Nikolai Hofmann, Marc Stamminger *Erlangen, 17/07/2025*

Global Illumination Tutorials (exercise sheet 5)

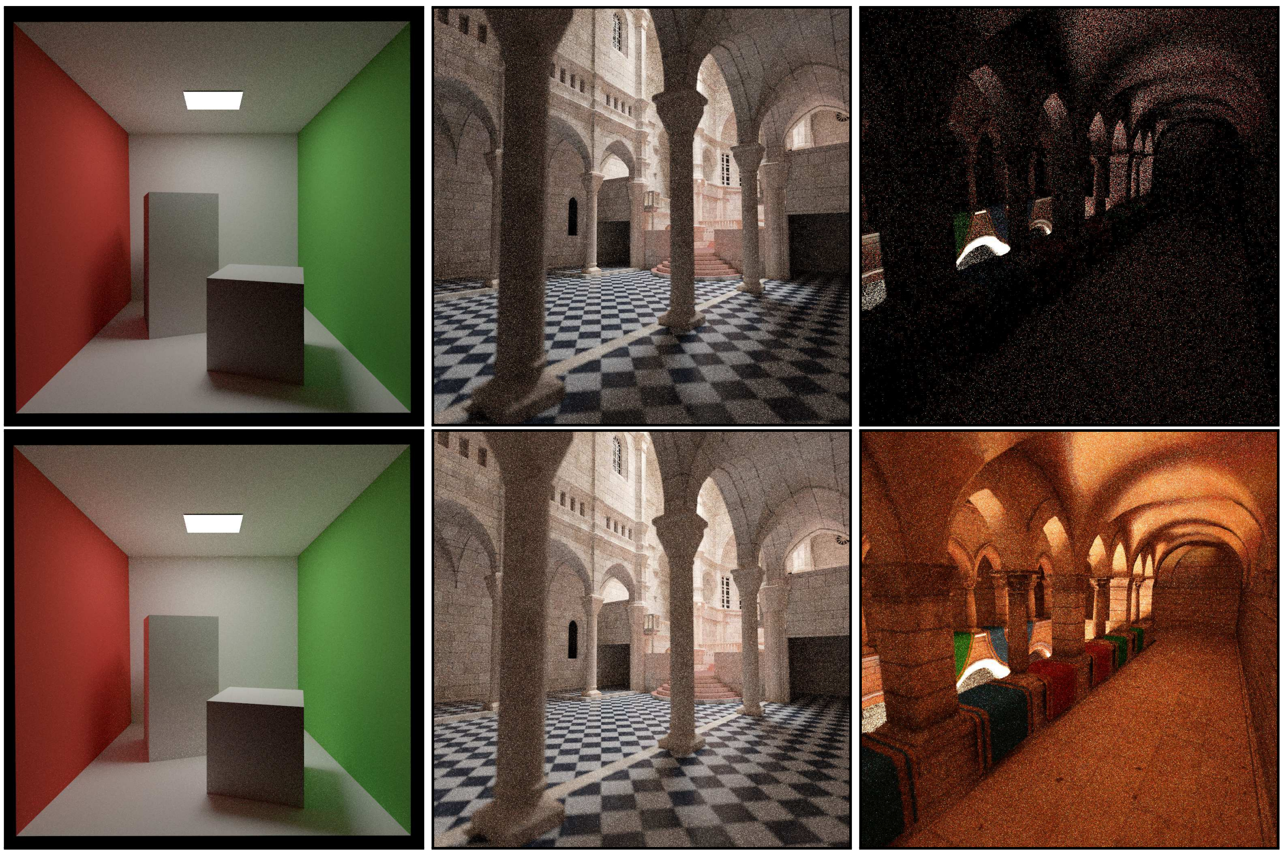

Figure 1: Example renderings using unidirectional path tracing from the last exercise (top row) and bidirectional path tracing from this exercise (bottom row). The config files used are: `a05_box.json` (left column), `a05_sibenik.json` (middle column) and `a05_spz.json` (right column).

