

Advanced Real-Time Volume Graphics

SIGGRAPH 2026 Course Notes

STEFAN ZELLMANN, University of Cologne, Germany

ALPER SAHISTAN, University of Utah, USA

INGO WALD, NVIDIA, USA

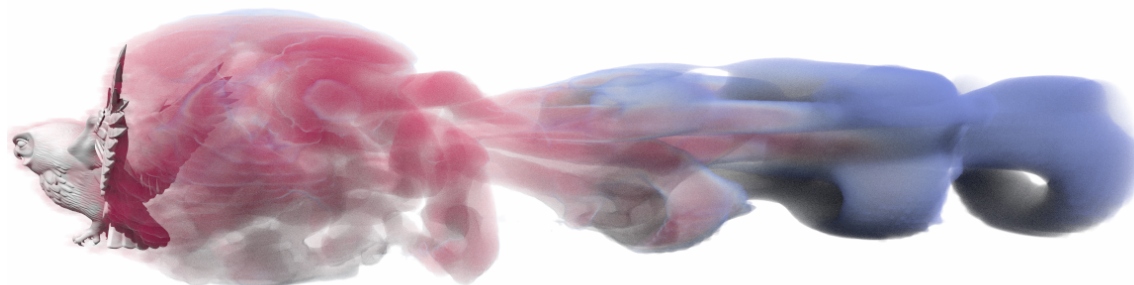


Fig. 1. Direct volume rendering is a standard method to visualize large fluid simulation data. In the visual effects (VFX) field volumetric path tracing is a common technique. Though the two techniques fundamentally solve the same light transport problems, the fields have diverged. Our course aims at bridging the two communities to adopt techniques from the other field.

Real-time volume graphics is one of the most matured methods in scientific visualization. About 25 years ago, when this topic became so popular, the most urgent challenge was how to use the limited programming models of GPUs from these times to efficiently implement volume rendering algorithms. Fastforward more than 20 years from the last SIGGRAPH course on this topic, the vast majority of volume renderers these days uses ray tracing models to accumulate samples, either driven by parallel C++ programs or by GPGPU kernels.

All the clever, yet arcane, methods to force the GPU hardware into sampling and compositing the volume rendering equation through its rasterization pipeline appear to be long obsolete by now; yet new challenges arose since the field has matured that can be largely attributed to the volumes' sizes. Different field types than voxel grids have become more important especially in times of Exascale. As the sample count is highly correlated with the volume size, the importance of space skipping techniques has increased dramatically. With the new compute power of modern GPUs at hand the use of Monte Carlo sampling for the volume rendering integral has become an option even for real-time graphics; and finally, the sheer sizes of the volumes demand a new look at data parallel rendering (and sample compositing) beyond such volumes that are easily decomposed into simple blocks.

The course develops a ray tracing framework for scientific volumes. It starts by introducing simple voxel grid integrators sampling at equidistant steps. The course proceeds by showing how to support different volume types within this same framework. A paradigm

Authors' Contact Information: Stefan Zellmann, University of Cologne, Cologne, Germany, zellmann@uni-koeln.de; Alper Sahistan, University of Utah, Salt Lake City, Utah, USA, asahistan@sci.utah.edu; Ingo Wald, NVIDIA, Salt Lake City, Utah, USA, iwald@nvidia.com.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

Manuscript submitted to ACM

Manuscript submitted to ACM

of segment traversal and sample accumulation along segments is established to show how to implement different ways of space skipping; this is complemented by adding Monte Carlo integrators to the framework that have a special relationship to transfer functions and space skipping techniques. We finally shed light on how all that ties into data parallel rendering, and provide cues how to integrate a framework like this into production systems like ParaView.

CCS Concepts: • **Computing methodologies** → **Rendering; Ray tracing**; • **Human-centered computing** → **Scientific visualization**.

Additional Key Words and Phrases: Volume Rendering, Scientific Visualization, GPGPU, Ray Tracing, Volume Path Tracing, Adaptive Volume Sampling, RT Cores

ACM Reference Format:

Stefan Zellmann, Alper Sahistan, and Ingo Wald. 2026. Advanced Real-Time Volume Graphics SIGGRAPH 2026 Course Notes. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Courses (SIGGRAPH Courses '26)*, July 19–23, 2026, Los Angeles, CA, USA. ACM, New York, NY, USA, 29 pages. <https://doi.org/10.1145/3799820.3812500>

