CHAPTER 28

Ray Tracing Inhomogeneous Volumes

Matthias Raab
NVIDIA

ABSTRACT

Simulating the interaction of light with scattering and absorbing media requires importance sampling of distances proportional to the volume transmittance. A simple method originating from neutron transport simulation can be used to importance-sample collision events of a particle like a photon with arbitrary media.

28.1 LIGHT TRANSPORT IN VOLUMES

When light passes through a volume along a ray, some of it may be scattered or absorbed according to the medium's light interaction properties. This is modeled by the medium's scattering coefficient σ and absorption coefficient α. Generally, both are functions that vary with position. Adding the two, we obtain the extinction coefficient $\kappa = \sigma + \alpha$, which characterizes total loss due to (out-)scattering and absorption.

The ratio of light that is not scattered out or absorbed for a distance s is called volume transmittance T and is described by the Beer-Lambert Law: if we follow a ray starting at position **o** in direction **d**, transmittance is

$$T(\mathbf{o},\mathbf{o}+s\mathbf{d}) = \exp\left(-\int_0^s \kappa(\mathbf{o}+t\mathbf{d})\,dt\right). \tag{1}$$

This term is prominently featured in the integral equations governing light transport. For example, the radiance scattered in along a ray for distance s is given by integrating the transmittance-weighted in-scattered radiance (according to scattering coefficient σ_s and phase function f_p):

$$L(\mathbf{o}, -\mathbf{d}) = \int_0^s T(\mathbf{o}, \mathbf{o}+t\mathbf{d})\left(\sigma_s(\mathbf{o}, \mathbf{o}+t\mathbf{d})\int_\Omega f_p(\mathbf{o}+t\mathbf{d}, -\mathbf{d}, \omega)\,d\omega\right)dt. \tag{2}$$

© NVIDIA 2019

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_28

A Monte Carlo path tracer will typically want to importance-sample a distance proportional to T. The physical interpretation is that one would stochastically simulate the distance at which an interaction occurs for a photon. The path tracer can then randomly decide if the event is absorption or scattering and, in case of the latter, continue to trace the photon into a direction sampled according to the phase function. The probability density proportional to T is

$$p(t) = \kappa(\mathbf{o} + t\mathbf{d})\,T(\mathbf{o}, \mathbf{o} + t\mathbf{d}) = \kappa(\mathbf{o} + t\mathbf{d})\exp\left(-\int_0^t \kappa(\mathbf{o} + t'\mathbf{d})\,dt'\right). \quad (3)$$

In cases where the medium is homogeneous (i.e., κ is constant), this simplifies to an exponential distribution $\kappa e^{-\kappa t}$ and the inversion method can be applied to obtain the distance

$$t = -\ln(1-\xi)/\kappa, \quad (4)$$

with the desired distribution for a uniformly distributed ξ. For an inhomogenous medium, however, this will not work, since for general κ the integral in Equation 3 cannot be solved analytically, or even if so, the inverse might not be available.

28.2 WOODCOCK TRACKING

In the context of tracking the trajectories of neutrons (where one deals with the same sort of equations as with photons), a technique to importance-sample distances in inhomogeneous media found widespread use in the 1960s. It is often called *Woodcock tracking*, referring to a publication by Woodcock et al. [5]

The idea is quite simple and based on the fact that homogeneous volumes can be handled easily. To obtain an artificial homogeneous setting, a *fictitious* extinction coefficient is added such that the sum of the actual and the fictitious extinctions equals the maximum κ_{max} everywhere. The artificial volume can now be interpreted as a mix of actual particles, which actually scatter and absorb, and the fictitious ones that will not do anything. See Figure 28-1.