In this final assignment, we will look into rendering with bidirectional path tracing (BDPT) and participating media, i.e. volumes. Bidirectional algorithms are more adept in gathering indirect lighting effects than standard unidirectional path tracing, since paths are traced from both the camera and the light source. This way, even hard-to-reach light sources will be sampled more often which can cause renderings to converge more quickly. Participating media breaks the assumption of rays travelling in a vacuum and is modeled by statistically independent particles with different absorption and scattering properties. White clouds have a high scattering probability, while dark smoke has a high absorption probability, for example.

Lecture: on Tuesdays from 12:15 - 13:45
Tutorials: on Thursdays from 09:00-12:00
Submission deadline: Wednesday 30.07.2025, 23:55
Grading period: Thursday 31.07.2025, 09:00-12:00

Assignment 1 [4 Points] (Bidirectional Path Tracing: Connecting Vertices)
[Files: `src/gi/bdpt.cpp`]

We will start with implementing bidirectional path tracing. The tracing of both camera and light paths is already implemented and provided to you (see `Bidirectional::sample_pixel`). You are provided with two `std::vector<PathVertex>`, containing all vertices along the camera and light paths, respectively. You may want to have a look at the `PathVertex` struct in `src/gi/bdpt.h`. Your task is to connect all camera and light vertices, efficiently generating `cam_path.size()` × `light_path.size()` radiance transporting paths.

When connecting two vertices, shoot a shadow ray between them to test for occlusion and compute the appropriate `BRDF` and `G` terms between the two vertices. Don't forget to apply the throughput and PDF of the preceding path for both the camera and light vertices. For simplicity, you may ignore escaped rays and handling environment lights here (see `PathVertex::infinite` and `PathVertex::escaped` flags). Note that the first vertex along a camera path is the primary hit point from the camera view ray (not a point on the camera lens) and the first vertex along a light path is a point on the light source. For vertices lying directly on a light source (see `PathVertex::on_light` flag), we want to skip evaluating the BRDF.

When restricting the light path to a length of one (`Context.MAX_LIGHT_PATH_LENGTH` = 1, see "Render settings" tab via the GUI or config files), i.e. containing only the vertex on the light source, your implementation of BDPT should produce identical results to a path tracer with next event estimation. You can verify your implementation against the renderings in Figure 1.

Assignment 2 [2 Points] (Volume Rendering: Ray Marching)
[Files: `src/gi/volume.cpp`]

In order to enable rendering with participating media, we need to support two queries: *transmittance evaluation* and *distance sampling*. Transmittance can be thought of the ratio of photons that passes through the volume without interacting with a particle. This is used to query volumetric attenuation along a shadow ray, for example. Distance sampling queries the free-flight distance along a ray, which is the distance light can travel through the volume until an interaction with a particle occurs. Such an interaction can either be absorption, in-scattering, or out-scattering. We will not consider volumetric emission in this assignment.

In general, a volume is modeled using the density of statistically independent particles in a unit volume ($1/m^3$). However, in order to sample for volume interactions along a ray, we need the probability of interaction per unit distance ($1/m$). To this end, we model the cross-sectional areas (m^2) of the absorbing (σ_a) and scattering (σ_s) particles in the volume, respectively. Using these cross-sectional areas, we can effectively tune the appearance of a volume between a white cloud with predominantly scattering interactions, or a dark smoke plume where absorption is predominant. By multiplying the cross-sectional areas with the density of particles at a point in the volume, we get the probability of the respective interaction per unit distance, which we call the absorption (μ_a) and scattering (μ_s) coefficients.

