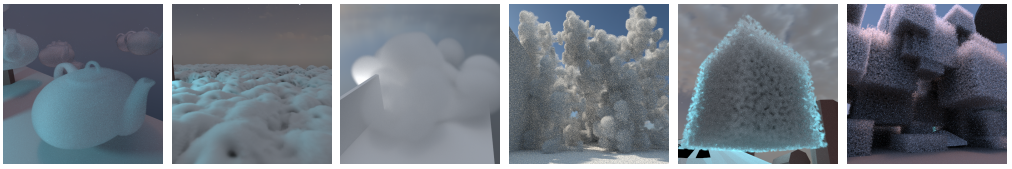
4.3.1 Instance Composition



Time p. sample	10 Inst.	50 Inst.	250 Inst.
RM (step s. 0.08)	1.93 ms	25.57 ms	136.2 ms
PinMS (mem. x1.1)	1.83 ms	20.16 ms	87.41 ms

Figure 4.10: *Above, we compare our method to ray marching when rendering many overlapping volume instances. For each intersected volume the respective sampling step is run separately for that volume. All instances are of the same volume. We use the same ray marching step size as in section 4.4, where we use the same volume. The error produced by both methods is thus similar. For the highest instance count, our method runs* ≈ 1.5 *times faster than full ray marching.*

4.3.2 Runtime Table



Frame time	Teapot *step=0.015*	Snow *step=0.015*	Cloud *step=0.04*	M. Clouds *step=0.04*	Cube *step=0.015*	M. Cubes *step=0.015*
RM	47.63 *ms*	**13.21** *ms*	7.49 *ms*	106.40 *ms*	36.25 *ms*	65.88 *ms*
PinMS	14.73 *ms*	**12.11** *ms*	5.65 *ms*	39.80 *ms*	7.79 *ms*	17.17 *ms*
Frame time	Teapot *step=0.03*	Snow *step=0.03*	Cloud *step=0.08*	M. Clouds *step=0.08*	Cube *step=0.03*	M. Cubes *step=0.03*
RM	**24.59** *ms*	11.47 *ms*	**6.02** *ms*	**56.12** *ms*	**9.46** *ms*	**34.72** *ms*
PinMS	**13.31** *ms*	11.80 *ms*	**5.46** *ms*	**36.16** *ms*	**6.55** *ms*	**16.34** *ms*
Frame time	Teapot *step=0.06*	Snow *step=0.06*	Cloud *step=0.16*	M. Clouds *step=0.16*	Cube *step=0.06*	M. Cubes *step=0.06*
RM	13.67 *ms*	9.16 *ms*	4.95 *ms*	**32.04** *ms*	10.18 *ms*	21.80 *ms*
PinMS	12.52 *ms*	11.48 *ms*	5.39 *ms*	**33.88** *ms*	6.65 *ms*	16.86 *ms*

Figure 4.11: *We here provide the average single sample frame times for the scenes evaluated in the next chapter for different step sizes. We mark the frame times in bold letters for the step size shown in the next chapter. In these cases ray marching performs the best considering error vs. runtime. This provides a fair comparison for our method. For the 3,4,5 and 6 row* $\approx$ *equal error is achieved after 10 seconds, for that respective step size. For the 5th test case the optimal step size is not obvious when considering the perceptual error. In the last row we achieve half the runtime compared to ray marching, for a reasonable step size. Results for the other step sizes are added to the appendix.*

4.4 Error Measurements

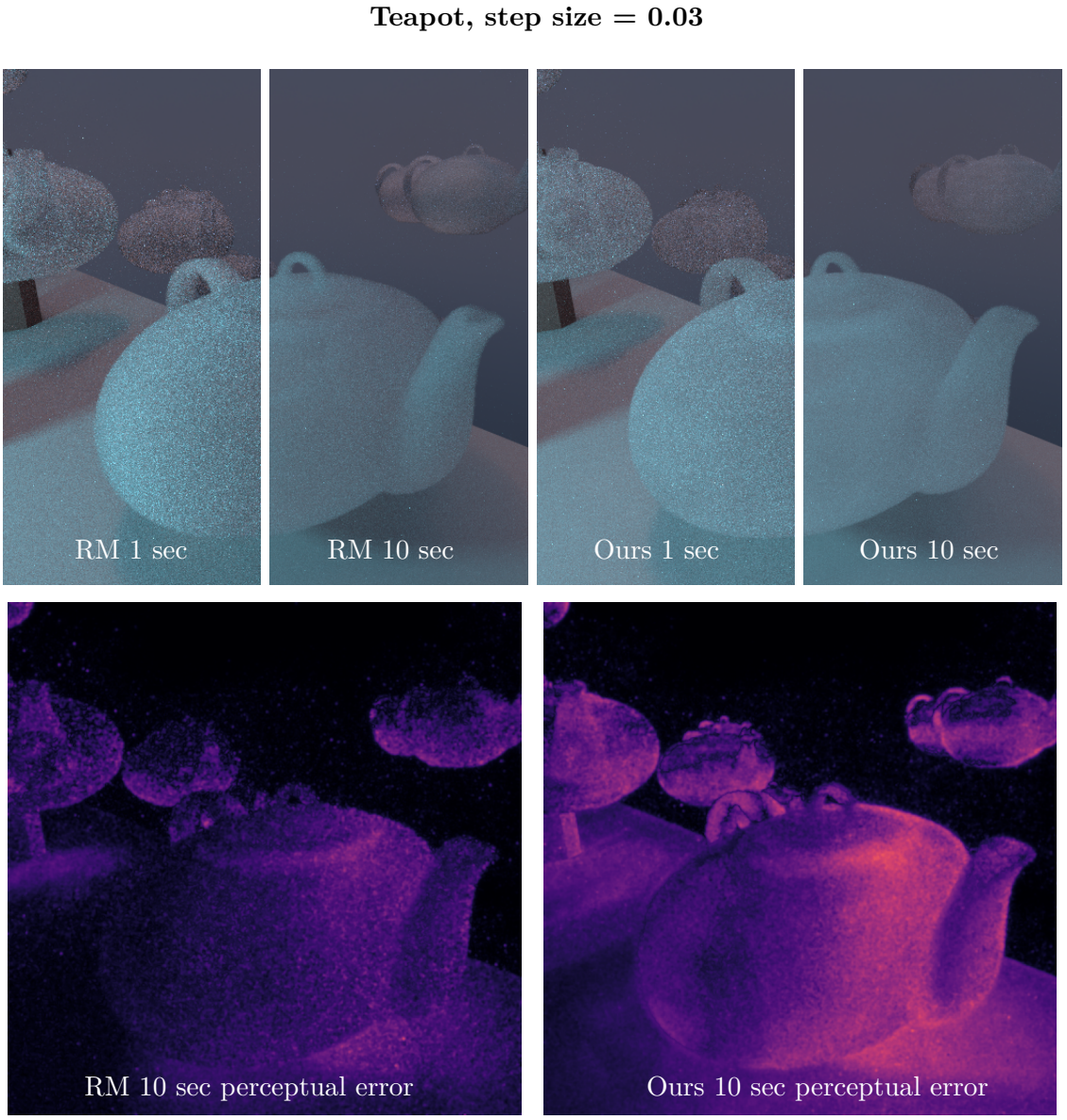


Figure 4.12: *When using our method, the most noticable error is produced for volumes with hard boundarys. We find a significant loss of contrast in these regions when compared to ray marching. This effect is amplified, when the volume is illuminated in a way that enhances these contrasts. In this case the volume is illuminated from the left side by an area light. Compared to ray marching the shadowed backside of our volume also appears generally brighter. The resulting bias is visible at the end of the error curve.*

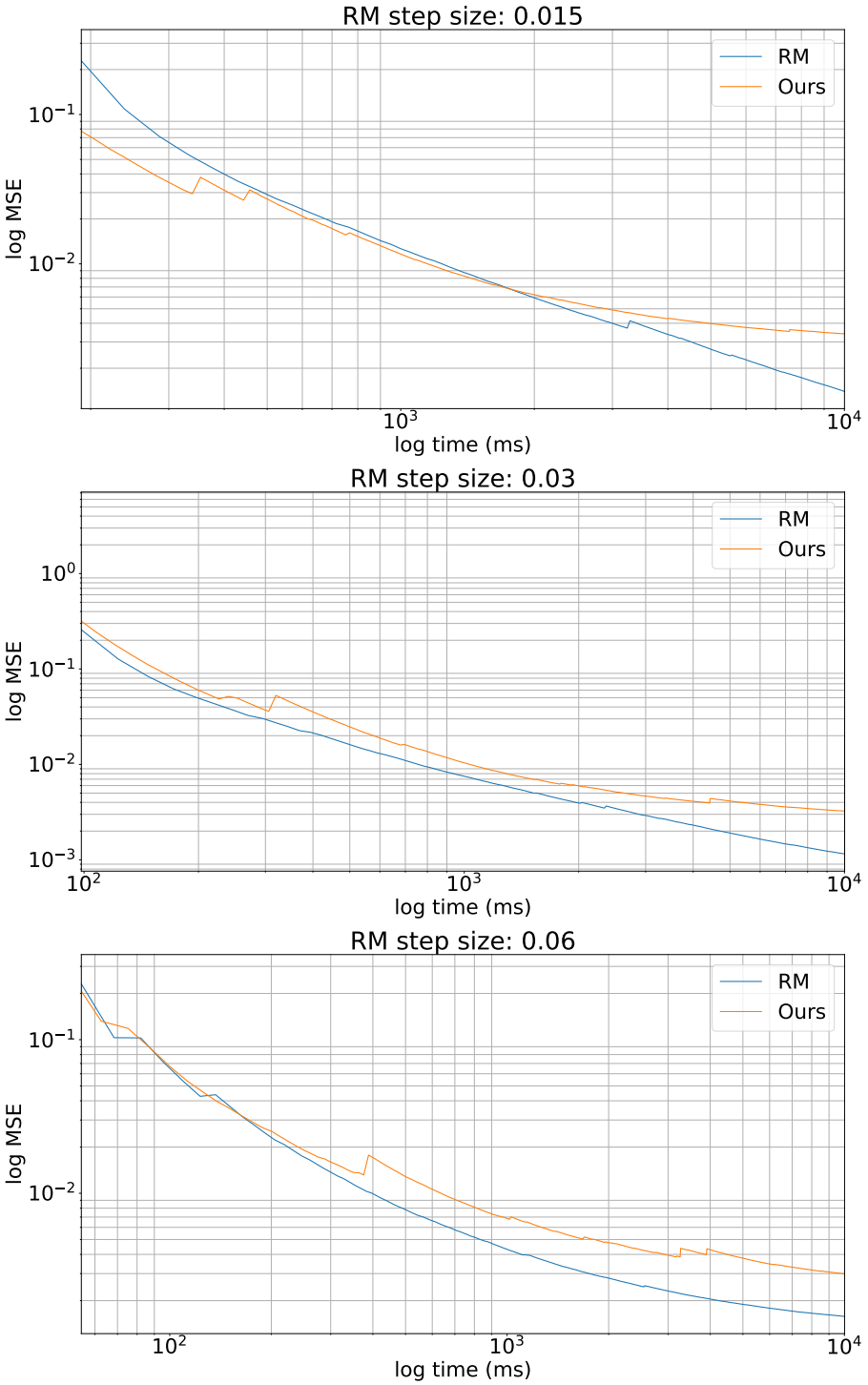


Figure 4.13: Teapot: MSE over time for different step sizes. In all cases our method produces a larger bias compared to ray marching.

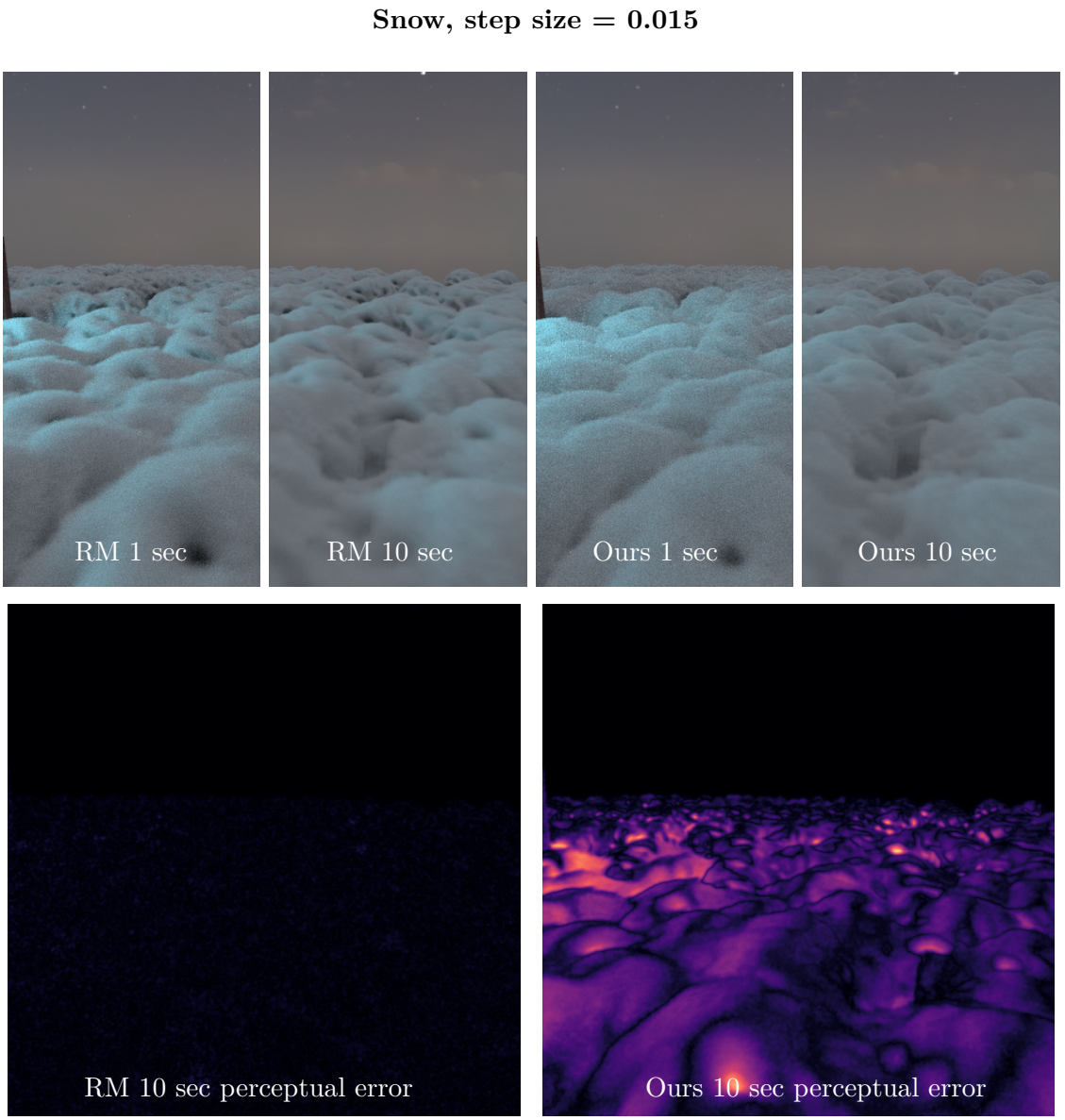


Figure 4.14: *We show another case where our method induces a large bias. Here we render many instances of the 'Tooth' with subtractive Perlin noise. The image produced with our method is slightly more washed out. Because we only use pins for higher order scattering, the resulting error is heavily blurred. As such this is often not noticeable, except when viewed in direct comparison.*

