Production Volume Rendering
SIGGRAPH 2017 Course

JULIAN FONG, Pixar Animation Studios
MAGNUS WRENNINGE, Pixar Animation Studios
CHRISTOPHER KULLA, Sony Pictures Imageworks
RALF HABEL, Walt Disney Animation Studios

Fig. 1. Te Ka, a character from Disney's Moana (2016)

With significant advances in techniques, along with increasing computational power, path tracing has now become the predominant rendering method used in movie production. Thanks to these advances, volume rendering can now take full advantage of the path tracing revolution, allowing the creation of photoreal images that would not have been feasible only a few years ago. However, volume rendering also provides its own set of unique challenges that can be daunting to path tracer developers and researchers accustomed to dealing only with surfaces. While recent texts and materials have covered some of these challenges, to the best of our knowledge none have comprehensively done so, especially when confronted with the complexity and scale demands required by production. For example, the last volume rendering course at SIGGRAPH in 2011 discussed ray marching and precomputed lighting and shadowing, none of which are techniques advisable for production purposes in 2017.

AUTHORS

Julian Fong

Julian Fong is a senior software engineer at Pixar Animation Studios. He has worked on RenderMan since 1999, with credits on all films since Finding Nemo. His key contributions to the renderer include enhancements to the REYES algorithm, hierarchical and Loop subdivision surfaces, deep compositing, optimization of ray tracing, and volume rendering.

Magnus Wrenninge

Magnus Wrenninge is the author of the book Production Volume Rendering and the Sci-Tech awarded open source project Field3D. He started his career writing fluid simulation and environment rendering software at Digital Domain, and in 2005 he joined Sony Imageworks where he worked both as a Senior Software Developer and Lead Technical Director on films such as Alice in Wonderland and Oz: The Great and Powerful. Since 2013 he is working as a Principal Engineer at Pixar Animation Studios, where he focuses on all things related to rendering.

Christopher Kulla

Christopher Kulla is a principal software engineer at Sony Pictures Imageworks where he has worked on the in-house branch of the Arnold renderer since 2007. He focuses on ray tracing kernels, sampling techniques and volume rendering. In 2017 he was recognized with a Scientific and Engineering Award from the Academy of Motion Picture Arts and Sciences for his work on the Arnold renderer.

Ralf Habel

Ralf Habel is a senior software engineer at the Walt Disney Animation Studios. He studied theoretical physics at the University of Stuttgart and did a PhD in computer graphics at the Vienna University of Technology. He was a Postdoctoral researcher at Disney Research Zurich before joining the Hyperion team at Walt Disney Animation Studios focusing on volumes and level set rendering.

Contents

Abstract 1
Contents 3
1 Introduction 5
2 Volume Rendering Theory 6
2.1 Properties of Volumes 6
2.1.1 Extinction and Single Scattering Albedo Parametrization 7
2.2 Light Propagation in Volumes 8
2.2.1 Radiative Transfer Equation 8
2.2.2 Volume Rendering Equation 9
2.3 Tracking Approach 10
2.3.1 Closed-form Tracking 10
2.3.2 Regular Tracking 12
2.3.3 Delta Tracking 13
2.3.4 Advanced Tracking Methods 15
2.4 PDF Approach 15
2.4.1 Transmittance Estimators 16
2.4.2 Path Directions and Reciprocity 17
2.4.3 Comparison of Approaches 18
2.4.4 Other Volume Rendering Methods 18
3 Adding volumes to a path tracer 19
3.1 Volume integrator execution model 20
3.2 Binding volumes to geometry 24
3.3 Light integrator 27
3.4 Transmittance rays 29
3.5 Heterogeneous volumes 31
3.6 Single scattering 33
3.6.1 Closed-form tracking approach 34
3.6.2 PDF approach 40
3.7 Multiple scattering 44
3.8 Emission 49
3.9 Overlapping volumes 51
4 Optimization 57
4.1 Multiple sampling 57
4.2 Importance sampling for Single Scattering 58
4.2.1 Uniform sampling and MIS 58
4.2.2 Equiangular Sampling 60
4.2.3 Equiangular Sampling from Area lights 62
4.2.4 Equiangular sampling in Heterogeneous media 62
4.2.5 Implementing equiangular sampling in a renderer 64
4.3 Importance sampling for Multiple Scattering 64
4.4 Tracking improvements 65
4.4.1 Residual and ratio tracking 65
4.4.2 Acceleration structures for tracking 67
5 Motion Blur 74

5.1 The problem with motion blur 74
5.2 Smeared motion blur 74
5.3 Microvoxel motion blur 75
5.4 Eulerian motion blur 75
5.5 Motion blur with tracking-based integrators 76
5.6 Temporal volumes 77
6 Aggregate Volumes 79
6.1 Integration intervals 79
6.2 Integration intervals and tracking-based integrators 79
6.3 The volume–integrator interface 80
6.4 The cost of providing meta-properties 81
6.5 Changing the interface 81
6.6 Multiple overlapping volumes 82
6.7 Volumes in aggregate 83
6.8 Volume aggregate structure 84
6.9 Implementing the aggregate method 85
6.10 Filter width and motion blur 87
6.11 Handling small volumes 88
6.12 Efficient volume aggregates 89
6.13 Performance examples 90
7 Volume Rendering in Hyperion 91
7.1 Hyperion Streaming Architecture 91
7.2 Light Cache Points 92
7.3 Hyperion Volume Architecture 93
7.3.1 Flux on Light Cache Points 93
7.3.2 Per-Volume Acceleration Grids and Estimate 93
7.3.3 Transmittance Estimate 93
7.3.4 Building the In-scattering PDF 94
7.3.5 Sampling the In-scattering PDF 94
7.4 Volumes in Production Scenes 94
7.5 User Parameters 95
7.6 Conclusion and Future 95
References 96

1 INTRODUCTION

The last SIGGRAPH courses on production volume rendering were held in 2011 (Wrenninge and Bin Zafar 2011; Wrenninge et al. 2011). A year later, a book (Wrenninge 2012) by the same name was published, summarizing and elaborating on much of the work. To say that the field has moved fast since then is an understatement: in only six years, the entire field has shifted, and nearly all of the techniques and methods presented at the time have since been superseded. Thanks to significant advances in techniques and increases in computational power, path tracing has now become the predominant rendering method used in movie production, entirely eliminating scanline algorithms. Where raymarching once served as the go-to rendering method, tracking algorithms (inherited from nuclear and plasma physics) are now the norm. Pre-computed lighting and multipass techniques are now also a thing of the past, along with the methods that were used for modeling the voxel buffers and the motion blur techniques of the time. A set of ad-hoc models for light propagation have made way for rigorous implementations of physically accurate light transport, and in the process a number of "holy grail" goals have been achieved: unbiased volume rendering, along with true area light sources, multiple scattering and accurate motion blur. Effects that were unthinkable only a half decade ago are now available in off-the-shelf renderers.

Nonetheless, volume rendering still provides its own set of unique challenges that can be daunting to path tracer developers and researchers. Volumes are computationally expensive to render, far more so than any other type of geometry. Yet the choice of integration technique to solve this expensive problem is not at all straightforward. Faster methods such as the use of tracking are unbiased and have low overhead, but suffer from high variance. Slower methods may be biased or require high setup costs, but have lower variance or may even be noise free. It may even be the case that no single approach is best for a single ray traversing the scene! Meanwhile, the sheer amount of data associated with large volumes means that traditional techniques for motion blur cannot be used. To add to the complications, production scenes vary tremendously in scale and complexity: complicated explosions may be combined with simple atmospherics in a single scene, and it is important to render each element as efficiently as possible.

This course picks up where the last ones left off, explaining not only the basic theory underpinning these recent advances, but also presenting a detailed and concrete look at their implementations. The authors bring the experience of producing two commercially available (RenderMan and Arnold) and one state-of-the-art proprietary renderer (Hyperion), and many of the key features of these are explored and described. The course will focus on the fundamentals of adding volume rendering to an existing path tracer, including all the practical aspects necessary for the demands of feature film work. The basic theory of volume rendering will be presented, which will lead to a comprehensive overview of different volume integration scenarios, along with the merits and weaknesses of techniques that are useful for dealing with each scenario. Recent advances for the optimization of motion blur, variance reduction, tracking, and dealing with scene complexity will also be presented.

Table 1. Nomenclature

Symbol	Description
$\mathbf{x}$	Position
t	Ray parameter
ω	Ray direction
$\mathbf{x}_t$	Parameterized position along a ray: $\mathbf{x}_t = \mathbf{x} + t\omega$
$\sigma_a(\mathbf{x})$	Absorption coefficient
$\sigma_s(\mathbf{x})$	Scattering coefficient
$\sigma_t(\mathbf{x})$	Extinction coefficient: $= \sigma_a(\mathbf{x}) + \sigma_s(\mathbf{x})$
$\alpha(\mathbf{x})$	Single scattering albedo: $= \sigma_s(\mathbf{x})/\sigma_t(\mathbf{x})$
$f_p(\mathbf{x}, \omega, \omega')$	Phase function
d	Ray length/domain of volume integration: $0 < t < d$
ξ, ζ	Random numbers
$L(\mathbf{x}, \omega)$	Radiance at $\mathbf{x}$ in direction ω
$L_d(\mathbf{x}_d, \omega)$	Incident boundary radiance at end of ray
$L_e(\mathbf{x}, \omega)$	Emitted radiance
$L_s(\mathbf{x}, \omega)$	In-scattered radiance at $\mathbf{x}$ from direction ω

2 VOLUME RENDERING THEORY

2.1 Properties of Volumes

Volumes, in the sense we are describing them for our purposes, are collections of particles, ranging from atoms and molecules to any particle size, including stars for galactic radiance transport for example. The average density of the particles needs to be relatively low so that the size of the particles is negligible compared to the average distance between particles. This is a prerequisite for statistically independent collisions and is usually the case in any gaseous media, but not for dense granular media such as sand and snow where these assumptions break down (Meng et al. 2015).

As a photon travels through a volume, it may collide with the particles making up the volume. These collisions define the radiance distribution throughout a volume. Because it is not feasible to model each and every particle in a volume, they are treated as collision probability fields; essentially, particle collisions are stochastically instantiated.

The chance of a photon collision is defined by a coefficient $\sigma(\mathbf{x})$, which is the probability density of collision per unit distance traveled inside the volume. The physical unit of a collision coefficient is inverse length. Another way to represent the coefficient is by its corresponding mean free path, which is the average distance traveled between collisions. In general, we consider only the case where the coefficients are a function of position and make them spectrally varying where appropriate.

The properties of volumes are defined by absorption and scattering coefficients, together with the phase function and emission:

Absorption. Absorption, expressed with the coefficient $\sigma_a(\mathbf{x})$, describes the collision when a photon is absorbed by the volume, simply disappearing from the domain we are interested in. Physically, the photon energy is transferred into some volume particle internal energy such as translational or vibrational modes in molecules, or more generally into heat.

Scattering. A scattering collision $\sigma_s(\mathbf{x})$ scatters the photon into a different direction defined by the phase function. The radiance value carried by the photon does not change with a scattering collision; any change in radiance is modeled with absorption and emission.

Phase Function. The phase function $f_p(\mathbf{x}, \omega, \omega')$ is the angular distribution of radiance scattered and is usually modeled as a 1D function of the angle θ between the two directions ω and ω'. Phase functions need to be normalized over the sphere:

$$\int_{S^2} f_p(\mathbf{x}, \omega, \omega')d\theta = 1 \tag{1}$$

otherwise phase functions would add or subtract radiance to a scattering collision. An important property of phase functions — similar to BSDFs — is reciprocity. We need to be able to interchange the directions without changing the value of the phase function. In a sense, the phase function has the same functionality as the BSDF in surface rendering and can be abstracted as a BSDF as described in section 3.6. Volumes that are isotropic have an equal probability of scattering incoming light in any direction, and have an associated phase function:

$$f_p(\mathbf{x}, \theta) = \frac{1}{4\pi} \tag{2}$$

Anisotropic volumes can exhibit complicated phase functions which can be accurately modeled by using the Mie solution to Maxwell's equations (Mie scattering), or by using the Rayleigh approximation. As an alternative to these expensive functions, in production volume rendering, the most widely used phase function is the Henyey-Greenstein phase function (Henyey and Greenstein 1941):

$$f_p(\mathbf{x}, \theta) = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g^2 - 2g \cos\theta)^{\frac{3}{2}}} \tag{3}$$

where $-1 < g < 1$. The single parameter g can be understood as the average cosine of the scattering directions, and controls the asymmetry of the phase function — it can model backwards scattering ($g < 0$), isotropic scattering ($g = 0$), and forward scattering ($g > 0$). Multiple lobes of this phase function can be combined to approximate more complicated phase functions. The Henyey-Greenstein phase function has the property that it can be easily perfectly importance sampled (Pharr and Humphreys 2010).

Emission. Volumes emit radiance in a volumetric fashion, but otherwise behave exactly like any other light source. The emitted radiance is expressed as a radiance field $L_e(\mathbf{x}, \omega)$ whose output gets absorbed and scattered like any non-volume light source. If a volume does not emit, $L_e(\mathbf{x}, \omega)$ is simply zero. In any real-world scenario, volumes do not emit directionally, and $L_e(\mathbf{x}, \omega)$ is the same in all directions w.

2.1.1 Extinction and Single Scattering Albedo Parametrization. In many cases, it is desirable to choose a different parametrization than absorption and scattering coefficients. We can define the extinction coefficient $\sigma_t = \sigma_a + \sigma_s$ (sometimes called the attenuation coefficient, and often informally referred to as the density) as the sum of the absorption and scattering coefficients. Extinction defines the net loss of radiance due to both absorption and scattering, and we can model that loss with a single coefficient. We can interpret an extinction collision as a collision where both scattering and absorption are involved and we need to make sure that the scattered radiance is correctly modulated in the integral formulations with the *single scattering albedo.*

The single scattering albedo $\alpha = \sigma_s/\sigma_t$ is defined in addition to the extinction to unambiguously define volume properties. α has a similar meaning to the surface albedo: both are a measure of overall reflectivity that determines the amount of scattered radiance. An $\alpha = 0$ means that all radiance is absorbed (such as in black coal dust) and an $\alpha = 1$ means that no absorption occurs and we have lossless scattering (such as in clouds).

In more practical terms, extinction and single scattering albedo provide a good parametrization to represent volumes since the albedo usually stays constant for a single type of volume while the extinction is driven by a density field represented by voxels or a procedural function. The parametrization can be changed at any time to the needs of computation or data storage.

2.2 Light Propagation in Volumes

2.2.1 Radiative Transfer Equation. The distribution of radiance in volumes is defined by the *radiative transfer equation* (Chandrasekhar 1950), or in short the RTE. It describes the equilibrium radiance field $L(\mathbf{x}, \omega)$ parametrized by position $\mathbf{x}$ and the direction ω for the given parameters of a volume, including any boundaries defined by geometry and light sources.

In the following, we describe each of the constituent terms of the RTE to arrive at its full formulation:

Absorption. For an exemplary radiance beam $L(\mathbf{x}, \omega)$, starting at $\mathbf{x}$ with direction ω, the derivative of the radiance in the direction of ω (expressed as $(\omega \cdot \nabla)$) is proportional to the radiance at that point. The proportionality factor is the previously introduced absorption coefficient σ_a:

$$(\omega \cdot \nabla)L = -\sigma_a(\mathbf{x})L(\mathbf{x}, \omega) \tag{4}$$

This is the three-dimensional formulation of the Beer-Lambert Law as a differential equation describing the loss of radiance due to absorption.

Out-Scattering. Our exemplary radiance beam $L(\mathbf{x}, \omega)$ also loses radiance due to out-scattering into other directions than its own direction ω. The radiance is not lost to the overall radiance field, but to the specific direction of the exemplary beam and contributes to other positions/directions in the $L(\mathbf{x}, \omega)$ field. The loss is again proportional to the radiance $L(\mathbf{x}, \omega)$ with the proportionality factor being the scattering coefficient σ_s:

$$(\omega \cdot \nabla)L(\mathbf{x}, \omega) = -\sigma_s(x)L(\mathbf{x}, \omega) \tag{5}$$

Emission. Emission is a separate radiance field $L_e(\mathbf{x}, \omega)$ that defines the radiance added to the radiance field $L(\mathbf{x}, \omega)$:

$$(\omega \cdot \nabla)L(\mathbf{x}, \omega) = \sigma_a(x)L_e(\mathbf{x}, \omega) \tag{6}$$

Please note that PBRT (Pharr and Humphreys 2010) as well as many other publications treat emission slightly different by defining a radiance or source term without the absorption coefficient $\sigma_a(x)$. A fully consistent description of the radiative transfer needs to contain the coefficient to correctly set up the emissive radiance $L_e(\mathbf{x}, \omega)$ in relation to absorption, which is important for the following derivations.

In-Scattering. In-scattering is a contribution from the out-scattering of all of the other directions ω' at $\mathbf{x}$, increasing the net radiance of the exemplary radiance beam:

$$(\omega \cdot \nabla)L(\mathbf{x}, \omega) = \sigma_s(\mathbf{x}) \int_{S^2} f_p(\mathbf{x}, \omega, \omega')L(\mathbf{x}, \omega')d\omega' \tag{7}$$

where S^2 denotes the spherical domain around the position $\mathbf{x}$. The σ_s in front of the integral models the scattering of the incoming radiance from all of the directions similar to the σ_s in equation 5. The radiance beam essentially picks up all of the radiance scattered into its own direction from all other directions.

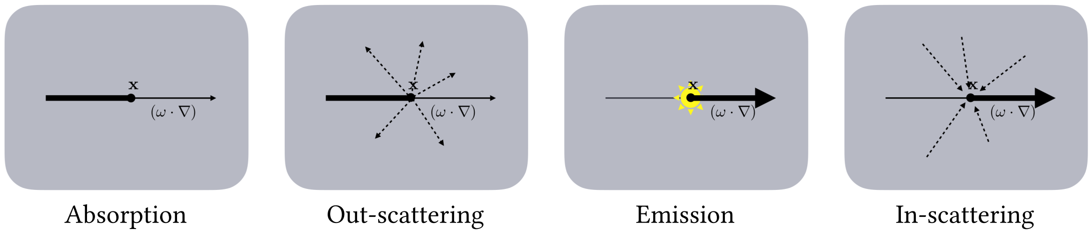


Fig. 2. The four constituent terms visualized

Assembling the RTE. For the final form of the RTE, we add up all constituent terms. Due to their fundamental similarity, we combine absorption and out-scattering into the previously discussed extinction coefficient $\sigma_t(\mathbf{x}) = \sigma_a(\mathbf{x}) + \sigma_s(\mathbf{x})$:

$$(\omega \cdot \nabla)L(\mathbf{x}, \omega) = -\sigma_t(\mathbf{x})L(\mathbf{x}, \omega) + \sigma_a(\mathbf{x})L_e(\mathbf{x}, \omega) + \sigma_s(\mathbf{x}) \int_{S^2} f_p(\mathbf{x}, \omega, \omega')L(\mathbf{x}, \omega')d\omega' \tag{8}$$

2.2.2 Volume Rendering Equation. The RTE formulates the radiance distribution in a forward-transport fashion using gradients $(\omega \cdot \nabla)$, defining what happens to a radiance beam as it travels forward. The RTE can be directly used in finite element methods such as radiosity (Cohen et al. 1993) and more generally in finite element methods, but not in a path tracing setting. The volume rendering equation can provide this formulation. To simplify the notation, we shorten the in-scattering with

$$L_s(\mathbf{x}, \omega) = \int_{S^2} f_p(\mathbf{x}, \omega, \omega')L(\mathbf{x}, \omega')d\omega'. \tag{9}$$

We can formally integrate both sides of the RTE, turning the gradient $(\omega \cdot \nabla)$ on the left side of Equation 8 into an integral on the right side, giving an explicit equation for $L(\mathbf{x}, \omega)$ and arriving at the volume rendering equation (VRE):

$$L(\mathbf{x}, \omega) = \int_{t=0}^{d} \exp\left(-\int_{s=0}^{t} \sigma_t(\mathbf{x}_s)ds\right)\Big[\sigma_a(\mathbf{x}_t)L_e(\mathbf{x}_t, \omega) + \sigma_s(\mathbf{x}_t)L_s(\mathbf{x}_t, \omega) + L_d(\mathbf{x}_d, \omega)\Big]dt \tag{10}$$

In order to express this formal solution, we parametrize the integrals along the negative part of the radiance beam, with the running ray parameters t and s, and where the ray ends at a boundary (surface or potentially other volume) modeled with the ray parameter d. The positions for the integrals over ds and dt in Equation 10 are $\mathbf{x}_t = \mathbf{x} - t\omega$ and $\mathbf{x}_s = \mathbf{x} - s\omega$. The incident radiance at the end of the ray is modeled with $L_d(\mathbf{x}_d, \omega)$. Choosing this parametrization along the negative part of a radiance beam follows naturally from the formal integration. Instead of describing where the radiance is traveling *to* with a forward-transport formulation as done by the RTE with *gradients*, the VRE describes the radiance in terms of where it comes *from*, looking backwards and accumulating all of the contributions with *integrals*. With that, it is also ensured that the direction ω is always pointing in the direction of radiance flow. Many descriptions such as PBRT (Pharr and Humphreys 2010) need to introduce switched directions and need to keep track of the direction using arrows in the notation which adds unnecessary complexity to the mathematical formulation.

For better structure, we shorten the exponential integral term with

$$T(t) = \exp\left(-\int_{s=0}^{t} \sigma_t(\mathbf{x}_s)ds\right) \tag{11}$$

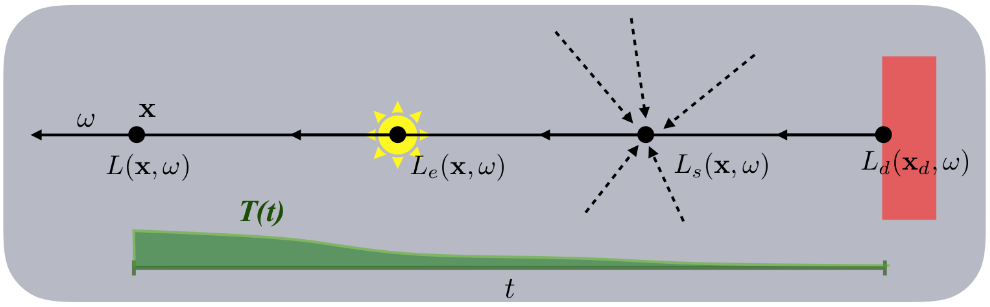


Fig. 3. The volume rendering equation (VRE) visualized.

which is the transmittance from the origin $\mathbf{x}$ of the ray to the position $\mathbf{x}_t = \mathbf{x} - t\omega$ on the ray parametrized by t. The transmittance $T(t)$ is the net reduction factor from absorption and out-scattering between $\mathbf{x}$ and $\mathbf{x}_t$, combining these constituents. Additionally, we can execute the integral for the boundary term since the term is not dependent on t, arriving at a more compact form of the VRE that fits our needs in path tracing:

$$L(\mathbf{x},\omega) = \int_{t=0}^{d} T(t)\Big[\sigma_a(\mathbf{x})L_e(\mathbf{x}_t,\omega) + \sigma_s(\mathbf{x})L_s(\mathbf{x}_t,\omega)\Big]dt + T(d)L_d(\mathbf{x}_d,\omega) \tag{12}$$

Any radiance picked up at a point on the ray needs to be extinguished by the transmittance up to that point to calculate its final contribution to $L(\mathbf{x},\omega)$. Figure 3 shows a visualization of the VRE corresponding to the contributions in Equation 12. The volume rendering equation can be interpreted as a generalization of the rendering equation (Kajiya 1986) to include volumetric structures while surfaces are nearly infinitely dense volumes with complex phase functions.

There are two principal stochastic approaches in Monte Carlo that can be used to solve the RTE. In the following section, we derive the specialized versions of the volume rendering equations for each of the approaches.

2.3 Tracking Approach

Tracking approaches employ Russian roulette and rejection sampling strategies to decide on a single type of collision being sampled instead of trying to estimate all of them at the same time. Tracking simulates how a photon (in the computer graphics sense) bounces around inside a volume, explicitly modeling absorption and scattering collisions. In a numerical sense, we do not allow the radiance beam to split itself up into fractional contributions, we only choose which collision type should be modeled.

Under those assumptions, the problem that arises is how to sample the free-path distances that are traveled by a photon between different types of collisions in order to model the physical process.

2.3.1 Closed-form Tracking. In simple volumes, e.g. those with constant, polynomial, or exponentially varying extinction, free paths can be sampled using inverse transform sampling, first applied to Monte Carlo integration by Ulam (Eckhardt 1987). In production rendering, the homogeneous volume is the most important one which we will discuss in the following derivation.

For sampling purposes, we can define a probability density function by normalizing the transmittance function in equation 11. If the corresponding cumulative distribution function (CDF) is analytically invertible, then the free-path distance t' can be sampled analytically using a single random number ξ.

For the homogeneous volume where σ_t is not spatially varying, the transmittance becomes

$$T(t) = \exp(-\sigma_t t) \tag{13}$$

which is the Beer-Lambert law: exponential extinction of radiance. For tracking, we are interested in creating a transmittance estimate that can double as a free-flight estimator; we want to sample and generate a distance that a photon travels in a volume defined by a constant transmittance. The problem breaks down to importance sampling an exponential function to produce a free-path distance t'. The PDF to do so is defined by normalizing the integral of the exponential function of the transmittance:

$$p(t) = \sigma_t \exp(-\sigma_t t). \tag{14}$$

We can perfectly importance sample the PDF with (Pharr and Humphreys 2010):

$$t' = -\ln(1-\xi)/\sigma_t \tag{15}$$

There is no explicit Monte Carlo weight involved since the underlying exponential PDF is perfectly importance sampled (producing a weight of 1) and we can interpret the t' directly as the distance a photon travels. With the PDF in (14), we have a stochastic substitution for the transmittance (Equation 11) in the VRE (Equation 10). Substituting Equation 14 into the simplified VRE for homogeneous volumes (i.e. with constant coefficients) is then

$$L(\mathbf{x},\omega) = \int_{t=0}^{d} p(t)\left[\frac{\sigma_a}{\sigma_t}L_e(\mathbf{x}_t,\omega) + \frac{\sigma_s}{\sigma_t}L_s(\mathbf{x}_t,\omega)\right]dt. \tag{16}$$

We need to divide the other terms by σ_t to account for the normalization factor in Equation 14. We drop the boundary term $L_d(\mathbf{x}_d,\omega)$ since it is cumbersome to express in math notation and will add it back at the end of the section.

The second part of the VRE which contains the radiance contributions is estimated by a Russian roulette that determines which type of collision is sampled: either absorption/emission or in-scattering. The stochastic estimate of the transmittance already brought the other terms into the right form, and we can interpret them as probabilities:

$$P_a = \frac{\sigma_a}{\sigma_t}, \qquad P_s(\mathbf{x}) = \frac{\sigma_s}{\sigma_t} \tag{17}$$

where $P_a(\mathbf{x}) + P_s(\mathbf{x}) = 1$ and like before $\sigma_t = \sigma_a + \sigma_s$. The probabilities are the ratios of each collision type to the sum of all collision types. A Russian roulette chooses which type is modeled for each instance. To arrive at a recursive formulation for Equation 16, recursing on a new estimate every time a scattering event is chosen, we define $\mathbf{x}_0 \equiv \mathbf{x}$ and $x_{j+1} = x_j - t_j\omega_j$

$$L(\mathbf{x}_j,\omega_j) = \int_{t=0}^{\infty} p(t_j)\Big[P_a L_e(\mathbf{x}_{j+1},\omega_j) + P_s L_s(\mathbf{x}_{j+1},\omega_j)\Big]dt. \tag{18}$$

To re-introduce boundaries into the approach, we need to split off the tail of the integral (d,∞):

$$L(\mathbf{x}_j,\omega_j) = \int_{t=0}^{d} p(t_j)\Big[P_a L_e(\mathbf{x}_{j+1},\omega_j) + P_s L_s(\mathbf{x}_{j+1},\omega_j)\Big]dt + L_d(\mathbf{x}_{d,j},\omega_j)\int_{t=d}^{\infty} p(t_j)dt \tag{19}$$

The second integrand is not dependent on t, there is only a single contribution at the boundary $\mathbf{x}_{d,j}$. This translates algorithmically into a simple test that if we hit a boundary before any of the collision types are sampled ($t > d$), we return the boundary radiance $L_d(\mathbf{x}_d,\omega_j)$.

Translating Equation 19 into pseudo code formulating the closed-form tracking is given in Algorithm 1. Figure 4 shows some example paths generated by tracking.

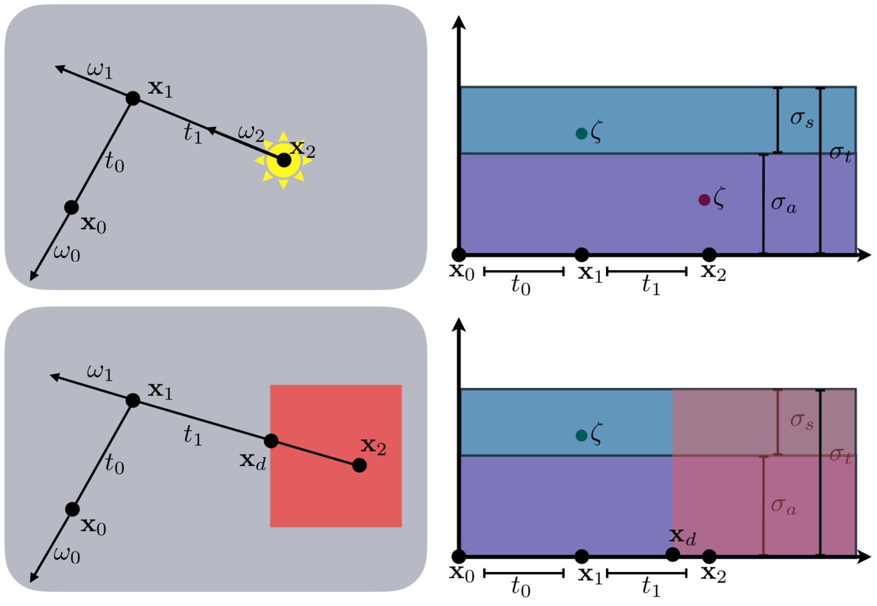


Fig. 4. A tracking path scattering once and absorbing, picking up some emission (top) and a path scattering before hitting a boundary (bottom).

Algorithm 1: *Closed-form tracking. We assume that the phase function $f_p(\omega)$ is perfectly importance sampled.*

```
function ClosedFormTracking($\mathbf{x}, \omega, d$)
    while true do
        $t \leftarrow \frac{\ln(1-\zeta)}{\sigma_t}$        // Set new $t$ for the current $\mathbf{x}$
        if $t > d$ then        // A boundary has been hit
            return $L_d(\mathbf{x}_d, \omega)$
        $\mathbf{x} \leftarrow \mathbf{x} - t \times \omega$
        if $\xi < \frac{\sigma_a}{\sigma_t}$ then        // An absorption/emission collision has occured
            return $L_e(\mathbf{x}, \omega)$
        else        // A scattering collision has occured
            $\omega \leftarrow$ sample $\propto f_p(\omega)$
            $d \leftarrow$ new ray end in $\omega$
```

Though simple, this method is highly relevant for production rendering — atmosphere and fog are almost always modeled using homogeneous volumes. For homogeneous volumes bounded by geometry and with a translucent surface shader, closed-form tracking can be used for path-traced subsurface scattering to render skin or any other subsurface scattering.