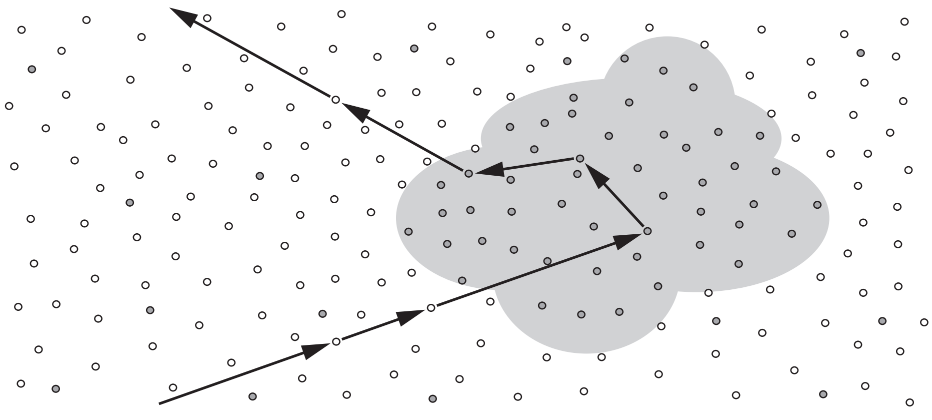

Figure 28-1. *Illustration of a path through inhomogeneous media, with high density in the cloud area and lower density around it. Actual "particles" are depicted in gray and fictitious ones in white. Collisions with fictitious particles do not affect the trajectory.*

Using the constant extinction coefficient κ_{max}, a distance can be sampled using Equation 4, and the particle will advance to that position. The collision could be a real one or a fictitious one, which can be randomly determined based on the ratio of actual to fictitious extinctions at that position (the probability of an actual collision is $\kappa(x)/\kappa_{max}$). In the case of a fictitious collision, the particle has prematurely been stopped and needs to continue its path. Since the exponential distribution is memoryless, we may simply continue along the ray from the new position by repeating the previous steps until an actual collision occurs. The precise mathematics have been described by Coleman [1], including a proof that the technique importance-samples the probability density function in Equation 3.

It is worth noting that Woodcock's original motivation was not to handle arbitrary inhomogeneous media, but to simplify and more efficiently handle piecewise homogeneous materials: treating the whole reactor as a single medium avoids all ray tracing operations with the complex reactor geometry.

Woodcock tracking is an elegant algorithm that works with any kind of medium where κ_{max} is known, and it can be implemented in a few lines of code:

```
1 float sample_distance(Ray ray)
2 {
3     float t = 0.0f;
4     do {
5         t -= logf(1.0f - rand()) / max_extinction;
6     } while (get_extinction(ray.o + ray.d*t) < rand()*max_extinction);
7
8     return t;
9 }
```

The only precaution that may be needed is to terminate the loop once the ray progresses to a surrounding vacuum. In this case no further interaction with the medium will occur and FLT_MAX may be returned. Since the procedure is unbiased, it is well suited for progressive Monte Carlo rendering.

28.3 EXAMPLE: A SIMPLE VOLUME PATH TRACER

To illustrate the application of Woodcock tracking, we present an implementation of a simple Monte Carlo volume path tracer in CUDA. It traces paths from the camera through the volume until they leave the medium. Then, it collects the contribution from the infinite environment dome, which can be configured to be an environment texture or a simple procedural gradient. For the medium we implicitly define the scattering coefficient to be proportional to the extinction coefficient by a constant albedo ρ, i.e., $\sigma(x) = \rho \cdot \kappa(x)$. All parameters defining the camera, volume procedural, and environment light are passed to the rendering kernel.

```
 1 struct Kernel_params {
 2     // Display
 3     uint2 resolution;
 4     float exposure_scale;
 5     unsigned int *display_buffer;
 6
 7     // Progressive rendering state
 8     unsigned int iteration;
 9     float3 *accum_buffer;
10     // Limit on path length
11     unsigned int max_interactions;
12     // Camera
13     float3 cam_pos;
14     float3 cam_dir;
15     float3 cam_right;
16     float3 cam_up;
17     float  cam_focal;
16
19     // Environment
20     unsigned int environment_type;
21     cudaTextureObject_t env_tex;
22
23     // Volume definition
24     unsigned int volume_type;
25     float max_extinction;
26     float albedo; // sigma / kappa
27 };
```

Since we need many random numbers per path and we require that they are safe for parallel computing, we use CUDA's curand.

```
1 #include <curand_kernel.h>
2 typedef curandStatePhilox4_32_10_t Rand_state;
3 #define rand(state) curand_uniform(state)
```