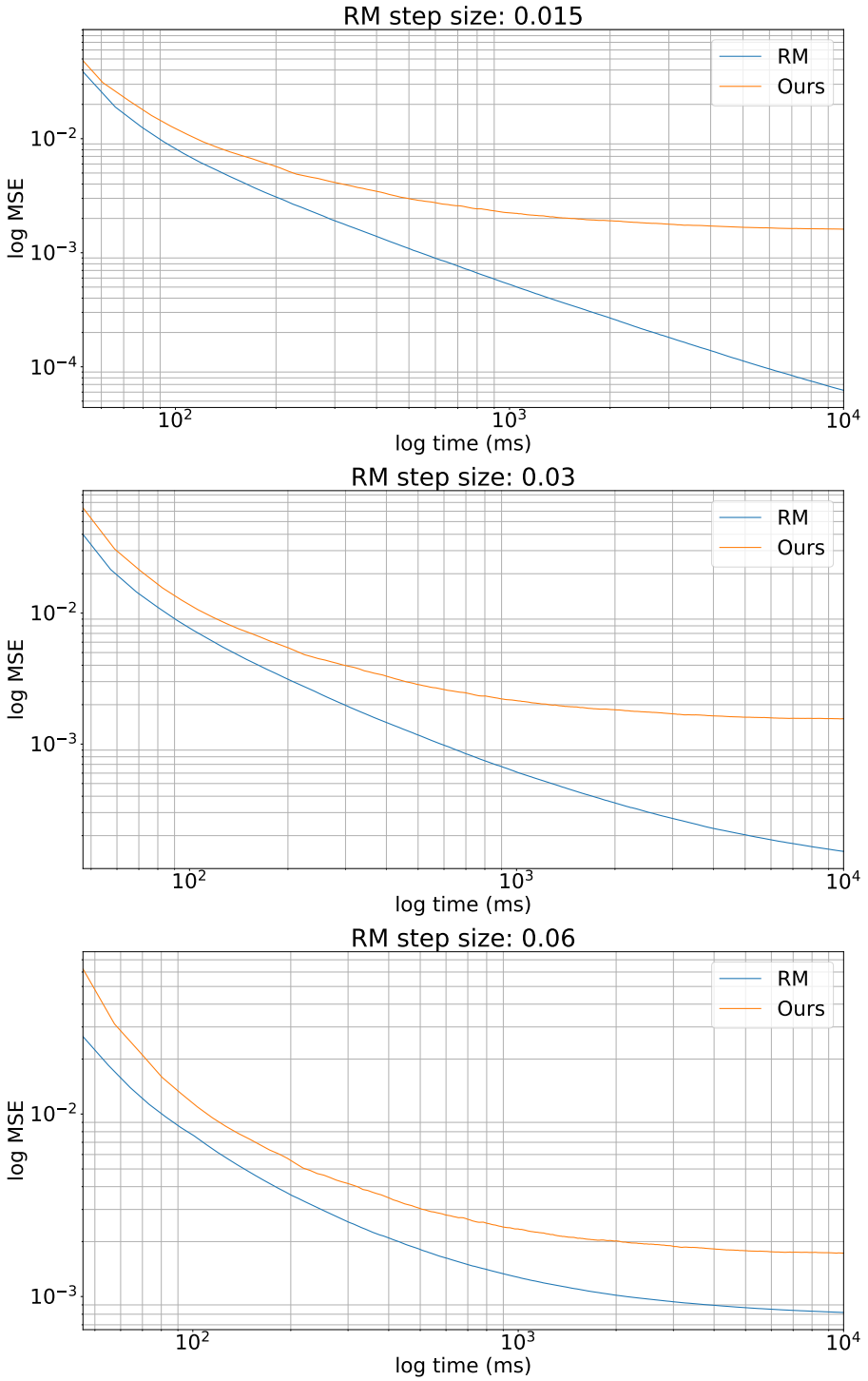


Figure 4.15: Snow: MSE over time for different step sizes. Again, ray marching is better for all evaluated step sizes.

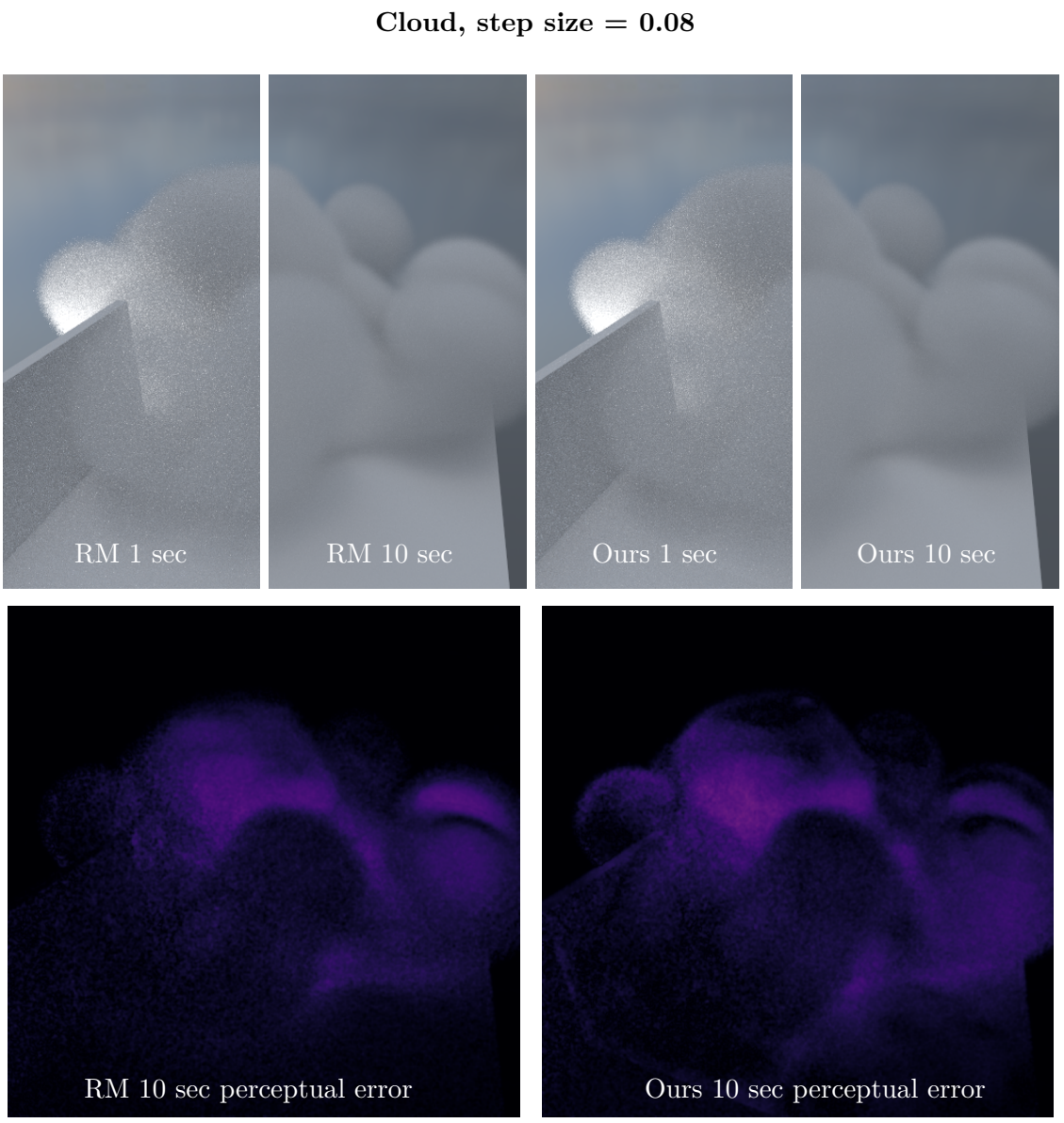


Figure 4.16: *Here we render a cloud illuminated by an area light source and an environment map. Our method produces little error as there are no hard boundaries. Because the medium only requires a coarse step size, ray marching also performes well.*

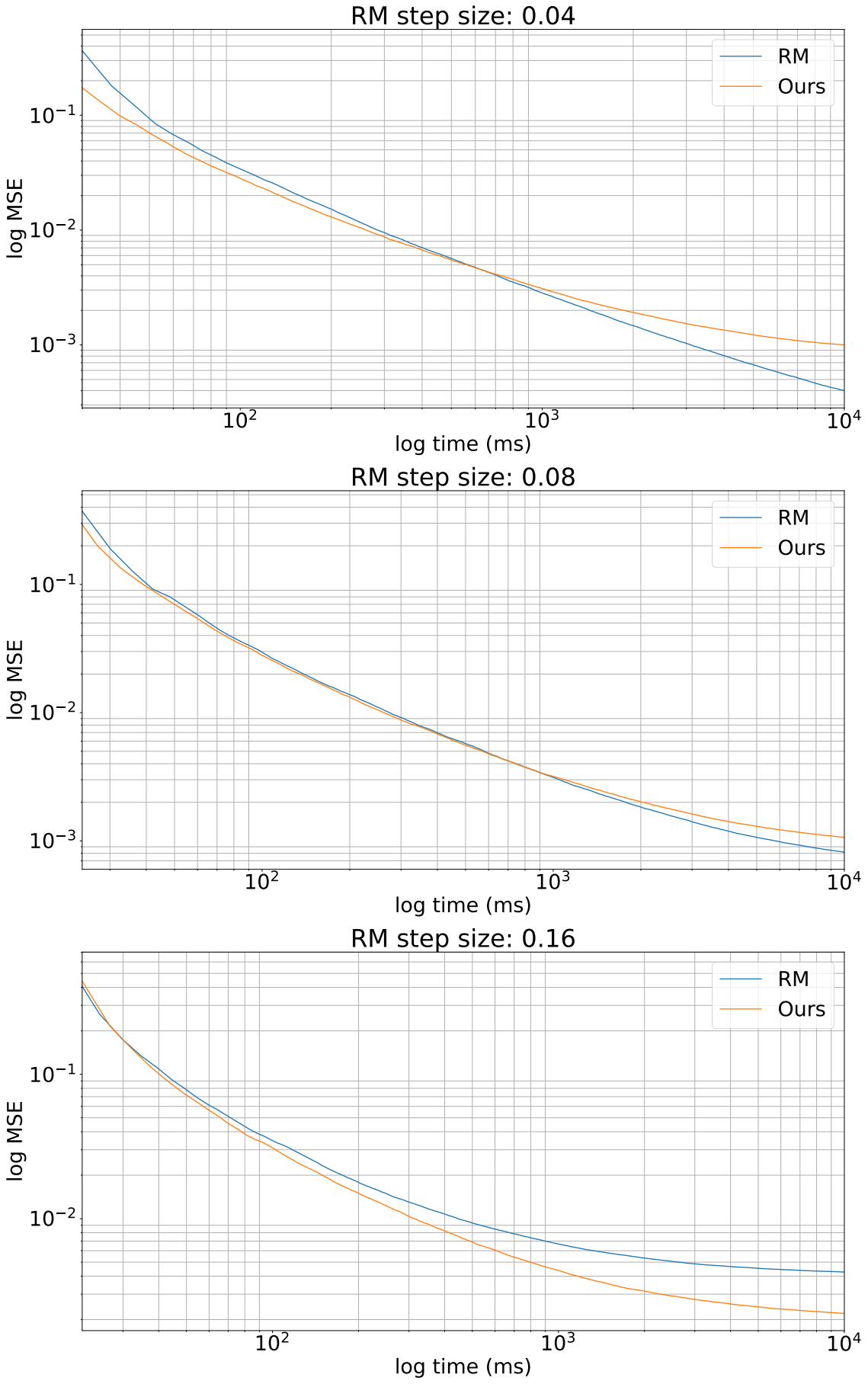


Figure 4.17: Cloud: MSE over time for different step sizes. We achieve similar results compared to ray marching in the middle case.

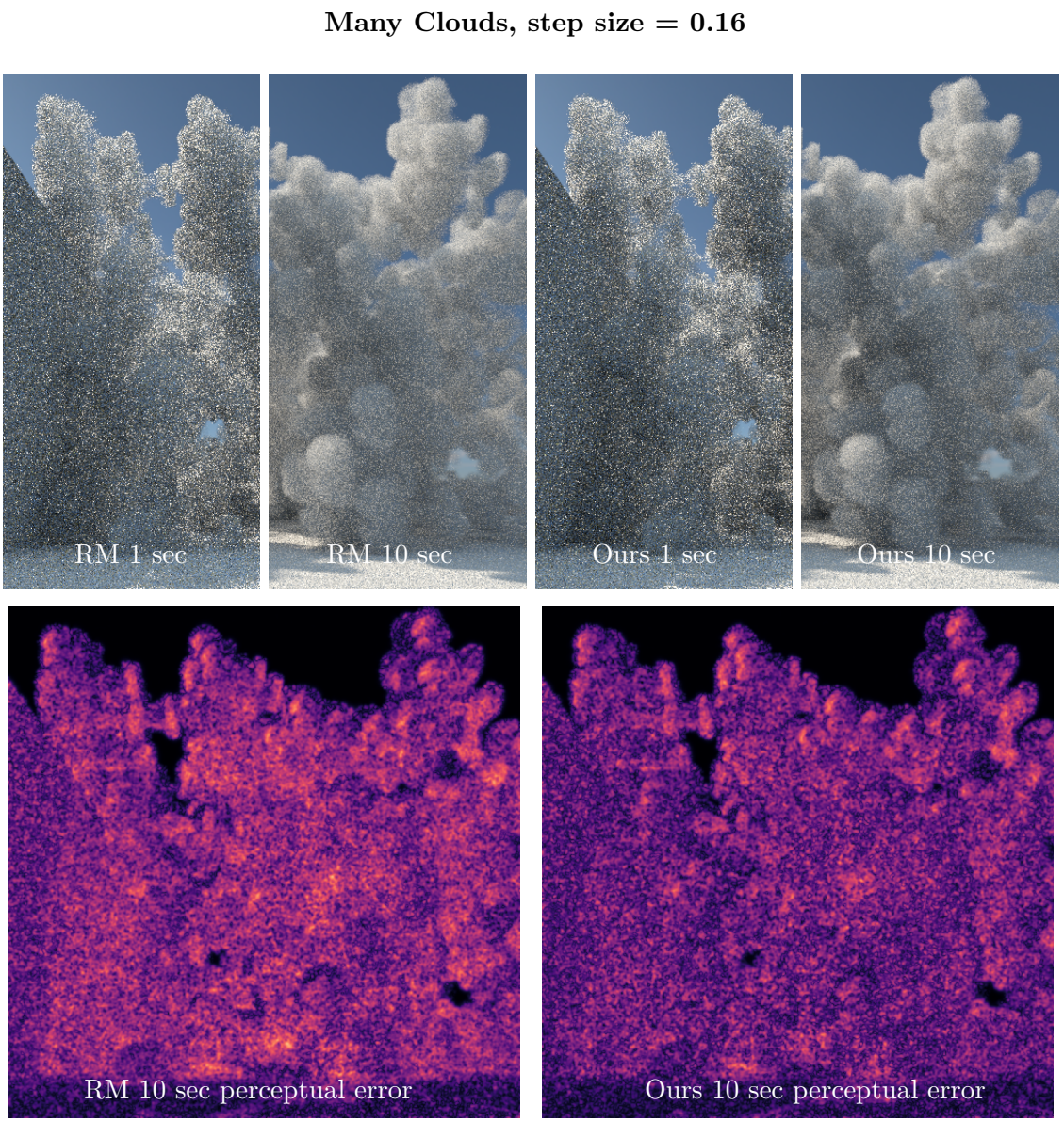


Figure 4.18: *In this image we render a large cloud, composed of many instances of the same volume. Again our method performs similar to ray marching. Though ray marching at this step size slightly blurrs the contours.*