2.3.2 Regular Tracking. In piecewise constant volumes, we can apply closed-form tracking to each of the piecewise constant domains, simply traversing into the next domain if we do not scatter. We can interpret the surface radiance $L_d(\mathbf{x}_d, j, \omega_j)$ as coming not from a hit surface, but rather a different

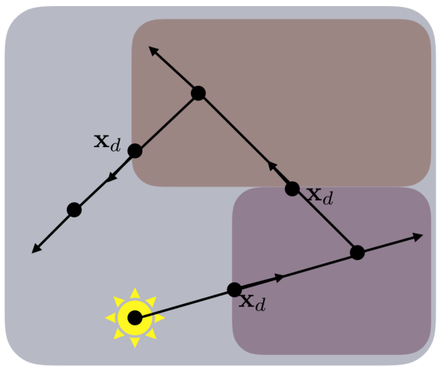


Fig. 5. Regular tracking applies closed-form tracking in piecewise constant domains, restarting the tracking at each boundary if the boundary is hit.

volume. In order to be efficient, this approach needs large parts of the volume to be homogeneous to be efficient.

2.3.3 Delta Tracking. Delta tracking is based on von Neumann's (1951) rejection sampling. It was independently developed in neutron transport (Bertini 1963; Woodcock et al. 1965; Zerby et al. 1961) and in plasma physics (Skullerud 1968), and is known also as Woodcock tracking, the null-collision algorithm, or pseudo scattering.

The key to sampling free-path distances in heterogeneous volumes is to introduce a fictitious collision type that homogenizes the total collision density in such a way that the sampling strategy as used in closed form tracking sees a homogeneous (constant) volume. In this new type of collision, called null-collision, the volume scatters in the same direction as the incoming direction, having no net effect on the light transport itself. We express this collision type with the null-collision coefficient $\sigma_n(\mathbf{x})$ which acts in the same way as the physical coefficients. The physical collision coefficients are now spatially variant, as is the null-collision coefficient $\sigma_n(\mathbf{x})$. To homogenize the overall volume, we choose $\sigma_n(\mathbf{x})$ in such a way that the sum of all coefficients, the *free-path coefficient* $\bar{\sigma}$, becomes constant:

$$\bar{\sigma} = \sigma_a(\mathbf{x}) + \sigma_s(\mathbf{x}) + \sigma_n(x) = \sigma_t(\mathbf{x}) + \sigma_n(x) \tag{20}$$

A consequence of the need to be constant is that $\bar{\sigma}$ is equal or greater to the maximum of $\sigma_t(\mathbf{x})$ (sometimes formulated as being a *majorant* of $\sigma_t(\mathbf{x})$):

$$\bar{\sigma} \geq \sigma_t(\mathbf{x}) \tag{21}$$

and we can easily calculate $\sigma_n(x)$ from

$$\sigma_n(x) = \bar{\sigma} - \sigma_t(\mathbf{x}). \tag{22}$$

Because $\bar{\sigma}$ is constant, it can take on the role of the constant extinction σ_t in Equation 14 and we can draw a distance sample in the very same way as in the closed-form tracking:

$$p_n(t) = \bar{\sigma} \exp(-\bar{\sigma} t). \tag{23}$$

We now have three collision types instead of two, resulting in the definition of a triple of probabilities:

$$P_a(\mathbf{x}) = \frac{\sigma_a(\mathbf{x})}{\bar{\sigma}}, \qquad P_s(\mathbf{x}) = \frac{\sigma_s(\mathbf{x})}{\bar{\sigma}}, \qquad P_n(\mathbf{x}) = \frac{\sigma_n(\mathbf{x})}{\bar{\sigma}} \tag{24}$$

that includes the null-collisions in the form of a collision probability with $P_a(\mathbf{x}) + P_s(\mathbf{x}) + P_n(\mathbf{x}) = 1$.

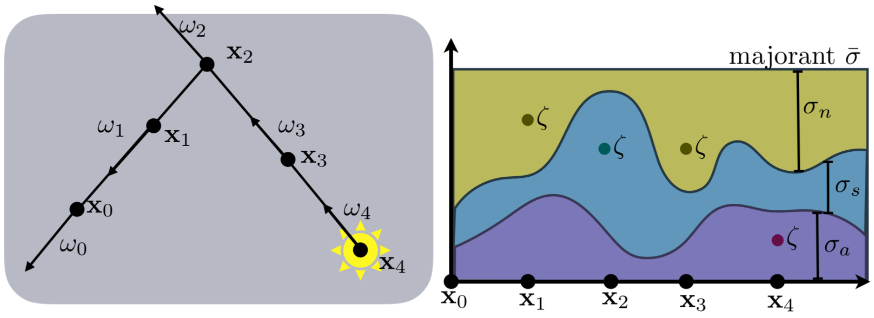


Fig. 6. A Delta-tracked path. If a null-collision event is chosen by the Russian roulette ($\mathbf{x}_1$ and x_3), the path is continued unaltered.

Applying the additional null-collision probability to the VRE gives the recursive form similar to Equation 18:

$$L(\mathbf{x}_j, \omega_j) = \int_{t=0}^{\infty} p_n(t_j)\Big[P_a(\mathbf{x})L_e(\mathbf{x}_{j+1}, \omega_j) + P_s(\mathbf{x})L_s(\mathbf{x}_{j+1}, \omega_j) + P_n(\mathbf{x})L(\mathbf{x}_{j+1}, \omega_j)\Big]dt. \tag{25}$$

The recursive evaluation as an algorithm of Equation 25 is very similar to the closed-form version of tracking. The Russian roulette at the position t now chooses between three instead of two collision types. If the null-collision type is chosen, the recursion simply goes into the next iteration without altering the ray beam. Figure 6 shows a visualization of an example Delta-tracked path. Algorithm 2 shows the addition of the null-collision to the tracking process. Since we treat a null-collision like any other collision, but continue in the same direction from the collision point $\mathbf{x}$ (set in line 6), we can calculate the updated d from the previous d instead of tracing an intersection ray like in the scattering collision.

Algorithm 2: *Delta tracking. We assume that the phase function $f_p(\omega)$ is perfectly importance sampled.*

```
1  function DeltaTracking($\mathbf{x}, \omega, d$)
2      while true do
3          $t \leftarrow -\frac{\ln(1-\zeta)}{\bar{\sigma}}$        // Set new t for the current x
4          if $t > d$ then        // A boundary has been hit
5              return $L_d(\mathbf{x}_d, \omega)$
6          $\mathbf{x} \leftarrow \mathbf{x} - t \times \omega$
7          if $\xi < \frac{\sigma_a(\mathbf{x})}{\bar{\sigma}}$ then        // An absorption/emission collision has occured
8              return $L_e(\mathbf{x}, \omega)$
9          else if $\xi < 1 - \frac{\sigma_n(\mathbf{x})}{\bar{\sigma}}$ then        // A scattering collision has occured
10             $\omega \leftarrow$ sample $\propto f_p(\omega)$
11             $d \leftarrow$ new ray end in $\omega$
12         else        // A null collision has occured
13             $d \leftarrow d - t$        // Update d in relation to the collision x, $\omega$ does not change
```

To work efficiently, we need $\bar{\sigma}$ to be as close to the maximum of σ_t as possible. While it is valid to choose a very large $\bar{\sigma}$, it will mean that we mostly do null-collisions, stopping just to continue again.

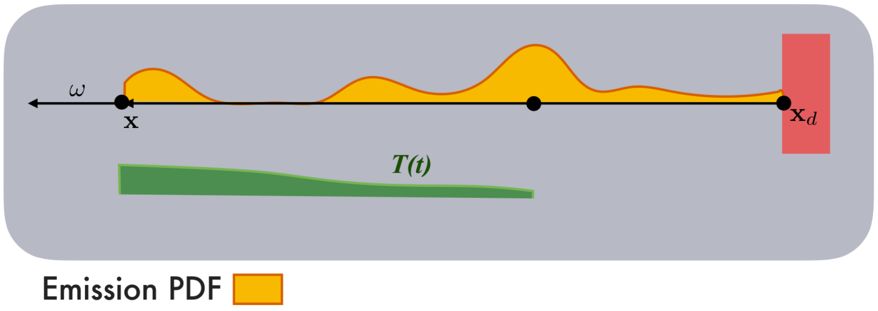


Fig. 7. Emission sampling in the PDF approach. A PDF is built, a sample is drawn and the result is extincted up to the sampled point.

The general solution is to subdivide volumes using an acceleration structure into domains where a tight $\bar{\sigma}$ can be delivered to the Delta tracking.

Delta Tracking in Extinction and Albedo Parametrization. Algorithm 2 can also be formulated by aggregating the probabilities $P_a(\mathbf{x})$ and $P_s(\mathbf{x})$ into a single probability $P_t = \frac{\sigma_t}{\bar{\sigma}}$, followed by a Russian roulette based on the single scattering albedo α. The drawback of this formulation is that emission is not easily integrated since absorption and emission is not separately modeled from scattering.

2.3.4 Advanced Tracking Methods. The restriction on $\bar{\sigma}$ needing to be a majorant can be remedied by applying weighted tracking (Carter et al. 1972; Cramer 1978). Recently Kutz et al. (2017) introduced trackers that can incorporate spectral data and minimizes lookups to the coefficients in addition to removing the majorant restriction. We are not principally restricted to the probabilities shown in equation 17; they can be manipulated to overcome the restrictions of Delta tracking. (Kutz et al. 2017) also contains a formal generalization of the null-collision principles, the *null-collision radiative transfer equation* which is compatible with the discussion in this section.

2.4 PDF Approach

In this approach, we allow the radiance beam or photon to split into fractions instead of doing Russian roulette as in the tracking methods. This has far-reaching consequences and creates both advantages and disadvantages in comparison to Delta tracking. We can separate equation 10 so that each of the terms is in a separate integral. The fundamental advantage of doing so is that the estimates can be driven by PDFs that are independently defined, instead of doing a combined Russian roulette between them like in the tracking approach, resulting in the separated VRE:

$$L(\mathbf{x}, \omega) = \int_{t=0}^{d} T(t)\sigma_a(\mathbf{x}_t)L_e(\mathbf{x}_t, \omega)dt + \int_{t=0}^{d} T(t)\sigma_s(\mathbf{x}_t)L_s(\mathbf{x}_t, \omega)dt + T(d)L_d(\mathbf{x}_d, \omega) \tag{26}$$

Since $L_d(\mathbf{x}_d, \omega)$ is not dependent on t, we already resolved its integral as defined in Equation 12.

Since we now have completely different integrals that can be added together, we can apply different sampling strategies to each of them. There is full freedom in choosing the sampling strategy, giving a lot of possibilities in how to estimate for each integral. Usually an estimate for the emission $L_e(\mathbf{x}_t, \omega)$ and in-scattering $L_s(\mathbf{x}_t, \omega)$ is generated by building a PDF which is then importance sampled to choose a point on the ray to sample emission or in-scattering. This process is visualized in Figure 7. The in-scattering term is sampled in a similar fashion: first a PDF is built, and a sample is drawn. The in-scattered radiance

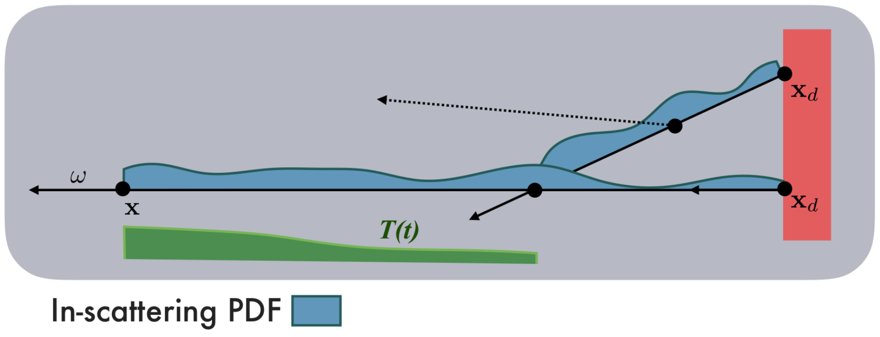


Fig. 8. In-scattering sampling in the PDF approach. A PDF is built, a sample is drawn and its in-scattering is sampled recursively. The sample needs to be extinguished up to the sample point according to the transmittance.

is sampled by creating a new ray which recursively needs to deliver an approximation of its radiance with a transmittance approximation and its own in-scattering approximation. Like in the emission case, the result needs to be extinguished up to the sample point (see Figure 8). We can build completely different PDFs for each term, optimizing for the underlying data in whichever way we deem necessary.

To correctly extinguish the contributions to $L(\mathbf{x}, \omega)$, we need an estimator for the transmittance $T(t)$ for all of the terms in Equation 26 in addition to the estimators for the radiances. The transmittance estimator needs to be evaluated for each sampling position, but as we will late see we can cache the transmittance estimate to avoid reevaluating the transmittance.

The inherent challenge to this approach,as well as opportunity to optimize, is to efficiently build the PDFs as accurately as possible to get efficient sampling. This PDF building is time consuming and usually several samples are drawn from the same PDF to amortize the building costs.

2.4.1 Transmittance Estimators.

Ray Marching Transmittance Estimator. The transmittance can be estimated with quadrature instead of Monte Carlo, essentially marching along the ray (Perlin and Hoffert 1989) and accumulating the transmittance. This introduces bias even if the marching is jittered (Raab et al. 2008), resulting in artifacts if the volume structure is undersampled. In the context of Monte Carlo, the ray marching transmittance estimator is noise free, but very expensive to do on a per ray basis. But a practical advantage of ray marching in the context of Monte Carlo integration is that no additional data is needed: ray marching does not need the free-path coefficient $\bar{\sigma}$ like the unbiased methods. In production, a ray marching transmittance with very small step sizes can be used as a reference to validate more sophisticated implementations.

Delta Tracking Transmittance Estimator. We can use Equation 23 as a transmittance estimator in the PDF approach, testing if a collision occurred before or after t. This estimator produces a transmittance estimate of $T(t) = 1$ or $T(t) = 0$ due to the Russian roulette and is therefore a binary estimator. The binary estimate is too inaccurate to be used efficiently in the PDF approach as it either does not extinguish or discards the whole estimate.

Ratio Tracking Transmittance Estimator. An unbiased transmittance estimator that is related to Delta tracking is introduced to computer graphics by Novák et al. (2014). The Russian roulette of Delta tracking is replaced by a Monte-Carlo weight T that is equal to the probability of a null collision in relation to

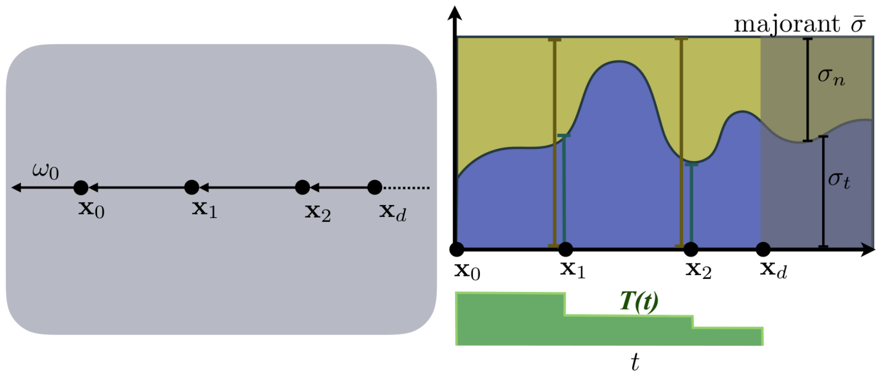


Fig. 9. Ratio Tracking producing and estimate for the transmittance T(t) to a boundary x_d.

extinction at a sampled collision point, resulting in the transmittance estimator

$$T(d) = \prod_{i=1}^{K} \left(1 - \frac{\sigma_t(\mathbf{x}_i)}{\bar{\sigma}}\right) \tag{27}$$

where K are all of the collisions created before reaching the end of the integration d. Like in Delta tracking, the free path coefficient $\bar{\sigma}$ needs to be constant and a majorant of $\sigma_t(\mathbf{x}_i)$. The resulting pseudo code is given in Algorithm 3. In contrast to the pseudo code of delta tracking (Algorithm 2), we do not restart the ray at every collision point, we travel backwards on the initial ray defined by $\mathbf{x}_0, \omega_0$ and update the weight until we hit the boundary. Kutz et al. (2017) discuss the deeper connections between Ratio and Delta trackers. While still called a tracking method, Ratio tracking always produces a transmittance estimate up to the end of the ray. The end of the domain d is either defined by a boundary or by a sampling point drawn from a PDF. Figure 9 shows a visualization of the Ratio tracking process to a boundary.

Algorithm 3: *Ratio Tracking*

```
function RatioTracking($\mathbf{x}_0, \omega_0, d$)
    $T = 1$        // Set Monte-Carlo weight to 1
    while true do
        $t \leftarrow t - \frac{\ln(1-\zeta)}{\bar{\sigma}}$        // Add a step to the total t
        if $t > d$ then        // We crossed a boundary or a sample point
            return $T$
        $\mathbf{x} \leftarrow \mathbf{x}_0 - t \times \omega_0$
        $T = T \times (1 - \frac{\sigma_t(x)}{\bar{\sigma}})$        // Update Monte-Carlo weight
```

Advanced Transmittance Estimators. In addition to Ratio tracking, Novak et al. (2014) introduced a more sophisticated Ratio tracking method, called Residual Ratio tracking that uses a coarse control transmittance to minimize the number of lookups to the volume parameters. It will be discussed in 4.4.1.

2.4.2 Path Directions and Reciprocity. While walking backwards on a ray or path representing $L(\mathbf{x}, \omega)$ is a consistent way to express the theory, adhering to the physical flow direction of radiance, it is inconvenient to always think backwards in an implementation. In Monte-Carlo path tracing, we make heavy use of the reciprocity of light transport. Camera rays transport importance and throughput in order to calculate a Monte-Carlo weight for the constructed path (Veach 1997). Importance and throughput form a dual to flux and radiance, and we can parametrize any path or ray segment for camera or light paths as we see fit as long as the radiance/throughput transport is modeled correctly. So in an implementation, we can simply invert the direction of the ray and path tracing to point in the counter direction of the radiance flow and model the flow of throughput with it.

Any known path tracer implementation always traces all rays and paths (e.g. camera paths, light paths) on the positive part of a ray and compensates by inverting the direction of ω where appropriate.

2.4.3 Comparison of Approaches. Comparing the tracking and PDF approach is a complicated task as the detailed attributes of the choices in each approach have strong influence on the efficiency. Also, the properties of the volume have a strong influence on the speed of an algorithmic configuration. Fortunately, all discussed methods can be mixed and matched to achieve the best performance.

Tracking methods produce fast but low-quality estimates while PDF approaches deliver relatively slow (due to PDF building) but high-quality estimates. Due to the Russian roulette in the tracking approach, its radiance estimates possess less variance (or even no variance) compared to the PDF approach where importance sampling can introduce fireflies due to very high variances in the weights.

In general, tracking methods are better suited for high order scattering such as in clouds and subsurface scattering whereas PDF approaches are better suited for low order scattering where it is important to efficiently find the main radiance contributions which can be done by sophisticated PDF building using all the data available.

2.4.4 Other Volume Rendering Methods. In the context of other rendering algorithms such as photon mapping or bidirectional path tracing, the discussed approaches can be used in order to construct camera and light paths to be further processed. Also, next event estimation is a standard method in volume rendering for light sampling (requiring a transmittance estimate) that should be applied in any volume rendering implementation. The discussed approaches can be adapted to deliver the necessary estimates.

3 ADDING VOLUMES TO A PATH TRACER

As hinted at in the previous section, there is no single preferred integration technique for volumes that works well across every different type of participating media. Simple cases like homogeneous absorption can be implemented by a straight forward beam transmittance calculation added to the throughput calculation of a path tracing algorithm. More complex cases involving light scattering through low albedo media (e.g. smoke) typically require algorithms which only need single scattering, whereas the most complicated types of volumes, such as clouds, may require high amounts of multiple scattering and complicated anisotropic phase functions. Rather than building a number of different volume integration models into a single lighting integrator, it is helpful to decouple volume integration from the general light integration problem: the integration domain can be broken into smaller volumetric subdomains when participating media is involved, and control of integration of those domains given to smaller volume modules.

Building on this, we assume a path tracing system similar to that described in PBRT (Pharr and Humphreys 2010) where shading and lighting have been decoupled. For every geometric primitive in the scene, there is an instance of a `Material` class bound to that primitive. When a ray hits a geometric primitive, the geometric properties of geometry at the hit point are encapsulated into a `ShadingContext`, which includes the position `P` of the hit point itself, the surface normal `N`, and the direction opposite to the incoming ray `V`.

Every `Material` implements a `CreateBSDF` method that returns a `BSDF` object, given a `ShadingContext`. The `BSDF` object implements both a `EvaluateSample` method and a `GenerateSample` method, which may use information from the `ShadingContext` (such as the geometric normal) to decide how to do their work. `EvaluateSample` is used to evaluate the response of the `BSDF` to a light sample, given an incoming light ray with direction `sampleDirection` and the outgoing ray direction `ShadingContext::GetV()`. `BSDF::GenerateSample` is used to sample the `BSDF`, and generates an outgoing ray direction `sampleDirection` based on evaluating a BRDF or a BTDF, as well as the associated PDF of that ray direction. If the `BSDF` object implements both a BRDF and a BTDF, it is responsible for randomly choosing which of the two to use.

```
class Material {
public:
    virtual BSDF *CreateBSDF(ShadingContext &ctx) = 0;
};

class BSDF {
public:
    BSDF(ShadingContext &ctx) : m_ctx(ctx) {}
    virtual void EvaluateSample(RendererServices &rs, const Vector &sampleDirection, Color
        &L, float &pdf) = 0;
    virtual void GenerateSample(RendererServices &rs, Vector &sampleDirection, Color &L,
        float &pdf) = 0;
protected:
    ShadingContext &m_ctx;
};

class ShadingContext {
public:
    Point GetP() const;
    Normal GetN() const;
    Vector GetV() const;
    Geometry &GetGeometry();
    void SetP(Point &P);
    void RecomputeInputs();
    float GetFloatProperty(int property) const;
    Color GetColorProperty(int property) const;
protected:
    Geometry m_geometry;
};
```

We assume the light integrator module is responsible for implementing a path tracing algorithm with the aid of a `RendererServices` object, which provides services for tracing a ray against the scene database, for sampling lights in the scene, and for creating a `ShadingContext` given a hit point on a piece of geometry.

```
class RendererServices {
public:
    float GenerateRandomNumber();
    float MISWeight(int nsamps1, float pdf1, int nsamps2, float pdf2);
    Point GetNearestHit(const Ray &ray, Geometry &g);
    ShadingContext CreateShadingContext(const Point &p, const Ray &wi, Geometry &g);
    void GenerateLightSample(const ShadingContext &ctx, Vector &sampleDirection, Color &L,
        float &pdf, Color &beamTransmittance);
    void EvaluateLightSample(const ShadingContext &ctx, const Vector &sampleDirection,
        Color &L, float &pdf, Color &beamTransmittance);
    void EvaluateLightSample(const ShadingContext &ctx, const Vector &sampleDirection,
        Color &L, float &pdf);
};
```

3.1 Volume integrator execution model

To begin integrating volumes into our path tracer, we extend our `Material` class to return a `Volume` object.

```
class Material {
public:
    virtual BSDF *CreateBSDF(ShadingContext &ctx) = 0;
    virtual bool HasVolume() = 0;
    virtual Volume *CreateVolume(ShadingContext &ctx) = 0;
};
```

Now we can describe the interface of a volume module - a `Volume` object:

Listing 1. The interface for volume integrators

```
class Volume {
```

```
public:
    Volume(ShadingContext &ctx) : m_ctx(ctx) {}
    virtual ~Volume() {}
    virtual bool Integrate(RendererServices &rs, const Ray &wi, Color &L, Color
         &transmittance, Color &weight, Point &P, Ray &wo, Geometry &g) = 0;
    virtual Color Transmittance(RendererServices &rs, const Point &P0, const Point &P1) = 0;
protected:
    ShadingContext &m_ctx;
};
```

We will also tie this interface to the compact form of the VRE as shown in equation 12 (repeated here):

$$L(\mathbf{x},\omega) = \int_{t=0}^{d} T(t)\Big[\sigma_a(\mathbf{x})L_e(\mathbf{x}_t,\omega) + \sigma_s(\mathbf{x})L_s(\mathbf{x}_t,\omega)\Big]dt + T(d)L_d(\mathbf{x}_d,\omega)$$

A `Volume` is constructed using a `ShadingContext`, which is initially the geometric environment of the hit point which begins the volume interval. The point returned by `ShadingContext::GetP()` is the position $\mathbf{x}_0$. As we will describe later, this hit point may not actually be from a piece of geometry to which the volume's originating `Material` has been bound.

Volume integration is initiated by the `Integrate` method. The input to the `Integrate` method is the ray with direction ω and origin $\mathbf{x}_0$, denoting the direction of the volume interval (chosen by the `BSDF::GenerateSample` executed at the hit point $\mathbf{x}_0$), and a `RendererServices` object that the `Volume` can use during integration. The outputs include the radiance along the ray `L` and the weight of this radiance estimate; these two outputs are the estimate of the integrand term in the VRE.

The outputs also include the beam `transmittance` over the volume integration interval, and the hit point `P`, incident ray `wo` into the hit point, and geometry corresponding to the end of the volume integration interval. In other words, the volume integrator actually picks the distance d and returns $\mathbf{x}_d$ in the output parameter `P`, and also returns $T(d)$ in the output parameter `transmittance`. The reason for this flexibility is to allow the light integrator to call the `Volume::Integrate` method to generate the next hit, instead of calling `RendererServices::GetNearestHit` with the outgoing ray from a `BSDF`; we will see in section 3.7 how these outputs can facilitate multiple scattering in the volume.

We will discuss how the `Volume::Transmittance` method is used in section 3.4, but for now consider it a method which returns the beam transmittance between two points inside the volume.

A complete but trivial volume implementation looks like this:

Listing 2. A trivial volume

```
class EmptyVolume: public Volume {
public:
    EmptyVolume(ShadingContext &ctx) : Volume(ctx) {}
    virtual bool Integrate(RendererServices &rs, const Ray &wi, Color &L, Color
         &transmittance, Color &weight, Point &P, Ray &wo, Geometry &g) {
        if (!rs.GetNearestHit(Ray(m_ctx.GetP(), wi.dir), P, g))
            // We escaped the volume somehow, not watertight?
            return false;
        L = Color(0.0);
        transmittance = Color(1.0);
        weight = Color(1.0);
        wo = Ray(P, wi.dir);
        return true;
    }
    virtual Color Transmittance(RendererServices &rs, const Point &P0, const Point &P1) {
        return Color(1.0);
    }
};
```

In EmptyVolume, the Integrate method does nothing more than call the RendererServices's GetNearestHit method to generate the next hit point and geometry. This renderer service will trace a ray against the geometry in the scene, and here we assume that when we trace the ray it will return a hit at the next interesting geometry interface; we will discuss this more in the next section. The incident ray into that hit point gets the same direction as the input ray wi passed to the Integrate method. An empty volume emits no radiance ($L_e = L_s = 0$) and has no effect on the transmittance ($T(d) = 1$), so the transmittance field is set to 1.0 in both the Integrate and Transmission methods.

A slightly less trivial volume which implements homogeneous absorption as described by the Beer-Lambert law looks like this:

Listing 3. Homogeneous absorption integrator

```
class BeersLawVolume: public Volume {
public:
    BeersLawVolume(const Color &absorption, ShadingContext &ctx) :
        Volume(ctx), m_absorption(absorption) {}
    virtual bool Integrate(RendererServices &rs, const Ray &wi, Color &L, Color
         &transmittance, Color &weight, Point &P, Ray &wo, Geometry &g) {
        if (!rs.GetNearestHit(Ray(m_ctx.GetP(), wi.dir), P, g))
            return false;
        L = Color(0.0);
        transmittance = Transmittance(rs, P, m_ctx.GetP());
        weight = Color(1.0);
        wo = Ray(P, wi.dir);
        return true;
    }
    virtual Color Transmittance(RendererServices &rs, const Point &P0, const Point &P1) {
        float distance = Vector(P0 - P1).Length();
        return Color(exp(m_absorption.r * -distance), exp(m_absorption.g * -distance),
             exp(m_absorption.b * -distance));
    }
protected:
    const Color m_absorption;
};
```

Similar to EmptyVolume, BeersLawVolume::Integrate calls RendererServices::GetNearestHit to generate the next hit point $\mathbf{x}_d$ and its associated Geometry. Unlike EmptyVolume, the absorption coefficient has an effect on the transmittance, so we measure the distance d between the beginning of the interval and the hit point, and assign $e^{-\sigma_t d}$ (equation 13) to the transmittance $T(d)$. Other than being used to measure the distance to the end of the volume, the hit point $\mathbf{x}_d$ is of no further use to the volume integration itself; it may in fact belong to an entirely different and unrelated piece of geometry. The volume integrator simply returns this hit point (and its associated geometry) back to the light integrator, which is ultimately responsible for treating this information as the next vertex on the integration path and computing L_d.

Bear in mind that in this system, a volume integrator does not exist by itself; it is a component of a `Material` controlling only the volumetric integration inside a piece of geometry. The surface properties of that geometry are represented by the BSDF returned by the `CreateBSDF` method of that `Material`. In order to finish defining a `Material`, we need to define a `TrivialBSDF`, whose `GenerateSample` method used for creating indirect rays simply continues a ray in the outgoing direction with full weight and PDF. The resulting material combining both the `TrivialBSDF` and the `BeersLawVolume` now fully defines a volumetric region with no surface properties which absorbs light.

Listing 4. A trivial BSDF representing an interface boundary with no effect, and a Material incorporating same

```
class TrivialBSDF : public BSDF {
public:
    TrivialBSDF(ShadingContext &ctx) : BSDF(ctx) {}
    virtual void EvaluateSample(RendererServices &rs, const Vector &sampleDirection, Color
        &L, float &pdf)
    {
        pdf = 0.0;
        L = Color(0.0);
    }
    virtual void GenerateSample(RendererServices &rs, Vector &sampleDirection, Color &L,
        float &pdf) {
        sampleDirection = -m_ctx.GetV();
        pdf = 1.0;
        L = Color(1.0);
    }
};

class BeersLawMaterial : public Material {
public:
    BeersLawMaterial(Color &extinction) : m_extinction(extinction) {}
    virtual BSDF *CreateBSDF(ShadingContext &ctx) {
        return new TrivialBSDF(ctx);
    }
    virtual bool HasVolume() {
        return true;
    }
    virtual Volume *CreateVolume(ShadingContext &ctx) {
        return new BeersLawVolume(m_extinction, ctx);
    }
private:
    const Color m_extinction;
};
```

Of course, the same `BeersLawVolume` may be combined with a more complicated `BSDF` to create materials with both interesting surface and volumetric behavior. Figure 10 shows a glass material without and without `BeersLawVolume`. (Note that the teapot in the figure is a properly modeled teapot with thin walls.)

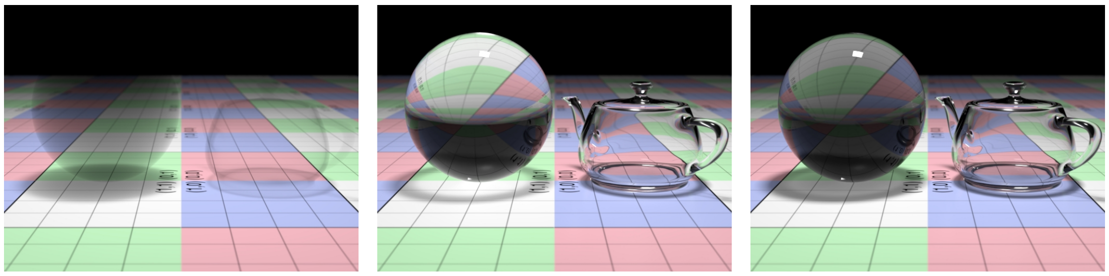


Fig. 10. Left: `BeersLawMaterial`, a material which combines `BeersLawVolume` with `TrivialBSDF`. Center: a material with a glass BSDF only and no volume. Right: a material combining a glass BSDF with `BeersLawVolume`.

3.2 Binding volumes to geometry

Now that we have some trivial volumes, we need to establish a convention for binding a volume to a region of space. Given the existence of ray traceable geometric primitives in our path tracer, we can use such primitives to establish the boundary of a volumetric region of space, and simply establish that a closed piece of geometry with outward facing normals is a valid container for a volume: the volume exists everywhere inside the geometry. (This assumes that the geometry is watertight when ray traced.) With this convention, a material that is bound to such a piece of geometry will return a `Volume` object when requested by the light integrator. Materials that return a `null Volume` are treated by the light integrator as a regular surface without a volume.