The volume data is defined to be restricted to a unit cube centered at the origin. To determine the entry point to the medium, we need an intersection routine, and to determine when a ray leaves the medium, we need a test for inclusion.

```
 1 __device__ inline bool intersect_volume_box(
 2     float &tmin, const float3 &raypos, const float3 & raydir)
 3 {
 4     const float x0 = (-0.5f - raypos.x) / raydir.x;
 5     const float y0 = (-0.5f - raypos.y) / raydir.y;
 6     const float z0 = (-0.5f - raypos.z) / raydir.z;
 7     const float x1 = ( 0.5f - raypos.x) / raydir.x;
 8     const float y1 = ( 0.5f - raypos.y) / raydir.y;
 9     const float z1 = ( 0.5f - raypos.z) / raydir.z;
10
11     tmin = fmaxf(fmaxf(fmaxf(
12           fminf(z0,z1), fminf(y0,y1)), fminf(x0,x1)), 0.0f);
13     const float tmax = fminf(fminf(
14           fmaxf(z0,z1), fmaxf(y0,y1)), fmaxf(x0,x1));
15     return (tmin < tmax);
16 }
17
18 __device__ inline bool in_volume(
19     const float3 &pos)
20 {
21     return fmaxf(fabsf(pos.x),fmaxf(fabsf(pos.y),fabsf(pos.z))) < 0.5f;
22 }
```

The actual density of the volume will be driven by an artificial procedural, which modulates the extinction coefficient between zero and κ_{max}. For illustration, we have implemented two procedurals: a piecewise constant Menger sponge and a smooth falloff along a spiral.

```
1 __device__ inline float get_extinction(
2     const Kernel_params &kernel_params,
3     const float3 &p)
4 {
5     if (kernel_params.volume_type == 0) {
6         float3 pos = p + make_float3(0.5f, 0.5f, 0.5f);
7         const unsigned int steps = 3;
8         for (unsigned int i = 0; i < steps; ++i) {
9             pos *= 3.0f;
```

```
10                const int s =
11                    ((int)pos.x & 1) + ((int)pos.y & 1) + ((int)pos.z & 1);
12                if (s >= 2)
13                    return 0.0f;
14            }
15            return kernel_params.max_extinction;
16        } else {
17            const float r = 0.5f * (0.5f - fabsf (p.y));
18            const float a = (float)(M_PI * 8.0) * p.y;
19            const float dx = (cosf(a) * r - p.x) * 2.0f;
20            const float dy = (sinf(a) * r - p.z) * 2.0f;
21            return powf (fmaxf((1.0f - dx * dx - dy * dy), 0.0f), 8.0f) *
22                  kernel_params.max_extinction;
23        }
24 }
```

Inside the volume, we use Woodcock tracking to sample the next point of interaction, potentially stopping early in case we have left the medium.

```
 1 __device__ inline bool sample_interaction(
 2     Rand_state &rand_state,
 3     float3 &ray_pos,
 4     const float3 &ray_dir,
 5     const Kernel_params &kernel_params)
 6 {
 7     float t = 0.0f;
 8     float3 pos;
 9     do {
10         t -= logf(1.0f - rand(&rand_state)) /
11               kernel_params.max_extinction;
12
13         pos = ray_pos + ray_dir * t;
14         if (!in_volume(pos))
15             return false;
16
17     } while (get_extinction(kernel_params, pos) < rand(&rand_state) *
18           kernel_params.max_extinction);
19
20     ray_pos = pos;
21     return true;
22 }
```

Now with all the utilities in place, we can trace a path through the volume. For that, we start by intersecting the path with the volume cube and then advance into the medium. Once inside, we apply Woodcock tracking to determine the next interaction. At each interaction point, we weight by the albedo and apply Russian roulette to probabilistically terminate paths with a weight smaller than 0.2 (and unconditionally