1 Introduction

This course teaches the current state of affairs in direct volume rendering for scientific visualization (sci-vis). Volume rendering is a mature field; as a visualization technique, volume rendering is usually quite data heavy and dominated by sampling and memory operations. GPUs with their distinct execution model and the way they streamline those sampling operations across many threads played a major role helping this field mature. A lot has changed in the field in recent years though, be it the availability of more capable hardware, including the addition of ray tracing units, the wide availability of performance portable compute APIs, and last but not least libraries like OSPRay [Wald et al. 2017] and OpenVKL [Intel Corporation 2023] pioneering more sophisticated interactive volume visualization by embracing ray tracing and path tracing techniques. In the remainder of this section we give a brief motivation why we think this course addresses a timely matter that is of importance to our community. We discuss how we intend a course participant to proceed through the course material and finally introduce the sample code framework accompanying the course. The course also comes with a slide deck presented at conferences; while the slides and sample code framework focus heavily on practical and implementation matters, the course notes are more detailed and also provide theoretical background that the other materials omit for brevity’s sake.

1.1 Motivation

Our course is an unofficial sequel to a course on real-time volume rendering held at conferences like SIGGRAPH or Eurographics between 2004 and 2006, by Engel *et al.* [Engel et al. 2004]. Engel’s course summarizes and provides recipes to write volume renderers with the means available at that time, including contributions from authors shaping this field and making interactive volume rendering a reality on contemporary GPU hardware. We leave a more detailed discussion of the then status quo (and how that influenced our contribution) to Section 2. We note though that at that time, programming interfaces and libraries for GPUs exposed a fixed set of capabilities supported by the hardware, rather than providing general purpose APIs. The 2004 course reflects this and describes ways to make the hardware from those times operate at high efficiency. With that, the course’s contribution (and the contribution of countless papers on this topic at that time) was enabling real-time volume graphics on consumer hardware and workstation PCs from that era. Many of the techniques live on in production-grade software libraries like VTK [Schroeder et al. 2006].

Given the new capabilities of modern hardware, modern APIs, wide availability of (hardware, and efficient software-accelerated) ray tracing, we observe a gradual shift where libraries use more sophisticated techniques to realize volume

renderers in sci-vis, by adopting methods from Monte Carlo rendering usually seen in the field of visual effects (VFX). The VFX field looks on an active community of practitioners from the industry and academics as well. Noteworthy are SIGGRAPH courses on production volume rendering in general [Fong et al. 2017], and on physically based Monte Carlo volume rendering in particular [Novák et al. 2018]. These courses take a completely different approach to volume rendering. At the time those courses were presented, the VFX industry had just recently pivoted to using physically based techniques and especially path tracing in production, and the courses address many technical issues, and present solutions for better importance sampling and variance reduction for volume rendering in this context.

In recent years those two communities—sci-vis and VFX—have gradually started converging on how they approach volume rendering; yet there are also fundamental differences in terms of demands for interactivity, data set sizes, visual fidelity, or faithfulness of the visualization. In short, we believe that members of the two communities can learn from each other in how they approach volume graphics, exchange lessons learned, and consolidate which techniques do not work in the light of certain assumptions, or what optimizations are of practical relevance, given the data typically used in the respective fields. It is thus ultimately the goal of our course to bring practitioners from those fields together to learn from each other, and hopefully helping physically based techniques being more widely embraced by sci-vis apps.

1.2 How to proceed through this course

Our course comes with three additional material items: a slide deck first presented at SIGGRAPH 2026, a sample code framework available on GitHub.com, and the course notes (this document). Each of those are viable entry points to familiarize with the course. We assume that course attendees reach for those resources offline and work through them at their own pace, possibly but not necessarily after attending the course in person.

The slide deck and sample code framework provide a more hands-on, implementation-focused view on the topic of volume rendering. While introducing some theory, the goal of those resources is to set the reader up to start writing their own volume renderers. The course materials provide simple recipes; as many of the topics discussed—especially the ones focusing on optimization—heavily depend on the specific data that is visualized, on its size, and on its topology, we do not provide “one fits all” solutions to all possible problems occurring in sci-vis, but rather provide example programs that are simple enough that users can extend them to their own needs. In many cases, multiple alternative solutions are viable, which is particularly the case in the later sections focusing on multi-GPU and data parallelism. The slides reflect this, by practically introducing the overall problem space, and referring to the individual samples as one tiny (yet from our perspective noteworthy) subject matter from that problem space and how to address it.

The course notes on the other hand provide a more gentle introduction. They approach the subject at a different pace, taking the time to introduce some theory in more detail where needed, while leaving the gory details to prior work where a theoretical discussion would be too excessive and does not support the point in question. The course notes are hence meant to be read more like you would read a book, rather than a technical document.