Suppose we have a ray, not currently inside a volume, with direction ω_i which hits an object at p_0 with a `Material` that indicates it has a volume binding, as shown on the left in figure 11. We know that the ray has hit the outside of the object because the incoming ray direction ω_i and the surface normal $\mathbf{n_0}$ point in opposite directions (their dot product is negative). When generating an indirect ray to continue the path, the `BSDF` can choose either to reflect or transmit. In the case of a reflection event, the `BSDF::GenerateSamples` method evaluates a BRDF and creates an outgoing ray with direction ω_r; no volume needs to execute over this ray as we have not entered the object and its associated volumetric region.

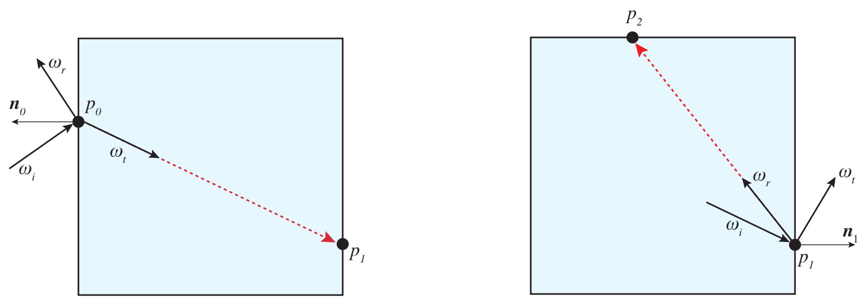


Fig. 11. Left: A ray intersecting the outside of a volume. In the case of a transmit event, a volume integration domain is created starting at point p_0; the volume integrator is responsible for calculating the end of its integration domain, potentially at point p_1. Right: A ray intersecting the inside of the same volume. In the case of a reflect event, another volume integration domain is created starting at point p_1; the volume integrator is responsible for ending its integration domain, potentially at point p_2.

In the case of a transmit event, the `BSDF::GenerateSamples` evaluates a BTDF and creates an outgoing ray with direction ω_t. In the case of a surface-only render, the light integrator would normally simply trace this ray against the scene database, create another shading event, and proceed as before. However, because our material is volume capable, we instead ask the material to create an instance of a `Volume`. The outgoing ray with direction ω_t is now an input to this `Volume` object, which is responsible for computing several outputs related to volume integration, as well as potentially tracing the ray against the scene database (by using the `GetNearestHit` method of the `RendererServices` object which is made available to the `Volume`) in order to find the end of the volume integration domain (in this case, the point p_1).

After the volume integration has executed, we are interested in what happens when the light integrator continues the integration and deals with a ray which hits the inside of the same object, as shown on the right in figure 11. Here, the incoming ray direction ω_i and the surface normal $\mathbf{n_1}$ point in the same direction: their dot product is positive. In this situation, again one of two situations may now occur: the `BSDF::GenerateSamples` method will either choose a reflection event or a transmit event. In the case of reflection, the resulting ray with direction ω_r has not left the interior of the object, and therefore has not left the volumetric region. Thus, another instance of the volume object must be created with input ω_r. In the case that the material instead selects a transmit event, the `BSDF` will create a ray with direction ω_t which has now left the volume object, and no volume needs to be created.

So far, we have considered only a single volume object, with a single material binding. Things are more complicated when multiple objects are involved. To help disambiguate situations such as this, we can utilize the concept of opposite and incident volumes: the *opposite* volume to an incident ray is the volume on the opposite side of the surface, while the *incident* volume to an incident ray is the volume on the same side of the surface. On the left of figure 11, the incident ray had an opposite volume but no incident volume, while on the right in the same figure the incident ray had an incident volume and no opposite volume. Now consider the situation of a volume object enclosing a smaller non-volumetric object, as shown in figure 12.

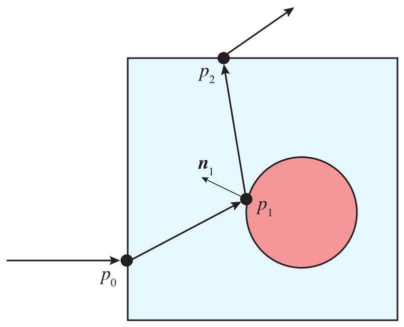


Fig. 12. A ray intersecting a non-volumetric object inside a volume. The volume binding between p_1 and p_2 has nothing to do with the `Material` of the non-volumetric object; it came from the `Material` bound to the `Geometry` at p_0

At p_1, there is no opposite volume because the enclosed red sphere is non- volumetric. However, there is definitely an incident volume. It is the same incident blue volume that was bound to the outer enclosing object. It is clear from this situation that the `Volume` object that needs to be executed at a hit point may actually have no relation with the `Material` binding of the intersected object; it may have been created from a previous `Material` encountered along the ray path.

This leads to an implementation where rays that are moving through our system must be aware of the `Material` that they are moving inside. Essentially, our rays must know what volume(s) they are inside. When a ray intersects an object, they may enter a new volume, or they may leave a volume. This can only occur on a transmit event; on a reflection event, they can neither enter a volume or leave a volume.

Now consider the case of two nested volumes: an outer volume V_O enclosing an inner volume V_I, as shown in figure 13.

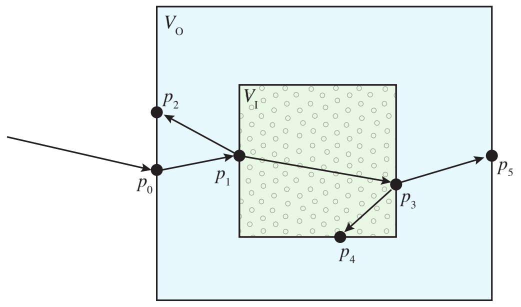


Fig. 13. A ray intersecting a volume V_I enclosed within another volume V_O. The intervals $[p_1, p_2]$ and $[p_3, p_4]$ require integration of both volumes.

At p_1, the incident volume is V_O. If a reflection event occurs at p_1, then the incident volume is V_0; this will be the volume that integrates over the interval $[p_1, p_2]$. If a transmit event occurs at p_1, then there are two opposite volumes: V_O and V_I. Both volumes must be integrated over the interval $[p_1, p_3]$.

At p_3, the incident volumes remain V_O and V_I. If a reflection event occurs at p_3, the incident volumes remain unchanged, and both volumes integrate over the interval $[p_3, p_4]$. If a transmit event instead occurs at p_3, then we must note that we have exited the volume V_I, leaving a single opposite volume V_O; this is the volume that integrates over the interval $[p_3, p_5]$.

In order to accommodate the need for tracking a list of `Materials` per ray, we can add `EnterMaterial` and `ExitMaterial` methods to the `Ray` data structure which add and remove a material from this list. We will see in the next section how these methods are used by the light integrator to track which volume should be used at an interface boundary.

Listing 5. Extensions to the Ray for tracking volumes

```
struct Ray {
    Ray() : org(0.0f), dir(0.0f) {}
    Ray(const Point &_org, const Vector &_dir) : org(_org), dir(_dir) {}
    Ray(const Ray &ray) : org(ray.org), dir(ray.dir), m_volumes(ray.m_volumes) {}
    Point org;
    Vector dir;

    void EnterMaterial(Material *b) { m_volumes.push_back(b); }
    void ExitMaterial(Material *b) {
        m_volumes.erase(std::remove(m_volumes.begin(), m_volumes.end(), b),
            m_volumes.end());
    }
    Volume *GetVolume(ShadingContext &ctx) const {
        if (!m_volumes.empty())
            return (*m_volumes.rbegin())->CreateVolume(ctx);
        else
            return nullptr;
    }
private:
    std::vector<Material *> m_volumes;
};
```

3.3 Light integrator

Now that we have established an interface for volumes, let's look at the changes to a light integrator necessary to incorporate our volume rendering system. The inner loop of a forward path tracer in our system before the introduction of volumes will look something like the following code. For every ray, we call `RendererServices::GetNearestHit` to find the next intersection point, create a shading context with `RendererServices::CreateShadingContext`, perform direct lighting, compute the direction of an indirect ray with `BSDF::GenerateSample`, and repeat until the maximum path length is achieved (or some other criterion is reached, such as minimum throughput to the eye). Along the way, the material response and the PDF from each call to the `BSDF` is used to attenuate the throughput to the eye, which is used to weigh the contribution of direct lighting.

Listing 6. Inner loop of a forward path tracing light integrator

```
    Color L = Color(0.0);
```

```
Color throughput = Color(1.0);
Ray ray = pickCameraDirection();
if (rs.GetNearestHit(ray, P, g))
    continue;

int j = 0;
while (j < maxPathLength)
{
    ShadingContext *ctx = rs.CreateShadingContext(P, ray, g);
    Material *m = g.GetMaterial();
    BSDF *bsdf = m->CreateBSDF(*ctx);

    // Perform direct lighting on the surface
    L += throughput * directLighting();

    // Compute direction of indirect ray
    float pdf;
    Color Ls;
    Vector sampleDirection;
    bsdf->GenerateSample(rs, sampleDirection, Ls, pdf);
    throughput *= (Ls / pdf);

    Ray nextRay(ray);
    nextRay.org = P;
    nextRay.dir = sampleDirection;
    if (!rs.GetNearestHit(nextRay, P, g))
        break;
    ray = nextRay;
    j++;
}
```

To incorporate volume rendering into this light integrator loop, we add some code after we have picked the direction of the indirect ray with `BSDF::GenerateSample`. As previously mentioned, we need to be aware of whether the indirect ray fired by the BSDF at the interface boundary is a transmit event (fired into the surface) or a reflect event (away), as this controls whether we need to consider a change in participating media. This is decided by comparing the sign of the dot product of the incoming ray against the surface normal, against the sign of the dot product of the outgoing ray against the surface normal. If a transmit event has indeed occurred, then a further decision is required: we need to decide whether we have entered or exited an object, which is signaled by the sign of the dot product of the ray direction against the normal. With the addition of the `EnterMaterial` and `ExitMaterial` methods to rays in order to track volumes, we can insert the following code into the light integrator in place of a simple call to `RendererServices::GetNearestHit`.

Listing 7. Extensions to the light integrator for volumes

```
    Volume *volume = 0;
    if (m->HasVolume()) {
       // Did we go transmit through the surface? V is the
       // direction away from the point P on the surface.
       float VdotN = ctx->GetV().Dot(ctx->GetN());
       float dirDotN = sampleDirection.Dot(ctx->GetN());
       bool transmit = (VdotN < 0.0) != (dirDotN < 0.0);
       if (transmit) {
           // We transmitted through the surface. Check dot
           // product between the sample direction and the
           // surface normal N to see whether we entered or
           // exited the volume media
           bool entered = dirDotN < 0.0f;
           if (entered) {
               nextRay.EnterMaterial(m);
           } else {
               nextRay.ExitMaterial(m);
           }
       }
       volume = nextRay.GetVolume(*ctx);
    }
    if (volume) {
       Color Lv;
       Color transmittance;
       float weight;
       if (!volume->Integrate(rs, nextRay, Lv, transmittance, weight, P, nextRay,
            g))
           break;
       L += weight * throughput * Lv;
       throughput *= transmittance;
    } else {
       if (!rs.GetNearestHit(nextRay, P, g))
           break;
    }
```

In this code, note that a single volume integrator will run over the interval, even if multiple volumes overlap. We will explain why in section 3.9. After determination of this `Volume`, its `Integrate` method is called instead of the call to `RendererServices::GetNearestHit` to allow the volume the chance to compute the end of the volume integration interval $\mathbf{x}_d$, and any radiance contribution from the volume `Lv` is added to the radiance estimate for the ray, weighted by the throughput so far. The beam transmittance from the volume $T(d)$ is itself multiplied into the throughput, and the inner loop of the path tracer then returns to its normal execution.

3.4 Transmittance rays

In the path tracer framework we are describing, we will assume that the direct lighting services provided by the renderer (`RendererServices::GenerateLightSample` and `RendererServices::EvaluateLightSample`) will compute and return the beam transmittance between the surface and the light, and that it will do so by the firing of a transmittance ray. In the surface-only path tracer, this calculation may have only involved the determination of shadow-casting occluders, but we will need to extend this mechanism to handle intervening volumes and their attenuating affect on the beam transmittance.

These transmittance rays are usually treated as a special case by a renderer because they can easily outnumber camera and indirect rays, and therefore deserve special optimization. One typical optimization performed for transmittance rays is to allow them to intersect geometry in any order, rather than in strict sorted depth-first order. This out-of-order execution is usually combined with early-out optimizations: any opaque object hit by a transmittance ray will immediately cause the ray to terminate. Furthermore, the importance of the efficient direct lighting may mean that in production rendering, transmittance rays may be allowed to break some laws of physics. For example, a glass material that normally must account for refraction on camera and indirect rays may choose to ignore such effects for transmittance rays; this optimization allows for much faster convergence when direct lighting objects behind glass, and may be visually acceptable if the glass is a thin material.

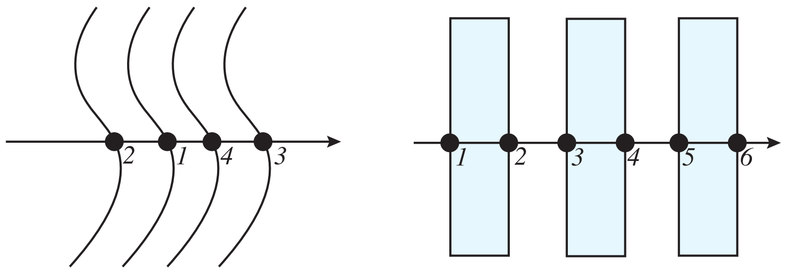


Fig. 14. Left: transmittance rays intersecting geometry can intersect the geometry in any order without affecting the transmittance result. Right: transmittance rays intersecting volumes need to find the correct volume extents, which requires intersection in depth first order. Moreover, correct tracking of volume entry and exit requires computation of the normal at every hit point.

Given this special treatment, integrating volumes into the framework of transmittance rays poses a few special problems. In a surface-only situation, either there is no need to shade a hit point (it is trivially opaque), or an opacity shader needs to be executed on that single hit point. With volumes, we need to intermingle the execution of `Volume::Transmittance` over a pair of hit points. This immediately curtails the ability to perform out of order hit testing, as we must ensure that the two hit points truly reflect the proper volume extents. Moreover, we must now track the proper volume bindings attached to the transmittance ray as it makes it way through the scene. This is simplified somewhat in this situation as by definition the transmittance ray must go through a surface and never reflects, so we never need to perform the dot product between the outgoing ray direction and the incoming ray direction. However, we still need to track whether the ray enters or leaves a volumetric material, which requires a dot product between the ray direction and the surface normal at every hit point.

Finally, it is important to realize that direct lighting of a volume requires self-shadowing, which means the transmittance ray from that volume must correctly track that it needs to compute the transmittance through the same volume. In order to accurately capture this state of affairs, we require that transmittance rays inherit the volume tracking state from the camera or indirect ray that intersected the hit point, and also require the convention that transmittance rays are fired from the point of illumination towards the light source.

3.5 Heterogeneous volumes

A heterogeneous, absorption-only volume represents the first real complexity associated with writing a volume integrator. So far, we have not described how texturing or pattern generation works in our system. In a typical production renderer, a material would actually be the root node of a shader graph, with the inputs to this material being connections to upstream nodes responsible for pattern generation or texturing. We now assume in our system that a facility exists whereby the combination of a `ShadingContext` bound to a `Material` allows for the evaluation of such input nodes. We assume such inputs are uniquely identified with an integer index, and that their value can be evaluated (triggering the necessary upstream graph evaluation) by invoking the `ShadingContext::GetFloatProperty` or `ShadingContext::GetColorProperty` methods.

With such a system, when rendering a textured surface, a material node may parameterize the diffuse color with an input texture map, where the texture map coordinates are driven by primitive variables associated with the geometry (encapsulated into a manifold as embodied by the `ShadingContext` class). Volumes are no different from surfaces in this regard, save that their parameterization for texturing usually require three coordinates instead of just two. However, they are also fundamentally different in one key aspect: while the shading context for a single hit point usually does not need alteration, a volume has an additional dimension for integration (along the length of the ray), and a single set of values associated with a single hit point will not suffice.

The volumes presented so far have not altered any properties of the `ShadingContext` passed into the `Volume`, and the geometry associated with that context has remained that of the hit point at the beginning of the volume. The properties of the volume itself have been described simply by using constant parameters to the volume itself. In order to implement a heterogeneous volume, however, we require the ability to alter the `ShadingContext` to be some other point within the volumetric region, and to reevaluate upstream inputs. This requires that the context object have mutability in its geometric properties. One approach to such a system is to allow the volume integrator to set a position on the `ShadingContext` using the `SetP` method, and then have the renderer services automatically recompute the properties of the geometric environment as well as all upstream inputs with `RendererServices::RecomputeShadingContext()`. Subsequent calls to the `ShadingContext::GetFloatProperty` or `ShadingContext::GetColorProperty` methods will now return updated values.

With these pieces, we can now extend `BeersLawVolume` to implement heterogeneous absorption, using delta tracking as described in algorithm 2 to implement a transmittance estimator. (An implementation of the transmittance estimator using ray marching/quadrature instead of a Monte Carlo method would also require the same renderer services; we will show such an implementation when we discuss the PDF approach to single scattering in section 3.6.2.) The majorant $\bar{\sigma}$ over the density $\sigma_t(\mathbf{x})$ required for delta tracking is in this implementation the parameter `maxAbsorption` passed to the constructor. Only the `Transmittance` method is shown here since `Integrate` is unchanged from `BeersLawVolume`.

Listing 8. Heterogeneous absorption integrator

```
class BeersLawHeterogeneousVolume: public Volume {
public:
    BeersLawHeterogeneousVolume(Color &maxAbsorption, int absorptionProperty,
         ShadingContext &ctx) :
        Volume(ctx), m_maxAbsorption(maxAbsorption.ChannelAvg()),
             m_absorptionProperty(absorptionProperty) {}
    virtual bool Integrate(RendererServices &rs, const Ray &wi, Color &L, Color
        &transmittance, float &weight, Point &P, Ray &wo, Geometry &g) {
        if (!rs.GetNearestHit(Ray(m_ctx.GetP(), wi.dir), P, g))
            return false;
        L = Color(0.0);
        transmittance = Transmittance(rs, P, m_ctx.GetP());
        weight = Color(1.0);
        wo = Ray(P, wi.dir);
        return true;
    }
    virtual Color Transmittance(RendererServices &rs, const Point &P0, const Point &P1) {
        float distance = Vector(P0 - P1).Length();
        Vector dir = Vector(P1 - P0) / distance;
        bool terminated = false;
        float t = 0;
        do {
            float zeta = rs.GenerateRandomNumber();
            t = t - log(1 - zeta) / m_maxAbsorption;
            if (t > distance) {
                break; // Did not terminate in the volume
            }

            // Update the shading context
            Point P = P0 + t * dir;
            m_ctx.SetP(P);
            m_ctx.RecomputeInputs();

            // Recompute the local absorption after updating the shading context
            Color absorption = m_ctx.GetColorProperty(m_absorptionProperty);
            float xi = rs.GenerateRandomNumber();
            if (xi < (absorption.ChannelAvg() / m_maxAbsorption))
                terminated = true;
        } while (!terminated);

        if (terminated)
            return Color(0.0);
        else
            return Color(1.0);
    }
protected:
    const float m_maxAbsorption;
    const int m_absorptionProperty;
};
```

3.6 Single scattering

In many production renderers, direct lighting (in-scattered radiance which comes directly from a light source to the point of illumination) is given special treatment. We assume that this is also true in the theoretical path tracer we are describing so far. For volumetric rendering, it is useful to consider the limited case of *single scattering*, where the in-scattered radiance takes into consideration only the direct lighting to a point on the volume (hence the term single scattering: the light has scattered only once on the way to the outgoing illumination direction). This limited case is analogous to a direct lighting integrator for surfaces. While manifestly non-physical, it is also an effect that has persisted even in otherwise physically based production rendering because single scattering is a useful approximation for dark volumes that have low albedo, where higher order multiple scattering effects are of lesser importance.

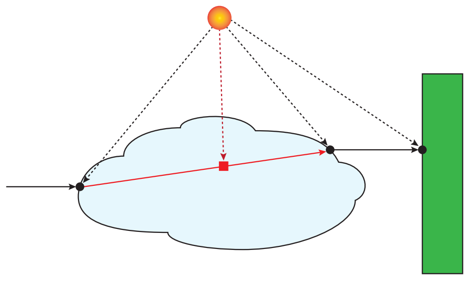


Fig. 15. Single scattering. The volume integrator is responsible for computing the hit point all the way through the volume, as well as an estimate of the in-scattered radiance (red dashed line) at a sample location (red square).

It is worth revisiting the simplified equation for the in-scattered radiance, equation 9, to determine how to compute the direct lighting contribution at a point x in the volume:

$$L(\mathbf{x},\omega) = \sigma_s(\mathbf{x}) \int_{S^2} f_p(\mathbf{x},\omega,\omega')L(\mathbf{x},\omega)d\omega'$$

By analogy to how multiple importance sampling (Veach 1997) works when computing direct lighting for surfaces: we can estimate the value of this integral either by sampling the phase function; or by sampling the light source; or we can do both and use multiple importance sampling to combine the results. A path tracer which implements a `BSDF` interface along with direct lighting for surfaces already has most of the pieces necessary to handle this integral estimation, and in fact can simply recast a phase function in terms of a `BSDF`. Let's assume that in such a path tracer, the `RendererServices` object provides a `GenerateLightSample` method to sample the light database given a shading context generated from a point; the method returns a sampled direction `sampleDirection`, the radiance `L` arriving at the point from the sample direction, along with the associated probability density `pdf` and the `beamTransmittance` between the light and the point. Similarly, a `EvaluateLightSample` method can be used for BSDF sampling: given a shading context generated from a point and a sample direction generated by a `BSDF`, the method returns the radiance `L` arriving at the point from the sample direction, along with the associated probability density `pdf` and the `beamTransmittance` between the light and the point. For both of these methods, the computation of `beamTransmittance` must involve the firing of a transmission ray, and the discussion from section 3.4 is therefore relevant: we assume that the implementation of transmission rays has already been altered as per the previous section to include the necessary calls to `Volume::Transmittance` as required for any intervening volumes (including the current volume we are dealing with itself).

Assuming the renderer provides all these pieces already, we can begin with the simplest phase function, isotropic scattering, and turn equation 2 into a BSDF:

Listing 9. Isotropic phase function

```
class IsotropicPhaseBSDF : public BSDF {
public:
    IsotropicPhaseBSDF(ShadingContext &ctx) : BSDF(ctx) {}
    virtual void EvaluateSample(RendererServices &rs, const Vector &sampleDirection, Color
         &L, float &pdf) {
        pdf = 0.25 / M_PI;
        L = Color(pdf);
    }
    virtual void GenerateSample(RendererServices &rs, Vector &sampleDirection, Color &L,
         float &pdf) {
        float xi = rs.GenerateRandomNumber();
        sampleDirection.z = xi * 2.0 - 1.0; // cosTheta
        float sinTheta = 1.0 - sampleDirection.z * sampleDirection.z; // actually square of
             sinTheta
        if (sinTheta > 0.0)
        {
            sinTheta = std::sqrt(sinTheta);
            xi = rs.GenerateRandomNumber();
            float phi = xi * 2.0 * M_PI;
            sampleDirection.x = sinTheta * cosf(phi);
            sampleDirection.y = sinTheta * sinf(phi);
        }
        else
            sampleDirection.x = sampleDirection.y = 0.0;
        pdf = 0.25 / M_PI;
        L = Color(pdf);
    }
};
```

Depending on the style of BSDF used for surfaces, you may be more used to a BSDF which multiplies surface properties into the `L` term - for example, some form of diffuse color. It is quite easy to make our phase functions look similar by using the parametrization described in section 2.1.1, in particular the single scattering albedo $\alpha = \sigma_s/\sigma_t$. As previously discussed, α is very similar to the surface albedo and could naturally appear in our phase function masquerading as a BSDF. For the sake of clarity, in the subsequent code we have chosen to keep the α term inside the single scattering volume integrator itself, where it appears when we need σ_s (because $\sigma_s = \alpha\sigma_t$).

3.6.1 Closed-form tracking approach. In the closed-form tracking approach (section 2.3.1) applied to single scattering, we estimate the radiance at a location $\mathbf{x}' = \mathbf{x}_0 + t'\omega$, using a Monte Carlo approach to choose this location. As described in section 2.3.1, one initial approach would be to use a PDF based on normalizing the transmittance (equation 14), which in the homogeneous case we can perfectly importance sample by using equation 15: $t' = -\ln(1-\xi)/\sigma_t$. If we implement things this way, we would end up with a volume integrator that when combined with the lighting integrator would essentially be the implementation of closed-form tracking as described in Algorithm 1.

We will instead depart from this approach slightly by noting that the PDF of equation 14 actually has the domain $0 \leq t < \infty$, which is why the distance t' chosen by equation 15 may be greater than d (thus necessitating the check $t > d$ on line 4 of Algorithm 1). In practical terms, this means that using this PDF may result in a sample location chosen outside the volume, and consequently no lighting estimate of the volume will occur. This can lead to high variance for low density volumes lit by a strong light source. We can instead create a new PDF which normalizes the integral of the exponential function of transmittance over the domain $0 \leq t < d$, resulting in

$$p(t) = \frac{\sigma_t \exp(-\sigma_t t)}{1 - \exp(-\sigma_t d)} \tag{28}$$

which leads to a slightly modified importance sample function:

$$t' = \frac{-\ln(1 - \xi(1 - \exp(-\sigma_t d)))}{\sigma_t} \tag{29}$$

The t' that results from this formulation is guaranteed to be less than d, ensuring that a sample location is within the extents of the volume.

In summary, to perform single scattering in a homogeneous medium we only need to extend the `BeersLawVolume` integrator to pick this sample location and perform direct lighting at the location.

Listing 10. Single scattering homogeneous integrator using closed form tracking

```
class SingleScatterHomogeneousVolume: public Volume {
public:
    SingleScatterHomogeneousVolume(Color &scatteringAlbedo, Color &extinction,
         ShadingContext &ctx) :
        Volume(ctx), m_scatteringAlbedo(scatteringAlbedo), m_extinction(extinction) {}
    virtual bool Integrate(RendererServices &rs, const Ray &wi, Color &L, Color
         &transmittance, Color &weight, Point &P, Ray &wo, Geometry &g) {
        if (!rs.GetNearestHit(Ray(m_ctx.GetP(), wi.dir), P, g))
            return false;

        // Transmittance over the entire interval
        transmittance = Transmittance(rs, P, m_ctx.GetP());

        // Compute sample location for scattering, based on the PDF
        // normalized to the total transmission
        float xi = rs.GenerateRandomNumber();
        float scatterDistance = -logf(1.0f - xi * (1.0f - transmittance.ChannelAvg())) /
             m_extinction.ChannelAvg();

        // Set up shading context to be at the scatter location
        Point Pscatter = m_ctx.GetP() + scatterDistance * wi.dir;
        m_ctx.SetP(Pscatter);
        m_ctx.RecomputeInputs();

        // Compute direct lighting with light sampling and phase function sampling
        IsotropicPhaseBSDF phaseBSDF(m_ctx);
        L = Color(0.0);
        Color lightL, bsdfL, beamTransmittance;
        float lightPdf, bsdfPdf;
        Vector sampleDirection;
        rs.GenerateLightSample(m_ctx, sampleDirection, lightL, lightPdf, beamTransmittance);
        phaseBSDF.EvaluateSample(rs, sampleDirection, bsdfL, bsdfPdf);
        L += lightL * bsdfL * beamTransmittance * rs.MISWeight(1, lightPdf, 1, bsdfPdf) /
            lightPdf;

        phaseBSDF.GenerateSample(rs, sampleDirection, bsdfL, bsdfPdf);
        rs.EvaluateLightSample(m_ctx, sampleDirection, lightL, lightPdf, beamTransmittance);
        L += lightL * bsdfL * beamTransmittance * rs.MISWeight(1, lightPdf, 1, bsdfPdf) /
            bsdfPdf;

        Color Tr(exp(m_extinction.r * -scatterDistance), exp(m_extinction.g *
            -scatterDistance), exp(m_extinction.b * -scatterDistance));
        L *= (m_extinction * m_scatteringAlbedo * Tr);

        // This weight is 1 over the PDF normalized to the total transmission
        weight = (1 - transmittance) / (Tr * m_extinction);
        wo = Ray(P, wi.dir);
        return true;
    }
    virtual Color Transmittance(RendererServices &rs, const Point &P0, const Point &P1) {
        float distance = Vector(P0 - P1).Length();
        return Color(exp(m_extinction.r * -distance), exp(m_extinction.g * -distance),
            exp(m_extinction.b * -distance));
    }
protected:
    const Color m_scatteringAlbedo;
    const Color m_extinction;
};
```

There are two items of note. First, the implementation of direct lighting within this volume should be very close to the implementation of direct lighting for a surface, save for one important difference: multiplication of the radiance by the geometric term $\omega \cdot n$ required for surfaces is replaced by multiplication by the scattering coefficient σ_s (which is itself the product of the extinction coefficient and the scattering albedo). Second, two separate transmittance terms are computed by this integrator: one is the transmittance over the entire volume interval $T(d)$ (which is returned in the `transmittance` output parameter), and the other is the transmittance $T(t')$ (the local variable `Tr`), computed only up to the location where scattering takes place. This second value is multiplied into the radiance returned (as required by the integrand of the VRE, equation 12), but is not otherwise needed by the light integrator.

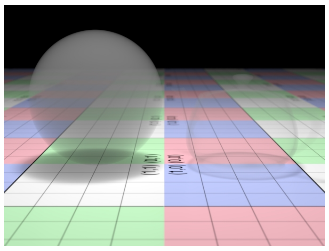


Fig. 16. Single scattering homogeneous integrator

Now, we can tackle the case of heterogeneous single scattering by combining `BeersLawHeterogeneousVolume` with `SingleScatterHomogeneousVolume`. Delta tracking can be used to determine the scatter location: the distribution of the delta tracking termination locations has a PDF proportional to the transmittance. Unlike the homogeneous case, there is no easy way to modify this PDF to be normalized over a finite domain, so an implementation based on delta tracking cannot guarantee that a sample will be taken inside the volume. The implementation of `Transmittance` is identical to that of `BeersLawHeterogeneousVolume` and is not shown here.