terminate paths that exceed the maximum length). If no termination occurs, we continue by sampling the (isotropic) phase function. Once we happen to leave the medium, we can look up the environment light contribution and end the path.

```
1 __device__ inline float3 trace_volume(
2     Rand_state &rand_state,
3     float3 &ray_pos,
4     float3 &ray_dir,
5     const Kernel_params &kernel_params)
6 {
7     float t0;
8     float w = 1.0f;
9     if (intersect_volume_box(t0, ray_pos, ray_dir)) {
10
11         ray_pos += ray_dir * t0;
12
13         unsigned int num_interactions = 0;
14         while (sample_interaction(rand_state, ray_pos, ray_dir,
15               kernel_params))
16         {
17             // Is the path length exceeded?
18             if (num_interactions++ >= kernel_params.max_interactions)
19                 return make_float3(0.0f, 0.0f, 0.0f);
20
21             w *= kernel_params.albedo;
22             // Russian roulette absorption
23             if (w < 0.2f) {
24                 if (rand(&rand_state) > w * 5.0f) {
25                     return make_float3(0.0f, 0.0f, 0.0f);
26                 }
27                 w = 0.2f;
28             }
29
30             // Sample isotropic phase function
31             const float phi = (float)(2.0 * M_PI) * rand(&rand_state);
32             const float cos_theta = 1.0f - 2.0f * rand(&rand_state);
33             const float sin_theta = sqrtf (1.0f - cos_theta * cos_theta);
34             ray_dir = make_float3(
35                 cosf(phi) * sin_theta,
36                 sinf(phi) * sin_theta,
37                 cos_theta);
38         }
39     }
40
41     // Look up the environment.
42     if (kernel_params.environment_type == 0) {
43         const float f = (0.5f + 0.5f * ray_dir.y) * w;
44         return make_float3(f, f, f);
```

```
45      } else {
46          const float4 texval = tex2D<float4>(
47              kernel_params.env_tex,
48              atan2f(ray_dir.z, ray_dir.x) * (float)(0.5 / M_PI) + 0.5f,
49              acosf(fmaxf(fminf(ray_dir.y, 1.0f), -1.0f)) *
50                      (float)(1.0 / M_PI));
51          return make_float3(texval.x * w, texval.y * w, texval.z * w);
52      }
53 }
```

Finally, we add the logic to start paths from the camera for each pixel. The results are progressively accumulated and transferred to a tone-mapped buffer for display after each iteration.

```
1 extern "C" __global__ void volume_rt_kernel(
2     const Kernel_params kernel_params)
3 {
4     const unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
5     const unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
6     if (x >= kernel_params.resolution.x ||
7             y >= kernel_params.resolution.y)
8         return;
9 
10     // Initialize pseudorandom number generator (PRNG);
11     // assume we need no more than 4096 random numbers.
12     const unsigned int idx = y * kernel_params.resolution.x + x;
13     Rand_state rand_state;
14     curand_init(idx, 0, kernel_params.iteration * 4096, &rand_state);
15 
16     // Trace from the pinhole camera.
17     const float inv_res_x = 1.0f / (float)kernel_params.resolution.x;
18     const float inv_res_y = 1.0f / (float)kernel_params.resolution.y;
19     const float pr = (2.0f * ((float)x + rand(&rand_state)) * inv_res_x
20             - 1.0f);
21     const float pu = (2.0f * ((float)y + rand(&rand_state)) * inv_res_y
22             - 1.0f);
23     const float aspect = (float)kernel_params.resolution.y * inv_res_x;
24     float3 ray_pos = kernel_params.cam_pos;
25     float3 ray_dir = normalize(
26             kernel_params.cam_dir * kernel_params.cam_focal +
27             kernel_params.cam_right * pr +
28             kernel_params.cam_up * aspect * pu);
29     const float3 value = trace_volume(rand_state, ray_pos, ray_dir,
30             kernel_params);
31 
```

```
32     // Accumulate.
33     if (kernel_params.iteration == 0)
34         kernel_params.accum_buffer[idx] = value;
35     else
36         kernel_params.accum_buffer[idx] =
37               kernel_params.accum_buffer[idx] +
38               (value - kernel_params.accum_buffer[idx]) /
39               (float)(kernel_params.iteration + 1);
40
41     // Update display buffer (simple Reinhard tone mapper + gamma).
42     float3 val = kernel_params.accum_buffer[idx] *
43           kernel_params.exposure_scale;
44     val.x *= (1.0f + val.x * 0.1f) / (1.0f + val.x);
45     val.y *= (1.0f + val.y * 0.1f) / (1.0f + val.y);
46     val.z *= (1.0f + val.z * 0.1f) / (1.0f + val.z);
47     const unsigned int r = (unsigned int)(255.0f *
48           fminf(powf(fmaxf(val.x, 0.0f), (float)(1.0 / 2.2)), 1.0f));
49     const unsigned int g = (unsigned int) (255.0f *
50           fminf(powf(fmaxf(val.y, 0.0f), (float)(1.0 / 2.2)), 1.0f));
51     const unsigned int b = (unsigned int) (255.0f *
52           fminf(powf(fmaxf(val.z, 0.0f), (float)(1.0 / 2.2)), 1.0f));
53     kernel_params.display_buffer[idx] =
54           0xff000000 | (r << 16) | (g << 8) | b;
55 }
```

Example renderings produced by the presented path tracer can be seen in Figure 28-2.