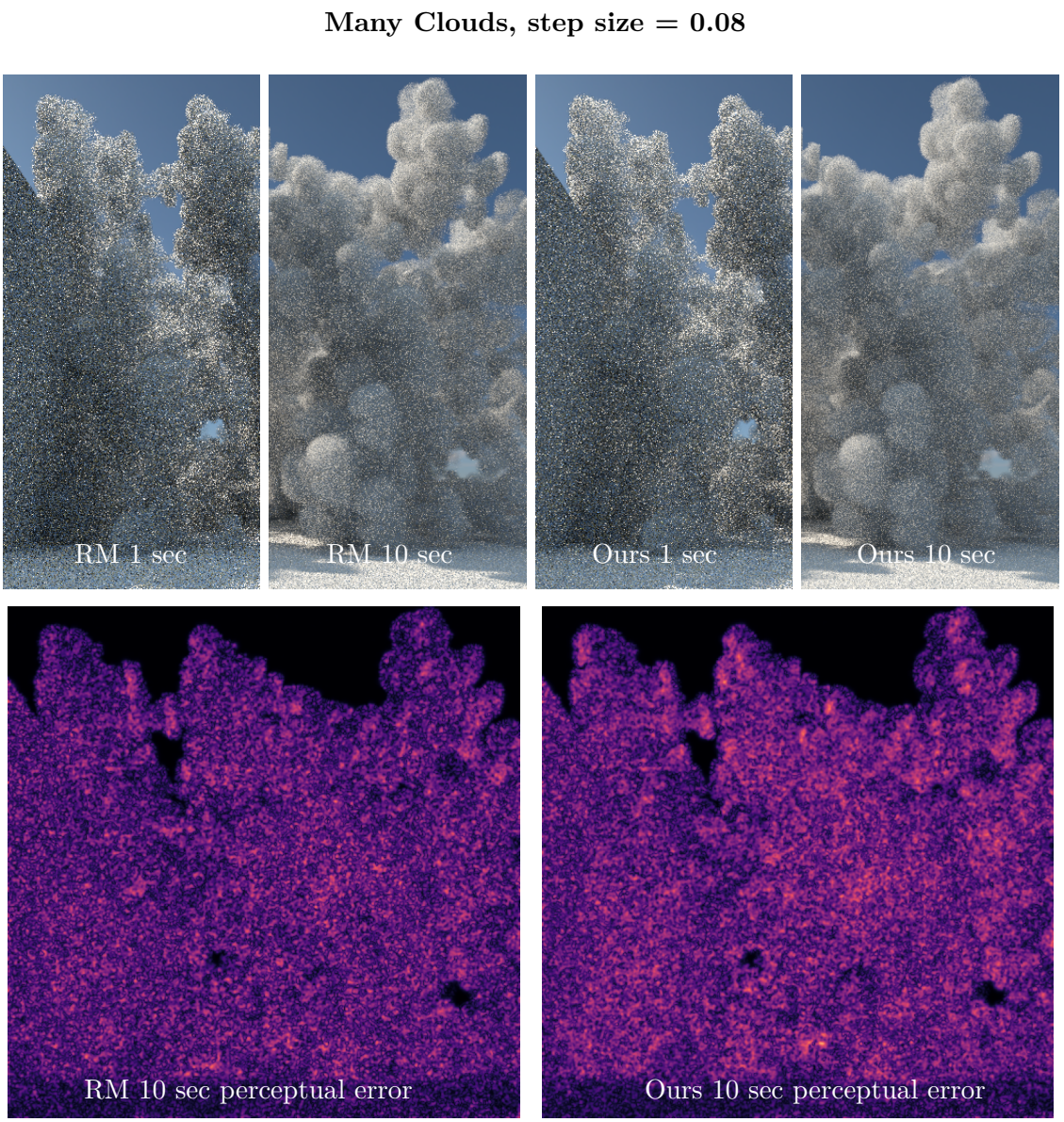


Figure 4.19: *When halving the step size compared to last render, the fully ray marched image appears slightly sharper than the image produced by our method. Therefore the level of noise is larger in the ray marched image. For that image, only* ≈ 0.64 *times the samples have been accumulated compared to our method in equal time.*

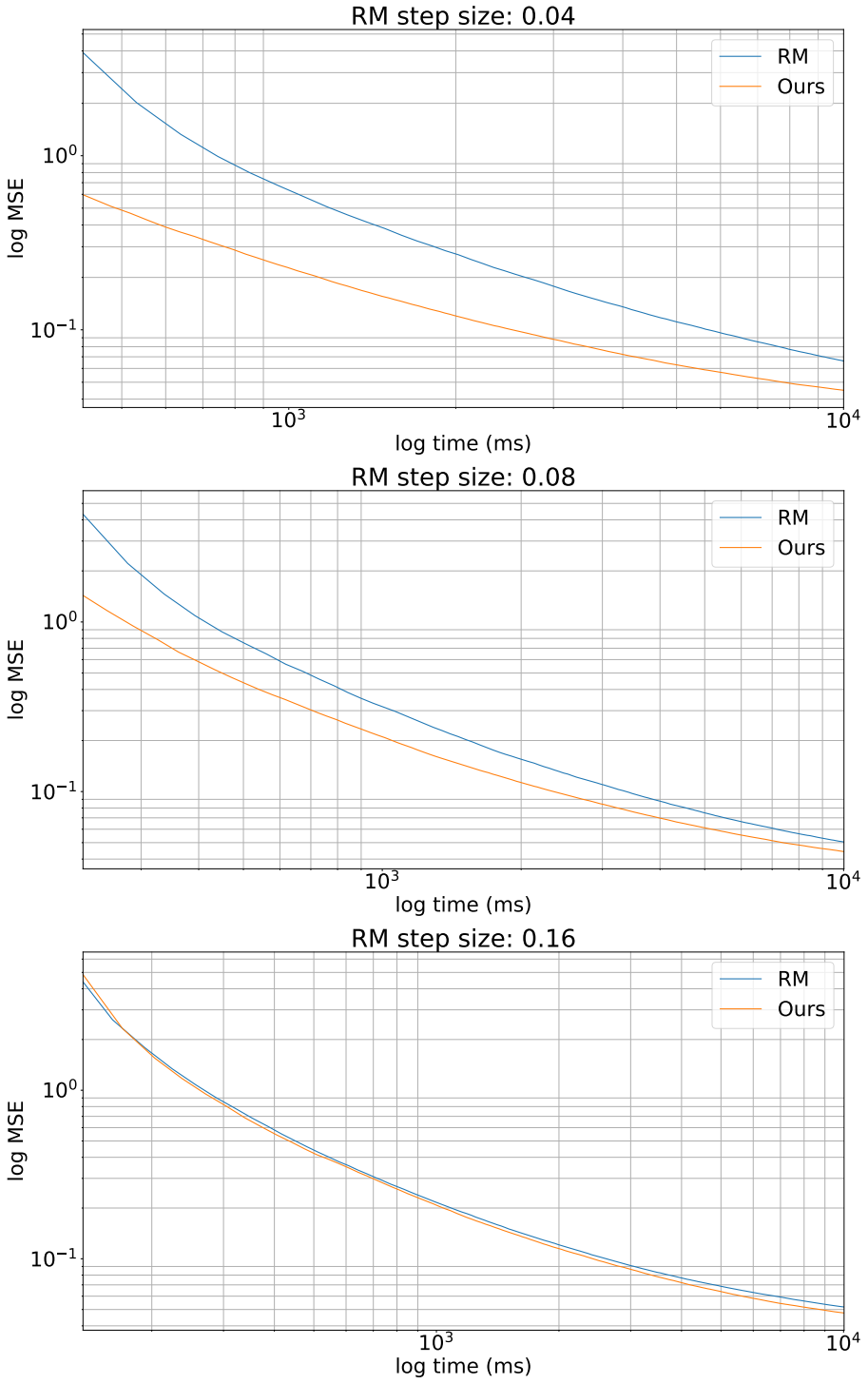


Figure 4.20: Many Clouds: MSE over time for different step sizes. We achieve a similar MSE compared to ray marching for the two largest step sizes. Convergence of our methods is faster for the middle case.

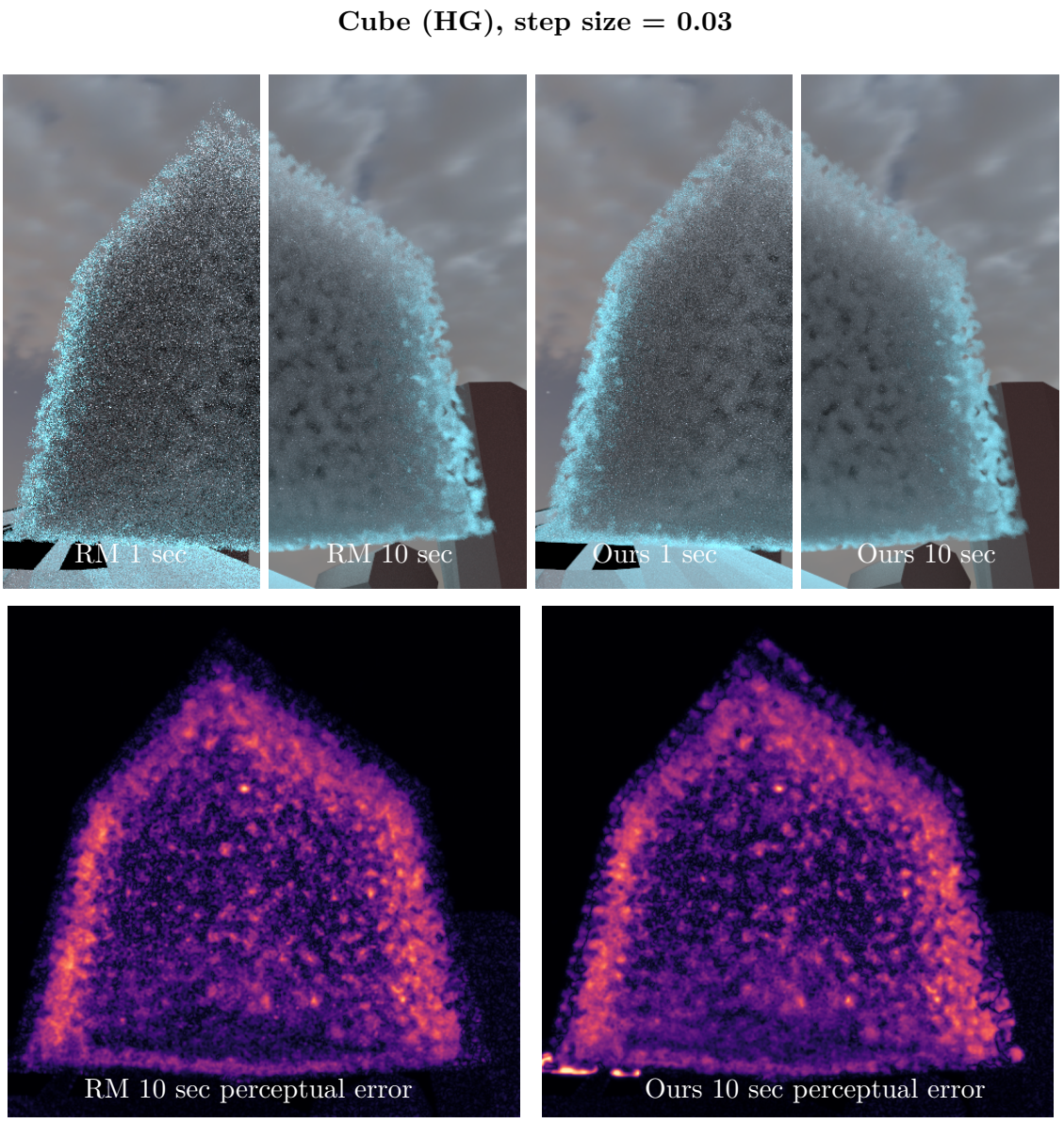


Figure 4.21: *In this scene we render a cube filled with a high frequency Perlin noise. This forces us to use a small step size for ray marching. Because the bounding box used to compute pins is tight, we produce less bias at the surface using our method. Also, the frequency of the noise is irrelevant for the runtime of our sampling algorithm. Thus, we achieve a significantly faster convergence rate compared to ray marching. This translates to less noise, which is most visible in the 1 second equal time render. Both ray marching and our method have the largest bias at the backlit edges of the volume.*