Listing 11. Single scattering heterogeneous integrator

```
class SingleScatterHeterogeneousVolume: public Volume {
public:
   SingleScatterHeterogeneousVolume(int scatteringAlbedoProperty, Color &maxExtinction,
       int extinctionProperty, ShadingContext &ctx) :
      Volume(ctx), m_scatteringAlbedoProperty(scatteringAlbedoProperty),
          m_maxExtinction(maxExtinction.ChannelAvg()),
          m_extinctionProperty(extinctionProperty) {}
   virtual bool Integrate(RendererServices &rs, const Ray &wi, Color &L, Color
       &transmittance, Color &weight, Point &P, Ray &wo, Geometry &g) {
      Point P0 = m_ctx.GetP();
      if (!rs.GetNearestHit(Ray(P0, wi.dir), P, g))
         return false;
      float distance = Vector(P - P0).Length();
      bool terminated = false;
      float t = 0;
      do {
         float zeta = rs.GenerateRandomNumber();
         t = t - log(1 - zeta) / m_maxExtinction;
         if (t > distance) {
            break; // Did not terminate in the volume
         }

         // Update the shading context
         Point Pl = P0 + t * wi.dir;
```

```
        m_ctx.SetP(Pl);
        m_ctx.RecomputeInputs();

        // Recompute the local extinction after updating the shading context
        Color extinction = m_ctx.GetColorProperty(m_extinctionProperty);
        float xi = rs.GenerateRandomNumber();
        if (xi < (extinction.ChannelAvg() / m_maxExtinction))
            terminated = true;
    } while (!terminated);

    if (terminated)
    {
        // The shading context has already been advanced to the
        // scatter location. Compute direct lighting after
        // evaluating the local scattering albedo and extinction
        Color scatteringAlbedo = m_ctx.GetColorProperty(m_scatteringAlbedoProperty);
        Color extinction = m_ctx.GetColorProperty(m_extinctionProperty);
        IsotropicPhaseBSDF phaseBSDF(m_ctx);
        L = Color(0.0);
        Color lightL, bsdfL, beamTransmittance;
        float lightPdf, bsdfPdf;
        Vector sampleDirection;
        rs.GenerateLightSample(m_ctx, sampleDirection, lightL, lightPdf,
             beamTransmittance);
        phaseBSDF.EvaluateSample(rs, sampleDirection, bsdfL, bsdfPdf);
        L += lightL * bsdfL * beamTransmittance * scatteringAlbedo * extinction *
             rs.MISWeight(1, lightPdf, 1, bsdfPdf) / lightPdf;

        phaseBSDF.GenerateSample(rs, sampleDirection, bsdfL, bsdfPdf);
        rs.EvaluateLightSample(m_ctx, sampleDirection, lightL, lightPdf,
             beamTransmittance);
        L += lightL * bsdfL * beamTransmittance * scatteringAlbedo * extinction *
             rs.MISWeight(1, lightPdf, 1, bsdfPdf) / bsdfPdf;
        transmittance = Color(0.0f);
        Color pdf = extinction; // Should be extinction * Tr, but Tr is 1
        weight = 1.0 / pdf;
    }
    else
    {
        transmittance = Color(1.0f);
        weight = Color(1.0f);
    }

    wo = Ray(P, wi.dir);
    return true;
}
```

```
protected:
    const int m_scatteringAlbedoProperty;
    const float m_maxExtinction;
    const int m_extinctionProperty;
};
```

There is one subtle difference between this and the homogeneous case: when using delta tracking, the transmittance up to the scatter location is simply 1, so it does not need to be multiplied into the lighting term. (The scattering coefficient σ_s does still need to be multiplied into the radiance, even if it is subsequently cancelled out by multiplication with the inverse of the pdf which also contains this term.)

We can extend our scattering volume integrators to support anisotropy by implementing the Henyey-Greenstein phase function (equation 3) as a BSDF. A single lobed variant parametrized by the anisotropy term g is shown below, with results shown in figure 17.

Listing 12. Anisotropic phase function

```
class AnisotropicPhaseBSDF : public BSDF {
public:
    AnisotropicPhaseBSDF(float g, ShadingContext &ctx) : BSDF(ctx), m_g(g) {
        if (m_g < -1.0f) m_g = -1.0f;
        if (m_g > 1.0f) m_g = 1.0f;
        m_isotropic = (fabsf(m_g) < 0.0001f);
        // Precompute terms used for the pdf/cdf
        if (!m_isotropic) {
            m_one_plus_g2 = 1.0f + m_g * m_g;
            m_one_minus_g2 = 1.0f - m_g * m_g;
            m_one_over_2g = 0.5f / m_g;
        }
    }
    virtual void EvaluateSample(RendererServices &rs, const Vector &sampleDirection, Color
         &L, float &pdf) {
        if (m_isotropic)
            pdf = 0.25 / M_PI;
        else {
            float cosTheta = Dot(-m_ctx.GetV(), sampleDirection);
            pdf = calcpdf(cosTheta);
        }
        L = Color(pdf);
    }
    virtual void GenerateSample(RendererServices &rs, Vector &sampleDirection, Color &L,
         float &pdf) {
        if (m_isotropic) {
            // same as IsotropicPhaseBSDF::GenerateSample
        } else {
            float phi = rs.GenerateRandomNumber() * 2 * M_PI;
            float cosTheta = invertcdf(rs.GenerateRandomNumber());
            float sinTheta = sqrtf(1.0f - cosTheta * cosTheta); // actually square of
                sinTheta
            Vector in = -m_ctx.GetV();
            Vector t0, t1;
            in.CreateOrthonormalBasis(t0, t1);
            sampleDirection = sinTheta * sinf(phi) * t0 + sinTheta * cosf(phi) * t1 +
                cosTheta * in;
            pdf = calcpdf(cosTheta);
        }
        L = Color(pdf);
    }
private:
    float calcpdf(float costheta) {
        return 0.25 * m_one_minus_g2 / (M_PI * powf(m_one_plus_g2 - 2.0f * m_g * costheta,
            1.5f));
    }
    // Assumes non-isotropic due to division by m_g
    float invertcdf(float xi) {
        float t = (m_one_minus_g2) / (1.0f - m_g + 2.0f * m_g * xi);
        return m_one_over_2g * (m_one_plus_g2 - t * t);
    }
    float m_g, m_one_plus_g2, m_one_minus_g2, m_one_over_2g;
    bool m_isotropic;
};
```

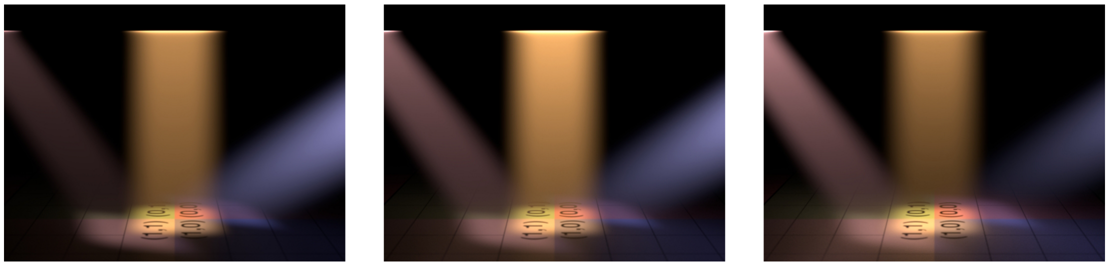


Fig. 17. Varying the anisotropic term g with `AnisotropicPhaseBSDF` and single scattering. Left: g = -0.5 (backwards scattering); center: g = 0.0 (isotropic scattering); right: g = 0.5 (forwards scattering). The red light is aimed towards the camera, while the blue light is aimed away from the camera.

3.6.2 PDF approach. As described in section 2.4, the PDF approach to single scattering amounts to building a PDF for sampling the integral $\int_{t=0}^{d} T(t)\sigma_s(\mathbf{x}_t)L_s(\mathbf{x}_t,\omega)dt$ from the separated VRE shown in equation 26.

In order to reduce variance in the estimate, it is desirable to have the PDF be proportional to the terms of the integrand which have the most variance. In the absence of this knowledge, especially in the heterogeneous volume case, a standard way of constructing the PDF is to try to approximate the entire integrand $p(t) \approx T(t)\sigma_s(\mathbf{x}_t)L_s(\mathbf{x}_t,\omega)$, and take samples of $p(t)$ at fixed intervals to create a discrete distribution. (If we let $p(t)$ actually be the integrand instead of an approximation, then we may as well compute the integral by quadrature rather than building and sampling a PDF.) We can then apply inverse transform sampling to the distribution. The table of PDF values is converted to a discrete cumulative density function (CDF) $c(t)$ by normalization. We can now importance sample the PDF by generating a random number ξ in the interval $[0, 1)$ and finding the value of t with $c(t)$ nearest to ξ by searching the distribution of CDF values.

While straightforward, using inverse transform sampling on discrete samples has the same drawbacks as using ray marching to estimate transmittance: it introduces bias, even if we jitter the interval size, and for a single sample is extremely expensive compared to the tracking approach. However, using a PDF $p(t)$ which approximates all the terms of the integrand also means for a single sample we have the potential to perform much better importance sampling; the PDF used in the tracking methods shown so far are only proportional to the $T(t)$ term, which means that if the variance in the integral is actually due to the lighting term L_s, the importance sampling will not capture the source of variance, and the tracking estimate will be much noisier.

The highest expense associated with this method is the computation of $p(t)$ at regular intervals, and most of that cost lies with approximating the $L(\mathbf{x}, \omega)$ component of the L_s term: the in-scattered radiance coming from lights (keeping in mind that here we are dealing strictly with single scattering, so $L(\mathbf{x}, \omega)$ is strictly from a light). Implementors of the PDF approach must weigh whether the computational cost of better approximations to $L(\mathbf{x}, \omega)$ are worth the decrease in variance. One trade off to consider is that much of the cost of lighting is in the cost of firing a transmission ray, and calculating the beam transmittance along that ray. Removing the beam transmittance from $L(\mathbf{x}, \omega)$ will speed up the calculation at the cost of making the estimate worse: areas of the volume that are shadowed (or self-shadowed) will be overly sampled.

The following listing gives a simple example of using the PDF approach for single scattering. `SingleScatterPDFVolume` takes as argument a stepsize and builds a PDF using $T(t)$, $\sigma_s(\mathbf{x}_t)$, and an approximation to $L_s(\mathbf{x}_t, \omega)$ using only a sample direction generated by the phase function and the lighting response from the light without the beam transmittance. This response is handled by the four argument variant of `RendererServices::EvaluateLightSample`, which will return results similar to the five argument variant that we have been using, except that it will not compute the beam transmittance and will not shoot a transmission ray. Note that in this listing, we do not specialize for the homogeneous volume case, but doing so would be trivial (by eliminating the re-evaluation of the local extinction coefficient and scattering albedo at each step). We also show in this listing how to evaluate transmittance by quadrature, since its computation closely mirrors much of the steps of the PDF approach to single scattering.

Listing 13. Single scattering integrator using the PDF approach

```
class SingleScatterPDFVolume: public Volume {
public:
    SingleScatterPDFVolume(float stepsize, int scatteringAlbedoProperty, int
         extinctionProperty, ShadingContext &ctx) :
        Volume(ctx), m_stepsize(stepsize),
            m_scatteringAlbedoProperty(scatteringAlbedoProperty),
            m_extinctionProperty(extinctionProperty) {}
    virtual bool Integrate(RendererServices &rs, const Ray &wi, Color &L, Color
         &transmittance, Color &weight, Point &P, Ray &wo, Geometry &g) {
        Point P0 = m_ctx.GetP();
```

```
if (!rs.GetNearestHit(Ray(P0, wi.dir), P, g))
    return false;
float distance = Vector(P - P0).Length();
int nsteps = (int) ceilf(distance / m_stepsize);
float adjustedStepsize = distance / nsteps;
Color accumTrans(1.0f);
float t = 0, pdfSum = 0.0f;
float *pdf = new float[nsteps];
for (int i = 0; i < nsteps; ++i) {
    // Update the shading context
    Point Pl = P0 + t * wi.dir;
    m_ctx.SetP(Pl);
    m_ctx.RecomputeInputs();

    // Recompute the local extinction after updating the shading context
    Color extinction = m_ctx.GetColorProperty(m_extinctionProperty);
    // Compute transmittance over the step, accumulate
    accumTrans *= Color(exp(extinction.r * -adjustedStepsize), exp(extinction.g *
        -adjustedStepsize), exp(extinction.b * -adjustedStepsize));

    // Evaluate lighting response without transmittance to a scattering direction
    // chosen by the BSDF
    Color lightL, bsdfL;
    float lightPdf, bsdfPdf;
    Vector sampleDirection;
    IsotropicPhaseBSDF phaseBSDF(m_ctx);
    phaseBSDF.GenerateSample(rs, sampleDirection, bsdfL, bsdfPdf);
    rs.EvaluateLightSample(m_ctx, sampleDirection, lightL, lightPdf);
    Color Ls = lightL * bsdfL / (bsdfPdf * lightPdf);

    // Compute T(t) x sigma_s x L_s term for pdf
    Color scatteringAlbedo = m_ctx.GetColorProperty(m_scatteringAlbedoProperty);
    pdf[i] = (accumTrans * scatteringAlbedo * extinction * Ls).ChannelAvg();
    pdfSum += pdf[i];

    t += adjustedStepsize;
}

// Create CDF from PDF
float total = 0.0f;
for (int i = 0; i < nsteps - 1; ++i) {
    total += pdf[i];
    pdf[i] = total / pdfSum;
}
pdf[nsteps - 1] = 1.0f;
```

```
    // Sample CDF
    float xi = rs.GenerateRandomNumber();
    for (int i = 0; i < nsteps; ++i) {
        if (pdf[i] > xi) {
            if (i == 0)
                t = (xi / pdf[i]) * adjustedStepsize;
            else
                t = (i + (pdf[i] - xi) / (pdf[i] - pdf[i - 1])) * adjustedStepsize;
            break;
        }
    }
    delete[] pdf;

    // Update the shading context to the scatter location
    Point Pl = P0 + t * wi.dir;
    m_ctx.SetP(Pl);
    m_ctx.RecomputeInputs();

    // Direct lighting
    Color scatteringAlbedo = m_ctx.GetColorProperty(m_scatteringAlbedoProperty);
    Color extinction = m_ctx.GetColorProperty(m_extinctionProperty);
    IsotropicPhaseBSDF phaseBSDF(m_ctx);
    L = Color(0.0);
    Color lightL, bsdfL, beamTransmittance;
    float lightPdf, bsdfPdf;
    Vector sampleDirection;
    rs.GenerateLightSample(m_ctx, sampleDirection, lightL, lightPdf, beamTransmittance);
    phaseBSDF.EvaluateSample(rs, sampleDirection, bsdfL, bsdfPdf);
    L += lightL * bsdfL * beamTransmittance * scatteringAlbedo * extinction *
         rs.MISWeight(1, lightPdf, 1, bsdfPdf) / lightPdf;

    phaseBSDF.GenerateSample(rs, sampleDirection, bsdfL, bsdfPdf);
    rs.EvaluateLightSample(m_ctx, sampleDirection, lightL, lightPdf, beamTransmittance);
    L += lightL * bsdfL * beamTransmittance * scatteringAlbedo * extinction *
         rs.MISWeight(1, lightPdf, 1, bsdfPdf) / bsdfPdf;

    transmittance = accumTrans;

    weight = Color(1.0);
    wo = Ray(P, wi.dir);
    return true;
}
virtual Color Transmittance(RendererServices &rs, const Point &P0, const Point &P1) {
    float distance = Vector(P0 - P1).Length();
    int nsteps = (int) ceilf(distance / m_stepsize);
    float adjustedStepsize = distance / nsteps;
```

```
        Color accumTrans(1.0f);
        float t = 0;
        for (int i = 0; i < nsteps; ++i)
        {
            // Update the shading context
            Point Pl = P0 + t * Vector(P1 - P0);
            m_ctx.SetP(Pl);
            m_ctx.RecomputeInputs();
            // Recompute the local extinction after updating the shading context
            Color extinction = m_ctx.GetColorProperty(m_extinctionProperty);
            // Compute transmittance over the step, accumulate
            accumTrans *= Color(exp(extinction.r * -adjustedStepsize), exp(extinction.g *
                 -adjustedStepsize), exp(extinction.b * -adjustedStepsize));
            t += adjustedStepsize;
        }
        return accumTrans;
    }
protected:
    const float m_stepsize;
    const int m_scatteringAlbedoProperty;
    const int m_extinctionProperty;
};
```

3.7 Multiple scattering

The primary difference between single scattering and multiple scattering in a volume is that in the latter, an indirect ray can be generated due to a scattering interaction within the volume itself. The single scattering implementation presented so far precludes any possibility of this happening; in our interface, only the light integrator will create indirect rays (by invoking the `BSDF::GenerateSample` method), and the volume integrator can only contribute a radiance estimate of the inscatter along the extent of the ray through the volume extent. Of course, it is entirely possible to bend the interface and have the volume integrator itself implement a full implementation of multiple scattering, but this is in essence writing a light integrator within a light integrator and is ill advised.

Instead, we can take advantage of the extra outputs in `Volume::Integrate`. Rather than shooting a ray entirely through the volume to the media interface and returning a hit point on that interface, an implementation of a volume that implements multiple scattering can choose instead to stop the ray somewhere inside the volume, and return this volumetric scattering location to the lighting integrator. Furthermore, instead of computing a radiance estimate of the interval, the volume integrator returns zero and lets the lighting integrator itself compute the radiance at the scattering location. In essence, we are implementing a variant of closed-form tracking as described in algorithm 1, where the light integrator is responsible for computing L_d, L_e, and ω, and the volume integrator is responsible for only computing t and x_t, as well as the weight (inverse PDF) associated with the choice of x_t.

The weight associated with x_t is a new quantity that normally would not exist in a surface-only path tracer. We can think of this term as follows: in a forward path tracer, every camera or indirect ray that is fired to advance the path is actually trying to compute a single sample estimate of the radiance along the infinite extent of that ray. It just so happens that if we are only dealing with surfaces, the in-scatter along the infinite length of the ray comes from a single location: the nearest surface that is hit. Therefore, the probability associated with that single location is 1, every other location along the ray has a probability of zero, and the weight associated with the location is trivially 1 and therefore unnecessary, as the in-scattering at the surface is the only single sample required for the in- scattering over the entire ray.

In the case where there are volumes along a ray, this is no longer the case: there is potential for in-scattering along the infinite ray extent wherever there is a volume, as well as the first opaque surface behind all volumes. Since a forward path tracer still must stop at a single location along the ray, we must now consider that wherever we stop can only be an estimate of the in-scatter along the infinite extent of the ray. Therefore, we must associate a weight of the estimate along with our location choice.

To complicate matters even further: in our decoupled shading model, a volume has only enough knowledge to compute a PDF $p(t)$ along a finite extent $0 < t < d$ where d is the end of the volume geometry. For the parts of the ray beyond the volume, the best that it can do is to compute a single probability $p(t \geq d)$ for the sample location being anywhere $t \geq d$, i.e. at the volume interface or beyond it.

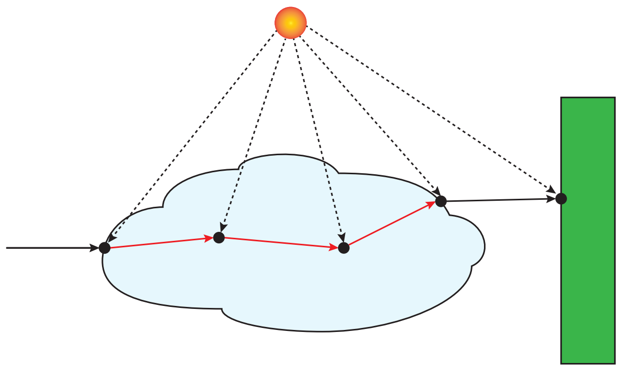


Fig. 18. Multiple scattering. Each invocation of the volume integrator (red lines) is responsible only for computing the next hit point, the transmittance over the interval, and the weight associated with the location; all radiance calculations are handled by the lighting integrator (black dashed lines).

To implement a tracking approach to multiple scattering, unlike the single scattering homogeneous integrator, we want to use the PDF with unbounded domain proportional to the transmittance (equation 14) to decide the sample location: $t' = -\ln(1-\xi)/\sigma_t$. The probability associated with sampling anywhere beyond the end of the volume $p(t \geq d)$ is computed by using the associated CDF $c(t)$ of the PDF: $p(t \geq d) = 1 - c(d)$, which happens to end up being $\exp(-\sigma_t d)$. With these PDFs, the implementation of a multiple scattering homogeneous volume is straightforward (the `Transmittance` method is the same as `SingleScatterHomogeneousVolume`, and is not shown):

Listing 14. Multiple scattering homogeneous integrator

```
class MultiScatterHomogeneousVolume: public Volume {
public:
    MultiScatterHomogeneousVolume(Color &scatteringAlbedo, Color &extinction,
         ShadingContext &ctx) :
```

```
        Volume(ctx), m_scatteringAlbedo(scatteringAlbedo), m_extinction(extinction) {}
    virtual bool Integrate(RendererServices &rs, const Ray &wi, Color &L, Color
         &transmittance, Color &weight, Point &P, Ray &wo, Geometry &g) {
        if (!rs.GetNearestHit(Ray(m_ctx.GetP(), wi.dir), P, g))
            return false;

        // Distance to the end of the volume
        float totalDistance = Vector(P - m_ctx.GetP()).Length();

        // Find unbounded scatter location
        float xi = rs.GenerateRandomNumber();
        float scatterDistance = -logf(1.0 - xi) / m_extinction.ChannelAvg();

        // If scatter location is within the volume, reset P and g
        if (scatterDistance < totalDistance) {
            P = m_ctx.GetP() + scatterDistance * wi.dir;
            g = m_ctx.GetGeometry();
            transmittance = Color(exp(m_extinction.r * -scatterDistance), exp(m_extinction.g
                 * -scatterDistance), exp(m_extinction.b * -scatterDistance));
            Color pdf = m_extinction * transmittance;
            // Note that pdf already has extinction in it, so we should avoid the
            // multiply and divide; it is shown here for clarity
            weight = m_scatteringAlbedo * m_extinction / pdf;
        } else {
            transmittance = Color(exp(m_extinction.r * -totalDistance), exp(m_extinction.g *
                 -totalDistance), exp(m_extinction.b * -totalDistance));
            Color pdf = transmittance;
            weight = 1.0 / pdf;
        }
        L = Color(0.0); // No estimate of radiance
        wo = Ray(P, wi.dir);
        return true;
    }
    virtual Color Transmittance(RendererServices &rs, const Point &P0, const Point &P1) {
        float distance = Vector(P0 - P1).Length();
        return Color(exp(m_extinction.r * -distance), exp(m_extinction.g * -distance),
             exp(m_extinction.b * -distance));
    }
protected:
    const Color m_scatteringAlbedo;
    const Color m_extinction;
};
```

The key difference between this integrator and the single scattering integrator (besides returning a radiance of zero) is that the P and Geometry g values returned are no longer always the value returned by GetNearestHit. Instead, they may correspond to a scatter location within the volume. When this happens, we assume that `ShadingContext::GetGeometry` will return the same geometry which started the volume renderer and return that to the lighting integrator, so that the next ray will invoke the same volume integrator.

Note the `weight` returned by this integrator not only includes the inverse of the PDF, but also includes σ_s when scattering inside the volume. The reason for this is that we assume the lighting integrator as described in section 3.3 does not include this term required by equation 10 in its implementation of direct lighting for surfaces, so it is convenient to include it directly in the `weight` output. (We should also point out that the implementation of direct lighting for surfaces may also include the geometric term $\omega \cdot n$, which is not required for volumes; if so, this needs to be accounted for in the lighting integrator implementation, as the normal in a volume is typically defined to be zero length.)

An inspection of the `weight`, `pdf`, and `transmittance` terms will reveal that they all cancel each other out when multiplied together in the lighting integrator. This is an expected result of using a PDF normalized to the transmittance. While it may be tempting to eliminate all of these terms, any use of an alternate PDF or multiple importance sampling will results in terms that no longer cancel out.

Note that the implementation of single scattering in the previous section involves a volume integrator that guarantees a direct lighting sample will take place somewhere along the volume on the way to the next interface. In other words, in the single scattering case, our path tracer which takes n steps will now invoke direct lighting $2n - 1$ times: n invocations at the surfaces, and $n - 1$ invocations at the volumes in front of every hard surface. This is not the case for our implementation of multiple scattering: for n steps, the lighting integrator will invoke direct lighting n times. This "branching factor" of 2 in the single scattering integrator leads to a more converged (but more expensive) estimate of radiance for a path for the same number of bounces due to the increased number of lighting calculations involved. As we will describe later, due to the strong importance of contributions closer to the eye, it may be worthwhile to blend between these two regimes depending on how far along we are on the path.

We can conclude our survey of integration techniques by examining the situation of multiple scattering in heterogeneous volumes, which combines `MultiScatterHomogeneousVolume` with `SingleScatter-HeterogeneousVolume`:

Listing 15. Multiple scattering heterogeneous integrator

```
class MultiScatterHeterogeneousVolume: public Volume {
public:
   MultiScatterHeterogeneousVolume(int scatteringAlbedoProperty, Color &maxExtinction, int
       extinctionProperty, ShadingContext &ctx) :
      Volume(ctx), m_scatteringAlbedoProperty(scatteringAlbedoProperty),
          m_maxExtinction(maxExtinction.ChannelAvg()),
          m_extinctionProperty(extinctionProperty) {}
   virtual bool Integrate(RendererServices &rs, const Ray &wi, Color &L, Color
       &transmittance, Color &weight, Point &P, Ray &wo, Geometry &g) {
      Point P0 = m_ctx.GetP();
      if (!rs.GetNearestHit(Ray(P0, wi.dir), P, g))
         return false;
      float distance = Vector(P - P0).Length();
      bool terminated = false;
      float t = 0;
      do {
         float zeta = rs.GenerateRandomNumber();
```

```
            t = t - log(1 - zeta) / m_maxExtinction;
            if (t > distance) {
                break; // Did not terminate in the volume
            }

            // Update the shading context
            Point P = P0 + t * wi.dir;
            m_ctx.SetP(P);
            m_ctx.RecomputeInputs();

            // Recompute the local extinction after updating the shading context
            Color extinction = m_ctx.GetColorProperty(m_extinctionProperty);
            float xi = rs.GenerateRandomNumber();
            if (xi < (extinction.ChannelAvg() / m_maxExtinction))
                terminated = true;
        } while (!terminated);

        if (terminated) {
            P = m_ctx.GetP() + t * wi.dir;
            g = m_ctx.GetGeometry();
            Color transmittance = Color(1.0f);
            Color extinction = m_ctx.GetColorProperty(m_extinctionProperty);
            Color pdf = extinction * transmittance;
            Color scatteringAlbedo = m_ctx.GetColorProperty(m_scatteringAlbedoProperty);
            // Note that pdf already has extinction in it, so we should avoid the
            // multiply and divide; it is shown here for clarity
            weight = scatteringAlbedo * extinction / pdf;
        } else {
            Color transmittance = Color(1.0f);
            Color pdf = transmittance;
            weight = 1.0 / pdf;
        }
        L = Color(0.0); // No estimate of radiance
        wo = Ray(P, wi.dir);
        return true;
    }
    protected:
    const int m_scatteringAlbedoProperty;
    const float m_maxExtinction;
    const int m_extinctionProperty;
};
```

The result of using `MultiScatterHomogeneousVolume` with increasing numbers of bounces (as controlled by the lighting integrator) is illustrated in figure 19. Note that even with 6 bounces of light, the render on the right does not very much resemble a real cloud, in which light may have actually bounced tens to hundreds of times before reaching the eye. While we are closer to physically accurate rendering of such clouds using today's hardware, in reality productions may not be able to afford the render times associated with hundreds of bounces. Approximation techniques are thus still useful and often required. A recent technique which we call *contrast approximation* or *contrast control* (Wrenninge et al. 2013) offers a useful way to simulate higher order scattering effects in an art directable fashion. This technique of lowering the scattering coefficient σ_s by an extinction multiplier a can be easily added to the `Transmittance` method of the multiple scattering integrators. Combining this with ideas from (Wrenninge 2015), we have found that by exposing the current ray depth i to the volume integrator (which we would need to make available from the lighting integrator), we can make use of the adjusted scattering coefficient $a^i\sigma_s$; this is computationally inexpensive and results in only a single parameter to adjust.

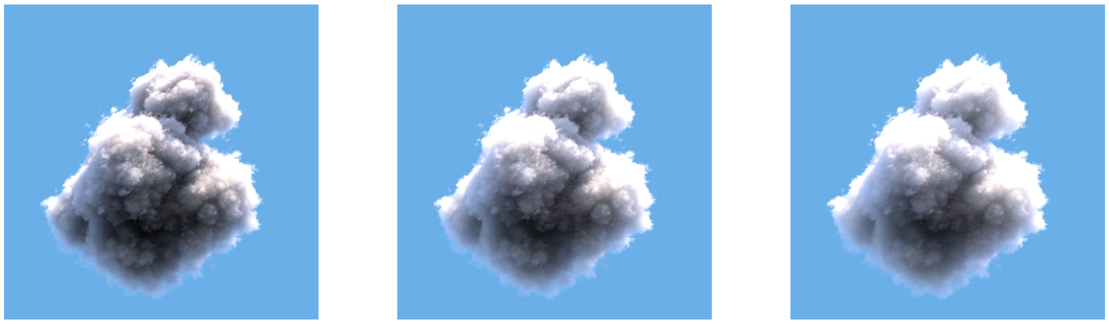


Fig. 19. Multiple scattering in clouds (highly anisotropic media). Left: 2 bounces of multiple scattering; center: 4 bounces of multiple scattering; right: 6 bounces of multiple scattering.

3.8 Emission

In the previous discussion, we have deliberately ignored the topic of emission and have assumed that we are dealing with volumes whose absorption coefficient $\sigma_a = 0$, and thus $\sigma_t = \sigma_s$. We have thus been parameterizing the volumes only with the extinction coefficient σ_t and the scattering albedo α.

For the tracking approach to integration, we can partially reintroduce emission into the implementation of `SingleScatterHomogeneousVolume` by adding the absorption coefficient $\sigma_a(\mathbf{x}_t)$ and volume emission color $L_e(\mathbf{x}_t, \omega)$ as parameters to the constructor. We have been picking values of t' for a scattering location based on σ_t; we can continue to use the same approach to determine the distance traveled by a photon before it is either absorbed or scattered, since this continues to be based on σ_t. After choosing t' we perform the extra random choice to determine the type of collision event that occurred there, using $P_a(\mathbf{x}) = \frac{\sigma_a}{\sigma_t}$ as shown in equation 17:

Listing 16. Extra step in homogeneous single scattering integrator to deal with emission

```
// Compute sample location for scattering or absorption, based on the PDF
// normalized to the total transmission
float xi = rs.GenerateRandomNumber();
float collisionDistance = -logf(1.0f - xi * (1.0f - transmittance.ChannelAvg())) /
    m_extinction.ChannelAvg();

// Determine if an absorption event or a scattering event has occurred
```

```
float zeta = rs.GenerateRandomNumber();
if (zeta < (m_absorptionCoefficient.ChannelAvg() / m_extinction.ChannelAvg()))
{
    Color Tr(exp(m_extinction.r * -collisionDistance), exp(m_extinction.g *
        -collisionDistance), exp(m_extinction.b * -collisionDistance));
    L = m_emissionColor * Tr;
} else {
    // Set up shading context to be at the scatter location
    // etc.
}
```

For the PDF approach to integration, listing 13 can be extended by considering the absorption integral $\int_{t=0}^{d} T(t)\sigma_a(\mathbf{x}_t)L_e(\mathbf{x}_t,\omega)dt$ separately from the scattering integral. The $T(t)$ component of the integrand is already computed when constructing the discrete PDF for single scattering. If we assume that $\sigma_a(\mathbf{x}_t)$ and $L_e(\mathbf{x}_t,\omega)$ are both relatively cheap to compute (because they are both simply properties of the volume) then we may as well compute the answer to the absorption integral by using quadrature instead of computing a PDF: there are no expensive terms that need approximation, and the quadrature calculation for absorption can take place at the same time as the construction of the scattering PDF.

Unfortunately, this is not a complete implementation of absorption. The in-scattered radiance L_s in the VRE also must take into account the light emitted by the volume itself (as well as all other emitting volumes in the direction of ω'), and we are missing this contribution. In fact, the direct lighting on all other geometry in the scene must be augmented by the L_e coming from volumes, considerably complicating matters.

One approach to this problem is to augment the volume interface with an `Emission` method, which is similar to the `Transmittance` method but instead of returning the transmittance, it must return an estimate of the emitted radiance between two points. Once we have such a method on all volumes, then in the same way that the implementation of the `RendererServices`'s `GenerateLightSample` and `EvaluateLightSample` must be augmented to call the `Transmittance` method of any volumes along the transmission ray, we can similarly require that an additional call to `Emission` be made to any volumes in order to gather the emission. This emission is added to the `L` radiance returned by the `RendererServices` methods. This ensures that emitted radiance is correctly accounted for in direct lighting, while the change to the integrators similar to that shown in listing 16 handles the emitted radiance for indirect lighting.