The *transmittance* $T(x, y)$ between two points, i.e., the loss of radiance due to absorption and out-scattering, is given via the Beer-Lambert law:

$$T(x,y) = e^{-\int_x^y \mu_t(x+t\omega)dt}\ , \tag{1}$$

where $\mu_t(x)$ is the extinction coefficient, which defines the probability of an absorption or out-scattering event in the volume: $\mu_t(x) = \mu_a(x) + \mu_s(x)$. To find the transmittance, we effectively need to integrate $\mu_t(x + t\omega)$ along the ray segment, where t is the distance along the ray segment between the

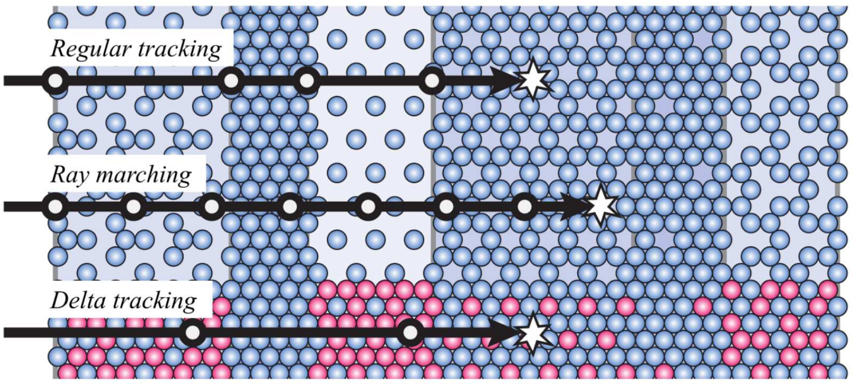

Figure 2: Illustration of different free-path sampling strategies in a medium composed of several homogeneous regions, such as voxels for example. Blue particles represent real, and pink particles represent fictitious matter. Regular tracking finds all boundary crossings and solves for the position analytically. Ray marching steps with a constant stride until it accumulates the sampled optical thickness. Delta tracking first homogenizes the volume using fictitious matter and returns to the correct distribution using rejection sampling.

entry point x and exit point y. This line integral is also called *optical thickness* τ, which simplifies the Beer-Lambert law to: $T(x,y) = e^{-\tau}$. Ray marching computes the optical thickness via taking constant steps through the volume, see Figure 2. Implement `Volume::transmittance_raymarching` and use `configs/a05_smoke_debug.json` and the top left image from Figure 3 to verify your implementation.

Since transmittance is the fraction of photons not interacting with the volume, the CDF to sample an interaction with the volume is: $1 - T(x,y)$. If we reorder for t, we get:

$$t = -\frac{\ln(1-\xi)}{\mu_t}\,. \tag{2}$$

However, this assumes the extinction coefficient μ_t as constant. With spatially varying extinction coefficients, however, we again need to integrate τ using ray marching. To this end, we sample a target optical thickness instead using $\tau_{target} = -\ln(1-\xi)$ and accumulate until we have exceeded the sampled τ_{target}. In this case, make sure to linearly revert to the point where the accumulated optical thickness matches the sampled optical thickness exactly. Implement `Volume::sample_raymarching` and use `configs/a05_smoke_debug.json` and the bottom left image from Figure 3 to verify your implementation.

Assignment 3 [2 Points] (Volume Rendering: Delta and Ratio Tracking)
[Files: `src/gi/volume.cpp`]

Ray marching is a *biased* method, which introduces some extent of error in the rendering, even with an infinitely large sampling budget. Hence, we now look into unbiased volume sampling methods, namely *ratio tracking* for transmittance estimation and *delta tracking* for distance sampling. The common idea for both approaches is to first homogenize the volume by introducing additional *null particles*, which have no effect on light transport. In other words, we essentially “fill” the volume with null particles until the maximum density value of the volume is reached everywhere, effectively turning it into a homogeneous volume. This significantly simplifies the sampling routines to as if the medium were homogeneous, but we now need to reject all collisions with null particles in order to return to the correct distribution.