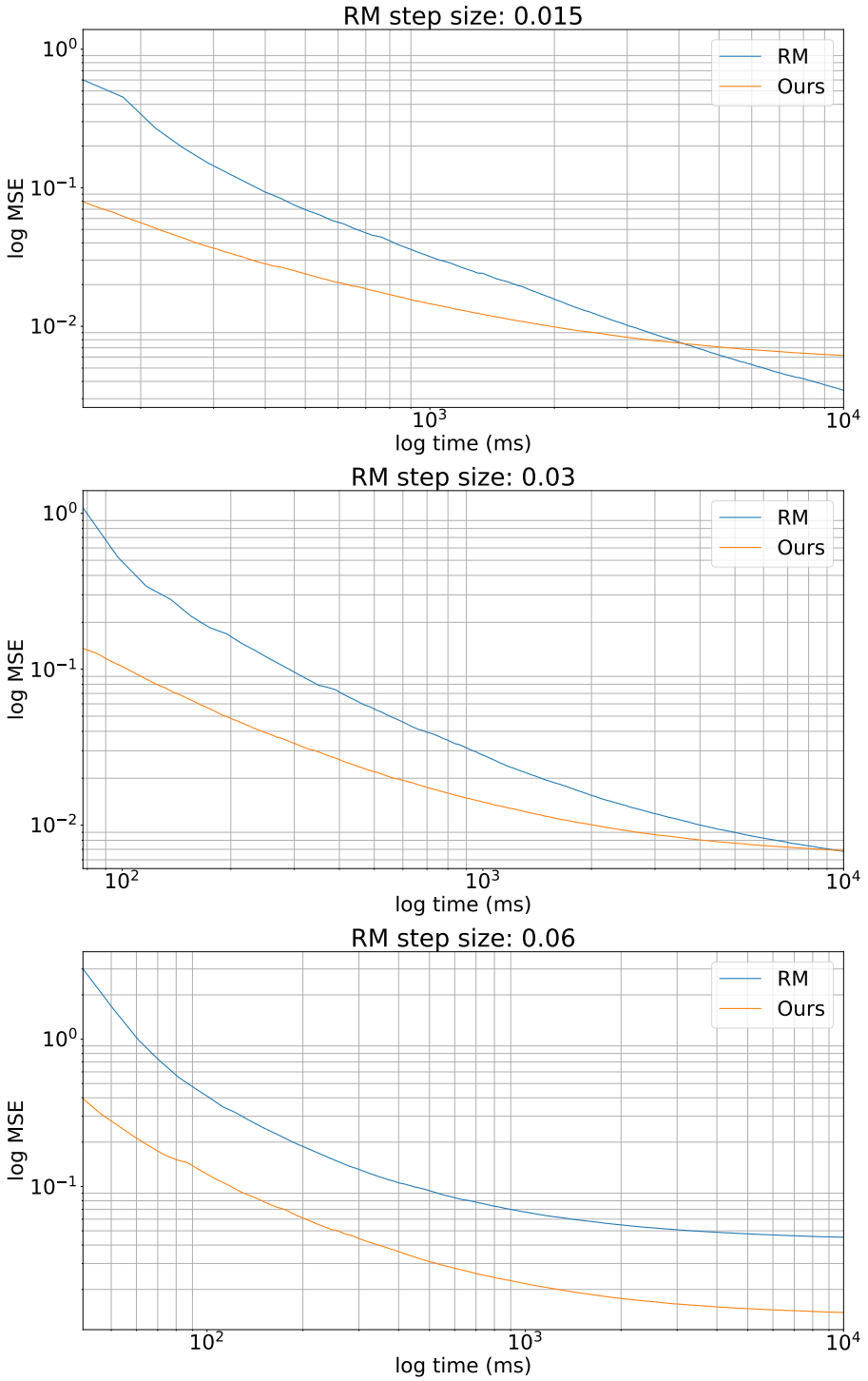


Figure 4.22: Cube: MSE over time for different step sizes. In each case our method initially converges faster than ray marching. Except for the last case, ray marching eventually catches up.

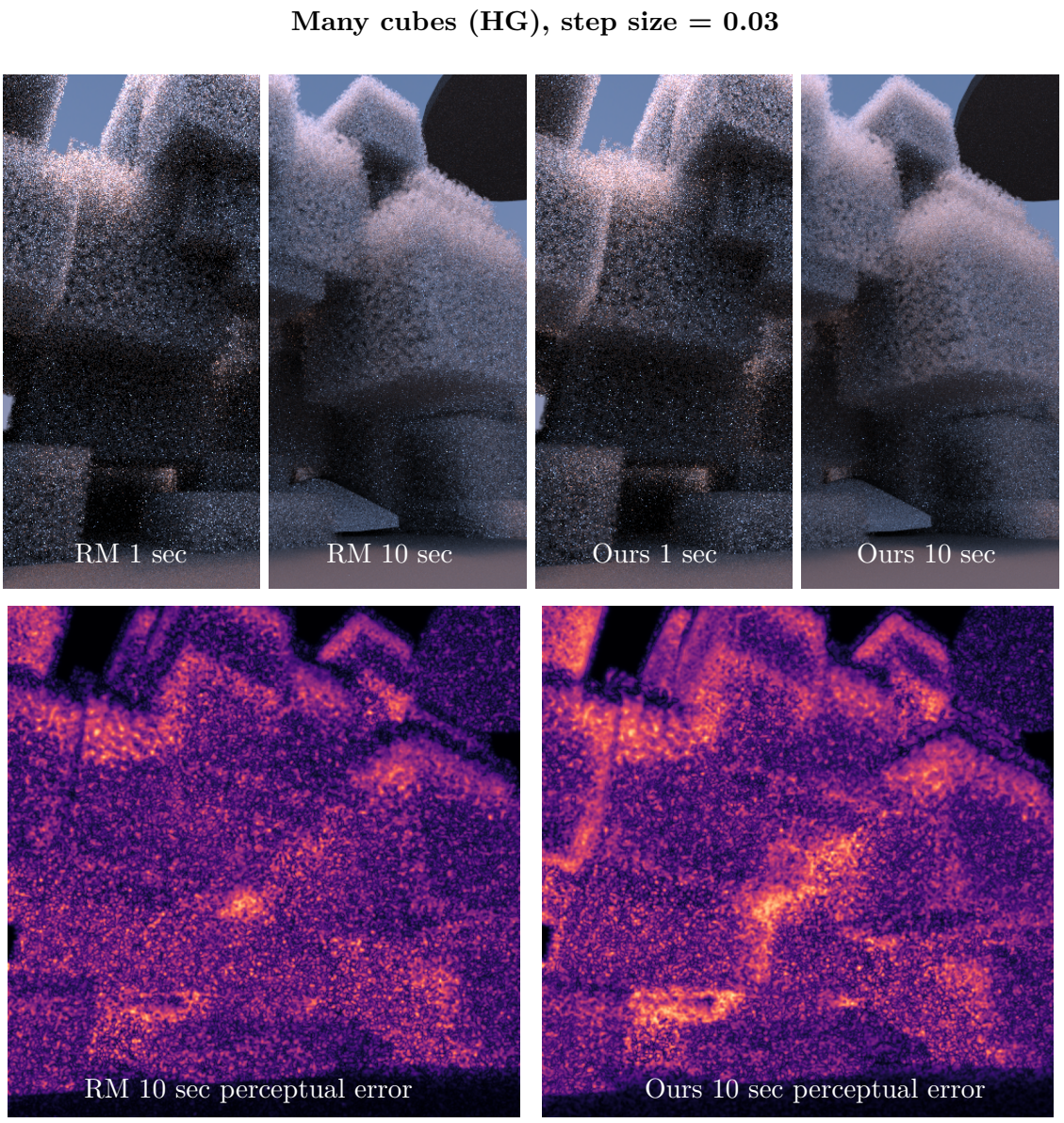


Figure 4.23: *Here, we render many instances of the cube from above. Again our algorithm produces less noise in an equal-time comparison. The bias produced by our method at the volume borders is more noticeable compared to ray marching at this step size.*

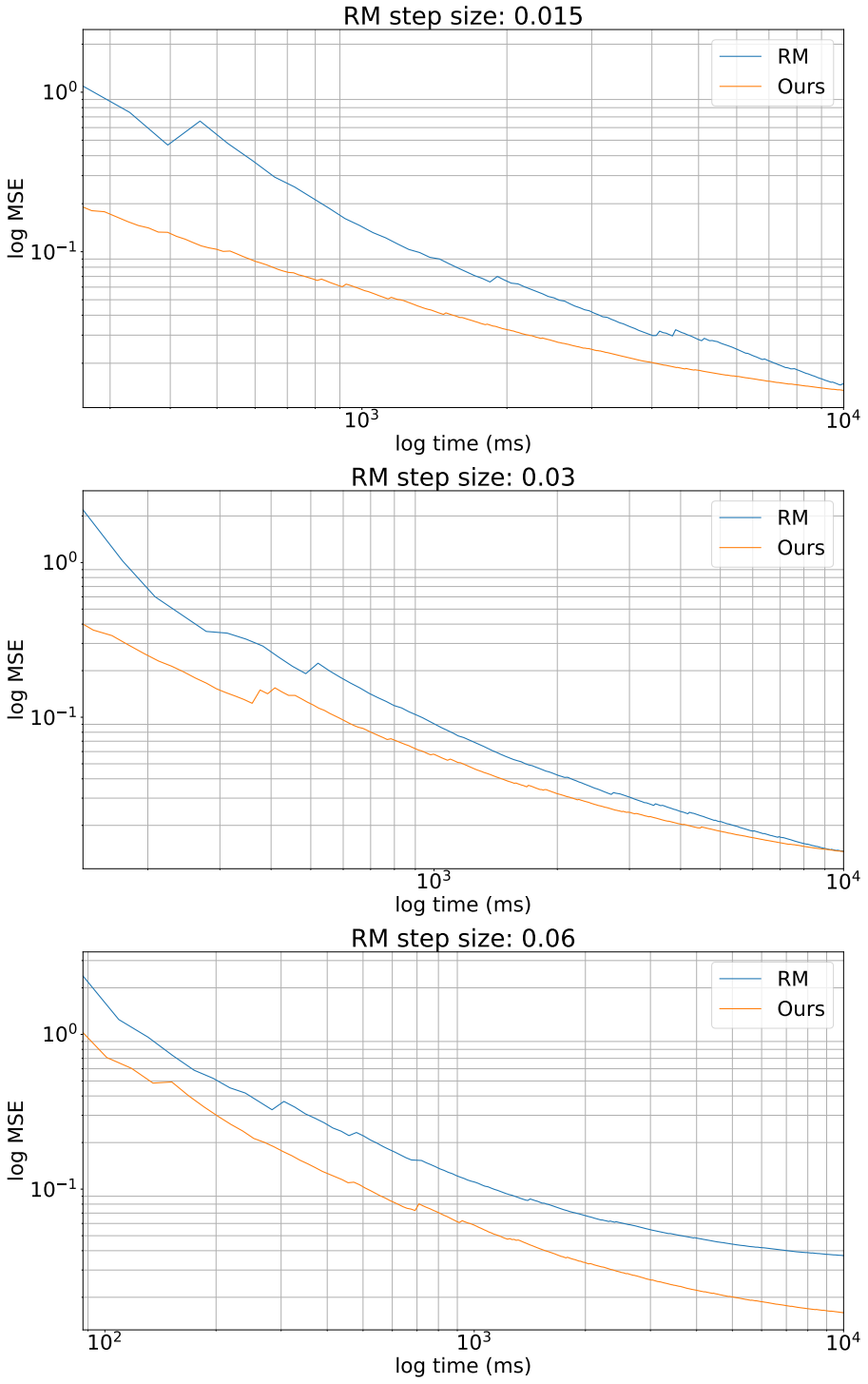


Figure 4.24: Many Cubes: MSE over time for different step sizes. The error curves are similar to the curves when rendering a single cube. The logarithmic plot enhances the effect of initial firefly samples on the shape of the curve. In this case these are more pronounced.

5. Discussion

In chapter 3 we evaluate the correctness of our bitmask sampling technique for a fully discrete scenario. To apply the respective methods of transmittance estimation and free path sampling to a real world application, using pins, a lot of approximations are made. The error made by such approximations is hard to judge from an analytic standpoint. In the last chapter we provide empirical data in order to reason about the quality of the result.

5.1 Ablation Study

Multiple scattering: Most importantly we find that our algorithm is suited for the computation of multiple scattering illumination. This of course does not account for all types of mediums and lightings. As with any heavily biased estimator, scenes can be constructed where the approximations produce large visible errors. In Figure 4.3 we see that the error produced by our algorithm does not lead to visible artifacts when it is blurred through multiple scattering. In 4.11 we see that jittering the grid access location also creates a more visually pleasing blur, compared to the hard lines induced by our space partition.

Single scattering: Of course, such a blur is not acceptable when computing the view ray. Also, when dealing with a highly directional illumination, such as the sun with a clear sky (figure 4.3), the artifacts and the respective jittered blur are starkly visible. Therefore our algorithm does not constitute a complete replacement for other tracking techniques at this moment. When used in combination with ray marching, the original scalar field is still needed.

Stream approach: We assess our stream approach in figure 4.2. The reduction in correlation of local distance and transmittance samples blurs the artifacts introduced by the 3D grid of our top-level data structure. We see this in figure 4.2 and 4.6. Though, we find that the effect is moderate and does not replace other methods of decorrelation. The stream is primarily useful to blur the box-like soft shadows on surfaces, inside the volume boundaries. The angular quantization remains visible for high sample counts, as we can see in 4.2 and 4.6.
When applying pins to multiple scattering and using jittering we obtain no visible improvement from updating pins. Therefore, computing the stream of pins for our currently viable use case is unnecessary. Nonetheless the stream implementation provides a very

fast way to update pins when changing the volume density in an interactive setting. Such a capability may be more useful in the context of an interactive volume visualizations.

Bitmask sampling: The results in figure 4.5 indicate that the error induced by our method stems primarily from the quantization too the pins and not along the pins. The bitmask sampling algorithms itself performs well for the types of volumes used in the last chapter.
While the error produced by the discretization along the pin might seem similar to the error produced by ray marching, it is not the same. A ray marched distance estimate may entirely step over a thin opaque layer. The bitmask representation, when computed with a high precision, will account for the corresponding optical mass, but may place the resulting distance sample behind that thin layer. We illustrate this in figure 5.1

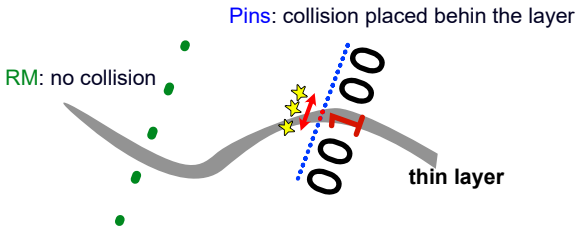


Figure 5.1: Comparison of errors made by ray marching and pins at thin layers.

5.2 Memory Requirements

Contrary to our initial beliefs we find that pins offer a compact representation of the medium. This may seem unintuitive because we store a high dimensional grid. But, when only applied to multiple scattering, we find that in the evaluated cases this grid can be very coarse. Because we still need the 3D texture for the primary ray and single scattering, the pin grid constitutes a memory overhead and not a compression of the original volume.

5.3 Performance Evaluation

For most of our evaluated test cases, our algorithm runs faster or in equal time compared to ray marching. The cases where ray marching is faster, are cases with large step sizes and few volumes. Our algorithm performs the best in the inverse case, thus low step sizes and many volume instances.

Fetching and sampling pins during path tracing comes at a constant low cost and is of course faster when ray marching needs to perform many steps. We pay for the reduced runtime by committing a larger bias in most of the evaluated cases.