The first part of the course notes concentrates on the status quo in sci-vis. This part starts with a motivational section on how volume rendering was done in the early days. That section mostly serves as motivation and can possibly be skipped upon first read by an audience only interested in learning how to write volume renderers today. This is followed by a short theoretical section on light transport in participating media. In the following, we assume that the reader has some general familiarity and at the very least an intuition for phenomena like scattering, absorption, or emission and their implications for the integral equations that a volume renderer has to evaluate numerically. We will later also refer to specific equations from this section from time to time; yet a reader with general familiarity can skim through this section as well. The following section gives a more practical introduction on how to implement absorption

plus emission-style ray marchers given modern APIs; this chapter fundamentally re-imagines the status quo of how many libraries and apps in sci-vis still implement volume rendering, yet now embracing modern techniques such as ray tracing APIs and sparse voxel libraries. The reader should at least familiarize themselves with the general approach and the most important concepts introduced in this section.

The next part introduces Monte Carlo techniques to the framework, gradually replacing and then extending the methods introduced before. This part of the notes introduces several techniques from Monte Carlo volume rendering—yet applied to sci-vis—that are at the heart of this course and contain the main arguments we intend to convey. These include a shift from ray marching techniques to adopting free-flight distance tracking and sample accumulation. Their applicability is discussed on several examples, including multi-channel and multi-volume rendering, or rendering with advanced lighting models.

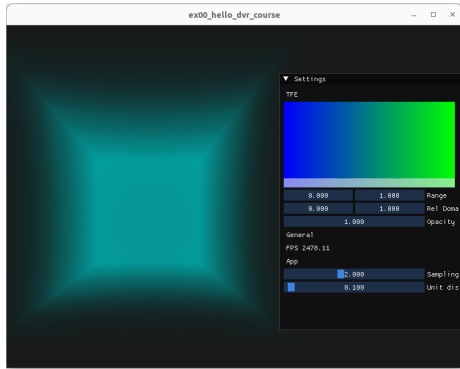
We then discuss topics related to performance, the use of ray tracing cores for sampling and traversal, and how to integrate the volume rendering algorithms into a more holistic framework, also supporting other primitives such as surfaces. This is followed by sections on more advanced subjects, such as space skipping and adaptive sampling (which we consider advanced as they provide optimizations, but they are also pivotal to our framework and in our opinion should not be skipped), data parallel volume rendering, and finally on how to integrate those techniques and the final framework as plug-ins into production software libraries.

1.3 Sample code framework

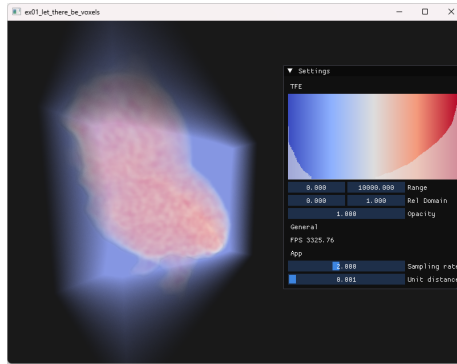
The course is accompanied by a set of small, portable C++ sample programs released under <https://github.com/dvr-course/dvr-course-cpp>. The purpose of this framework is to provide recipes that show how the techniques discussed in this course can be realized in practice. The samples claim to be nowhere near complete, nor do they represent a production-ready rendering system. They are self-contained, simplified examples that incrementally build on each other and can form a basis for a rendering system implemented by the reader that has “all the bells and whistles”. The final examples present integration with a full rendering stack, including support for triangle geometry, different volume types living in the same application context, and can be loaded as a plug-in into VTK. This is realized using the Khronos ANARI API.

The software framework embraces hardware ray tracing in CUDA/C++ using NVIDIA OptiX. To hide some of OptiX’s complexity we use NVIDIA’s OptiX wrapper library OWL [NVIDIA Corporation 2026]. The framework can be compiled without OWL, in which case not all of the samples have full functionality—particularly the later ones making use of ray tracing features for volume element sampling or for rendering triangle geometry. For bootstrapping, the framework uses CMake. The sample applications can be compiled in an interactive mode, using SDL3 for windowing, or in command line mode where the app outputs an image to the current working directory upon termination. The final sample includes an ANARI “device”, which compiles into a shared library loaded as a plug-in into ANARI client applications. Possible client applications would be ParaView or VTK; a simplistic ANARI client is also included in the final sample. Some of the sample apps (run in interactive mode) are shown in Fig. 2. We later refer to the samples whenever it is relevant for the course notes.