There is still one more additional wrinkle if we consider renderers whose implementation of `Render-erServices::GenerateLightSample` involves importance sampling of lights. In the case of volumetric emission, the volume is essentially a light source and may itself need to be considered during the importance sampling. Consider the case where a scene is illuminated by fire or explosive effects; in order to reduce noise, the sampling of lighting for direct illumination should importance sample the emission function of the volume, as well as potentially the transmittance to the volume. One approach, particularly useful when the volume is heterogeneous, is to voxelize the light emission distribution; for further discussion, see (Villemin and Hery 2013). In the absence of such functionality, the substitution of light sources in the middle of the volume solely for the purposes of "steering" the light importance sampling may be required, and has certainly been used in recent production rendering.

3.9 Overlapping volumes

So far we have glossed over the correct treatment of overlapping volumes, but in production rendering it is important to handle this correctly. A simple case is a glass object with homogeneous absorption, embedded inside a cloud of smoke. A more complicated case is a large bank of clouds; due to limited computational resources, it may be desirable to run simulation on smaller clouds, and combine them to a merged result at render time.

It is useful to establish situations where overlapping volume integration is actually not needed. The aforementioned glass object inside smoke is one such case: the glass object is not actually permeable to smoke, so in fact the associated volume integrator never needs to worry about any situation where it overlaps with another volume. We can enshrine this situation in our system by adding a query method to the `Material`, where it can be interrogated as to whether it can participate in overlapping volume integration. If it chooses not to, everything proceeds as described in the system so far.

It is also important to note that for the case of only calculating the beam transmittance between two points (as described in section 3.4), we do not actually need to worry about whether or not two volumes overlap. Their individual transmittances can be calculated in isolation and accumulated by multiplication without error, even if they overlap each other.

Otherwise, we are in the situation where two volumes do overlap and require overlapping integration — two overlapping puffs of smoke, for example. In production volume rendering, it is typical to assume that such overlapping volumes are comprised of particles that do not react to each other — the particles simply co-exist in the same region of space. Because of this non- interaction model, we need to account for the increase in particle density and the corresponding increase in the probability of particle interaction, which only requires that the combined volume has an absorption coefficient equal to the sum of the absorption coefficients; the same holds for the scattering coefficient, and therefore the extinction coefficient. While these coefficients are additive, the same is not true of the radiance computed from each overlapping volume. This is due to the presence of the exponential in the transmittance term (equation 11). Intuitively, we cannot consider the light transport from each volume in isolation: the mere presence of particles from any other volume has an absorptive or scattering effect on the outgoing radiance. This means that it is physically incorrect to extend the light integrator as described in section 3.3 to iterate over the list of volumes that are tracked by a ray and simply sum the radiances. We cannot simply adjust the transmittance after the fact either, as the radiance estimate from each volume may have included a transmittance term up to the scatter location (such as in the case of the single scattering integrator).

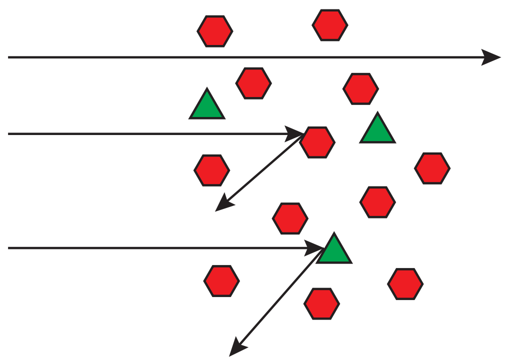


Fig. 20. A low density volume (green triangles, σ_t^g) overlapping a high density volume (red hexagons, σ_t^r). The chance of collision with either volume is the sum of the extinction coefficients $\sigma_t^g + \sigma_t^r$. If a collision occured, the chance that the collision was with the green volume is $\frac{\sigma_t^g}{\sigma_t^g+\sigma_t^r}$.

One solution this problem is to choose a single primary volume integrator, which has full access to the properties of all other volumes which coexist over the interval, and is responsible for a single result which takes into account all properties of the overlapping volumes. To pick the single integrator correctly, we require a priority scheme for these overlapping volume integrators. A simple homogeneous absorption-only volume integrator cannot be trivially extended to deal with heterogeneous scattering, whereas a heterogeneous integrator can correctly deal with the absorption-only integrator while integrating its own results. In essence, the priority scheme ranks more capable integrators higher than less capable integrators, and is used by `Ray::GetVolume` to pick the most capable integrator from the list of volumes: the implementation in listing 5 is altered to find the most recently entered volume with sufficient priority that can handle all other volumes.

Once we select a single integrator, we need to expose a list of properties from the `Volume` interface necessary for an integrator to be able to perform a summed integration. Consider one of the scattering heterogeneous integrators, which implement delta tracking. In the combined case, delta tracking would require the sum of the maximum extinction coefficients in order to take the delta tracking step, and the sum of the extinction coefficients at the potential interaction location. Each volume therefore needs to expose the ability to query the maximum extinction coefficient and the local extinction coefficient.

Once a scattering location has been chosen, the transmittance can be computed using the summed extinction properties of each volume. Now we need to resolve how to deal with scattering in the combined volumes. By appeal to the non- interacting particle model, we can simply decide that the collision event has occurred with a single particle from a single volume, and the only question is deciding which volume. The probability of interaction with a volume is simply the extinction coefficient of that volume divided by the sum of the extinction coefficients of all the volumes. Therefore, we only need to build a CDF from the scattering coefficients and use a random number to pick one of the volumes. Once we have picked a volume, a phase function from that volume can be used which abstracts away all further details of the scattering away from the volume integrator — in essence, the integrand of the in-scattering integral in equation 7 is now restricted to a single volume.

The following listing shows a modified version of `SingleScatterHomogeneousVolume` which can account for other overlapping homogeneous volumes. Note the changes to the `Volume` interface: the addition of the `CanOverlap` and `GetOverlapPriority` methods, along with a list of common property accessor methods. We assume that heterogeneous volumes have an overlap priority greater than one. The lighting integrator is now responsible for creating instances of all `Volume` over the list of all `Materials` on the ray and storing them on the shading context, so that `ShadingContext::GetAllVolumes` can return them. A render demonstrating the correct overlapping behavior is shown in figure 21.

Listing 17. Single scattering homogeneous integrator extended for overlapping volumes

```
cclass SingleScatterHomogeneousVolume: public Volume {
public:
    SingleScatterHomogeneousVolume(Color &scatteringAlbedo, Color &extinction,
         ShadingContext &ctx) :
        Volume(ctx), m_scatteringAlbedo(scatteringAlbedo), m_extinction(extinction) {}
    virtual bool Integrate(RendererServices &rs, const Ray &wi, Color &L, Color
         &transmittance, Color &weight, Point &P, Ray &wo, Geometry &g) {
        if (!rs.GetNearestHit(Ray(m_ctx.GetP(), wi.dir), P, g))
            return false;

        // Sum the extinction over all volumes. Build a CDF of their densities
        float distance = Vector(P - m_ctx.GetP()).Length();
        std::vector<Volume*> &volumes = m_ctx.GetAllVolumes();
        std::vector<float> cdf;
        Color summedExtinction(0.0);
        for (auto v = volumes.begin(); v != volumes.end(); ++v) {
            summedExtinction += (*v)->GetExtinction();
            cdf.push_back(summedExtinction.ChannelAvg());
        }
        for (auto f = cdf.begin(); f != cdf.end(); ++f) {
            *f /= summedExtinction.ChannelAvg();
        }

        // Transmittance over the entire interval, computed over all volumes
        transmittance = Color(exp(summedExtinction.r * -distance), exp(summedExtinction.g *
             -distance), exp(summedExtinction.b * -distance));

        // Compute sample location for scattering, based on the PDF
        // normalized to the total transmission
        float xi = rs.GenerateRandomNumber();
        float scatterDistance = -logf(1.0f - xi * (1.0f - transmittance.ChannelAvg())) /
             summedExtinction.ChannelAvg();

        // Set up shading context to be at the scatter location
        Point Pscatter = m_ctx.GetP() + scatterDistance * wi.dir;
        m_ctx.SetP(Pscatter);
        m_ctx.RecomputeInputs();

        // Pick one of the overlapping volumes
        float zeta = rs.GenerateRandomNumber();
        unsigned index;
        for (index = 0; index < cdf.size(); ++index) {
            if (zeta > cdf[index]) break;
        }

        // Compute direct lighting with light sampling and phase function sampling
        BSDF *phaseBSDF = volumes[index]->CreateBSDF(m_ctx);
        L = Color(0.0);
        Color lightL, bsdfL, beamTransmittance;
        float lightPdf, bsdfPdf;
        Vector sampleDirection;
        rs.GenerateLightSample(m_ctx, sampleDirection, lightL, lightPdf, beamTransmittance);
        phaseBSDF->EvaluateSample(rs, sampleDirection, bsdfL, bsdfPdf);
        L += lightL * bsdfL * beamTransmittance * rs.MISWeight(1, lightPdf, 1, bsdfPdf) /
             lightPdf;

        phaseBSDF->GenerateSample(rs, sampleDirection, bsdfL, bsdfPdf);
        rs.EvaluateLightSample(m_ctx, sampleDirection, lightL, lightPdf, beamTransmittance);
        L += lightL * bsdfL * beamTransmittance * rs.MISWeight(1, lightPdf, 1, bsdfPdf) /
             bsdfPdf;

        Color Tr(exp(m_extinction.r * -scatterDistance), exp(m_extinction.g *
             -scatterDistance), exp(m_extinction.b * -scatterDistance));
        L *= (volumes[index]->GetScatteringAlbedo() * volumes[index]->GetExtinction() * Tr);

        weight = Color(1.0);
        wo = Ray(P, wi.dir);
        return true;
    }
    virtual bool CanOverlap() const { return true; }
    virtual int GetOverlapPriority() const { return 1; }
    virtual Color GetExtinction() const { return m_extinction; }
    virtual Color GetScatteringAlbedo() const { return m_scatteringAlbedo; }
    virtual BSDF* CreateBSDF(ShadingContext &ctx) const { return new
         IsotropicPhaseBSDF(ctx); }
    virtual Color Transmittance(RendererServices &rs, const Point &P0, const Point &P1) {
        float distance = Vector(P0 - P1).Length();
        return Color(exp(m_extinction.r * -distance), exp(m_extinction.g * -distance),
             exp(m_extinction.b * -distance));
    }
protected:
    const Color m_scatteringAlbedo;
    const Color m_extinction;
};
```

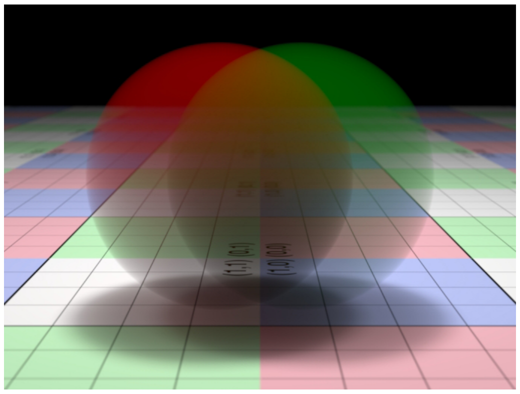


Fig. 21. Two overlapping single scattering homogeneous volumes. Each volume has a different density, scattering albedo, and phase function. The use of random selection based on a CDF built using each volume's local extinction coefficient correctly accounts for each volume's different scattering properties.

A problematic situation arises when the camera is itself inside a volume, such as would happen in an underwater scene. In the system described so far, we only consider a volume to exist when a ray hits a piece of geometry with outward facing normals. There is no such geometry to intersect when the camera is inside the volume, and we must add a special case to deal with this situation. At minimum, user intervention is required to specify a volume-capable `Material` bound to the camera itself; all rays originating from the camera are now immediately tagged as having entered the material as soon as they are fired. A similar situation exists for bidirectional path tracing when tracing rays originating from the light that are inside a volume.

Our discussion of overlapping volumes was inspired by the treatment of nested dielectrics by Schmidt and Budge (2002), which describes a system for rendering refractive objects within each other. In fact, a system which implements index of refraction tracking and intersection priorities as described in that paper can be easily extended to track volumetric materials: a fish inside water contained within glass can be robustly rendered with volumetric scattering (the thin body of the fish), and with absorption (liquid and glass), while correctly tracking the relative indices of refraction at each interface, and not suffering from precision problems at the geometric boundaries.

However, when rendering overlapping volumes which have identical scattering properties, and which do not require boundary effects (i.e. they use a `TrivialBSDF` from listing 4), our system suffers from performance problems as compared to the case of the equivalent single non-overlapping volume. To deal with two overlapping volumes, we require three separate intervals of volume integration: two intervals have only one volume, while the third has two overlapping volumes. Our simple single scattering integrators always guarantee a lighting estimate at each interval, so we have now effectively tripled the cost of direct lighting by creating three intervals. On top of this, the cost of dealing with overlapping volumes requires density evaluations in each volume, even if we can get away with only integrating and lighting one of them. The situation grows rapidly worse with many overlapping volumes. Section 6 describes a system for dealing with this particular case.

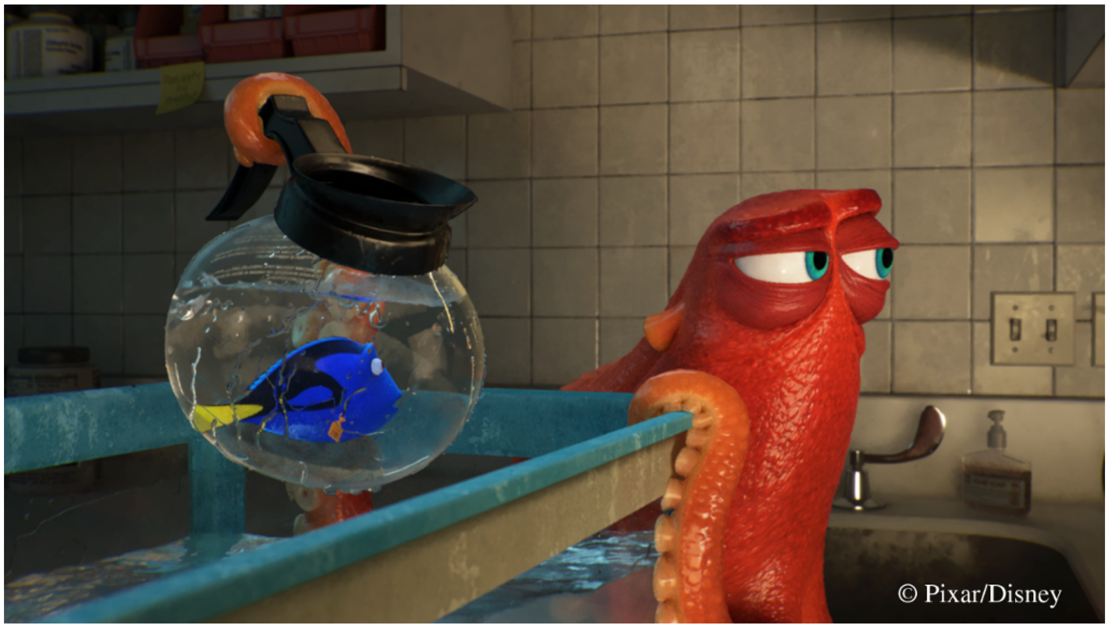


Fig. 22. A scene from Finding Dory (2016) involving nested dielectrics and volumes. Dory's material includes a volumetric single scattering component. Hank's subsurface scattering also took place in a volumetric framework similar to that described in this chapter.

4 OPTIMIZATION

4.1 Multiple sampling

Like any other compute-intensive problem, renderers can derive large speed gains by taking advantage of locality of reference. The benefit of locality can be achieved in multiple domains: on-chip cache, main memory, or disk. This becomes very important when dealing with complicated volume assets that may involve gigabytes of voxel data. Even highly optimized routines for organizing access to this data benefit from coherent access patterns.

In terms of locality, the tracking approach for heterogeneous volumes are much worse than the PDF approach: the use of the majorant allows the tracker to skip over large parts of the volume, which leads to more random access patterns. Even in a rendering architecture which batches rays for shading, the number of steps taken by each ray can vary significantly and the integrator can often end up having to finish single rays that take more steps than its siblings in the same batch.

We can at least improve locality significantly in the single scattering integrators by taking more than a single estimate of the direct lighting. Both the tracking and PDF implementations in section 3.6 can be easily extended to add up multiple samples of direct lighting over a single volume interval, and divide by the number of samples taken for a higher quality estimate of in-scattered radiance. This will almost certainly be faster than firing the equivalent number of rays from the camera: the integrator can calculate the distance to the end of the volume interval by firing a single ray, and the locality of access to the volume texture will be improved significantly. The PDF approach to single scattering benefits the most from taking multiple samples, since the high cost of building the discrete PDF can be amortized over multiple samples in the volume.

In the case of the multiple scattering integrator described in section 3.7, the volume integrator is unable to trivially perform multiple sampling as it cannot return multiple hit points to the light integrator. In practice, we wouldn't want to do this anyways; in our system, we imagine that the light integrator would have its own set of rules governing how many indirect rays to spawn, and in practice this may involve a set of heuristics such as ray depth, usage of Russian roulette, etc., not all of which should be exposed to the volume integrator.

However, for many scenes, it is well worth considering a hybrid between the single scattering integrators and multiple scattering integrators. It is often the case that the first few bounces of light in a volume are the most important, and we want to focus more computational resources onto those bounces for a higher quality estimate of direct lighting. After those first few bounces, however, the relative importance of each additional path becomes less important to the eye and it is more important to take faster, lower quality estimates of direct lighting.

The multiple scattering integrators we have described so far make no estimate of the in-scattered radiance, preferring only to return the next hit point to the light integrator. If we instead alter the multiple scattering integrator's `Integrate` routine to selectively perform an estimate of the in- scattered radiance — in effect, combining their implementation with the single scattering integrators — then in conjunction with the lighting integrator, the integrator has now delivered a higher quality estimate of the direct lighting. We would need to augment our interfaces to the integrator to deliver the current ray depth or perhaps the current throughput to the eye, in order to allow the volume integrator to perform this extra computation only when it is important to do so.

Note there is now a new complication: the multiple scattering integrator and the lighting integrator are both computing direct lighting, and we are now injecting too much radiance into the system unless extra steps are taken. As the `weight` calculated by the volume integrator was used to correctly scale the contribution of both the direct and indirect contributions calculated by the light integrator, we need to separate this weight out into two separate terms: a new term to scale the lighting integrator's direct lighting contribution, and the old term still used to scale the lighting integrator's indirect lighting.

4.2 Importance sampling for Single Scattering

The basic techniques for importance sampling the transmittance presented in earlier section are effective when little is known about the incoming radiance. However for single scattering, that is light that only has a single scattering event the light source and the camera, it is possible to use additional importance sampling techniques to further reduce noise. In this section, we will review some of these techniques, and update the reader on best practices for making them fit into a general volume rendering framework.

4.2.1 Uniform sampling and MIS. A drawback with tracking methods is that their PDF is proportional to the transmittance $T(t)$ of a volume with unbounded interval $0 \leq t < \infty$. In practical terms, this means that `SingleScatterHeterogeneousVolume`, which implements delta tracking, is highly unlikely to terminate in a bounded volume when σ_t is low. For such low density volumes, this leads to very high variance in the estimate for direct lighting, particularly when the volume is lit by a very bright light source. Compare this to `SingleScatterHomogeneousVolume`, which will always find a scattering location in the volume because the PDF is normalized to the total transmittance over the volume $T(d)$ and can thus be perfectly sampled.

Because it is the transmittance-based PDF which is the source of the problem, we look for an alternate sampling strategy to use. The simplest such strategy is uniform sampling (sometimes referred to as distance sampling): simply pick a random t where $0 \leq t < d$. We have avoided this strategy for homogeneous volumes because it is almost always inferior to density sampling when σ_t is constant, but as a fallback mechanism for sparse heterogeneous volumes it is useful. We also introduce it here as a way to bring multiple importance sampling (MIS) (Veach 1997) into our discussion. We have already been using MIS to combine samples from both the phase function and the light source during direct lighting; because we now have two sampling strategies for t to consider, and it may not be clear which one is preferable, we can instead just use both strategies simultaneously and combine the results with MIS. (In practice, if the lower bound of the density over the volume is known, we can avoid MIS and switch over to uniform sampling when the lower bound of the density is below some threshold. This ties into the general discussion of generating bounds for tracking methods in section 4.4.2.)

To perform MIS between uniform sampling and delta tracking, we note that the PDF for uniform sampling is simply:

$$p_u(t) = \begin{cases} \frac{1}{d} & \text{if } 0 \leq t < d \\ 0 & \text{if } t \geq d \end{cases} \tag{30}$$

The PDF for delta tracking is essentially given by equation 14: $p_t(t)$ is the product of $\sigma_t(t)$ and $T(t)$, the latter of which is estimated by delta tracking itself.

The standard MIS approach to blending samples requires the computation of a weight for each sample, which requires calculation of the PDF $p(t)$ for every technique. Consider the case of drawing one sample for uniform sampling at location t_u, and one sample by delta tracking at location t_t: we need to calculate $p_u(t_u)$, $p_u(t_t)$, $p_t(t_u)$, and $p_t(t_t)$. The complication here is the need to compute two PDFs for delta tracking: $p_t(t_u)$ and $p_t(t_t)$. Rather than performing two separate invocations of delta tracking in order to estimate the two transmittances $T(t_u)$ and $T(t_t)$ necessary for the PDFs, we can estimate both transmittances in a single delta tracking invocation using a priori knowledge of t_u as follows: if delta tracking proceeds past $t > t_u$ (and thus has not found a location t_t) then $T(t_u) = 1$. If delta tracking stops at a distance $t_t < t_u$, then we can $T(t_u) = 0$. If delta tracking terminates because $t > d$, then $T(t_u) = T(t_t) = 1$.

This technique of combining delta tracking with another importance sampling metric is broadly applicable; for example, we can replace uniform sampling with equiangular sampling as described in the next section. Note that in order to perform delta tracking to provide both transmittance estimates, we require knowing the location of the other sample first.

A modification of SingleScatterHeterogeneousVolume to perform MIS is shown below.

Listing 18. Single scattering heterogenous integrator extended with MIS

```
 Color SingleScatterHeterogeneousVolume::directLighting(RendererServices &rs) {
    Color scatteringAlbedo = m_ctx.GetColorProperty(m_scatteringAlbedoProperty);
    Color extinction = m_ctx.GetColorProperty(m_extinctionProperty);
    IsotropicPhaseBSDF phaseBSDF(scatteringAlbedo, m_ctx);
    Color L = Color(0.0);
    Color lightL, bsdfL, beamTransmittance;
    float lightPdf, bsdfPdf;
    Vector sampleDirection;
    rs.GenerateLightSample(m_ctx, sampleDirection, lightL, lightPdf, beamTransmittance);
    phaseBSDF.EvaluateSample(rs, sampleDirection, bsdfL, bsdfPdf);
    L += lightL * bsdfL * beamTransmittance * extinction * rs.MISWeight(1, lightPdf, 1,
        bsdfPdf) / lightPdf;
    phaseBSDF.GenerateSample(rs, sampleDirection, bsdfL, bsdfPdf);
    rs.EvaluateLightSample(m_ctx, sampleDirection, lightL, lightPdf, beamTransmittance);
    L += lightL * bsdfL * beamTransmittance * extinction * rs.MISWeight(1, lightPdf, 1,
        bsdfPdf) / bsdfPdf;
    return L;
 }
virtual bool SingleScatterHeterogeneousVolume::Integrate(RendererServices &rs, const Ray
    &wi, Color &L, Color &transmittance, Color &weight, Point &P, Ray &wo, Geometry &g)
    {
    Point P0 = m_ctx.GetP(), Pl;
    if (!rs.GetNearestHit(Ray(P0, wi.dir), P, g))
        return false;
    Color extinction;
    float distance = Vector(P - P0).Length();
    bool terminated = false;
    float t = 0;

    do {
        float zeta = rs.GenerateRandomNumber();
        t = t - log(1 - zeta) / m_maxExtinction;
        if (t > distance) {
            break; // Did not terminate in the volume
        }

        // Update the shading context
        Pl = P0 + t * wi.dir;
        m_ctx.SetP(Pl);
        m_ctx.RecomputeInputs();

        // Recompute the local extinction after updating the shading context
        extinction = m_ctx.GetColorProperty(m_extinctionProperty);
        float xi = rs.GenerateRandomNumber();
        if (xi < (extinction.ChannelAvg() / m_maxExtinction))
            terminated = true;
    } while (!terminated);

    // Generate a uniform sampling location
    float uniformDistance = rs.GenerateRandomNumber() * distance;
    L = Color(0.0);
    // Lighting and transmittance estimator from delta tracking
    float uniformPDF = 1.0 / distance, deltaPDF, w;
    if (terminated) {
        extinction = m_ctx.GetColorProperty(m_extinctionProperty);
        deltaPDF = extinction.ChannelAvg(); // extinction x transmittance, which is one
        w = rs.MISWeight(1, deltaPDF, 1, uniformPDF) / deltaPDF;
        L += directLighting(rs) * w;
        transmittance = Color(0.0f);
    } else
        transmittance = Color(1.0f);

    // Lighting and transmittance estimator from uniform sampling
    Pl = P0 + uniformDistance * wi.dir;
    m_ctx.SetP(Pl);
    m_ctx.RecomputeInputs();
    if ((terminated && t > uniformDistance) || !terminated) {
        extinction = m_ctx.GetColorProperty(m_extinctionProperty);
        deltaPDF = extinction.ChannelAvg();
    } else
        deltaPDF = 0.0;
    w = rs.MISWeight(1, deltaPDF, 1, uniformPDF) / uniformPDF;
    L += directLighting(rs) * w;
    transmittance += Color(0.0f) * w;

    weight = Color(1.0);
    wo = Ray(P, wi.dir);
    return true;
}
```

4.2.2 Equiangular Sampling. The general volume rendering equation is not easy to reason about because it contains many non-analytical integrals. We simplify the problem by focusing on the case with the fewest number of integrals, single scattering from a singular point light source in a homogeneous medium. Rewriting Equation 9 we obtain:

$$L(x, \vec{\omega}) = \int_a^b \sigma_s e^{-\sigma_t (t + \Delta + \sqrt{D^2 + t^2})} \frac{\Phi}{D^2 + t^2} dt \tag{31}$$

Figure 23 describes the involved parameters. Note that to simplify the notation, we omit the inclusion of visibility and phase function in the equation. We also reparameterize t so the origin is at the orthogonal projection of the light onto the ray. This change modifies the integration bounds a and b (which can be negative) and adds an extra term Δ which is the distance between the real ray origin and the new one. The use of integration bounds a and b is also convenient when dealing with spotlights as we can restrict integration only to the part of the ray that intersects the light's cone.

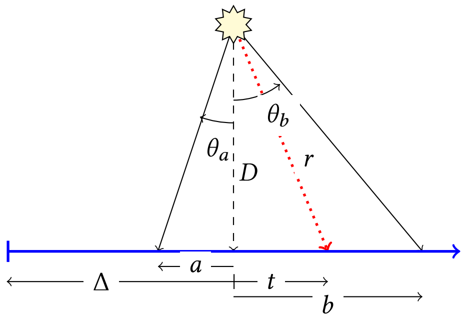


Fig. 23. Single scattering from a point light source

While the transmission term is bounded by 1, the incoming radiance Φ/r^2 presents a weak singularity that starts to dominate as rays approach the light source. This reveals why distance sampling (distributing samples proportionally to $e^{-t\sigma_t}$) can be suboptimal. Distance sampling focuses most samples close to the viewer, missing the visually important light source (see Figure 24). In fact, the results for distance sampling can be made almost arbitrarily bad by increasing σ_t, pushing the sample points closer to the viewer and away from the light.

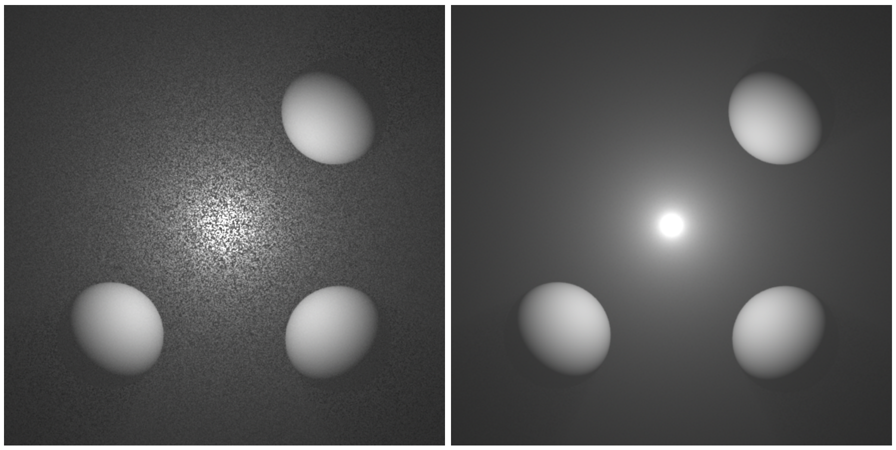
We can importance sample proportionally to the $1/r^2$ term instead to focus more samples where the incoming light is strong and cancel out the weak singularity directly. This results in a Cauchy distribution which has the following normalized pdf and associated sampling function over the interval $[a, b]$ (ξ is a random number in $[0, 1)$):

$$\text{pdf}(t) = \frac{D}{(\theta_b - \theta_a)(D^2 + t^2)} \tag{32}$$

$$t(\xi) = D \tan\left((1 - \xi)\theta_a + \xi\theta_b\right) \tag{33}$$

$$\theta_x = \tan^{-1} x/D \tag{34}$$

Equation 33 reveals that this technique linearly interpolates the angle between the bounds of integration, so we refer to this technique as *equiangular sampling*. The results in Figure 24(b) show the dramatic improvement compared to distance sampling. Due to the smoothness of the transmittance term, the noise present using our method quickly vanishes and yields nearly noise free images with as few as 16 paths per-pixel.



(a) Distance Sampling (b) Equiangular Sampling

Fig. 24. Point light source in homogeneous media (16 paths/pixel, each method rendered in 2 seconds)

It should be noted that the equations above still contain a singularity when $D = 0$. This corresponds to the case where the viewing ray passes through the point light exactly. While this may seem like a corner case that could be ignored, it can occur in the common "flashlight" setup where a point light source is parented to the camera. To solve this case we can use the following specialization:

$$\mathrm{pdf}(t) = \frac{ab}{(b-a)t^2} \tag{35}$$

$$\mathrm{t}(\xi) = \frac{ab}{b+(a-b)\xi} \tag{36}$$

4.2.3 Equiangular Sampling from Area lights. It turns out that the scheme above generalizes very well to area lights. Even though the integral over a light's area is more complicated, the solid angle subtended by the light source varies as $1/r^2$. Therefore, if the renderer supports solid angle sampling from a point towards the light source, it suffices to choose a point along the ray using equiangular sampling first as in the point light case, and then applying the solid-angle sampler from the chosen point.

The original description of equiangular sampling (Kulla and Fajardo 2012) gives another strategy for arbitrary lights that do not have convenient solid angle based importance sampling schemes. One can still benefit from equiangular sampling by first choosing the point on the light source (which is independent of the ray being traced) and then applying equiangular sampling from this chosen point. In practice however, this approach is always inferior to schemes that can importance sample the solid angle of the light source directly. We would therefore recommend investing the time in such samplers (Ureña et al. 2013).

4.2.4 Equiangular sampling in Heterogeneous media. The last generalization we must discuss is how to handle heterogeneous media. The presence of an integral for transmittance requires the use of the techniques from section 3.6. The basic idea is to store a representation of the transmittance along the viewing ray such that we may evaluate the transmittance at any point along the ray. This in turn means we can choose points along the ray freely and thus combine heterogeneous media with equiangular sampling.

A particularly simple way (though not necessarily the most efficient) to achieve this is the classic ray marching algorithm (Perlin and Hoffert 1989). The ray is divided into shorter segments over which we can assume the medium properties are locally constant. This produces a piecewise exponential representation of the volume that can be quickly evaluated at any distance t along the ray via binary search. This means that we only need to march along the ray once and then can perform as many lighting calculations as needed (taking multiple samples per light, or evaluating multiple light sources). Because the computation of transmittance being *decoupled* from the lighting calculations, this technique was called *decoupled ray marching*. It should be noted that this technique may be used with the unbiased tracking methods presented in the previous section.. The important point is that transmittance can be cached, and subsequently evaluated.