Bias: We have found that the bias produced by pins is the worst at the boundaries of dense volumes (see figure 4.12). Contrast is lost, especially for highly directional illumination. We judge that pins sampled around the surface regions may generate collisions outside the volume.
While ray marching can decrease its bias by using a smaller step size, reducing the bias of pins is more difficult. Because pins use a fixed quantization over 32 intervals, decreasing the ray marching step size while computing pins does not help much in reducing the error made along the pin. Increasing the number of stored pins also comes at a diminishing return rate in reducing the error from quantizing to the pin. We see this in Figure 4.8.
For diffuse volumes such as clouds, the bias produced by our method is far less.

In a comparative analysis to ray marching, our algorithm performs the best when the cost of ray marching dominates the total runtime. We list 3 scenarios where this is the case.

Composition: Transmittance and distance can be evaluated for overlapping volumes independently as we have already seen with decomposition- and ratio-tracking.

When tracking media that is defined by additive overlapping volumes, ray marching becomes more expensive. The density needs to be read and interpolated separately for each overlapping instance. Our method also needs to be performed for each volume, but our memory footprint is lower as we only need to read 64 bit per instance.

In some of our test cases we additively combine many scaled instances of the same volume. In these cases our method performs strong when compared to ray marching. In the future work section we briefly discuss the inverse case, that is decomposing a volume into additive components.

High frequent data: Tracing pins can be performed as a precomputation. The results show that resampling them is not necessary in practice, when used for multiple scattering. Hence, the cost of pin sampling does not increase when the underlying volume requires a small step size for ray marching. In the comparative evaluation for a high frequency medium we find that our method converges faster than ray marching, for a well chosen step size. In return, ray marching achieves less bias in the long run. Our current number of test cases for using pins with high frequent media, is still too low to make any final assumptions. We believe that comparing the bias made by pins for different types of procedurally generated noise could be useful to assess for which types media to apply pins.

Large volumes: Ray marching large volumes can be an expensive task. Out of practical limitations our application we cannot evaluate our method for large volumes on the GPU at this time. The largest volume we evaluate in this thesis has a memory size of $\approx 0.25GB$. Because of the fixed discretization to 32 intervals along each pin, it might prove impractical to use pins over the entire volume. In that case, we suggest computing pins for local regions of the volume and stepping through these regions akin to regular tracking. We illustrate this in the future work section in figure 7.2.

6. Conclusion

We find that our algorithm, achieves its goal of reducing the total per sample runtime of MC path tracing in most evaluated cases. We have assessed the individual components and validated most of them. During that process, we found out that the stream computation offers no improvement for multiple scattering. We have verified the correctness of our sampling algorithms and find that the main error is produced by the positional-directional quantization. In the last chapter we have analyzed the strengths and weaknesses of our method. In a trade-off for a lower runtime, our method produces a larger bias than ray marching in most cases. Decreasing this bias by increasing the size of the pin grid comes at a diminishing return rate in terms of memory requirement. In return, when only used for multiple scattering, the bias is acceptable already for a small number of pins. For that case, our method can be implemented alongside other tracking methods with little memory overhead. In such a scenario our algorithm achieves good results for computing multiple scattering in diffuse media, for instance, when rendering clouds. We outperform full ray marching by a large margin, when the medium requires a small step size or when many overlapping volumes need to be tracked individually.

Because of our stream approach, our volume representation is designed to be continuously updated. This allows us to respond at runtime to varying medium densities. Therefore, we believe that at the moment, our method is best suited for computing multiple scattering for an interactive environment, where the tracking costs dominate the total runtime. But primarily, we provide our algorithm and its analysis, as a basis for future work to expand on, in particular by extending pins to large volumes.

7. Future Work

We believe that our method in its current state provides ample room for optimization. Throughout this thesis we have already mentioned several points that can be explored for future work. Here we list the 4 ideas we would most consider further investigating.

Tighter bounds: We judge that a significant error comes from sampling pins at locations outside the actual extent of the medium. Currently, when intersecting a volume from the outside, we sample pins at the ray intersection point with the oriented bounding box (OBB). By computing a tighter bound we could obtain an intersection point closer to the medium. We judge that this would cause fewer pins to 'miss' the volume entirely. This may prove especially beneficial when using pins for the primary ray and single scattering.

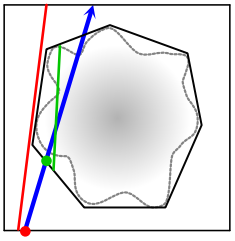


Figure 7.1: *Red: Pin sampled at OBB intersection point. Green: pin sampled closer to the medium.*

Octree + octahedral map: As we have already mentioned we suggest using a better parameterization of the hemisphere. We currently use a uniform grid over the polar coordinates. This parametrization leads to a high discrepancy in stored pin densities over the angle. We suggest using a different parametrization such as an octahedral map. Additionally, we suggest experimenting with using an octree to store pins at adaptive resolutions.

Volume decomposition: To apply pins to large volumes we recommend decomposing these into distinct regions, and store a pin grid per respective region. Alternatively we suggest decomposing the volume into additive components. For components with a low variation in density, a low resolution pin grid may be sufficient for large regions. In our current application we use this scheme to combine multiple scaled instances of the same volume, but of course it may also be applied to decomposition.

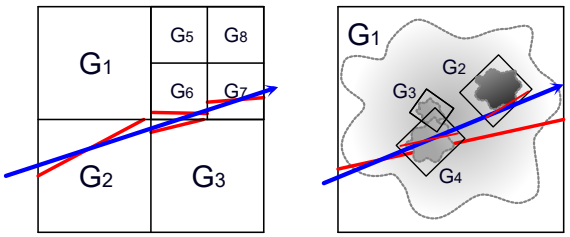


Figure 7.2: *Left: spatial decomposition. Right: additive decomposition. Pin grids are computed only for their respective component.*

Replace scalar field: If large sample counts are accumulated for a static scene, we suggest precomputing the depth-dependent transmittance from the camera and all light sources. We have seen different methods for this in Section 2.6. Combining these with pins for multiple scattering could be used to completely replace the need to step through the original scalar field while rendering.

8. Appendix

8.1 Code Samples

```
Pin create_pin(vec3 inOrigin, vec3 inDir, inout uint seed)
{
   vec3 origin = clamp(inOrigin, 0.0 , 0.99999);
   vec3 start, end;
   get_pin_segment(origin, inDir, start, end);
   vec3 dir = normalize(end - start);
   Pin pin;
   uint stepCount = 32;
   float stepLength = distance(end,start) / float(stepCount);
   float minTransmittance = 1.0;
   for(uint i = 0; i<stepCount; i++)
   {
      vec3 pos = start + dir * float(i) * stepLength;
      float transmittance = rayMarcheMediumTransmittance(pos, dir, stepLength,
          seed);
      minTransmittance = min(minTransmittance, transmittance);
   }
   pin.minTransmittance = minTransmittance;

   uint mask = 0;
   for(uint i = 0; i<stepCount; i++)
   {
      vec3 pos = start + dir * float(i) * stepLength;
      float transmittance = rayMarcheMediumTransmittance(pos, dir, stepLength,
          seed);
      float p = (1.0 - transmittance)/(1.0 - minTransmittance);

      if(p > unormNext(seed))
      {
         mask |= 1 << 31 - i;
      }
   }
   pin.mask = mask;

   return pin;
}
```

Figure 8.1: Code sample: Pin computation

```
float sample_msb_distance(uint mask, vec3 origin, vec3 direction, float
    maxLength, inout uint seed)
{
   if(mask == 0)
   {
      return TMAX;
   }
   bool inverseDir = direction.z < 0;
   vec3 start, end;
   unitCubeIntersection(origin, direction, start, end);

   float relOffset = distance(origin, start) / distance(start, end); // [0, 1]
   relOffset = clamp(relOffset, 0.0, 0.9999); // [0, 1)
   uint bitOffset = uint(relOffset * 32); // [0, 31]
   float perBitDistance = distance(start, end) / float(32);

   if(inverseDir)
   {
      mask = bitfieldReverse(mask);
   }
   // x = position relative to pin
   // start   x           end
   //     00101001101110011010110
   mask = mask << bitOffset; // Shift
   //     x            end
   //     10111001110101100000000
   uint sampledDiscreteDist = 31 - findMSB(mask); // [0, 32] (32 == infinity)
   if(sampledDiscreteDist == 32)
   {
      return TMAX;
   }
   float sampledDist = float(sampledDiscreteDist) * perBitDistance;
   sampledDist += unormNext(seed) * perBitDistance; // randomize hit pos within
       the bit

   if(sampledDist > maxLength)
   {
      return TMAX;
   }
   return sampledDist;
}
float pin_sample_distance(Pin pin, vec3 origin, vec3 direction, float maxLength,
    inout uint seed)
{
   float majorant = 1.0 - pin.minTransmittance;
   uint sampleMask = gen_sample_mask(majorant, RNG_SAMPLE_MASK_ITERATIONS, seed);
   uint mask = pin.mask & sampleMask;
   return sample_msb_distance(mask, origin, direction, maxLength, seed);
}
```

Figure 8.2: Code sample: Pin distance sampling

```
uint pin_count_bits(uint mask, vec3 origin, vec3 direction, float maxLength,
    inout uint seed)
{
   if(mask == 0)
   {
      return 0;
   }

   bool inverseDir = direction.z < 0;
   vec3 start, end;
   unitCubeIntersection(origin, direction, start, end);

   float perBitDistance = distance(start, end) / float(32);

   float relOffset = distance(origin, start) / distance(start, end); // [0, 1]
   relOffset = clamp(relOffset, 0.0, 0.9999); // [0, 1)
   uint bitOffset = uint(relOffset * 32); // [0, 31]

   float relEndOffset = relOffset + maxLength / distance(start, end); // [0, 1]
   float invRelEndOffset = 1.0 - relEndOffset;
   invRelEndOffset = clamp(invRelEndOffset, 0.0, 0.9999); // [0, 1)
   uint invBitEndOffset = uint(invRelEndOffset * 32); // [0, 31]

   uint maxLengthBitMask = 0xFFFFFFFF << invBitEndOffset;

   if(inverseDir)
   {
      mask = bitfieldReverse(mask);
   }
   // [ ] range over which we want to evaluate transmittance
   // start   [    ]     end
   //     00101001101110011101011O
   mask &= maxLengthBitMask;
   // start   [    ]     end
   //     00101001101110011000000O
   mask = mask << bitOffset; // Shift
   //       [   ]     end
   //     10111001100000000000000O
   return bitCount(mask); // [0, 32]
}
float pin_estimate_transmittance(Pin pin, vec3 origin, vec3 direction, float
    maxLength, inout uint seed)
{
   uint hitCount = pin_count_bits(pin.mask, origin, direction, maxLength, seed);
   float transmittance = pow(pin.minTransmittance, hitCount);
   return transmittance;
}
```

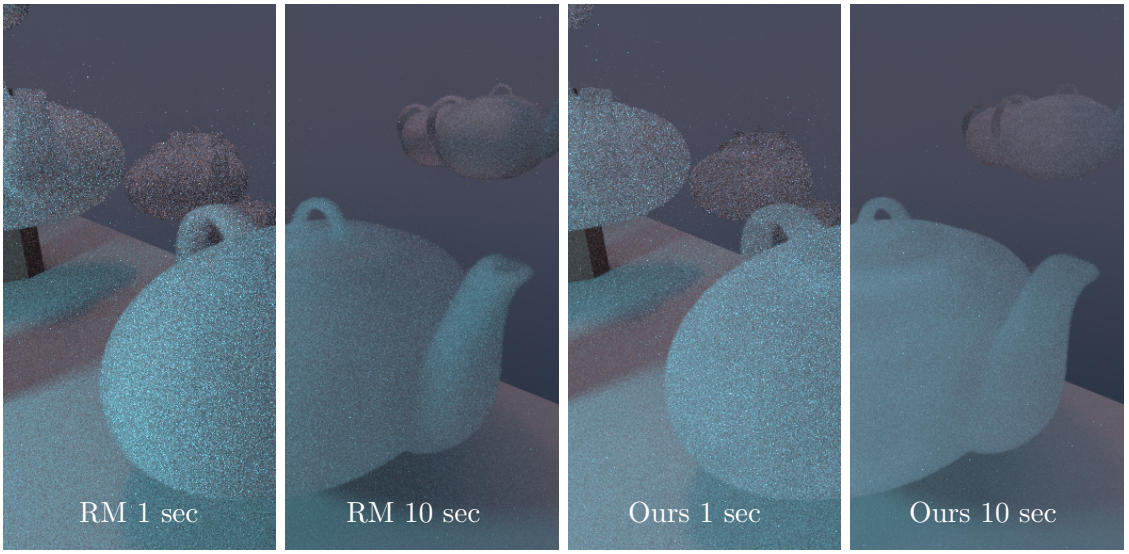
Figure 8.3: Code sample: Pin transmittance sampling

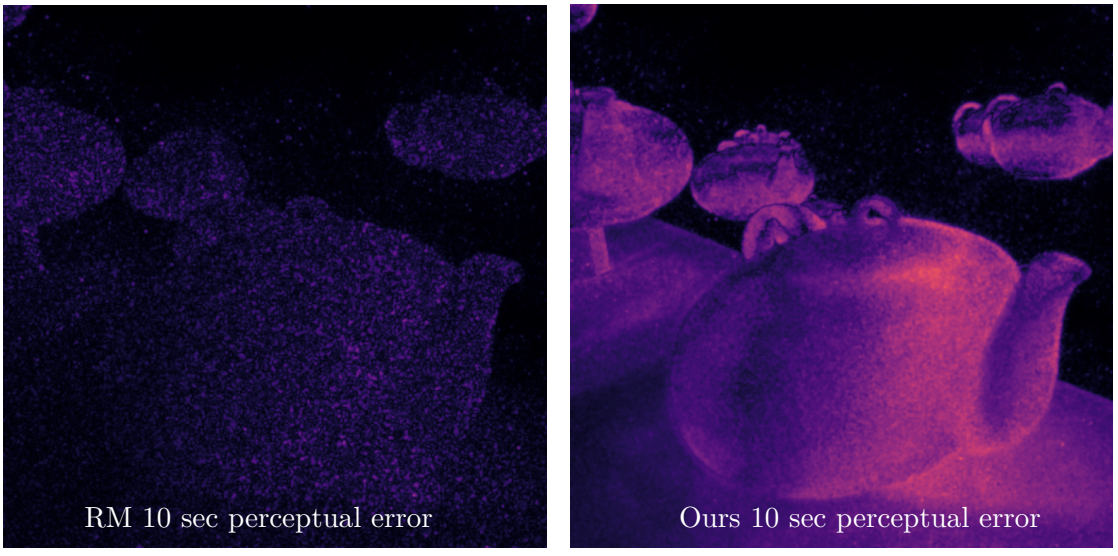
```
uint gen_sample_mask(float p, uint iterations, inout uint seed)
{
   uint upperMask = 0xFFFFFFFFU; // 32 bits all 1
   uint lowerMask = 0x00000000U; // 32 bits all 0
   float upperP = 1.0;
   float lowerP = 0.0;
   uint middleMask;
   for(uint i = 0; i<iterations; i++)
   {
      uint h = hash(seed); // We assume P(h[i] = 1) = 0.5
      seed = h;
      middleMask = h & lowerMask | (~h) & upperMask;
      float middleP = mix(lowerP, upperP, 0.5);
      if(middleP > p)
      {
         upperMask = middleMask;
         upperP = middleP;
      }
      else
      {
         lowerMask = middleMask;
         lowerP = middleP;
      }
   }
   return middleMask;
}
```