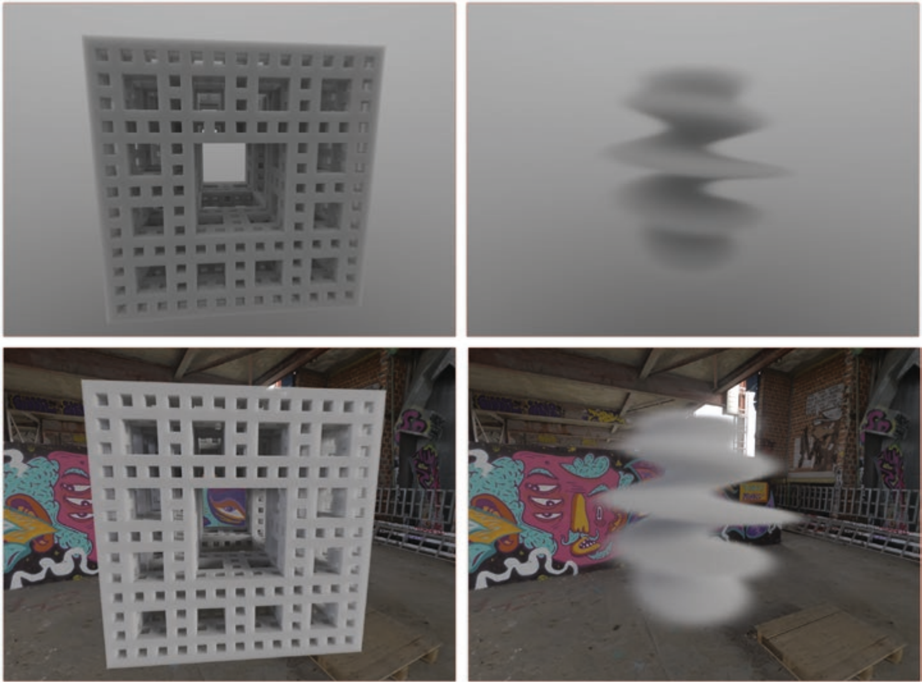

Figure 28-2. *The two procedural volume functions implemented in the sample path tracer, lit by a simple gradient (top) and an environment map (bottom). The albedo is set to 0.8 and the maximum number of volume interactions is limited to 1024. (Environment map image courtesy of Greg Zaal,* `https://hdrihaven.com`*.)*

28.4 FURTHER READING

The Woodcock tracking method can also be used to probabilistically evaluate the transmittance, as, e.g., required when tracing shadow rays through volumes. This can be achieved by sampling (potentially multiple) distances and using the ratio of those that "survive the trip" as estimate [4]. As an optimization, the random variable for continuing the path may be replaced by its expected value: instead of continuing the path with probability $1 - \kappa(x)/\kappa_{\text{max}}$, the product of those probabilities (until the distance is covered) may be used [2].

If the maximum extinction coefficient in a scene is much higher than the one typically encountered, many iterations are necessary and the method becomes inefficient. The detailed state-of-the-art report by Novák et al. [3] provides a good summary for further optimization.

REFERENCES

[1] Coleman, W. Mathematical Verification of a Certain Monte Carlo Sampling Technique and Applications of the Technique to Radiation Transport Problems. *Nuclear Science and Engineering 32* (1968), 76–81.

[2] Novák, J., Selle, A., and Jarosz, W. Residual Ratio Tracking for Estimating Attenuation in Participating Media. *ACM Transactions on Graphics (SIGGRAPH Asia) 33*, 6 (Nov. 2014), 179:1–179:11.

[3] Novák, J., Georgiev, I., Hanika, J., and Jarosz, W. Monte Carlo Methods for Volumetric Light Transport Simulation. *Computer Graphics Forum 37*, 2 (May 2018), 551–576.

[4] Raab, M., Seibert, D., and Keller, A. Unbiased Global Illumination with Participating Media. In *Monte Carlo and Quasi-Monte Carlo Methods*, A. Keller, S. Heinrich, and N. H., Eds. Springer, 2008, pp. 591–605.

[5] Woodcock, E. R., Murphy, T., Hemmings, P. J., and Longworth, T. C. Techniques Used in the GEM Code for Monte Carlo Neutronics Calculations in Reactors and Other Systems of Complex Geometry. In *Conference on Applications of Computing Methods to Reactor Problems* (1965), pp. 557–579.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.