When dealing with lights that are both inside and outside the volume, neither equiangular sampling or transmittance based sampling may be optimal on their own. Equiangular sampling works when close to the light source while transmittance based sampling works well for lights outside the volume. Multiple importance sampling (Veach and Guibas 1995) is a natural way to combine equiangular sampling with density sampling (see Figure 26). As explained above, binary search can also be used to importance sample from the representation of transmittance computed during the initial ray march.

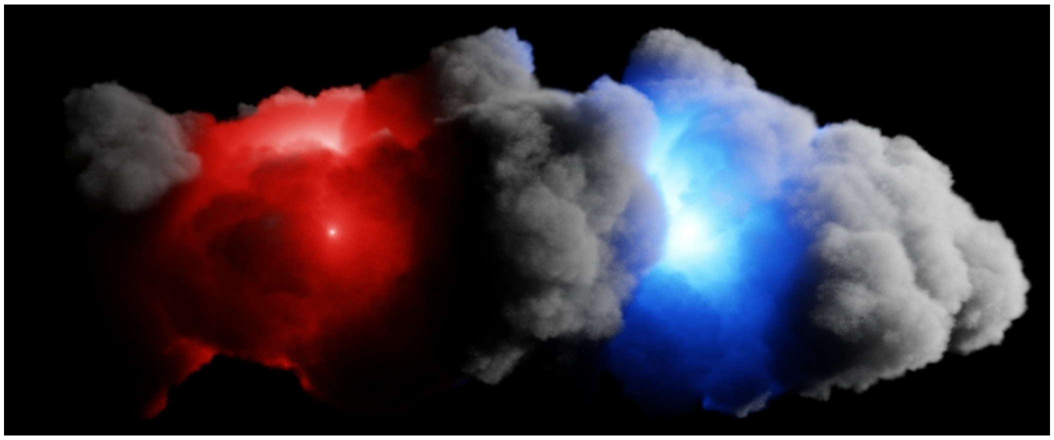


Fig. 25. Heterogeneous medium rendered using decoupled ray marching (16 paths/pixel, ~5 min). Previous stochastic methods would have a hard time capturing single scattering from the light sources embedded in the medium as seen in Figure 24.

The biggest drawback of using classical ray marching is that it introduces a bias in the calculation. When working with voxel based volumes, the bias is not noticeable because the step size can be fixed to be on the order of a single voxel. However efficiency suffers because any homogeneous regions still perform many lookups. The more recent work of Novák et al. (2014) can instead be used to obtain a fully unbiased algorithm. The *residual ratio tracking* method (see Section 4.4.1) creates a similar piecewise exponential representation for transmittance and can be mostly used as a drop-in replacement. It should be noted that the original method does allow importance sampling proportionally to $\sigma_s(\mathbf{x}_t)T(t)$ while unbiased methods only support sampling proportionally to $T(t)$.

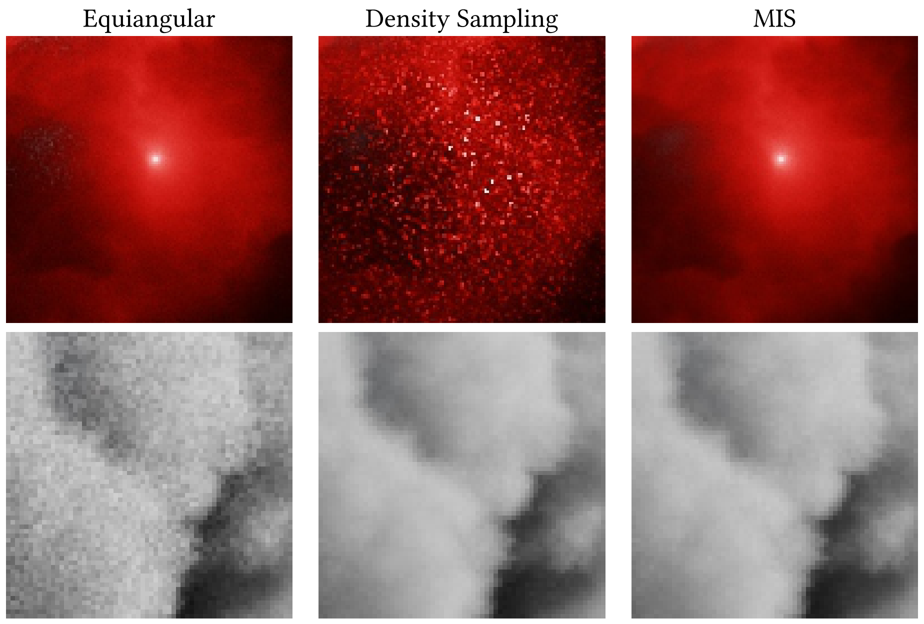


Fig. 26. Closeups of Figure 25. MIS combines the strengths of each line-sampling technique.

4.2.5 Implementing equiangular sampling in a renderer. When implementing equiangular sampling in a renderer, a few choices will present themselves. We discuss some of these choices along with how they were handled in the original implementation (Kulla and Fajardo 2012).

Every ray has a choice to integrate direct lighting for the volume hits or surface hits when both are present. The choice that introduces the least variance is simply to compute direct lighting for both. However, there are many situations where the volume is either thick enough for the surface to have low contribution, or for the volume to be thin enough that its contribution is negligible. In such situations, we can regain a bit of efficiency at the cost of variance by stochastically choosing if we should shade the surface or the volume.

Another important optimization for production rendering concerns the tracking of transparent surface within a volume. A surprisingly common case is that of fur inside smoke. The naive approach of repeating a direct lighting loop for each volume segment and surface hit at each transparent interaction can quickly become prohibitive. Instead, we can recognize that the change in transmittance from transparent surfaces is trivially folded into the CDF representation discussed in Section 4.2.4. This means one should keep tracing a ray through multiple transparent hits, deferring both surface and volume integration until full opacity is reached, and only then execute a single (heterogeneous) integration along a segment that will have discrete changes in opacity in addition to the continuous one provided by the volume.

4.3 Importance sampling for Multiple Scattering

In high albedo mediums such as clouds or snow, many bounces might be required before we exit the volume. In these cases, performing a light integration for each path segment can be prohibitive, particularly when each segment is short. Much like the case of transparent surface hits described in Section 4.2.5, we can relax the idea of generating a sampling position along the ray to generating a sampling position along the entire *path*. This requires additional bookkeeping, but one can build a single discrete cdf representing all the scattering segments encountered along a path. To lower variance, the first and last segments can be always integrated, while all intermediate segments can perform a single lighting calculation. One can also employ splitting and perform multiple lighting calculations across the entire path, but then care should be taken that the number of segments being sampled is larger than the number of samples requested. The exact cross over point where this technique pays off is highly implementation dependent.

4.4 Tracking improvements

4.4.1 Residual and ratio tracking. As mentioned in section 2.4.1, the ratio tracking and residual tracking methods, introduced by Novák et. al (2014), extend delta tracking. The improvement in both of these methods is a better estimator of the transmittance. Instead of terminating in the volume at a tentative collision point and returning a binary 1 or 0 estimate of the transmittance, ratio tracking continues to step through the volume, accumulating into the estimator the probability of each potential interaction with the volume, as shown in equation 27 shown again here:

$$T(t) = \prod_{i=1}^{K} \left(1 - \frac{\sigma_t(\mathbf{x}_i)}{\bar{\sigma}}\right)$$

Ratio tracking can easily replace an implementation of delta tracking, such as the implementation of `Transmittance` method for `BeersLawHeterogeneousVolume`, which follows algorithm 3.

Listing 19. Calculating transmittance with ratio tracking

```
virtual Color BeersLawHeterogeneousVolume::Transmittance(RendererServices &rs, const
    Point &P0, const Point &P1) {
   float distance = Vector(P0 - P1).Length();
   Vector dir = Vector(P1 - P0) / distance;
   float t = 0;
   Color transmittance = Color(1.0);
   do {
      float zeta = rs.GenerateRandomNumber();
      t = t - log(1 - zeta) / m_maxExtinction;
      if (t > distance) {
         break; // Did not terminate in the volume
      }

      // Update the shading context
      Point P = P0 + t * dir;
      m_ctx.SetP(P);
      m_ctx.RecomputeInputs();

      // Recompute the local extinction after updating the shading context
      Color extinction = m_ctx.GetColorProperty(m_extinctionProperty);
      transmittance *= Color(1.0) - (extinction / m_maxExtinction);
   } while (true);
   return transmittance;
}
```

Ratio tracking can also be used in the implementation of `Integrate` of single scattering heterogenous volumes (listing 10). The location used for direct lighting remains unchanged from the delta tracking implementation (the first potential interaction), but instead of terminating at the first step, we can continue until the end of the volume, accumulating the transmittance as above. Ratio tracking can also be easily combined with other importance sampling techniques such as equiangular sampling to improve the estimate of transmittance, and is a straightforward replacement for delta tracking's usage in multiple importance sampling (see section 4.2.1).

While ratio tracking delivers a much better estimator of the transmittance than delta tracking, unlike delta tracking it must also track all the way through the volume, and must evaluate the extinction coefficient at every step. This means it is a higher cost estimator than delta tracking, especially in volumes with a high majorant of extinction. This cost can be greatly mitigated if we can tolerate a slight bias: terminating the ratio tracking if some arbitrarily low transmittance is reached. Even so, this leads to the question: is it faster to converge to take a higher number of lower quality delta tracking estimates, or a lower number of higher quality ratio tracking estimates? We have generally found in production that the latter is true, especially when the extinction lookup is expensive and can benefit from locality of execution.

Residual tracking is another incremental improvement that increases the accuracy of the transmittance estimator. Residual tracking introduces a control extinction coefficient σ_c which is constant over the volume, and ideally is very similar to the actual extinction coefficient σ_t. With σ_c, the transmittance $T(t)$ is now broken into two parts: a control transmittance $T_c(t)$, which can be written in closed form, and a residual transmittance $T_r(t)$, which must be estimated:

$$\begin{aligned} T_c(t) &= \exp(-\sigma_c t) \\ T_r(t) &= \exp\Big(-\int_{s=0}^{t} \sigma_t(\mathbf{x}_s) - \sigma_c ds\Big) \\ T(t) &= T_c(t) T_r(t) \end{aligned} \tag{37}$$

By applying the ratio tracking estimator to $T_r(t)$ we arrive at the combined technique of residual ratio tracking, with resulting pseudocode shown in algorithm 4. The key improvement of residual ratio tracking over ratio tracking is the use of $\bar{\sigma}_r$, the majorant over the residual extinction coefficient $\sigma_r = \sigma_t - \sigma_c$. As long as $\bar{\sigma}_r$ is smaller than $\bar{\sigma}$, the mean free length of the steps between tentative collisions is lengthened, and residual ratio tracking will correspondingly evaluate the extinction with less frequency.

Algorithm 4: *Residual Ratio Tracking*

```
function ResidualRatioTracking($\mathbf{x}, \omega, d$)
    $T_c = \exp(-\sigma_c \times d)$
    $T_r = 1$
    $t = 0$
    while true do
        $t \leftarrow t - \frac{\ln(1-\zeta)}{\bar{\sigma}_r}$
        if $t > d$ then
            return $T_c \times T_r$
        $T_r = T_r \times (1 - \frac{\sigma_t(x + t \times \omega) - \sigma_c}{\bar{\sigma}_r})$
```

Novák et. al describe three different choices for the control extinction coefficient σ_c, and also describe that while the theoretical best choice is the average over σ_t, this may result in certain situations which lead to increased variance. We have found in practice it is best to avoid these situations and use the minimum value (or minorant) of σ_t. This is the chief drawback of residual ratio tracking: the need for the minorant as well as the majorant of the extinction coefficient. In the next section, we will discuss acceleration structures for tracking which can also be used to return the minorant of the extinction coefficient. Finally, we note that in practice, residual ratio tracking can be used for implementations of `Transmittance`, but are not as straight forward to use for single scattering implementations of `Integrate`; the control transmittance must be taken into account when selecting a location for direct lighting.

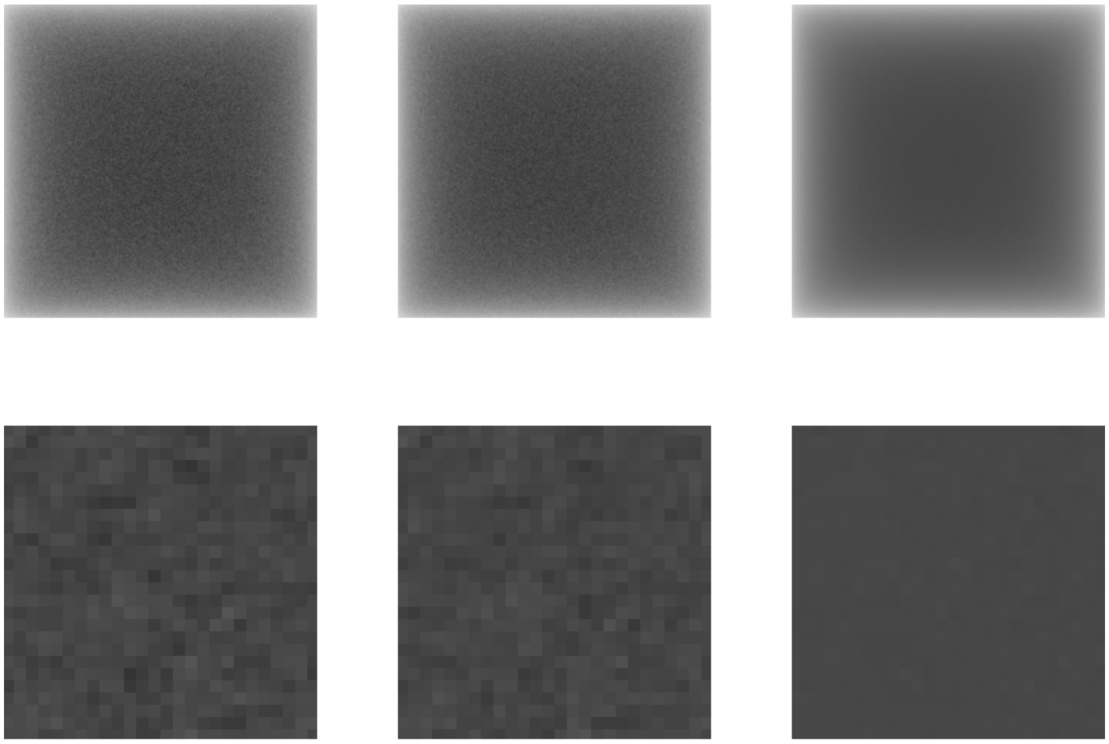


Fig. 27. Transmittance through heterogeneous but low variance volume. Left: delta tracking; center: ratio tracking; right: residual ratio tracking. Bottom row images are zoomed by 10 x.

4.4.2 Acceleration structures for tracking. While tracking methods can be very attractive due to being unbiased techniques which require low overhead, they have one major disadvantage: in order to remain unbiased, the initial tracking step size must be calculated from the maximum extinction coefficient σ_t of the volume. If the volume has a very heterogeneous σ_t, the majorant of σ_t over the volume may be far higher than the average value of σ_t in many places. For example, consider a mostly homogeneous volume of very low density which is dominated by a "spike" in density, as shown in figure 15. This high maximum density would cause the initial tracking step size to be small. Since the actual density sampled at the step would more likely than not be small compared to the maximum density, the chance of interaction with the volume itself would also be small. Over many rays, this would result in a large number of steps taken on average. Since every step requires a computation of the actual density, on production scenes these per-step density calculations will be the dominant cost of volume rendering.

To ameliorate this problem, we can break up the volume into smaller, non- overlapping subregions, and bound σ_t inside each region. Subregions with a high $\bar{\sigma}$ would be sampled at the appropriate higher initial stepsize, while subregions with low $\bar{\sigma}$ would be sampled at the lower densities. In effect, by doing this, denser regions of the volume are sampled by the tracking method with the appropriate higher frequency, while less dense regions are sampled less often. However, dividing the volume up into subregions means that the region of volume integration along the ray is no longer a single interval: the ray which formerly intersected a single volume region now intersects multiple non- overlapping subregions to create multiple subintervals, each subinterval having its own $\bar{\sigma}$. We must determine the intersection of the ray against these subregions, and we must be able to do so quickly. One convenient and well-studied approach to this problem is to use a k-d tree (Bentley 1975) to spatially subdivide the volume region into subregions; this approach was first suggested for the acceleration of delta tracking by Yue et al. (2010). The leaf nodes of the k-d tree store the maximum density of a single subregion. We can then intersect a ray against the k-d tree to come up with the list of overlapping volume subregions; these regions divide the extent of the ray into subintervals, each with an associated $\bar{\sigma}$ value.

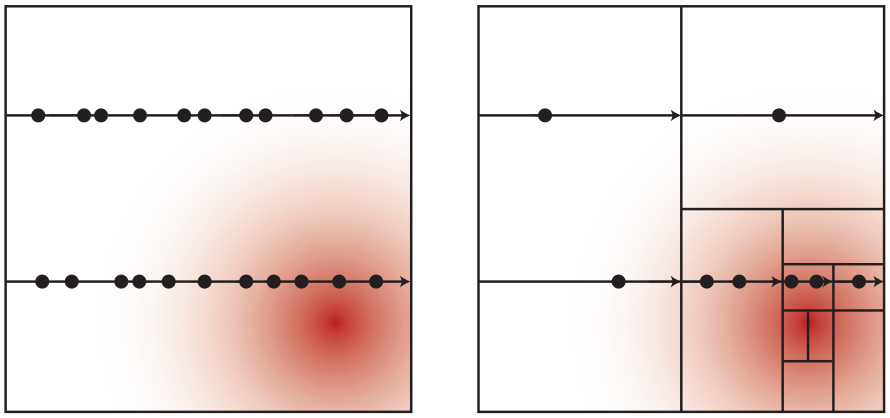


Fig. 28. A heterogeneous volume with a spike in density results in a high $\bar{\sigma}$ everywhere, requiring many short tracking steps and many evaluations of the extinction coefficient (left). Dividing the volume with a kd-tree lowers the majorant in large regions, resulting in many fewer evaluations (right).

With a well-constructed k-d tree, the dominant cost of rendering a heterogeneous volume when using a tracking method is now no longer the per- step evaluation of σ_t; it instead shifts to the intersection of the ray against the k-d tree in order to find the list of subregions which intersect the ray. This is very much analogous to the inner workings of a ray tracer: we avoid the costs of many per-geometry intersection tests by using a ray acceleration data structure. A simple stack-based traversal algorithm such as the one published by Jansen (1986) works well here, and can be reasonably optimized by using tail recursion, and by reducing the memory usage of each node (for example, by using an implicit representation which only stores the split plane, and avoiding explicit pointers to the children nodes). For a more general overview of k-d tree traversal algorithms, please see (Hapala and Havran 2011).

While a well optimized implementation of k-d tree traversal is important, the details of the construction of the k-d tree are just as important, as reported by Yue et al in their follow-up paper (2011). A naively implemented k-d tree can offer a tremendous speedup compared to the non accelerated implementation, but a well-tuned k-d tree can offer several times speedup on top of that. We will depart somewhat from (Yue et al. 2011) and focus instead on a well researched heuristic for building k-d trees, the Surface Area Heuristic (SAH) (MacDonald and Booth 1990), which is commonly used for building k-d trees used for ray intersection against geometry. In brief, this heuristic tries to minimize the recursive cost function $C(V)$, defined for a non-leaf node V of the k-d tree as follows:

$$C(V) = K_T + \frac{SA(V_l)}{SA(V)} C(V_l) + \frac{SA(V_r)}{SA(V)} C(V_r) \tag{38}$$

where K_T is the cost of a traversal step through the k-d tree, $SA(V)$ is the surface area of V, V_l is the left subchild of V and V_r is the right subchild of V. When dealing with geometry, the cost function of a leaf node $C(V)$ would normally be a simple function of the number of triangles in the leaf node. As we are interested in minimizing the number of delta tracking steps in the volume, a more useful yet also simple cost function is

$$C(V) = \frac{K_D \operatorname{diag}(V)\bar{\sigma}(V)}{-\log(0.5)} \tag{39}$$

where K_D is the cost of evaluating σ_t somewhere in V, $\operatorname{diag}(V)$ is the length of the diagonal of the volume V and $\bar{\sigma}(V)$ is the majorant of σ_t over V. This cost function can be understood simply as the maximum number of delta tracking steps taken in the volume assuming the chance of termination at every step is zero, which is the diagonal of the volume, divided by the length of the average step $\frac{-\log(0.5)}{\bar{\sigma}}$. This cost function can be further improved by removing the zero-termination assumption, and instead taking into account the conditional probability of termination, which at every step is the average value of σ_t divided by $\bar{\sigma}$ in V.

Because equation 38 is a recursive function, minimizing it over all possible subtrees is a very expensive problem to solve. The usual simplifying assumption is to use a greedy algorithm which at each step computes only an approximation to $C(V)$ rather than recursing: it assumes that when computing the cost of $C(V)$, the cost of the subvolumes $C(V_l)$ or $C(V_r)$ can be approximated by assuming they are leaf nodes, thus allowing the use of equation 39 as their cost, leading to a modified cost equation:

$$C(V) = SA(V_l) \operatorname{diag}(V_l)\bar{\sigma}(V_l) + SA(V_r) \operatorname{diag}(V_r)\bar{\sigma}(V_r) \tag{40}$$

In this formulation, the terms that are invariant for a volume can be dropped, since we are just interested in minimizing the function locally in the greedy algorithm, and there is no longer a recursive aspect. Now, instead of computing a globally optimal solution, we simply start at the top level volume V and compute the splitplane which divides the volume V into two subvolumes V_l and V_r which minimizes equation 40. We recurse over each subvolume until some termination metric is satisfied.

For more discussion on building SAH-optimized k-d trees, please refer to the overview by Wald et al. (2006).

In order to calculate $\bar{\sigma}$ over a subvolume, we must of course be able to compute the σ_t over the entire subvolume. A typical approach to this problem is to voxelize the entire subvolume and run the full pattern generation inputs in a prepass. If the finest possible resolution of the input to the extinction coefficient calculation is known a priori — perhaps because the input is sourced from a simulation run at some finite frequency — the voxelization can be set to this resolution, and the resulting maximum will be guaranteed to be correct. If however the input resolution cannot be determined in advance — perhaps because some procedural shading will add additional advection to the volume at runtime — the voxelization may miss some detail, which will result in biased results. In practice, this bias may be acceptable, especially if the fine voxelization otherwise required would necessitate a high preprocessing time. In either case, if a shading prepass is required, it is possible to intertwine the k-d tree construction with an architecture which minimizes the in-memory requirements of shading: for example, using a REYES-like architecture which splits and dices to grids of microvoxels, evaluates the shading over the grid, and discards the grids. For very large volumes, however, this may come at a considerable startup cost which will affect the time to first pixel of rendering. If it is known that shading will not add additional detail to the volume (usually because σ_t is baked into a file), another approach is to bake the maximum into the file format at runtime. This is useful because complicated volume simulations are often written only once, but read multiple times over the course of their lifetime in production. However, this baking step ideally requires knowledge of the k-d tree optimization employed by the renderer. If the maximums are simply a coarser subsampling of the volume data, a refitting step may be required by the renderer in order to create an optimal k-d tree.

We will not show how the pre-processing of the volume k-d tree takes place in our hypothetical path tracer; we will either assume it is handled by an implementation of the `Volume` interface, or handled by the renderer and provided by the `RendererServices` object. Furthermore, we assume that the facilities for intersecting a ray against a k-d tree are also handled in one of these two places; the choice of which is highly dependent on the implementation of shading in the system. We will simply assume that some function `intersectSegmentAgainstKDTree` exists where a line segment denoting the region of volume integration can be intersected against the k-d tree to produce a list of subinterval endpoints `rangeDists` and their σ_t majorants `maxDensities`. With this information in hand, we need to modify our tracking integration algorithms. The implementation of `BeersLawHeterogeneousVolume::TrackingVolume` from listing 8 can be augmented by simply changing how the first delta tracking step takes place:

Listing 20. Delta tracking using bounded subintervals from a k-d tree

```
bool takeInitialStep(RendererServices &rs, RtInt rangeCounts, float const * rangeDists,
    float const * maxDensities, float *accumDistance, int *currentSeg) {
    bool finished = false;
    do {
        // Find first interesting segment
        int i = *currentSeg;
        float maxRayDensity = maxDensities[i];
        // Skip over empty segments entirely
        if (maxRayDensity == 0) {
            if (i == rangeCounts - 1) {
                // We're done with all the segments
                finished = true;
                break;
            } else {
                *accumDistance = rangeDists[i];
                *currentSeg = i + 1;
                continue;
            }
        }
        float xi = rs.GenerateRandomNumber();
        // Woodcock tracking: pick distance based on maximum density
        *accumDistance += -logf(1 - xi) / maxRayDensity;
        if (*accumDistance >= rangeDists[i]) {
            // Skipped past the end of the current segment
            if (i == rangeCounts - 1) {
                finished = true;
                break;
            } else {
                // Went past the segments with no interaction. Move on to
                // the next segment, resetting the accumulated distance to
                // the beginning of that segment
                *accumDistance = rangeDists[i];
                *currentSeg = i + 1;
            }
        }
        else
            // Check for actual interaction
            break;
    } while (true);
    return finished;
}

virtual void Transmittance(RendererServices &rs, const Point &P0, const Point &P1,
     Color &transmittance) {
    float distance = Vector(P0 - P1).Length();
    Vector dir = Vector(P1 - P0) / distance;
    bool terminated = false;
    float t = 0;
    int currentSeg = 0;
    int rangeCounts;
    float *rangeDists, *maxDensities;
    intersectSegmentAgainstKDTree(P0, P1, &rangeCounts, &rangeDists, &maxDensities);
    do {
        if (takeInitialStep(rs, rangeCounts, rangeDists, maxDensities, &t, &currentSeg))
            break; // Did not terminate in the volume

        // Update the shading context
        Point P = P0 + t * dir;
        m_ctx.SetP(P);
        m_ctx.RecomputeInputs();

        // Recompute the local absorption after updating the shading context
        Color absorption = m_ctx.GetColorProperty(m_absorptionProperty);
        float xi = rs.GenerateRandomNumber();
        if (xi < (absorption.ChannelAvg() / m_maxAbsorption))
```

```
            terminated = true;
    } while (!terminated);

    if (terminated)
        transmittance = Color(0.0f);
    else
        transmittance = Color(1.0f);
}
```

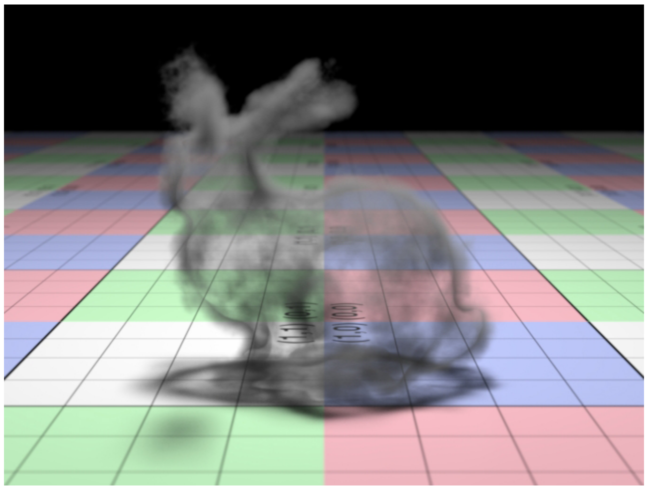


Fig. 29. The use of a k-d tree data structure to bound the majorant of σ_t results in a reduction in the number of σ_t evaluations in this scene by over a factor of 6, which in this implementation reduces the runtime by over 30%.

In order to implement residual ratio tracking, we not only require the majorant of σ_t over a volume, we need the minorant. This is a trivial extension of the pre-processing necessary to build the k-d tree, except that the cost function needs to be modified. At the bare minimum, equation 39 would be changed to reflect that the average step is lengthened by the minorant $\breve{\sigma}_t(V)$:

$$C(V) = \frac{K_D \operatorname{diag}(V)(\bar{\sigma}(V) - \breve{\sigma}_t(V))}{-\log(0.5)} \tag{41}$$

With knowledge of the minimum and maximum densities in subintervals over the extent of an interval, we can perform several other very useful optimizations. First, if the $\bar{\sigma}$ at every subinterval is zero, we know that the beam transmittance of the entire volume interval is also trivially 1. This is essentially an empty volume optimization, and can arise in practice if the volume loosely fills its bounding box. Because this optimization can be significant, it is well worth ensuring that the k-d tree is built to maximize regions of zero density. Second, we can use the minorant of σ_t over each subinterval to compute a lower bound on the beam transmittance of the volume interval:

$$T_r \leq \prod_{i=1}^{n} e^{-\breve{\sigma}_t(i) d(i)} \tag{42}$$

If we accept a slightly biased result and establish some minimum transmittance ϵ such that $T_r \leq \epsilon$ is considered an opaque transmittance, this condition can be used to skip a more expensive tracking operation that would otherwise be incurred by the computation of transmittance.

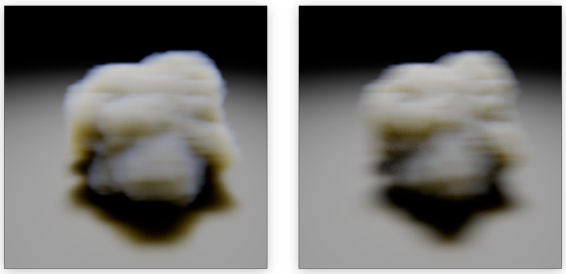


Fig. 30. Left: A volume with smeared motion blur. Note the change in light bleed. Right: The correct motion blur computed using transformation motion blur

5 MOTION BLUR

5.1 The problem with motion blur

At a glance, motion blur in the path tracing context should be trivial. Time is simply another dimension over which samples are distributed, and the accurate solution that handles all types of motion blur is to simply let $t = \xi$ for each ray. Given enough samples the resulting image will be imbued with photorealistic motion blur and the goal will be accomplished. Although a gross oversimplification, this rationale may be the underlying reason why motion blur often gets glossed over or ignored in the rendering literature. In reality, the addition of time and deforming geometry in a rendering system greatly complicates the implementation, effectively turning a (usually) tractable three-dimensional problem into a four-dimensional engineering nightmare.

Note: In this chapter, we are considering *deformation motion blur*, meaning motion that varies within an individual object. *Transformation motion blur*, by which individual objects are rigidly animated using their transformation matrices, can be achieved identically for both geometry and volumes.

Volumetric motion blur is different from geometric motion blur in the fact that volumes lack the topology that defines a geometric object. Whereas a surface renderer generally treats a deforming object as the interpolation between two states (e.g. at the start and end of the shutter), a volume lacks the scaffolding to do so. Instead, volumes are *fields* that describe the value of a function (usually density) across space, and the only way to define motion is to assign a vector-valued *motion field* to the volume, whose magnitude and direction describes the instantaneous velocity of the medium at any given point in space.

5.2 Smeared motion blur

Prior to path tracing's rise to popularity in production rendering, motion blur in volume rendering was achieved using several different solutions, depending on the architecture of the host renderer.

The first, and simplest form of volumetric motion blur was "smeared" motion blur, in which the motion effect was pre-computed at the voxel level, effectively performing a 3D line convolution on the voxel data itself. This method had the benefits of being easy to implement and (once pre-computed) very fast to render. However, because the method has to compute a convolution per voxel, the process can be quite expensive in the pre-computation step. Also, the visual appearance is not correct. Because the voxel densities are averaged along the motion vector, the lighting response is softened and features that were once sharp do not form well-defined streaks but rather unnaturally soft smears.