Figure 8.4: Code sample: Sample mask generation

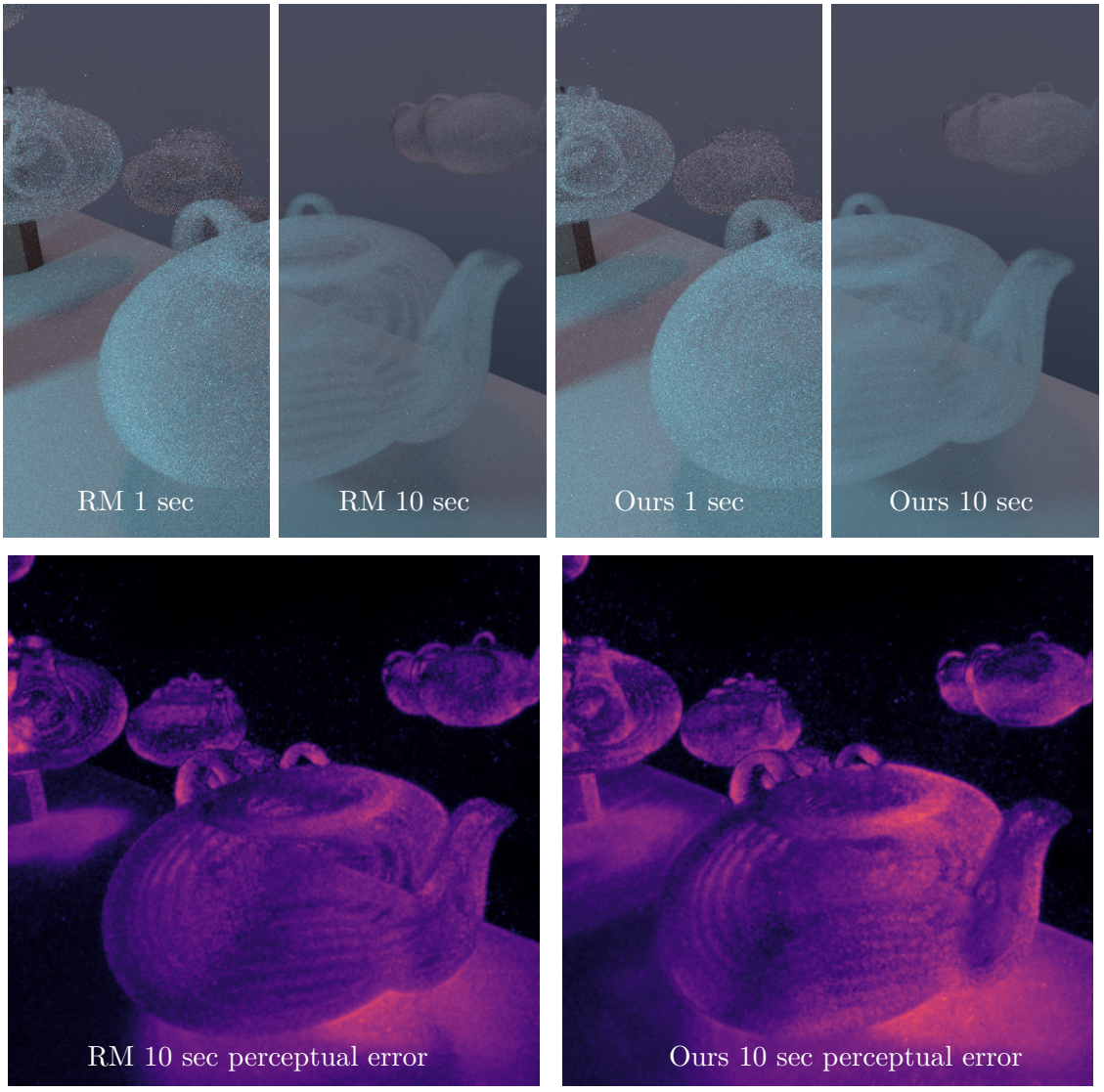
8.2 Additional Results

Teapot, step size = 0.015

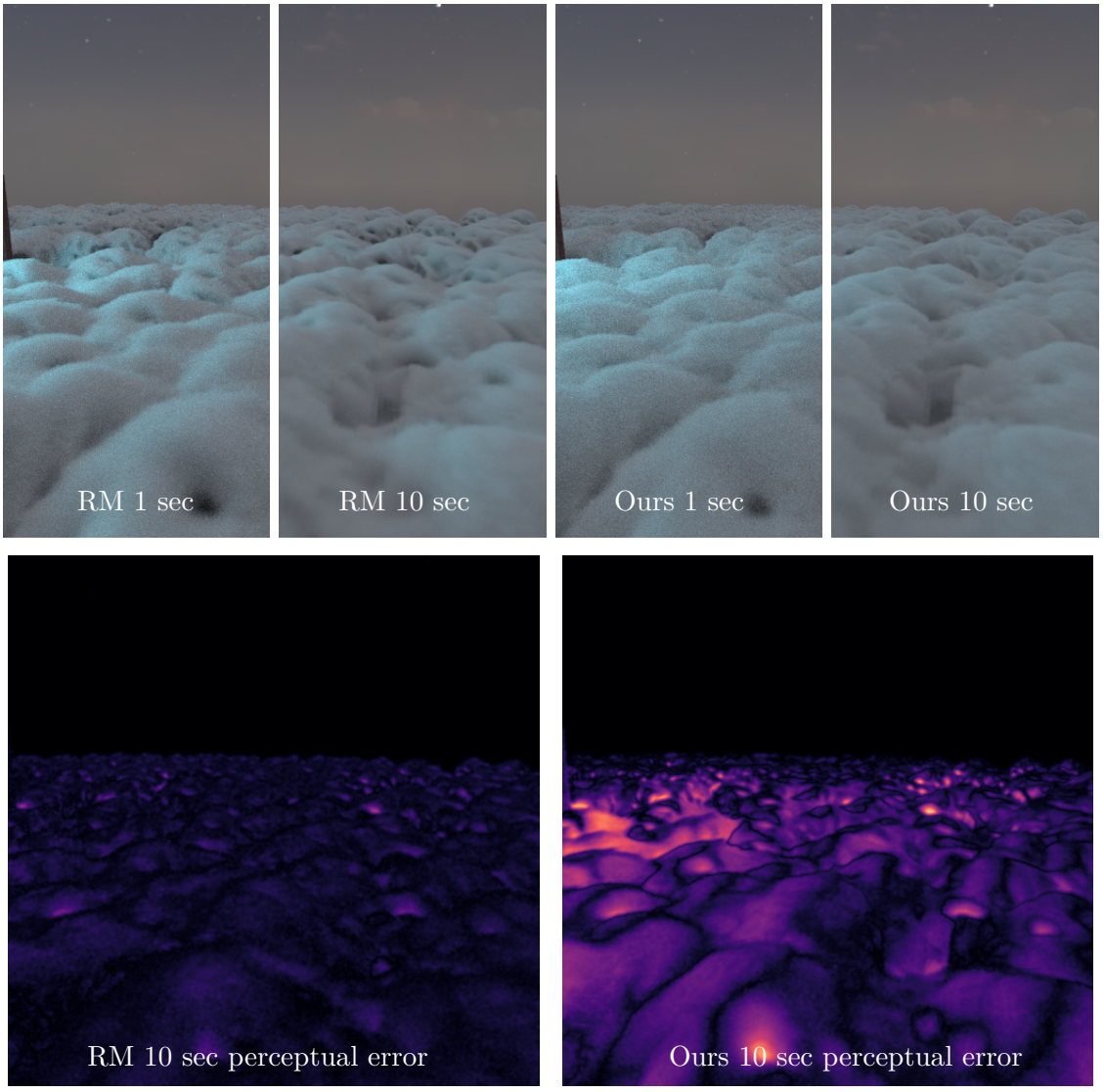




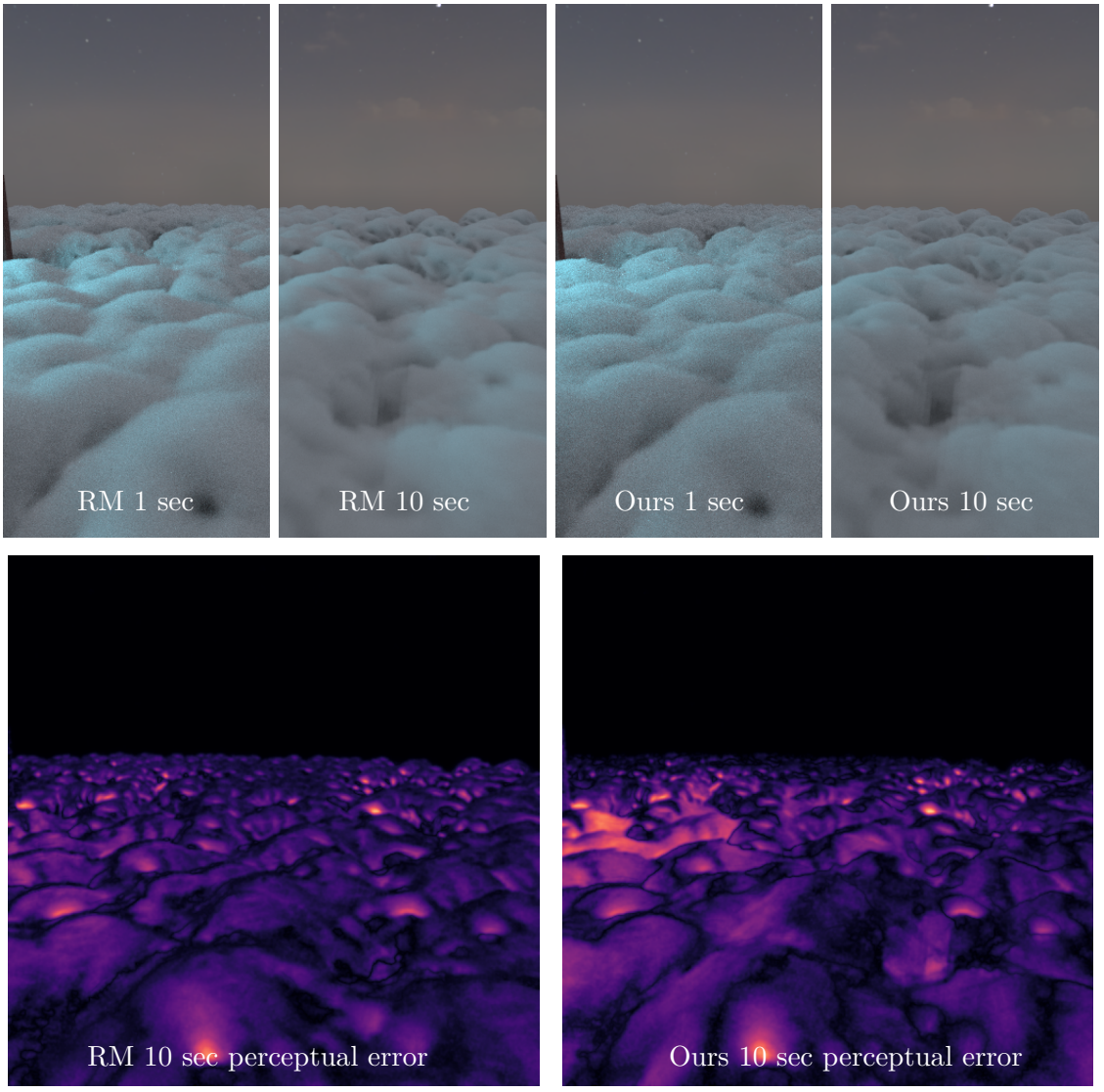
Teapot, step size = 0.06



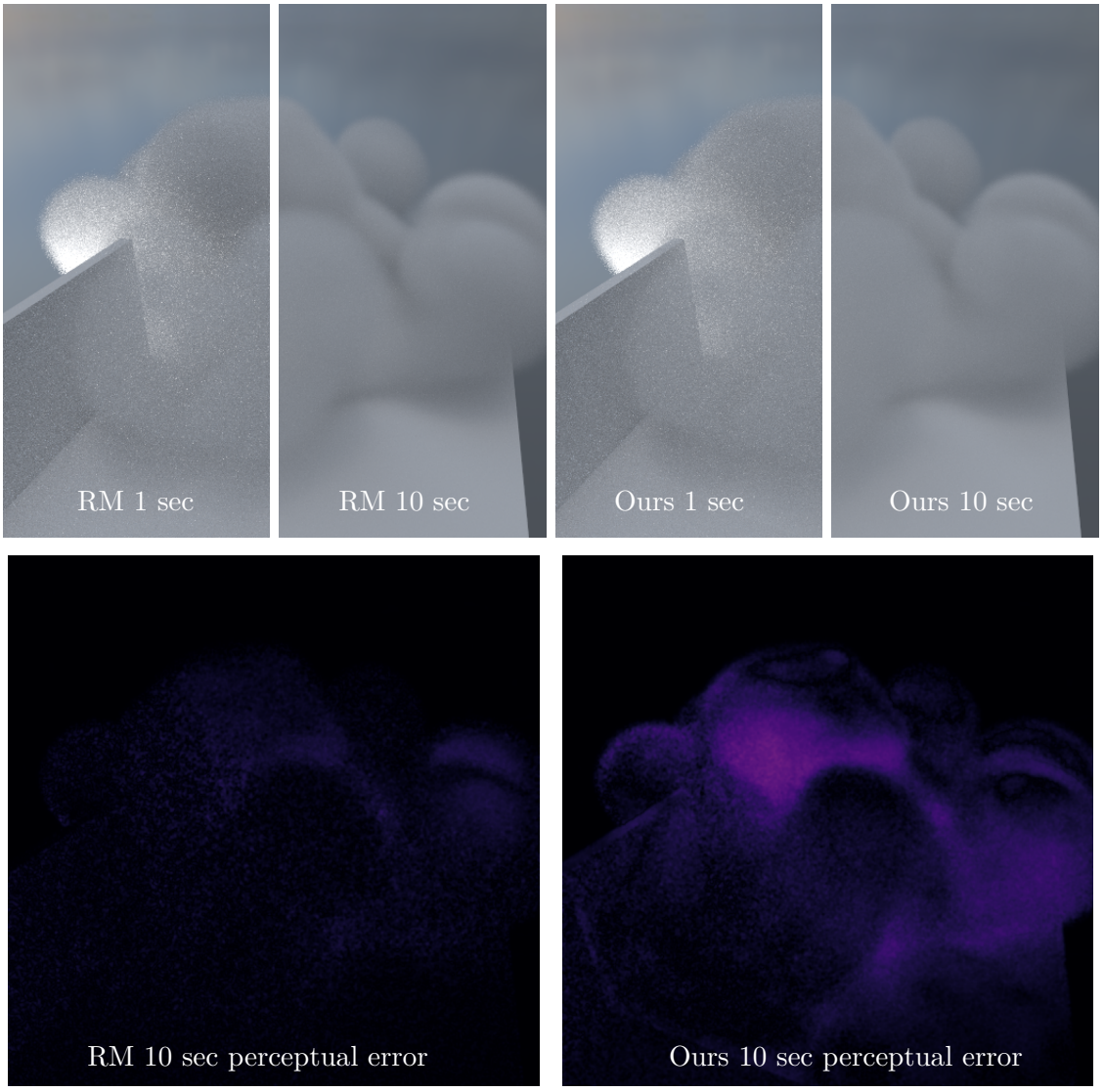
Snow, step size = 0.03



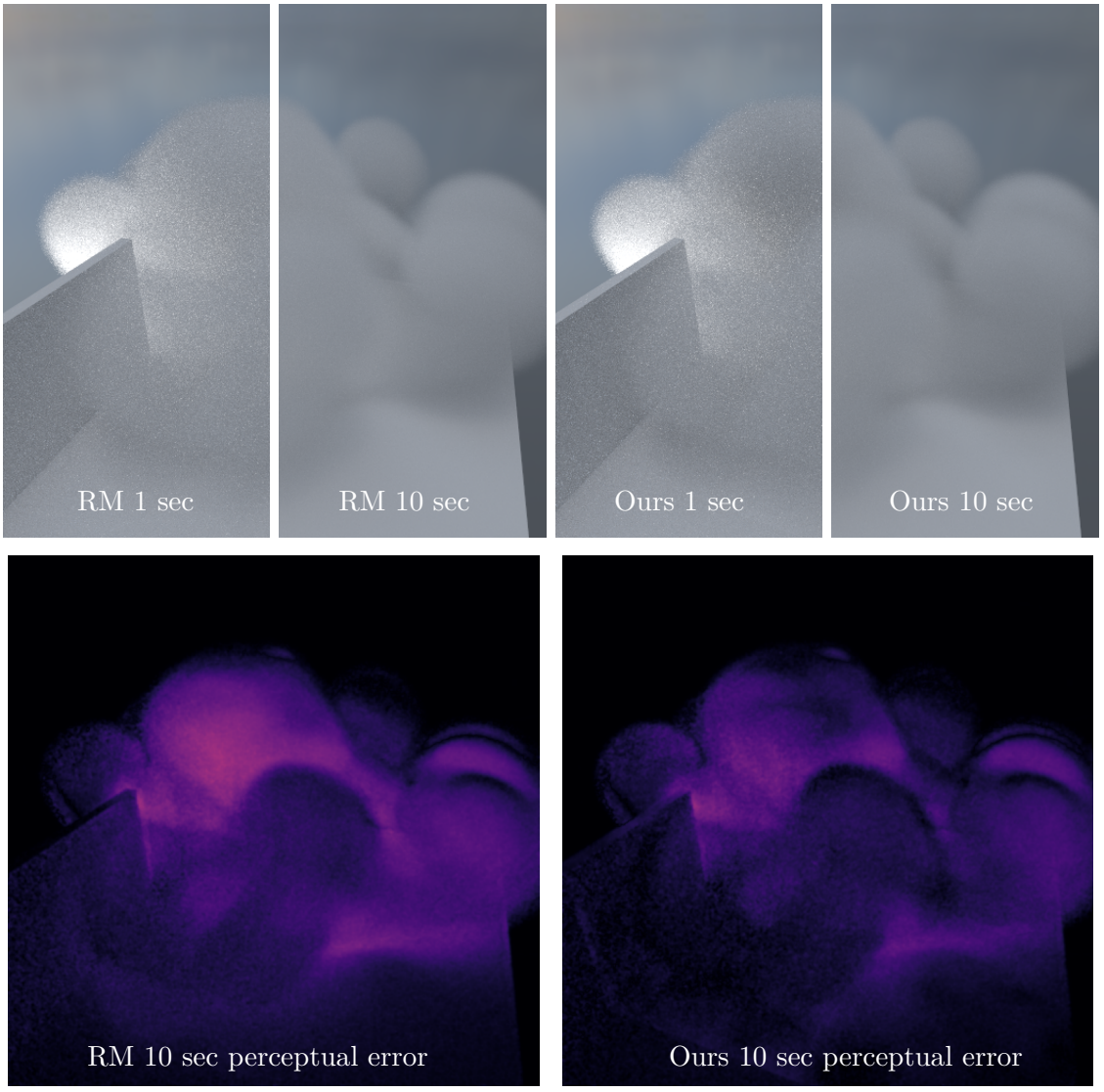
Snow, step size = 0.06



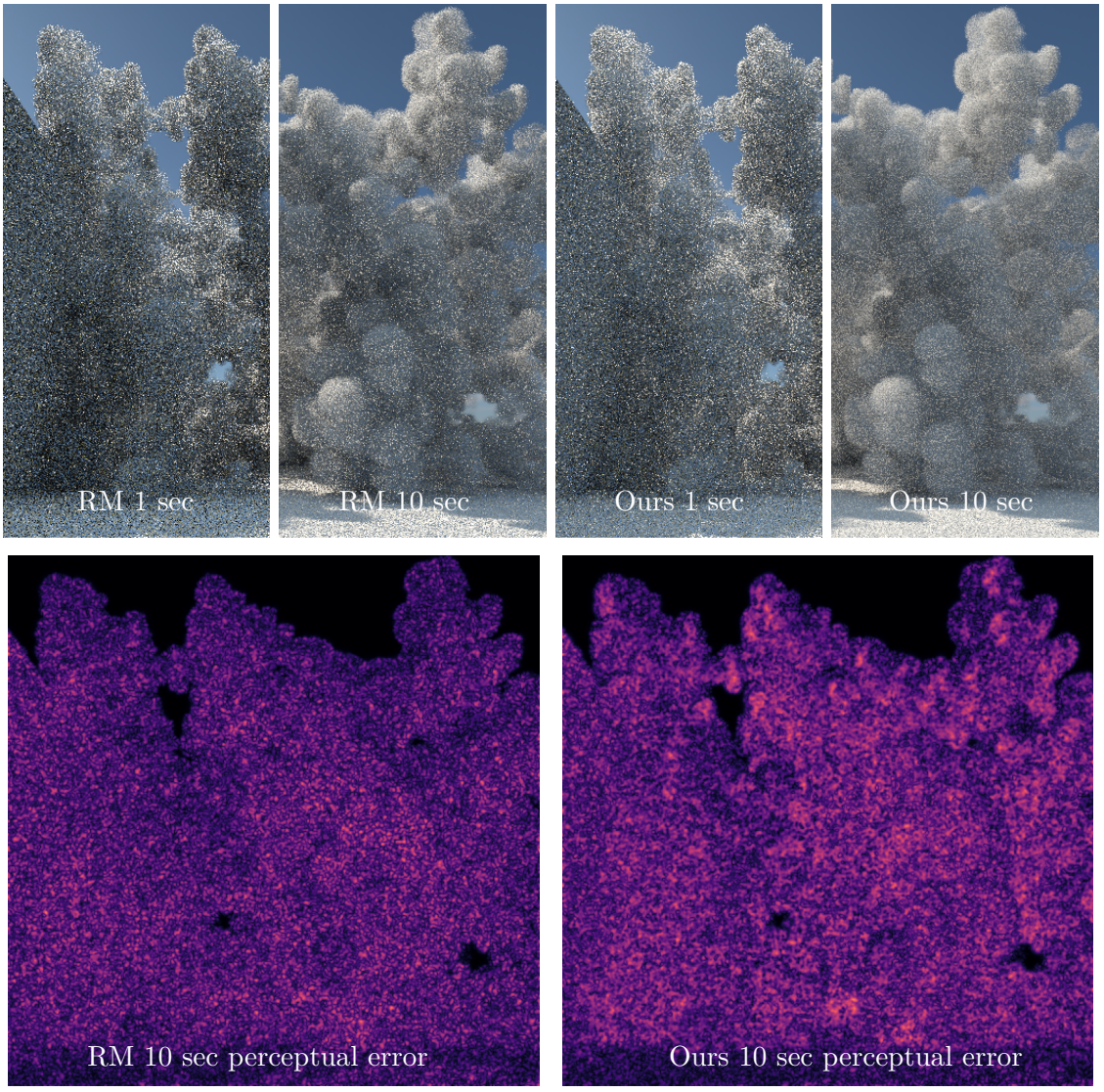
Cloud, step size = 0.04



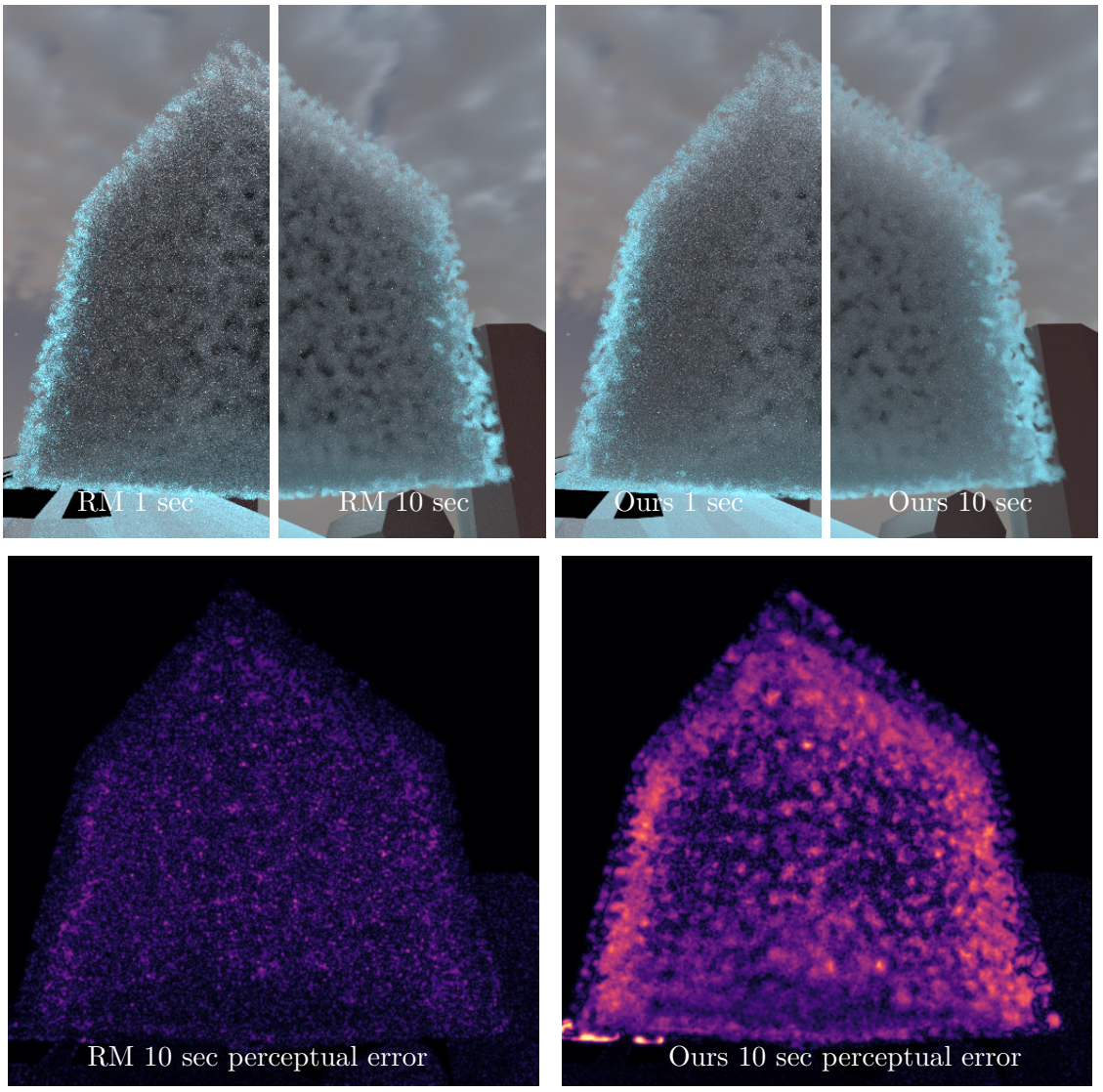
Cloud, step size = 0.16



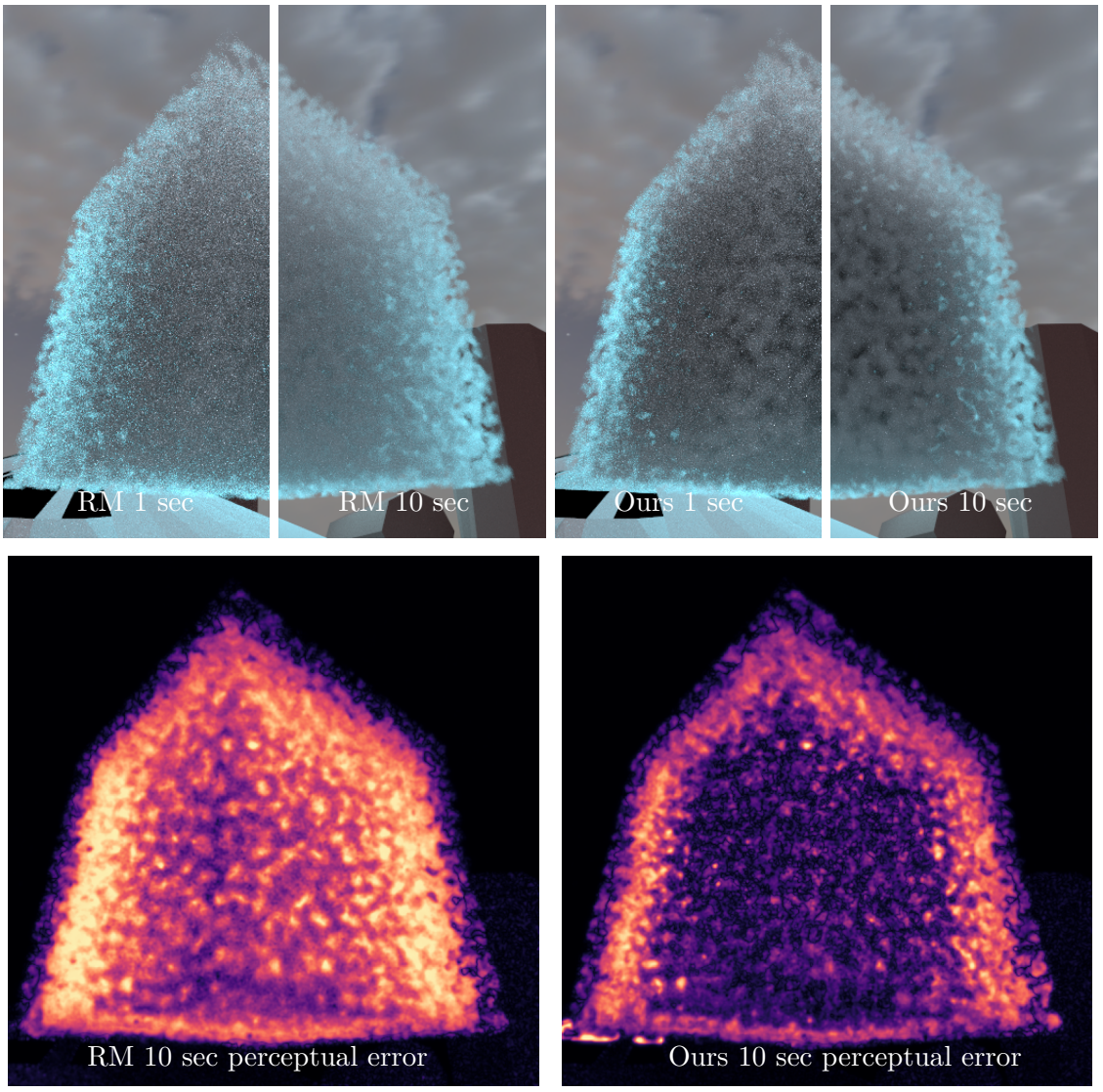
Many cloud, step size = 0.04



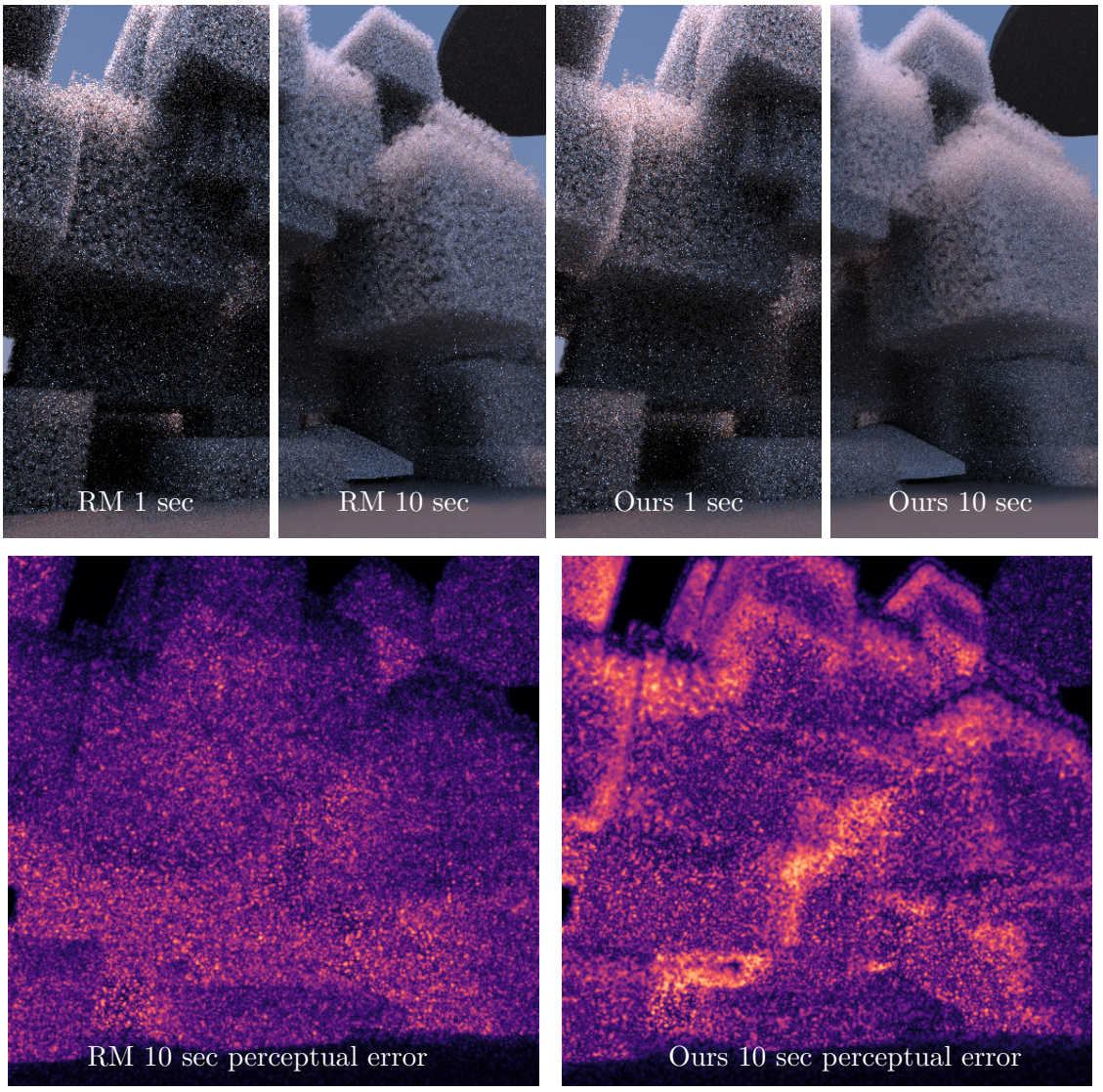
Cube (HG), step size = 0.015



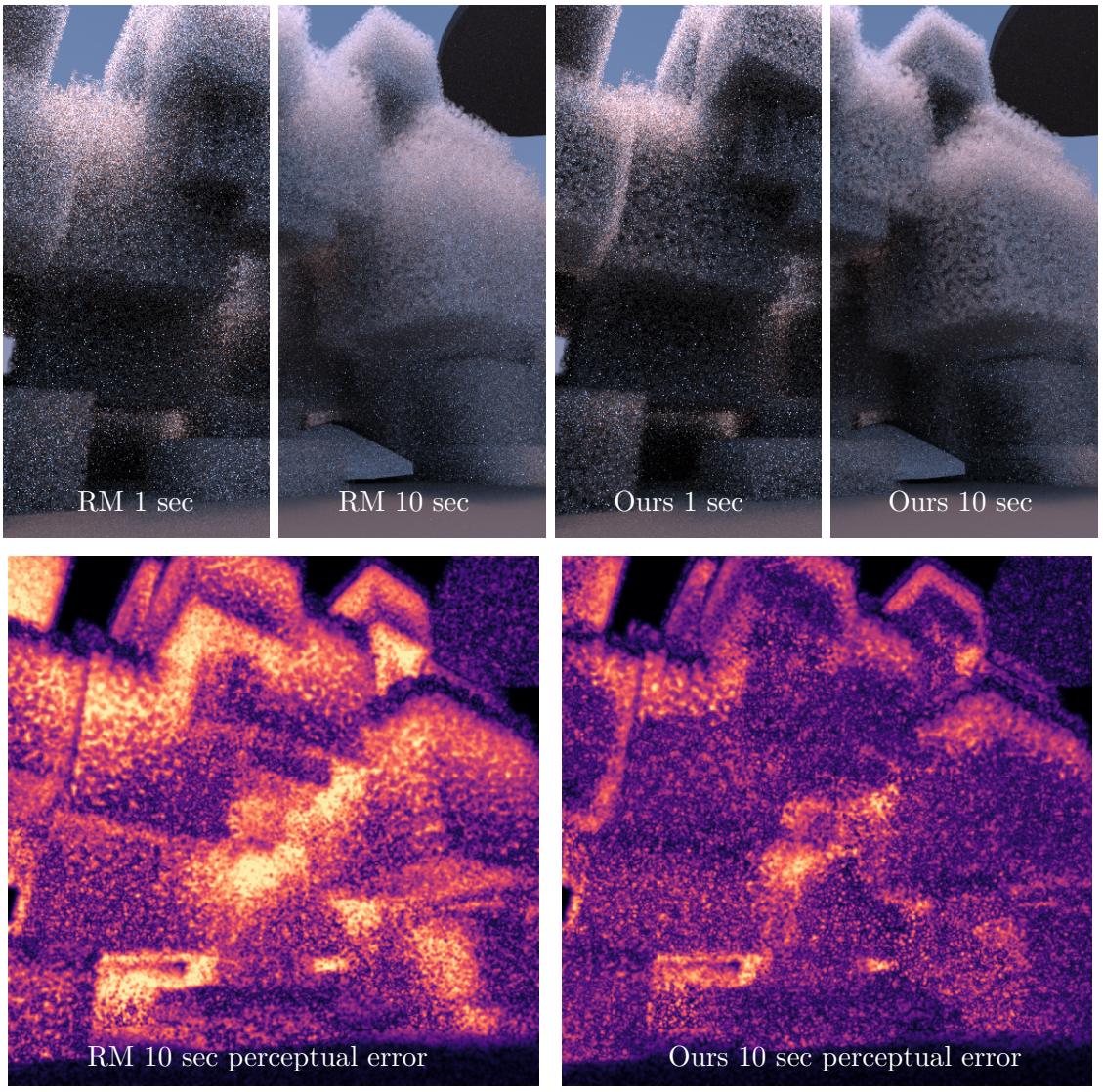
Cube (HG), step size = 0.06



Many cubes (HG), step size = 0.015



Many cubes (HG), step size = 0.06



Bibliography

[ANSA21] P. Andersson, J. Nilsson, P. Shirley, and T. Akenine-Möller, "Visualizing Errors in Rendered High Dynamic Range Images," in *Eurographics Short Papers*, May 2021.

[CPP+05] E. Cerezo, F. Pérez, X. Pueyo, F. J. Seron, and F. X. Sillion, "A survey on participating media rendering techniques," pp. 303–328, 2005.

[ED08] T. Engelhardt and C. Dachsbacher, "Octahedron environment maps." in *VMV*. Citeseer, 2008, pp. 383–388.

[GBC+13] M. Galtier, S. Blanco, C. Caliot, C. Coustet, J. Dauchet, M. El Hafi, V. Eymet, R. Fournier, J. Gautrais, A. Khuong *et al.*, "Integral formulation of null-collision monte carlo algorithms," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 125, pp. 57–68, 2013.

[Jen06] J. L. W. V. Jensen, "Sur les fonctions convexes et les inégalités entre les valeurs moyennes," *Acta mathematica*, vol. 30, no. 1, pp. 175–193, 1906.

[Jen96] H. W. Jensen, "Global illumination using photon maps," in *Rendering Techniques' 96: Proceedings of the Eurographics Workshop in Porto, Portugal, June 17–19, 1996 7*. Springer, 1996, pp. 21–30.