We first implement distance sampling using delta tracking. To this end, we step through the volume as if it were homogeneous (using its maximum density value) to find collisions with *both* real and null particles:

$$\Delta t = -\frac{\ln(1-\xi)}{\mu_{t\_max}} \,. \tag{3}$$

In order to reject collisions with null particles and return to the correct free-flight distribution, we need to classify each collision as real or null. The chance of a collision being accepted as real is equivalent to the ratio of real to null particles. Thus, we can stochastically classify the collision via drawing a random sample in $[0,1)$ and comparing to the ratio of real to null particles. In case of a null collision, we continue the algorithm unaffected. As soon as a collision is classified as real, we terminate and return the free-flight distance. Implement `Volume::sample_delta_tracking` and use `configs/a05_smoke_debug.json` and the bottom right image from Figure 3 to verify your implementation.

In order to find the ratio of particles passing through the volume unaffected, i.e. the transmittance, we replace the binary termination criterion in the delta tracking algorithm and keep track the ratio of null to real particles instead. We start with $T(x,y) = 1$ and at each tentative collision, we update the transmittance by the ratio of null to real particles, i.e. remove the fraction of photons which have collided with real particles from the estimate. Implement this approach in `Volume::transmittance_ratio_tracking` and use `configs/a05_smoke_debug.json` and the top right image from Figure 3 to verify your implementation. Compare your results to the ones using ray marching. Where are the differences most obvious? What is the cause of this bias?

Assignment 4 [2 Points] (Volumetric Path Tracing)
[Files: `src/gi/volpath.cpp`]

So far, we considered volume sampling strategies in isolation. In order to render images with participating media, we obviously need to account for volume interactions in a path tracer as well. The volumetric rendering equation is defined as follows:

$$L_i(x,\omega) = T(x,y)L_o(y,-\omega) + \int_x^y T(x,x')L_s(x',-\omega)dt \,, \tag{4}$$

where $T(x,y)$ is the transmittance between x and y, $L_o(y,-\omega)$ is the emitted radiance of a surface at position y in direction $-\omega$, y the point on the closest surface along the ray, and $x' = x + t\omega$. We already know how to compute $L_o(y,-\omega)$ from the previous assignment. $L_s(x,\omega)$ is the in-scattered radiance, which is defined as follows:

$$L_s(x,\omega) = \mu_s(x) \int_{S^2} f_p(x,\omega')L_i(x,\omega')d\omega' \,, \tag{5}$$

where $\mu_s(x)$ is the scattering coefficient at x and $f_p(x,\omega')$ the *phase function*, which defines the directional scattering properties of a volume, and you may think of it as the equivalent of a BRDF.

In other words: during rendering, we sample for collisions with the volume and for each volume interaction, we apply NEE as with surfaces. However, we now attenuate the light's contribution by transmittance and weight it using the phase function and scattering coefficient, instead of the BRDF and cosine of the incoming angle for surfaces. We then scatter the ray according to the phase function, update the throughput and continue the path. Surface intersections are treated like before, except that we additionally need to account for attenuation by the volume.

Add volume support to the path tracer in `src/gi/volpath.cpp`. Render and compare the configs `a05_cloud_march.json`, `a05_cloud_track.json` and `a05_sibenik_smoke.json` to the renderings in Figure 3. Rendering might take a bit longer with participating media, by the way.