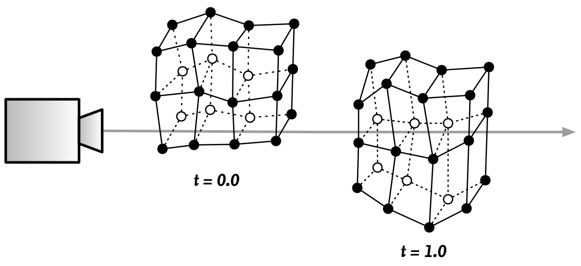


Fig. 31. Two motion samples of a single microvoxel grid. Because the topology is identical, simple interpolation can be used to find in-between states.

The technique of smearing voxel data eventually gave way to two improved techniques: microvoxel motion blur and advection motion blur. The two methods fit in the two main rendering architectures at the time: microvoxel motion blur fit well within Reyes-based renderers, and the Eulerian motion blur method (sometimes also called advection motion blur) worked within the ray marching (i.e. ray tracing) framework.

5.3 Microvoxel motion blur

A method by Clinton and Elendt (Clinton and Elendt 2009) tackles the motion blur problem in a classic Reyes (Cook et al. 1984) fashion: by subdividing a complex shape into micro-shapes, in this case called "microvoxels". The volume is divided into a lattice grid with spacing roughly equal to the projected pixel size. This structure provides the topology needed to directly track motion, and the velocity field is simply sampled at each grid vertex to define the motion over time.

Overall, the cost of microvoxel motion blur is variable. There is a fixed cost related to shading the grid points in the lattice, where the volume's density and velocity fields are evaluated. There is also an increased cost due to tracking the displacement of the grid vertices, which increases with additional motion magnitude.

For raymarching, these costs are neatly amortized across the number of steps along each ray, which is often in the 100s, by using a DDA-like traversal from one lattice cell to the next. However, for tracking algorithms the cost of random point sampling makes microvoxel blur impractical since the number of steps along each ray is low and more randomly spaced.

5.4 Eulerian motion blur

Similarly to the microvoxel motion blur method, the Eulerian method, first described by Kulla and Fajardos (Kulla and Fajardo 2012), assumes that the change to a volumetric property is due to *motion* rather than material appearing or disappearing over time (as would happen in e.g. a fluid simulator). That is, given a property σ_t and a velocity field v, the function over time is

$$\sigma_t(P, t) \approx \sigma_t(P - v(P) \cdot t, 0), \tag{43}$$

and by offsetting the sample point based on $t \in [0, 1]$, we deform the field σ_t by the velocity field, which produces a motion blur effect. This process is equivalent to *advection* in fluid dynamics, and the method is sometimes also called it advection-based motion blur.

The benefit to the method compared to smeared motion blur is that secondary rays see the correct volume configuration at any given time, so motion blurred shadowing effects are correctly preserved.

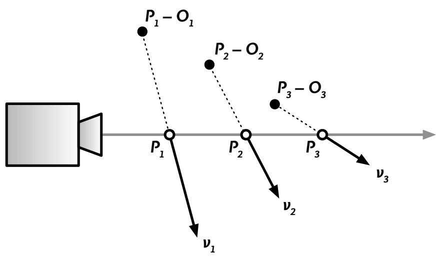


Fig. 32. Eulerian motion blur, also called advection motion blur. Each sample point is offset by a motion vector by a magnitude determined by the ray's time value.

However, the method is only fully accurate for velocity fields that are not spatially varying. We can see why if we consider a "curved" velocity field: It is not necessarily possible to take a step backwards from P to $P' = P - v(P)$, and then return to the original point by stepping forward along the velocity of the new position, i.e. $P \neq P' + v(P - v(P))$. Intuitively this means that the material that we transported from P' to P by tracing backwards along $v(P)$ would actually have traveled to the point $P'' = P' + v(P')$.

The cost of Eulerian motion blur depends on two factors. First, there is an increased cost with each sample taken, since both the velocity field and the density field must be evaluated. This tends to amount to an increase of roughly 100% in the time spent computing density values, compared to just a single density lookup. The second factor depends on the magnitude of the motion and how the acceleration structures are designed. Similarly to microvoxel motion blur, the deformation aspect has to be taken into account when performing any kind of empty space optimization during the ray traversal. For example, if a superstructure that identifies empty space is constructed at $t = 0$, then a padding must be applied so that even after deforming a lookup point, the empty space is accurate. This is equivalent to *displacement bounds* in surface rendering, and can make volumetric acceleration structures much less efficient, especially if the motion blur magnitude is large.

5.5 Motion blur with tracking-based integrators

Volume integration methods that are based on raymarching have a constraint on their step length: it needs to be roughly the size of the smallest voxel in order not to miss any features. Conversely, tracking-based integrators have a different constraint: they need to choose their step length based on the extinction coefficient of the volume. For Woodcock tracking, the majorant is used, and for others such as residual ratio tracking, both the majorant and a control are needed.

This has multiple implications for a production renderer. First, the renderer's API towards volumes must provide hooks that allow querying of the volume's "meta properties" (i.e. majorant) along any ray. Furthermore, this query must be extremely fast, as it will be made multiple times per ray, for every ray in the scene. One relatively common approach in accelerating these queries is to use a grid superimposed on each volume (Novák et al. 2014) where the meta properties are stored at a coarser granularity than the density function itself. Again, the deformation bounds problem rears its head: if combining a tracking method with Eulerian motion blur, each cell of the super structure must be padded according to the largest motion vector, which reduces the efficiency of the acceleration structure.

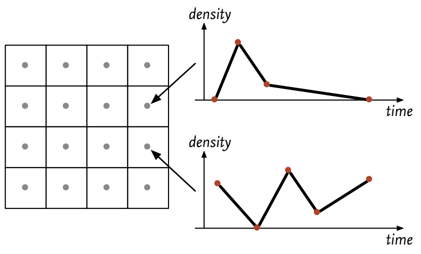


Fig. 33. Illustration of the temporal function stored in two temporal voxels. Note that the placement and number of samples varies between the two.

When it comes to tracking integrators, having tightly bounded statistics on the extinction property is absolutely key to good performance (even for integrators that don't need a strict majorant), and both microvoxel- and Eulerian motion blur make the problem intractable.

5.6 Temporal volumes

In the rendering context, the idea of *motion blur* is itself somewhat biased: what we are interested in is not motion itself, but the passage of time in an image. In fact, in the path tracing context, individual rays never see explicit motion; rather, they see instances of the scene at different *times*. Motion is only apparent once we average the samples and observe the final image, perceptually inferring the arc that an object traveled.

Looking back to Equation 43, we note that the renderer is actually querying σ_t as a spatiotemporal function, and that the motion just serves as a mechanism for computing the temporally varying property. With that insight in mind, we could imagine volumes that were defined not just in three dimensions, but as fully temporally dependent volumes of four dimensions. In a way, this is not particularly different from the animation of the volumetric data to begin with: it is stored on disk as a sequence of frames. In general, there is only one file per image to be rendered, and the lack of topology makes state interpolation difficult, but fundamentally we have a 4D representation. If we increased the sampling rate of the volumetric data to 100x the actual frame rate, we can picture the resulting image as having quite accurate motion blur, once each ray has the ability to "see" the appropriate sub-frame volume state. The only impractical aspect to this solution is storage on disk and in memory: a dense 500^3 volume would occupy roughly 500MB of disk and memory, but the oversampling factor of 100 would balloon that number to 50GB. Ignoring the practicality of the storage requirements, we can see the appeal of this representation: rather than solving the difficult problem of optimizing ray traversal in the face of deformation, we can attack the much more tractable problem of compressing a large but highly coherent dataset.

In 2016, Wrenninge presented a solution (Wrenninge 2016) to this storage problem that was inspired by deep shadow maps (Lokovic and Veach 2000). Where deep shadow maps made a highly variable depth dimension tractable be storing an unstructured piecewise linear function in each pixel of an image, temporally unstructured volumes (hereby referred to as "temporal volumes") store an unstructured piecewise linear function in each voxel of an otherwise ordinary voxel grid. The one-dimensional function encodes how the volumetric property changes over time, making it possible to (lossily) compress the data in the temporal dimension. Intuitively, this means that the voxel grid itself stays stationary, and

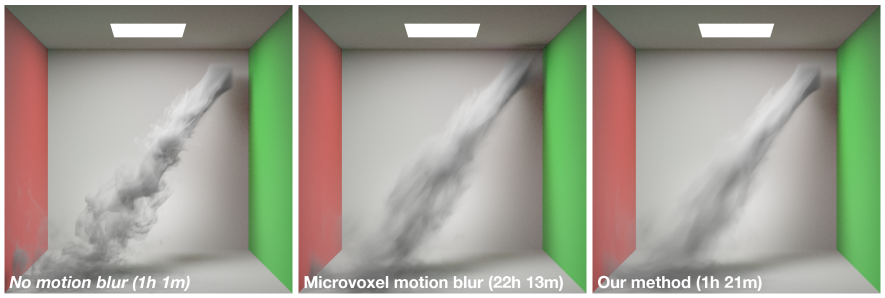


Fig. 34. Left: Volume with no motion blur. Center: Microvoxel-based motion blur. Note that the fluid source in the upper right is incorrectly blurred. Right: A temporal volume renders faster and correctly handles the continuous injection of density.

that each voxel now measures the fluctuation in density over time. Even though it captures no actual motion, it is a perfectly valid representation of the physical state of the volume seen over time. In fact, because the quantity and location in time of the sample points can vary from voxel to voxel, voxels do not necessarily ever see the same "frame" of animation.

The rendering performance of temporal volumes is very predictable. Because the lookup cost is independent of any motion apparent in the data, the only overhead compared to a non-temporal volume is the additional interpolation of the one-dimensional function in each voxel. Although technically an $O(\log N)$ operation, the number of time samples is generally small enough that the speed impact seen in density calculation is roughly 30%. On a production-scale shot with significant motion blur, the fact that motion magnitude no longer affects render time may mean that a shot renders several times faster than when using the Eulerian method. This is largely due to the fact that a temporal volume can produce meta property calculations (i.e. min/max) that are tightly bounded, resulting in efficient acceleration structures independently of visual motion blur magnitude.

Apart from rendering efficiency, temporal volumes have the advantage of being able to handle all forms of motion blur, including curved motion as well as material appearing or disappearing during the course of the shutter interval. The latter is especially useful for fluid simulations where density is injected at high speed – see Figure 34.

In practice, there are multiple ways of generating temporal volume data. The original paper describes the Reves algorithm, essentially "Reyes for volumes", which is suitable for procedurally modeled volumes. It also discusses how to generate temporal volume data given non-temporal volumes plus a velocity field, in effect shifting the Eulerian motion blur method to a pre-computation step. Lastly, it is possible to generate temporal volume data directly in a fluid simulator, by caching in memory each simulation sub-step and reconstructing a full animation frame's worth of temporal data once the simulation step is complete. The latter version has the benefit of being able to capture intricate sub-frame motion without affecting the final render time, which can be especially beneficial for fast-moving simulations such as fire and explosions.

The temporal volume method has been used in multiple Pixar films, such as The Good Dinosaur (Wrenninge and Rice 2016), Cars 3 (Marshall et al. 2017) as well as the upcoming Coco.

6 AGGREGATE VOLUMES

6.1 Integration intervals

The traditional view of volumes in the rendering context is that each one lives in a box. This granularity is, in a way, the first form of optimization in volume rendering: there is no need to perform calculations in areas that are known to be empty. When volume rendering was performed using ray marching, techniques such as empty space optimization (see Wrenninge (2012) for an overview) were used to answer questions about the internal structure of the volume, such that the integrator could optimize its calculations. The idea was that a volume could communicate to the integrator regions that needed attention vs. regions that were known to be empty. Also, the step length of a ray marcher was tightly coupled to the Nyquist limit, and it was important for each volume to communicate to the integrator its frequency content. As such, the granularity of such queries were strictly in the hands of the volume itself, and each volume could tell the integrator to divide the ray into an arbitrary number of integration intervals.

Listing 21. Pseudo-interface for raymarching-based integration

```
struct Segment
{
    float start, end;
    float nyquistLimit;
};

class Volume
{
    // ...
    virtual void getSegments(const Ray &ray,
                             std::vector<Segment> &segs) const = 0;
    // ...
};
```

In the case of ray marching, this was a reasonable arrangement: the empty space was often analyzed in terms of the volumetric data structure itself, and both OpenVDB (Museth 2013) and Field3D (Wrenninge 2009) have built-in mechanisms for checking whether a given region of space is empty (i.e. unallocated). Likewise, the sampling frequency (i.e. the voxel size) was known only to the volume itself.

6.2 Integration intervals and tracking-based integrators

With the move to path tracing and tracking-based integrators, some of these previously reasonable couplings become problematic. The integrator is no longer interested in a binary present/not-present state, and there is no longer a Nyquist-type sampling frequency to adhere to. Rather, the integrator is now concerned with the entry and exit points into the volume (which are still useful for skipping empty space) and various meta-properties of the volume, such as the maximum and minimum extinctions over a particular ray segment. Whereas ray marching suffers performance issues in thin but high resolution volumes, a Woodcock integrator grinds to a halt if the majorant extinction is poorly bounded, due to the large number of steps required to traverse each segment. We thus note that the sampling frequency information has become irrelevant, and that the presence query now needs to report a quantity rather than simply a binary value. Figure 35 illustrates the concept.

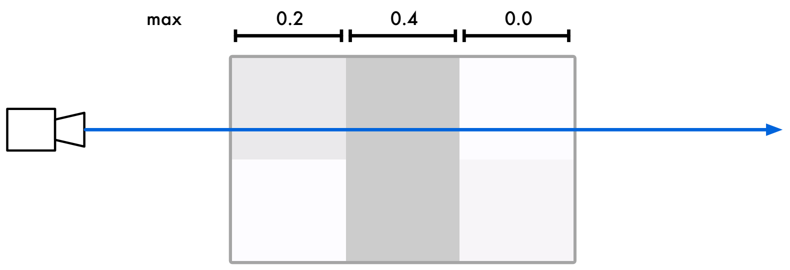


Fig. 35. Gathering integration intervals from the blocks/tiles of a voxel buffer. The intervals are of uniform size independently of the contents of the volume. Empty space can be culled by recognizing that max=0 for certain intervals.

Listing 22. Pseudo-interface for tracking-based integration

```
struct Segment
{
    float start, end;
    float maxDensity;
};

class Volume
{
    // ...
    virtual void getSegments(const Ray &ray,
                             std::vector<Segment> &segs) const = 0;
    // ...
};
```

6.3 The volume–integrator interface

At this point we are faced with a potential problem: the volume is tasked with providing information about the volume at a granularity that makes the integrator as efficient as possible, but the volume doesn't have any information about how the integrator works, only that it is requesting certain meta-properties about the volume (in this case, the maximum density). Conversely, the integrator knows how the meta-properties will be used and what an efficient granularity may be, but it does not have knowledge of the volume's structure. OpenVDB and Field3D's sparse data structure can both be leveraged to store these meta-properties at branch nodes or as auxiliary fields (usually at 8x or 16x lower resolution), but there is no reason to believe that this is an inherently suitable granularity. If we consider a high-resolution voxel volume, the older ray marching methods would need to visit each voxel at least once due to the Nyquist sampling limit, irrespective of the actual density of the volume. For a tracking integrator, things are different: if a high resolution volume is thin, then the integrator will be able to take very large steps (see Section 2.3.3) regardless of the actual voxel size, while still providing a correct and unbiased result. But in order to specify long integration intervals only where the volume is thin, the volume would now need information about all other volumes that may overlap it, or else risk providing segments that are ultimately inefficient.

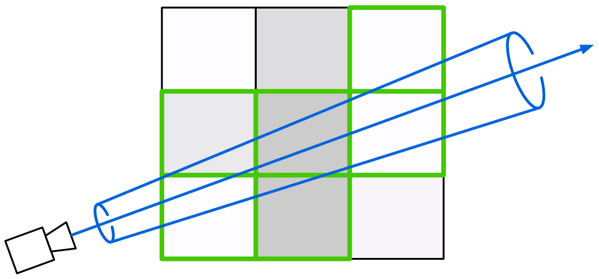


Fig. 36. Explicitly computing meta-properties along a ray involves checking each voxel that contributes to points along the ray.

6.4 The cost of providing meta-properties

A second concern is the cost of providing these meta-property queries on a per-ray basis. In order to find the maximum extinction value along a ray, we either need to resort to looking at a coarse granularity (which provides suboptimal bounds), or to traverse all the high-resolution voxels along a ray in order to determine the maximum. To make matters worse, we would have to traverse a “fat” line through the volume as seen in Figure 36, because the linear interpolation commonly used in voxel lookups will “pull” non-zero values part-way into a voxel that may itself have zero value[1]. The cost of this sort of traversal quickly makes it impractical, since the marching process for gathering meta-properties along a ray becomes much more costly than the final integration itself.

Ignoring the efficiency issue, we could design the Volume’s interface to at least allow the integrator to query the relevant information:

Listing 23. Pseudo-interface for meta-property query

```
class Volume
{
    // Query the maximum density along a ray segment with given width
    virtual void getMaxDensity(const Ray &ray, float filterWidth,
                               float start, float end) const = 0;
    // ...
};
```

6.5 Changing the interface

An important observation to make at this point is that although the bounding box intersections are used to start and end the integration process, in the most common case of trivial BSDFs (see Section 3.1) for the volume’s enclosure, where the interface is a no-event, there is nothing inherently special about these start- and end points. In fact, densities tend to be zero at the edges of volumes, to hide their very existence. Thus, if we were to bring the integration start point closer to the camera, or push the end point further away than the actual intersection, the integration would still be valid.

[1]This happens because the voxel sample is considered to lie in the center of the voxel

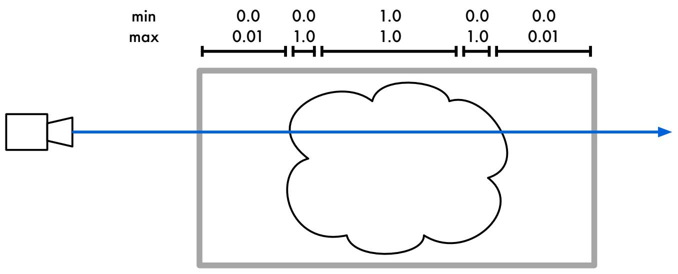


Fig. 37. Efficient integration intervals for a cloud with uniform interior density. Only the short segments near the boundary of the cloud require small steps by a tracking integrator.

A second observation to make is that the ideal granularity of the meta-properties vary within a volume. For example, in a cloud, we would like the region around the edge to be as short as possible, so that we can take large or no steps in the thin or empty area around the cloud, then quickly cross the part where density goes from low-to high (which is where a Woodcock integrator would need to take short steps and reject many null-events), then proceed into the thick part of the cloud where step lengths again can be long, due to the minimum and maximum densities being close in range. Figure 37 illustrates the configuration.

With these in mind, we would ideally like to change the interface between the volume and the integrator such that volumes can be more agnostic of the integrator's design, and provide only a bare minimum of operations such that the integrator can point sample at individual marching steps, and query meta-properties at a granularity that is appropriate both for the integration method as well as the global arrangement of volumes.

6.6 Multiple overlapping volumes

So far we have only discussed the interaction between a single volume and the integrator. Once we introduce the possibility of having multiple volumes in a scene, especially volumes that overlap, the problems are compounded. There is a certain overhead involved in querying multiple volumes and converting overlapping integration intervals to disjoint regions, but these disjoint regions now break the design goal of adapting integration intervals to the distribution of density. In fact, by tracking the entry and exit points we can wind up with arbitrarily short integration intervals, each of which cause a reset of the tracking algorithm, as seen in Figure 38.

We again note that the "99% case" for volumes is the one where the hull of the volume is a no-event. Thus, the only information that the integrator really needs from these intersections is the list of active volumes, since for efficiency reasons we only want to draw samples from volumes that can have non-zero values in a given interval. Apart from this requirement however, explicitly tracking the interfaces of each overlapping volumes is unnecessary in the no-op BSDF case; in fact, it greatly harms performance.

One approach that avoids having to deal with multiple volumes is to *resample* all of the scene's volumes into a single voxel buffer, reducing the more complex multiple-volume rendering problem to a one-volume case. However, this approach has several drawbacks: resampling can take a long time and use a significant amount of memory. It is also difficult to manage overlapping volumes that employ different phase functions. Lastly, it is also diffusive, and very high voxel resolutions are required in order to preserve detail.

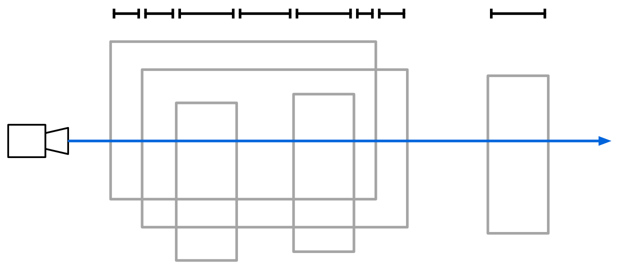


Fig. 38. Overlapping volumes create many short integration intervals (seen on top) if volume boundaries are tracked explicitly. Even if each interval is nearly homogeneous and could use a long step length, the integrator is forced to reset the tracking algorithm at each interval change, causing poor performance.

6.7 Volumes in aggregate

We can draw a few conclusions from these discussions: First, letting the integrator communicate directly with multiple volumes quickly gets complicated. And second, apart from having a general idea of which volumes to try and sample in a given region, the integrator doesn't particularly care about the exact intersection points nor the individual makeup of each volume. Instead, the integrator wants only "nicely" defined integration intervals: short enough that the meta-properties conform well to the underlying density function, but long enough that we can traverse the ray quickly without too many re-starts.

So far, we have only made incremental modifications to the Volume interface that we inherited from the raymarching method of integrating volumes. However, we can also turn the problem around and design a new interface. From the integrator's perspective, the goals are:

- Integrator-friendly intervals: We care primarily about finding a scattering event in any region, and are less concerned with the specific length of that segment: a thin, long segment can be computationally equivalent to a dense, short one.
- Fast meta-property queries: Ideally, we would amortize the cost of computing meta-properties across a region of space and share the results for all rays that intersect the given region. This avoids costly per-ray calculations.
- Integrator-agnostic volumes: The modularity of our system would be improved if volumes needn't be aware of exactly *how* the integrator goes about computing radiance and transmittance.
- Volume-agnostic integrator: The integrator should not need to consider the underlying data structures of volumes. This also improves modularity and makes it practical to efficiently render a wider set of volume types, including purely procedural ones.

The best way to accomplish these goals without increasing the coupling between the volumes and the integrator is to insert an intermediary data structure between the two. In our system we refer to this intermediary as the *volume aggregate*, as it aggregates information about the volumes without explicitly combining them.

In 2010, Yue et al. presented a paper where a k-d tree spanning the entire space of the scene was used to optimize a Woodcock tracking integrator. Their follow-up 2011 paper (Yue et al. 2011) further explored the optimal subdivision of this k-d tree. Although many of the assumptions don't fit the production rendering requirements, the papers nonetheless presented a key insight: it is possible to design a hierarchical spatial data structure that provides optimal (or at the very least deliberately tailored) integration intervals for a tracking-based integrator.

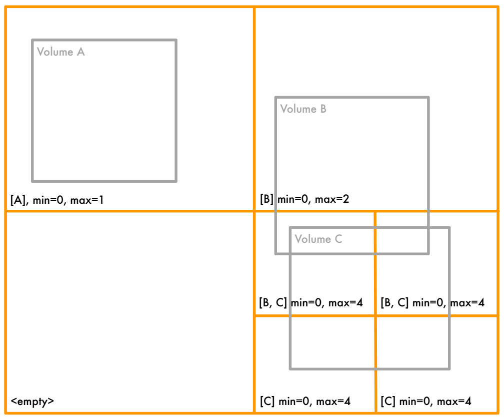


Fig. 39. The aggregate data structure. Each node records a list of volumes that overlap (partially or completely) as well as the min and max value of the contained volumes.

6.8 Volume aggregate structure

The design of our volume aggregate is as follows:

- An octree is constructed, spanning the region of all volume primitives in the scene.
- Each node of the tree stores a list of volumes that overlap the region (even partially), as well as the relevant meta-properties for a given integrator. Typically, these are the minimum and maximum extinction coefficients across the region.
- Each node is subdivided as long as a given heuristic is true. In our case, the heuristic for a region R is simply $(\max(R) - \min(R)) \cdot \texttt{diag}(R) > T$, where T is an arbitrary cutoff value, typically in the range $[1, 4]$. This approach purposely ignores the physical boundaries of each volume.
- Any ray that needs to integrate the volumes in the scene performs a DDA-like walk of the octree nodes until a scattering event is found. During traversal, all the relevant meta-properties are available and pre-computed for efficiency.

By design, this structure is simple, in fact even simpler than the Yue paper's version. The purpose of this simplicity is generality: in production scenes we simply do not have enough information about the underlying extinction function to be able to choose a sensible cut plane for a k-d tree, so an octree is used instead[2]. As we will see below, we also need to incorporate support for motion blur, ray differentials and other production rendering cornerstones, and having a simple underlying structure makes this more feasible.

At its core, the volume aggregate achieves all of the design goals that were specified above:

- Integrator-friendly intervals: The length of each integration interval is relative to the octree node size, which in turn is implicitly relative (thanks to the heuristic) to the probability of finding a scattering event in the interval. In fact, the structure unifies heterogeneous and homogeneous volumes, since by virtue of the subdivision heuristic, for any region where $\max(R) = \min(R)$ the subdivision will stop and a Woodcock-type tracker will either find a scattering event or pass through the segment in a single step (which is the optimal case).

[2]This is in contrast to the case in Section 4.4.2, where the volume is rasterized at a fine resolution, giving us detailed knowledge of the extinction function.

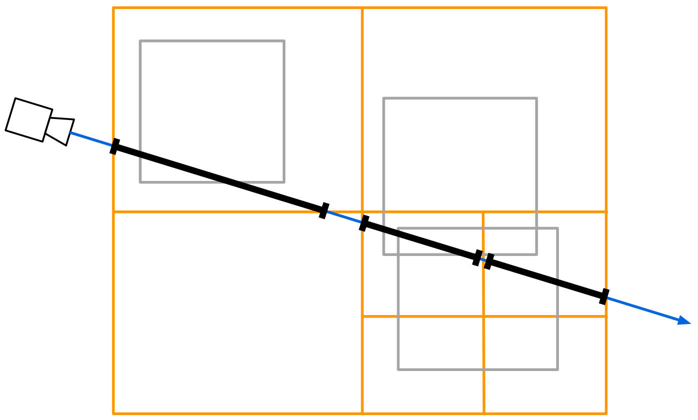


Fig. 40. Integration intervals gathered from the octree have boundaries at the octree nodes, not at the volumes' interfaces. Thus, traversal complexity is independent of how many volumes are intersected by a given ray.

- Fast meta-property queries: The meta-properties can be computed once per region, with the cost amortized across all subsequent rays that traverse the node. This dramatically decreases the cost of querying meta-properties from the scene, even when constructing fine-grain representations thereof.
- Integrator-agnostic volumes: The aggregate hides the exact bounds of the underlying volumes from the integrator, and instead exposes a hierarchical structure that can be traversed linearly without consideration of precisely where the underlying volumes' interfaces lie. Thus, individual volume never receive ray-specific queries. The volumes only need to respond to point sampling (for the tracking) and range queries (for constructing the meta-property representation), which simplifies the interface, improves efficiency while at the same time providing better meta-information to the integrator.
- Volume-agnostic integrator: The aggregate also hides all the irrelevant properties of the underlying volumes, such as their exact boundary location and the data structure used to store them.

6.9 Implementing the aggregate method

The interface presented by the aggregate to the integrator is minimal. The DDA traversal of the octree is encapsulated in an iterator class, which not only hides code complexity but also facilitates early stopping in the volume. The integrator can do its work per-segment and only increment the iterator in cases where no scattering event was found.

Listing 24. Pseudo-interface for the volume aggregate

```
class VolumeAggregate
{
    class Iterator
    {
        float start() const;
        float end() const;
        float maxExtinction() const;
        float minExtinction() const;
        bool finished() const;
        void operator++();
        int numVolumes() const;
        const Volume* volumes() const;
    }
    void insert(const Volume *v);
    Iterator iterator(const Ray &ray) const;
};
```

For a production renderer, we need to consider details such as per-sample time for motion blur and filter width for texture filtering when sampling a volume. Nonetheless, the volume aggregate approach requires a very small API to the volumes.

Listing 25. Pseudo-interface for individual volumes

```
class Volume
{
    virtual float extinction(const Point &p, float time, float filterWidth) const = 0;
    virtual std::pair<float, float> extinctionMinMax(const Region &r) const = 0;
    virtual PhaseFunction* phaseFunction() const = 0;
};
```

A simple (recursive) Woodcock-type integrator can then handle an arbitrary set of volumes in a scene trivially. Because the aggregate hides the explicit structure of the scene and the iterator hides the traversal, the resulting code remains concise.

Listing 26. Pseudo-interface for aggregate-based integrator

```
Color WoodcockIntegrator::integrate(const Ray &ray, const VolumeAggregate &aggregate)
{
    Iterator iter = aggregate.iterator(ray);
    for (; !iter.empty(); ++iter) {
        // Skip empty space
        if (iter.numVolumes() == 0) {
            continue;
        }
        // Track current position and end of segment
        float t = iter.start(), tEnd = iter.end();
        while (t < tEnd) {
            float xiStep = rng(), xiInteract = rng();
            // Take a step
            t += deltaStep(xiStep, iter.maxExtinction());
            // Did we escape the segment?
            if (t > tEnd) {
                break;
            }
            // Determine real or null-collision
            Point p = ray(t);
            float extinction = sumVolumeExtinction(iter.volumes(), p);
            if (xiInteract < extinction / iter.maxExtinction()) {
                Color L = computeDirectLighting(p);
                if (ray.depth < MAX_DEPTH) {
                    Ray indirectRay = setupIndirectRay(p);
                    L += integrate(indirectRay);
                }
                // Terminate since a real collision was found
                return L;
            }
        }
    }
};
```

6.10 Filter width and motion blur

As mentioned earlier, texture filtering and motion blur are two key aspects of production rendering. In order to support these, we must ensure that they are handled appropriately in the meta-property calculations.

The most efficient way to address motion blur is to use temporal volumes (see Section 5.6). In the temporal case, the meta-property calculation is trivial: instead of inspecting a single value for each individual voxel, we can simply perform the inspection for all temporal values over the entire shutter time interval.

Robustly handling filter width is more complex. The filter width varies along a ray, and thus varies along the line segment defined by a ray intersecting an octree node. A simple way to guarantee that the meta-property is conservative across the entire region of the aggregate node is to *pad* the lookup bounds by the filter width. This way, we capture all the values that could possibly blend into a sample point within the node region. Figure 41 illustrates this.

The question is, when we compute the meta-properties for a given aggregate node, what filter width should we assume will be used? The answer depends on the renderer: if filtering is performed using some global fixed function, for example distance-to-camera, then this constant value (constant for each aggregate node, at least) can be used and the aggregate node's value computed accordingly. However, most modern path tracers tend to have a *footprint* and *spread* associated with each ray. When spawned from the camera, the ray spread is often set to the pixel-to-pixel differential, but at scattering events, the spread may be increased to account for diffuse material response. In the volume aggregate context, this means that a given aggregate node may see a wide range of filter widths and needs to respond with an optimal meta-property range in each case.

This problem can be resolved by storing multiple instances of the meta-property at each node. In the RenderMan implementation, each time a filter width is computed, the value is rounded up to the nearest power-of-two, and the meta-property query is computed with the additional width (which still satisfies the requirement of producing a conservative majorant), but allows for better re-use of the computed value for multiple ray queries with similar ray footprints.