[JNSJ11] W. Jarosz, D. Nowrouzezahrai, I. Sadeghi, and H. W. Jensen, "A comprehensive theory of volumetric radiance estimation using photon points and beams," *ACM transactions on graphics (TOG)*, vol. 30, no. 1, pp. 1–19, 2011.

[JNT+11] W. Jarosz, D. Nowrouzezahrai, R. Thomas, P.-P. Sloan, and M. Zwicker, "Progressive photon beams," *ACM Trans. Graph.*, vol. 30, no. 6, p. 1–12, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2070781.2024215

[Kaj86] J. T. Kajiya, "The rendering equation," in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986, pp. 143–150.

[KHLN17] P. Kutz, R. Habel, Y. K. Li, and J. Novák, "Spectral and decomposition tracking for rendering heterogeneous volumes," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, pp. 1–16, 2017.

[Kla17] P. Klacansky, "Open scivis datasets," December 2017, `https://klacansky.com/open-scivis-datasets/`. [Online]. Available: https://klacansky.com/open-scivis-datasets/

[KVS+14] R. Khlebnikov, P. Voglreiter, M. Steinberger, B. Kainz, and D. Schmalstieg, "Parallel irradiance caching for interactive monte-carlo direct volume rendering," in *Computer Graphics Forum*, vol. 33, no. 3. Wiley Online Library, 2014, pp. 61–70.

[LV23] T. Lokovic and E. Veach, *Deep Shadow Maps*, 1st ed. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3596711.3596746

[MKKP18] C. Münstermann, S. Krumpen, R. Klein, and C. Peters, "Moment-based order-independent transparency," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 1, no. 1, pp. 1–20, 2018.

[MLJ+13] K. Museth, J. Lait, J. Johanson, J. Budsberg, R. Henderson, M. Alden, P. Cucka, D. Hill, and A. Pearce, "Openvdb: an open-source data structure and toolkit for high-resolution volumes," in *Acm siggraph 2013 courses*, 2013, pp. 1–1.