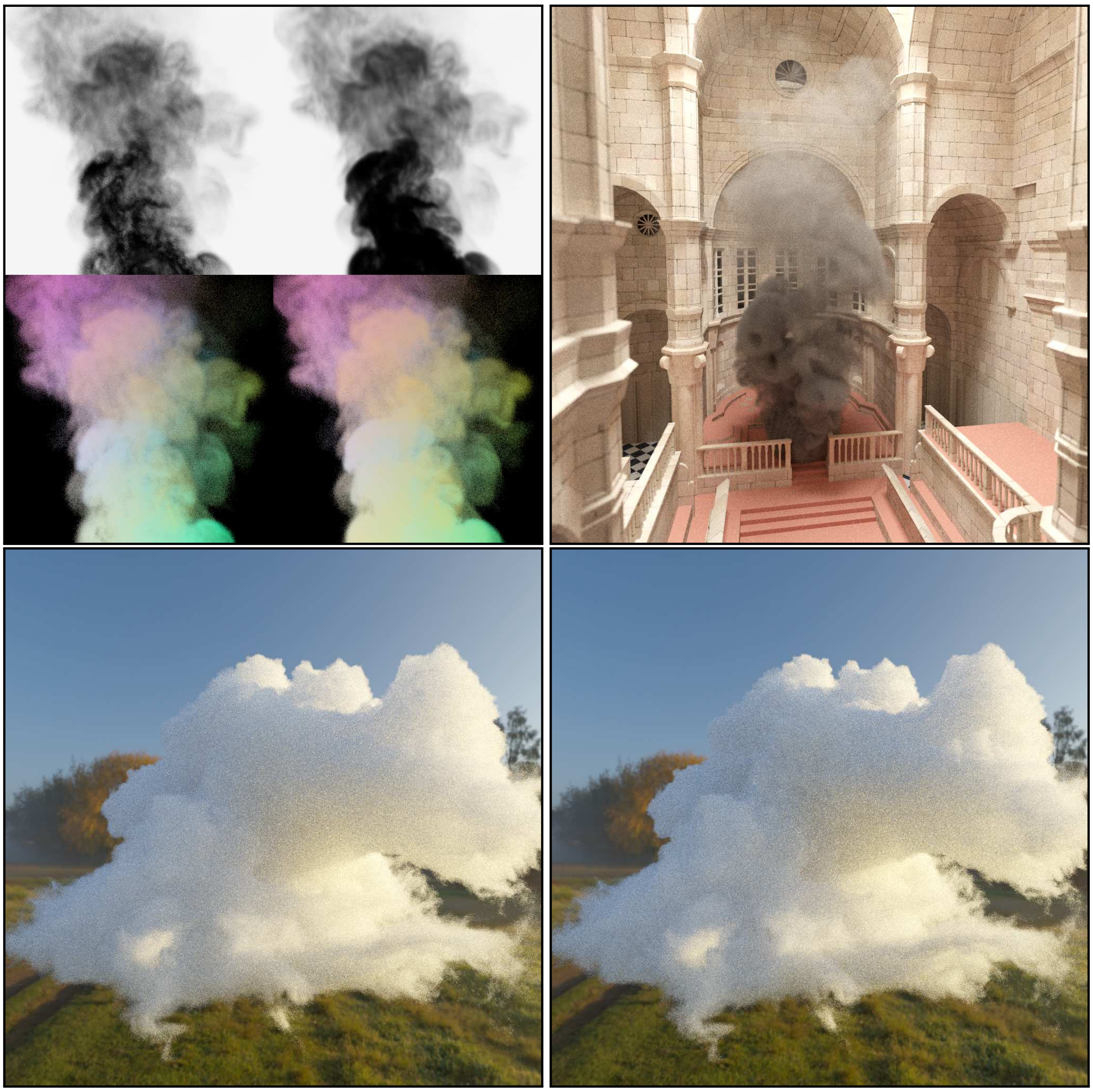

Figure 3: Example volumetric renderings for reference. Top left: Debug visualizations for transmittance estimation and distance sampling using both ray marching and ratio/delta tracking (`configs/a05_smoke_debug.json`). Top right: The Sibenik cathedral rendered using participating media (`configs/a05_sibenik_smoke.json`). Bottom left: A cloud using volumetric path tracing and ray marching (`a05_cloud_march.json`). Bottom right: The same scene rendered using delta and ratio tracking (`a05_cloud_track.json`).