We note a potential optimization here. As aggregate nodes are subdivided they successively get smaller, but the the padding that must be applied to account for the filter width is constant. As a result,

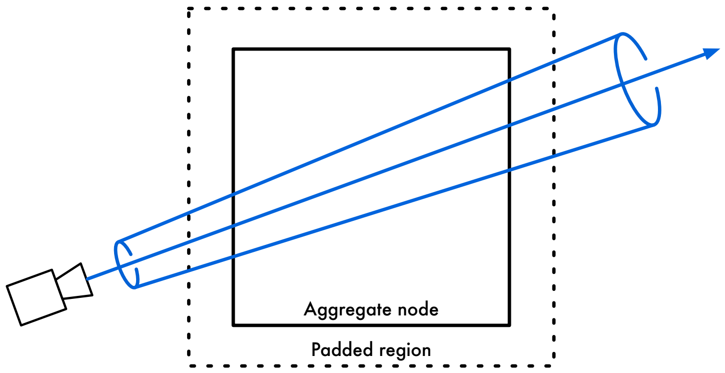


Fig. 41. When computing min/max for a given aggregate node, we expand the region by the ray's filter width in order to get a conservative value. In order to limit the number of filter width queries stored at each aggregate node, we round the value up to the nearest power-of-two.

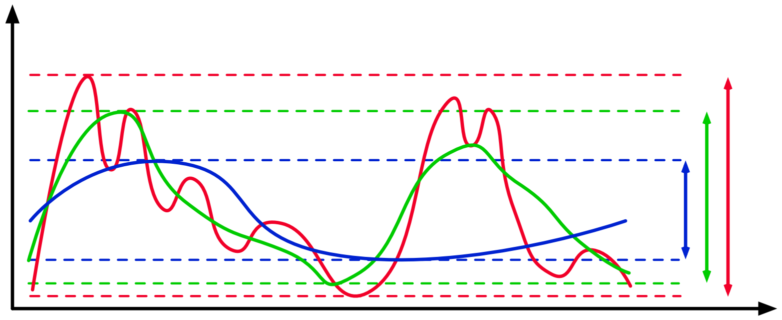


Fig. 42. The spread between max and min converges as a given signal is low-pass filtered (e.g. in MIP mapped volumes). The high resolution signal (red) is filtered at half frequency (green) and again at quarter frequency (blue). This can be leveraged for min/max queries with a large filter width.

the heuristic may force excessive subdivision since the min/max delta remains high at each division. However, if we have access to a properly filtered representation of the volumetric properties (e.g. MIP mapped volume data), then we can compute the meta-property on data which is low-pass filtered, and for which the spread between min and max is guaranteed to be narrower than for the full-resolution data. By computing a unique set of meta-property values for multiple filter widths, as described above, we avoid the problem of excessive subdivision.

6.11 Handling small volumes

One particular problem for integrators such as Woodcock trackers is heterogeneous areas with many pockets of zero-value extinction. If we recall the termination probability being the ratio between the actual point sampled extinction and the majorant used for selecting the step length, we see that any area where the true extinction is zero has zero chance of terminating! In the volume aggregate context, this often occurs when small and low-density volumes are present. The aggregate may be requested to insert a volume of density 10^{-4} and size 10 into a node of size 10^3. The volume will not likely contribute much to the image, but as far the integrator could tell, there is potentially density to find. However, the small size of the volume means that nearly all samples taken in the node have zero extinction and force the integrator to keep stepping.

A convenient solution to this corner case is to force subdivision of aggregate nodes that are larger than the volume contained, regardless of the volume's majorant extinction.

6.12 Efficient volume aggregates

So far we have assumed that the volume aggregate is pre-computed before rendering occurs, but especially with the introduction of filter width-dependence, the calculation can take several minutes, greatly hurting the time to first bucket and the overall user experience. Likewise, many renderers support deferred loading of data, and in this case there simply aren't any volumes present in the scene when the renderer first launches.

The solution to both of these problems is to create the aggregate on-the-fly. In this approach, the subdivision heuristic is evaluated each time a ray visits an aggregate node and instead determines whether or not the ray should visit the node's children or use the current node as the integration interval. We note that this implies a traversal where rays with high filter width is able to stop traversal at branch nodes whereas primary rays with narrow filter width traverse all the way to the leaf level. Likewise, if the *contrast control* method (see Section 3.7) is used, the min/max extinction values will be multiplied down, and each individual ray can adapt the subdivision heuristic with this knowledge in mind, which improves performance.

The process for creating the aggregate is:

- Start by intializing a unit cube at the origin.
- For each ray, trace against any procedural primitives to see if new volumes need to be added.
  - If the new volume is not entirely contained by the existing root node, recursively add a parent to the root in order to expand the octree.
  - Any new volumes are inserted at the root and then recursively tested to see if they overlap existing children. Where needed, the new volume's min/max properties are merged with the child node's existing values at each filter width combination, at each node.
- Once all volume that can potentially intersect the ray are added, the DDA traversal is initiated by starting at the root.
  - Compute the filter width range for the intersection between the node and the ray.
  - Compute min/max for the given filter width range, if not already present.
  - Using the heuristic function, compute whether the min/max and the length of the segment are acceptable. If not, recurse into child nodes.
    - Is child node present? If not, allocate new node.
    - Recurse into child nodes, repeating all the above steps.
  - If the segment is acceptable, break the traversal and allow integrator to process. If the integrator needs to proceed to the next segment, continue along to the next neighboring node.

The process for subdividing the volume aggregate on-the-fly is straightforward, but the recursive subdivision and insertion of volumes can be a severe bottleneck in multithreaded rendering code if not implemented carefully. Wherever possible, it is desirable to implement lock-free designs, as the number of rays that will access the data structure is likely to range in the billions. A lock-free but wasteful

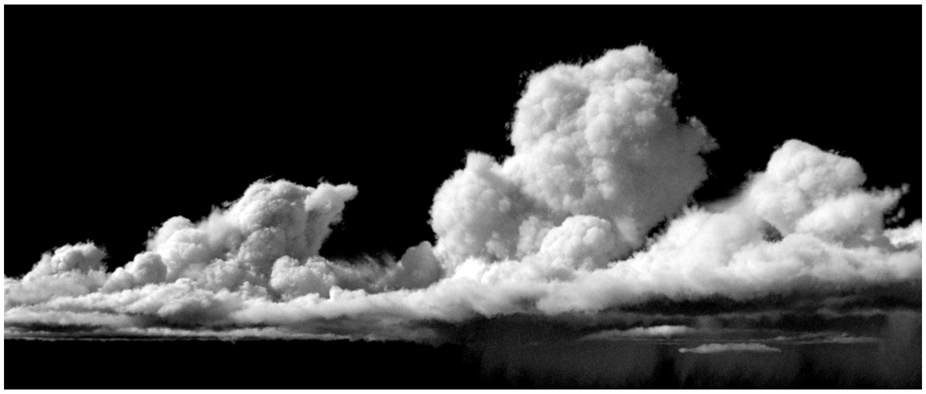


Fig. 43. 87 individual cloud pieces make up the cloudscape. 21 minutes (8 cores) at 1024 samples per pixel using 32 bounces of multiple scattering.

Table 2. Performance breakdown. Steps per camera ray and shadow ray indicates average number of steps. Rays per second indicate the speed of individual pixel samples, including all indirect and shadow rays.

Integrator	Bounces	Time	Steps / camera ray	Steps / shadow ray	Rays / second
Woodcock	1	2m 15s	15	11	2.9M
PDF	1	1h 14m 40s	182	118	87K
Woodcock	32	21m 36s	21	30	305K
PDF	32	8h 6m 10s	183	126	13K

approach will generally have better performance in this case where race conditions exists but are very rare.

6.13 Performance examples

The set of clouds in Figure 43 were rendered in a bare-bones volumetric path tracing renderer using single and multiple scattering (32 bounces) using Woodcock tracking and volume aggregates, and using a naïve implementation of the PDF method (see Section 2.4). The clouds are composed of 87 pieces, each with varying extinction multiplier. For the PDF method, integration intervals were drawn from the volume aggregate, but the subdivision heuristic was limited to ensuring that nodes were at least as small as the individual volumes, in order to provide efficient empty-space optimization. The test avoids the overhead associated with tracking the individual interfaces of each volume, and only measures the cost of integration itself.

Looking at Table 2, we note that the Woodcock tracking method is able to take very few steps per ray, thus making the cost of each individual ray low. Shadow and indirect rays tend to require fewer samples per ray than camera rays, since they originate in areas with non-zero densities and are likely to find a scattering event sooner than rays originating outside the volume. Because the PDF method traverses the entire ray and its step length is bound by the voxel size, each ray requires a much higher number of steps. Here, shadow rays use somewhat fewer steps since they tend to be shorter.

7 VOLUME RENDERING IN HYPERION

Hyperion is the in-house path tracer used at the Walt Disney Animation Studios since 2014, starting with Disney's Big Hero 6. Hyperion has been developed alongside the production of Big Hero 6 and marks the switch from a REYES Renderman to a physically-based path traced pipeline. The renderer since has been used in the Zootopia (2016) and Moana (2016).

7.1 Hyperion Streaming Architecture

A fundamental problem of physically based rendering is that incoherent secondary rays produce incoherent access to texture and geometry data as they bounce around in a scene, leading to frequent cache misses. As of 2017, production scale scenes can easily reach more than 100 GB of data containing assets with many texture layers. The incoherent nature of standard path tracing algorithms does not lend itself to coherent data accesses.

To allow for efficient path tracing of large data, Hyperion employs a two-stage ray sorting framework to create coherent memory accesses (Eisenacher et al. 2013). The strategy is to create large ray batches (30-60 million rays) during traversal that are then sorted according to their cardinal directions (see Figure44). The result of the traversal is a list of hit points which is then sorted by mesh ID and the face ID. The hit points are then shaded, accessing the textures through a texture cache in a highly coherent manner. This amortizes per-file texture costs such as opening the file and per-face costs such as decompressing besides the coherent memory access. This approach also allows for efficient lazy-loading and paging.

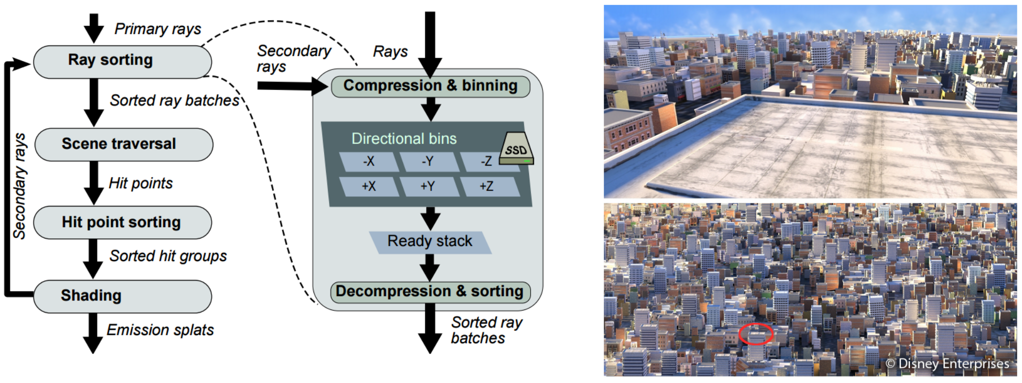


Fig. 44. Batch rendering pipeline with two sorting stages (left) and a large city data set rendered with Hyperion (right) (Eisenacher et al. 2013).

While the volume rendering in Hyperion hooks into the streaming system, it does not use the sorting mechanism. There might be possibilities to accelerate memory accesses in volumes, but the three-dimensional nature of volume data sets in context of accesses along even coherent ray directions is challenging. Hyperion uses the PDF approach, though a new system based on tracking is in development as of 2017. The volume calculations are done before the ray batches are processed and any in-scattering rays generated by the volume rendering put in the streaming batches and treated like any other ray.

7.2 Light Cache Points

Production scale scenes usually contain many complex light sources, potentially millions in the case of cities for example. In order to perform efficient light sampling, we need an approximation of incident radiance to do importance sampling. We need to be able to inform a shading point about the important lights for its position and normal direction. We distribute points, we call them light cache points, in a scene that carry this information for the valid vicinity of a light cache point. They are generated in a pre-rendering phase by tracing a small but representative number of paths and filtering and aggregating their hit points to arrive at a relatively even distribution on the surfaces and inside volumes. Figure 45 shows a simple scene and the cache point distribution in it.

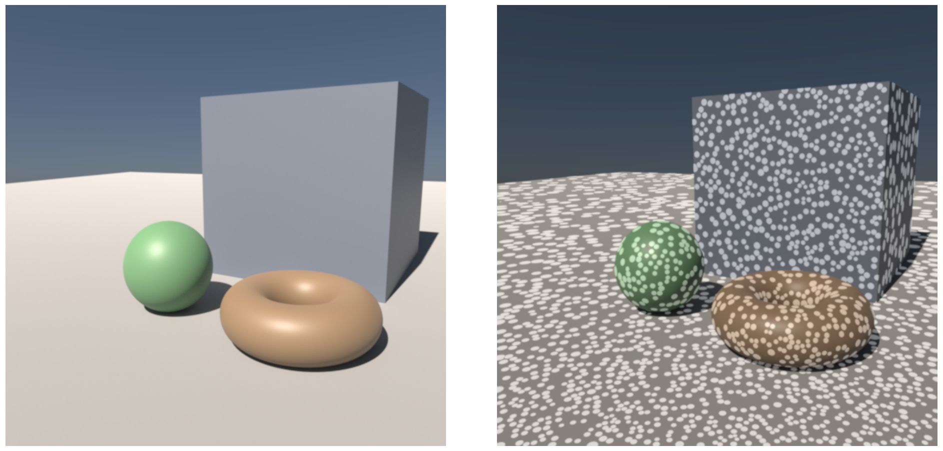


Fig. 45. A simple scene (left) and the positions of the cache points in the scene overlayed (right).

Preparing the Cache Points. In another pre-render step, we analyze each light for each cache point (with some pre-culling to avoid unnecessary tests). For each light, we calculate the solid angle and unoccluded irradiance incident to the cache point. For light primitives such as quad lights and sphere lights we can use analytic expression to calculate the analytic irradiance. After culling unimportant lights through a heuristic, the remaining lights are separated into lists, the nearby list and six distant light lists for each cardinal direction (x,y,z and negative directions). All lights that have a large solid angle from the position of the cache point go into the nearby light list. The reason to treat nearby lights separately is because the irradiance is changing significantly in position and shading normal direction. Lights that have been categorized to be distant lights are sorted into the 6 lists depending on their contribution to the cardinal directions a light list represents. The light cache points directly store a precomputed irradiance for each light. Any non-importance sampled lights are sampled along with the BSDF sampling.

Once the light cache points are generated, a kd-tree is built to be able to query the closest cache point efficiently during rendering. When a shading point queries for a light to sample, the nearest light cache point evaluates the irradiance contribution of all the nearby lights. The corresponding cardinal direction list depending on the shading normal is determined in order to generate weights for each light in the combined lists. A PDF and CDF over the weights is then generated in order to determine the light to be sampled for the shading point.

On-line Visibility Learning. In addition to the light lists, the cache points also hold a visibility weight for each light in the cache point. During rendering, a cache point keeps track of the ratio of successful light hits to occluded hits and a heuristic includes this ratio into the weight calculations. The heuristic is very conservative to avoid undersampling of highly occluded lights such as a sun light through a roof with small holes.

Efficiency of Light Cache Points. In spirit, this approach is similar to (Georgiev et al. 2012) and (Vorba et al. 2014) where an initial estimate is generated from a pre-pass which is then refined during rendering. We see a lot of the same behavior described in these publications in Hyperion. At first, the image converges slowly but once the quality of the visibility learning is high enough, the image converges fast.

Light cache points play an essential role to achieve good efficiency in complex production scenes. Especially scenes with hundreds of lights and up have a significant speedup by several factors. Also outdoor scenes where large parts of the scene is in shadow see huge performance gains as the shadowed areas are identified by the visibility learning.

7.3 Hyperion Volume Architecture

The streaming architecture of Hyperion poses several challenges to incorporate volume rendering. We do not want to interfere with the sorting and we do not have the full history of a path available like in a recursive path tracer.

We are representing any heterogeneous volume as vdb grids (Museth 2013) in memory and access them with a linear interpolator. Advection motion blur is rendered using eulerian motion blur (Kim and Ko 2007).

The volume rendering is optimized for single scattering, though for Zootopia (2016) and Moana (2016), 5-8 bounces were usually used. We apply the PDF approach with a highly optimized PDF builder. For the transmittance estimator we use residual ratio tracking (Novák et al. 2014).

7.3.1 Flux on Light Cache Points. The light cache points for surface, lights are cached according to their irradiance in cardinal directions. We can take a similar approach for volumes. The quantity relevant for volumes is flux, the overall intensity incident to a point from all sides without any cosine projection term (we do not have any defined normal to begin with in volumes). We transport a flux analogous to the irradiance with the simplification that we do not need 6 different lists to account for the surface normal.

7.3.2 Per-Volume Acceleration Grids and Estimate. Our solution is that we built our acceleration structures which performs the empty space detection and carries the information for the residual ratio tracking per volume, compared to a global data structure. We build an acceleration vdb grid at a predetermined lower resolution for each of the volumes. This allows for the acceleration grids to be optimally tuned towards its underlying voxel grids in terms of acceleration data representation and alignment of the empty space detection.

7.3.3 Transmittance Estimate. For a transmittance estimate on a ray, we perform the transmittance estimate per volume using residual ratio tracking (Novák et al. 2014), traversing through the acceleration grids one at a time and arrive at the contribution of each volume. Transmittance has the property of being multiplicative, so we can simply multiply the different contributions of volumes to arrive at to overall transmittance estimate. The main advantage of doing transmittance per volume is that no hierarchical structure needs to be traversed, so the traversal is relatively fast assuming that the volume overlap is not very high. Similar to (Kulla and Fajardo 2012), we cache the transmittance to avoid re-tracking of the transmittance estimate.

An additional advantage is that the integrator can keep track of contributions from different volumes, so it is possible to write the per volume contributions into different frame buffers to separate them out for compositing.

7.3.4 Building the In-scattering PDF. The transmittance calculation is very straight forward, the intelligence and optimizations are almost all in building a high quality in-scattering PDF with moderate computational cost. To build the PDF, we need to consider all volumes at the same time, but we have the freedom to built it as conservative or aggressive as we need to. We use an adaptive ray-marching scheme after the per volume transmittance that walks along the ray, heuristically determining the ray-march step size and and weights from all available sources (volumes and acceleration structures, cache points, etc.). We accumulate them in three separate PDFs in order to perform multiple importance sampling between them to arrive at a high-quality PDF for in-scattering sampling.

Transmittance PDF. During the transmittance estimate, we already generated the data needed for the transmittance PDF in the form of the cache that is used to avoid multiple transmittance estimates in the integration. Therefore this PDF is not generated during the adaptive ray-marching step. We simply need to normalize the cached transmittance estimate to produce the PDF.

Extinction/Albedo PDF. The second PDF is produced with the adaptive ray-marching stepping along the ray. The acceleration grids inform about the homogeneity of the volumes on intervals along the ray, besides detecting empty space. Heuristics decide if the density and albedo is probed on a finer level to increase the quality of this PDF as deemed necessary. We also incorporate values looked up during the transmittance estimate in the heuristics.

Light PDF. The third PDF is the heaviest to generate, we probe for the flux incident on a ray position. This requires a nearest-point traversal on the light cache point kd-tree, making up for the bulk of cost to generate this PDF. This PDF has similar properties to PDFs generated and sampled by equiangular sampling of lights (Kulla and Fajardo 2012), but allows to integrate any kind of light sources. But due to the kd-traversal, it takes much more resources than equiangular sampling.

7.3.5 Sampling the In-scattering PDF. To sample all three PDFs in the theoretically optimal way would be to use product importance sampling, multiplying them and drawing a sample from the resulting PDF. Unfortunately, this can lead to strongly undersampled segments on a ray since the three PDFS are not perfect. Multiplying PDFs for product sampling can lead to still valid but far too small values in the resulting PDF that would take several tens of thousand samples to converge out.

In order to combine the three PDFs, multiple-importance sampling is performed. Since we are dealing with 1D PDFs on a ray and they are in the same domain, we weight them and add them up to arrive at a single PDF that is then importance sampled.

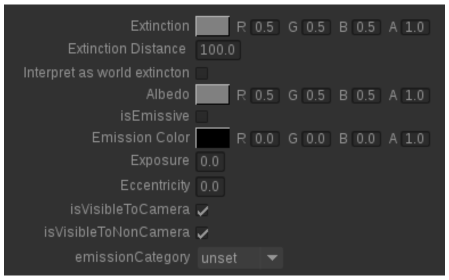
7.4 Volumes in Production Scenes

One of the biggest challenges in production rendering is that the volume rendering needs to work at least somewhat efficiently in all situations. The heuristics used need to be conservative in order to not break down in extreme situations. Additionally, production scenes usually contain hundreds of lights that may all influence volumes in the scene. Many of them may be placed inside or behind volumes for light shaping and dramatic effect. Not considering any of the three PDFs usually results in very high noise that does not converge in a feasible time.

7.5 User Parameters

We expose extinction, albedo, emission and phase function (Eccentricity in the UI) to the artists. All with the exception of the phase function can be driven by vdb grids. An expression system based on SeExpr allows to map any grid into the fundamental volume properties. For example a temperature grid can map into the resulting emission. Shaders are binding color multipliers to the underlying data in order to define constant values (such as a constant albedo) and allow for easy tuning of the extinction.

While coefficients σ are good for the mathematical formulations, they are difficult to grasp for artists to understand. A change in a multiplier to the extinction does not linearly correspond to visual changes in the volume. One advantage is though that an increase in the parameter increases the perceived density. We expose the extinction coefficient as the inverse, which is the extinction mean free path. The extinction mean free path (Extinction distance in the UI) is a distance that is expressed in scene units. Figure 46 shows the basic volume shader UI of Hyperion. Artists are educated to think in terms of how deep light penetrates a volume. In this way of thinking, thin volumes are volumes where light goes through or very deep, so the extinction distance is large. In dense volumes the light travels less deep into a volume due to a short extinction distance. Artists can build up intuition since the length of the mean free path can be related to the size of a volume, for example a cloud is 0.5 miles large, the extinction distance is 20 feet, around a hundredth of the size of the cloud.



```
                 Extinction  R 0.5  G 0.5  B 0.5  A 1.0
        Extinction Distance  100.0
Interpret as world extincton
                     Albedo  R 0.5  G 0.5  B 0.5  A 1.0
                 isEmissive
             Emission Color  R 0.0  G 0.0  B 0.0  A 1.0
                   Exposure  0.0
               Eccentricity  0.0
         isVisibleToCamera   ✓
      isVisibleToNonCamera   ✓
          emissionCategory   unset
```

Fig. 46. The basic volume shader used interface of Hyperion

7.6 Conclusion and Future

While the current system can efficiently render low-order scattering, the desire is to also be able to render the subtle effects of higher order scattering. Especially volumes with an extremely high albedo such as clouds where the look is defined by high-order scattering pose a problem in the current system.

As of 2017, Hyperion is transitioning to a new volume rendering system which allows for very high order scattering (Kutz et al. 2017) using advanced tracking methods.

REFERENCES

Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517.

H. W. Bertini. 1963. *Monte Carlo Simulations on Intranuclear Cascades.* Technical Report ORNL–3383. Oak Ridge National Laboratory, Oak Ridge, TN, USA.

L. L. Carter, E. D. Cashwell, and W. M. Taylor. 1972. Monte Carlo Sampling with Continuously Varying Cross Sections Along Flight Paths. *Nuclear Science and Engineering* 48, 4 (1972), 403–411.

Subrahmanyan Chandrasekhar. 1950. *Radiative Transfer.* Clarendon Press.

Andrew Clinton and Mark Elendt. 2009. Rendering Volumes with Microvoxels. In *SIGGRAPH 2009: Talks (SIGGRAPH '09).* ACM, New York, NY, USA, Article 47, 47:1–47:1 pages.

Michael F. Cohen, John Wallace, and Pat Hanrahan. 1993. *Radiosity and Realistic Image Synthesis.* Academic Press Professional, Inc., San Diego, CA, USA.

Robert L. Cook, Thomas Porter, and Loren Carpenter. 1984. Distributed ray tracing. *Computer Graphics (Proc. of SIGGRAPH)* 18, 3 (Jan. 1984), 137–145.

S. N. Cramer. 1978. Application of the Fictitious Scattering Radiation Transport Model for Deep-Penetration Monte Carlo Calculations. *Nuclear Science and Engineering* 65, 2 (1978), 237–253.

Roger Eckhardt. 1987. Stan Ulam, John von Neumann, and the Monte Carlo Method. *Los Alamos Science, Special Issue* (1987), 131–137.

Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. 2013. Sorted Deferred Shading for Production Path Tracing. *Computer Graphics Forum* (2013).

Iliyan Georgiev, Jaroslav Křivánek, Stefan Popov, and Philipp Slusallek. 2012. Importance Caching for Complex Illumination. *Computer Graphics Forum* 31, 2pt3 (2012), 701–710. EUROGRAPHICS '12.

M. Hapala and V. Havran. 2011. Review: Kd-tree Traversal Algorithms for Ray Tracing. *Computer Graphics Forum* 30, 1 (March 2011), 199–213.

L. G. Henyey and J. L. Greenstein. 1941. Diffuse radiation in the Galaxy. *Astrophysical Journal* 93 (Jan. 1941), 70–83.

Frederik W Jansen. 1986. Data Structures for Ray Tracing. In *Proceedings of a Workshop (Eurographics Seminars on Data Structures for Raster Graphics.* Springer-Verlag New York, Inc., 57–73.

James T. Kajiya. 1986. The Rendering Equation. *Computer Graphics (Proc. of SIGGRAPH)* (1986), 143–150.

Doyub Kim and Hyeong-Seok Ko. 2007. Eulerian Motion Blur. In *Eurographics Workshop on Natural Phenomena*, D. Ebert and S. Merillou (Eds.). The Eurographics Association.

Christopher Kulla and Marcos Fajardo. 2012. Importance Sampling Techniques for Path Tracing in Participating Media. *CGF (Proc. of Eurographics Symposium on Rendering)* 31, 4 (June 2012), 1519–1528.

Peter Kutz, Ralf Habel, Yining Karl Li, and Jan Novák. 2017. Spectral and Decomposition Tracking for Rendering Heterogeneous Volumes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2017)* 36, 4 (2017).

Tom Lokovic and Eric Veach. 2000. Deep Shadow Maps. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 385–392.

David J. MacDonald and Kellogg S. Booth. 1990. Heuristics for Ray Tracing Using Space Subdivision. *Vis. Comput.* 6, 3 (May 1990), 153–166.

Stephen Marshall, Tim Speltz, Greg Gladstone, Krzysztof Rost, and Jon Reisch. 2017. Racing to the Finish Line: Effects Challenges on Cars 3. In *ACM SIGGRAPH 2017 Talks (SIGGRAPH '17).* ACM, New York, NY, USA.

Johannes Meng, Marios Papas, Ralf Habel, Carsten Dachsbacher, Steve Marschner, Markus Gross, and Wojciech Jarosz. 2015. Multi-Scale Modeling and Rendering of Granular Materials. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 34, 4 (July 2015).

Ken Museth. 2013. VDB: High-resolution Sparse Volumes with Dynamic Topology. *ACM TOG* 32, 3 (July 2013), 27:1–27:22.

Jan Novák, Andrew Selle, and Wojciech Jarosz. 2014. Residual Ratio Tracking for Estimating Attenuation in Participating Media. *ACM TOG (Proc. of SIGGRAPH Asia)* 33, 6 (Nov. 2014), 179:1–179:11.

Ken H. Perlin and Eric M. Hoffert. 1989. Hypertexture. *Computer Graphics (Proc. of SIGGRAPH)* 23, 3 (July 1989), 253–262.

Matt Pharr and Greg Humphreys. 2010. *Physically Based Rendering: From Theory to Implementation* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Matthias Raab, Daniel Seibert, and Alexander Keller. 2008. Unbiased Global Illumination with Participating Media. In *Monte Carlo and Quasi-Monte Carlo Methods 2006.* Springer, 591–606.

Charles M. Schmidt and Brian C. Budge. 2002. Simple Nested Dielectrics in Ray Traced Images. *Journal of Graphics Tools* 7, 2 (2002), 1–8.

H. R. Skullerud. 1968. The stochastic computer simulation of ion motion in a gas subjected to a constant electric field. *Journal of Physics D: Applied Physics* 1, 11 (1968), 1567–1568.

Carlos Ureña, Marcos Fajardo, and Alan King. 2013. An Area-preserving Parametrization for Spherical Rectangles. In *Proceedings of the Eurographics Symposium on Rendering (EGSR '13)*. Eurographics Association, 59–66.

Eric Veach. 1997. *Robust Monte Carlo Methods for Light Transport Simulation.* Ph.D. Dissertation. Stanford University, Stanford, CA, USA.

Eric Veach and Leonidas J. Guibas. 1995. Optimally combining sampling techniques for Monte Carlo rendering *(ACM SIGGRAPH 1995)*. 419–428.

Ryusuke Villemin and Christophe Hery. 2013. Practical Illumination from Flames. *Journal of Computer Graphics Techniques (JCGT)* 2, 2 (31 December 2013), 142–155.

John von Neumann. 1951. Various Techniques Used in Connection with Random Digits. *Journal of Research of the National Bureau of Standards, Appl. Math. Series 12* (1951), 36–38.

Jiří Vorba, Ondřej Karlík, Martin Šik, Tobias Ritschel, and Jaroslav Křivánek. 2014. On-line Learning of Parametric Mixture Models for Light Transport Simulation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33, 4 (Aug 2014).

Ingo Wald and Vlastimil Havran. 2006. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006* (18-20). 61–69.

E.R. Woodcock, T. Murphy, P.J. Hemmings, and T.C. Longworth. 1965. Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry. In *Applications of Computing Methods to Reactor Problems*. Argonne National Laboratory.

Magnus Wrenninge. 2009. Field3D. (2009). http://github.com/imageworks/Field3D

Magnus Wrenninge. 2012. *Production Volume Rendering: Design and Implementation.* CRC Press.

Magnus Wrenninge. 2015. Art-directable Multiple Volumetric Scattering. In *ACM SIGGRAPH 2015 Talks (SIGGRAPH '15)*. ACM, New York, NY, USA, Article 24, 1 pages.

Magnus Wrenninge. 2016. Efficient Rendering of Volumetric Motion Blur Using Temporally Unstructured Volumes. *Journal of Computer Graphics Techniques (JCGT)* 1 (31 January 2016), 1–34.

Magnus Wrenninge and Nafees Bin Zafar. 2011. Production Volume Rendering 1: Fundamentals *(ACM SIGGRAPH 2011 Courses)*.

Magnus Wrenninge, Nafees Bin Zafar, Ollie Harding, Gavin Graham, Jerry Tessendorf, Victor Grant, Andrew Clinton, and Antoine Bouthors. 2011. Production Volume Rendering 2: Systems *(ACM SIGGRAPH 2011 Courses)*.

Magnus Wrenninge, Christopher D. Kulla, and Viktor Lundqvist. 2013. Oz: The Great and Volumetric.. In *SIGGRAPH Talks*. ACM, 46:1.

Magnus Wrenninge and Michael Rice. 2016. Volume Modeling Techniques in The Good Dinosaur. In *ACM SIGGRAPH 2016 Talks (SIGGRAPH '16)*. ACM, New York, NY, USA, Article 63, 1 pages.

Yonghao Yue, Kei Iwasaki, Bing-Yu Chen, Yoshinori Dobashi, and Tomoyuki Nishita. 2011. Toward Optimal Space Partitioning for Unbiased, Adaptive Free Path Sampling of Inhomogeneous Participating Media. *CGF (Proc. of Pacific Graphics)* 30, 7 (2011), 1911–1919.

Yonghao Yue, Kei Iwasaki, Bing-Yu Chen, Yoshinori Dobashi, and Tomoyuki Nishita. 2010. Unbiased, Adaptive Stochastic Sampling for Rendering Inhomogeneous Participating Media. *ACM TOG (Proc. of SIGGRAPH Asia)* 29, 6 (Dec. 2010), 177:1–177:8.

C. D. Zerby, R. B. Curtis, and H. W. Bertini. 1961. *The Relativistic Doppler Problem.* Technical Report ORNL-61-7-20. Oak Ridge National Laboratory, Oak Ridge, TN, USA.

Received May 2017; revised July 2017; revised May 2018