[Mus21] K. Museth, "Nanovdb: A gpu-friendly and portable vdb data structure for real-time rendering and simulation," in *ACM SIGGRAPH 2021 Talks*, 2021, pp. 1–2.

[NGHJ18] J. Novák, I. Georgiev, J. Hanika, and W. Jarosz, "Monte carlo methods for volumetric light transport simulation," in *Computer graphics forum*, vol. 37, no. 2. Wiley Online Library, 2018, pp. 551–576.

[Ols07] J. Olsson, "Ray tracing animations using 4d kd-trees," Master's thesis, 2007.

[PK15] C. Peters and R. Klein, "Moment shadow mapping," in *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, 2015, pp. 7–14.

[RPD22] T. Rapp, C. Peters, and C. Dachsbacher, "Image-based visualization of large volumetric data using moments," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 6, pp. 2314–2325, 2022.

[SKS23] P.-P. Sloan, J. Kautz, and J. Snyder, "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments," in *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*, 2023, pp. 339–348.

[SVL+18] M. Salvi, K. Vidimče, A. Lauritzen, A. Lefohn, and M. Pharr, "Adaptive volumetric shadow maps," *GPU Pro 360 Guide to Shadows*, pp. 97–113, 2018.

[Vea98] E. Veach, *Robust Monte Carlo methods for light transport simulation*. Stanford University, 1998.

[WMHL65] E. Woodcock, T. Murphy, P. Hemmings, and S. Longworth, "Techniques used in the gem code for monte carlo neutronics calculations in reactors and other systems of complex geometry," in *Proc. Conf. Applications of Computing Methods to Reactor Problems*, vol. 557, no. 2. Argonne National Laboratory, 1965.

[ZHWG08] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Transactions on Graphics (TOG)*, vol. 27, no. 5, pp. 1–11, 2008.

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen.

Karlsruhe, den January 20, 2025

(Sidney Hansen)