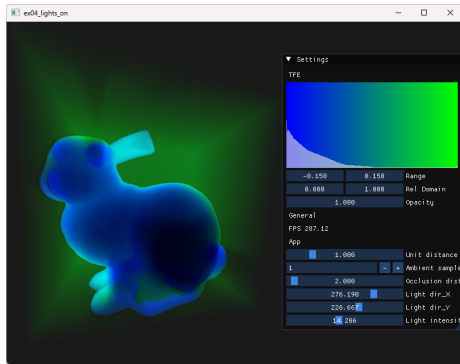
To compile the sample code framework, the reader should **git-clone** the repository and create a separate build directory where CMake and other temporary files will go during bootstrapping and compilation. Third-party software required are a C++ compiler, CMake, CUDA on systems where this is supported, and NVIDIA OWL on systems with support for OptiX (OptiX headers come bundled with OWL, while OptiX binaries are part of the NVIDIA driver). For the final example the ANARI SDK (<https://github.com/KhronosGroup/ANARI-SDK>) must be installed. The framework



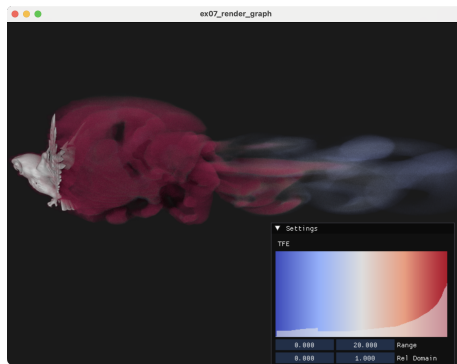
(a) “Hello DVR Course” sample, uses sci-vis ray marching to sample a homogeneous volume with an RGBA transfer function.



(b) Voxel rendering sample, adds NanoVDB volumes as a data source for voxel grids.



(c) Direct lighting sample, extends the pipeline to support single scattering, and is used for discussion of different tracking algorithms.



(d) Render graph sample, combines the techniques discussed in this course and adds them into a more versatile renderer supporting complex geometry.

Fig. 2. Sample apps. The course is centered around a framework of ten sample apps coming with source code that we gradually extend with more advanced volume rendering features. A subset of those apps is shown in this figure.

exposes two CMake flags: **DVR_COURSE_WITH_OWL** and **DVR_COURSE_PIPELINE_INTERACTIVE**, each of which enable fallback modes when initialized to the value **OFF**. When compiled in interactive mode, CMake will either try to find a pre-installed version of the SDL3 library, or downloads a copy of the SDL3 source code and compiles it during bootstrapping. That allows a CPU fallback version to be built with minimal dependencies. When compiled with OptiX and OWL, the C define **RTCORE** is available to the CUDA/C++ code in files both compiled for the host and for the device, to identify that OptiX features are used. We also refer to the unofficial OWL clone fakeOWL that our framework was tested with: <https://github.com/szellmann/fakeOWL>. fakeOWL provides an emulation layer with a software fallback for device programs and bounding volume hierarchy (BVH) traversal (if compiled with fakeOWL, the flag **RTCORE** will be defined); i.e., the full OWL feature set is available, yet with limited performance due to execution on the CPU. This variant of OWL is experimental and intended to provide another way to make the framework more approachable.

The course notes will from time to time refer to the sample apps, or discuss code snippets included in the samples. A detailed discussion is beyond the scope of the course notes though. The sample code framework includes extensive

documentation and **readme** files and as such should provide a self-contained learning resources in themselves. The slide deck accompanying the course also discuss the samples in more detail.

2 Real-Time Volume Graphics: Status Quo and How it Evolved

This section is meant as a gentle introduction to direct volume rendering (DVR) in general. It gives a brief overview of how volume rendering in sci-vis has evolved, and includes a brief theory section to bring the reader up to speed to be able to follow the explanations on volumetric path tracing in later chapters. It also presents volume ray marching with the absorption plus emission model as the baseline in sci-vis.

2.1 Real-time volume graphics and its role in scientific visualization

Volume graphics plays a major role in three different fields: visual effects (VFX) rendering, computer games, and scientific visualization. Each of those fields has different demands for volumetric rendering. While in production rendering interactivity and real-time aspects are less important than realism, in computer games the opposite is the case. In first approximation, in VFX creating volume rendered images within minutes is acceptable; for scientific visualization, interactive frame rates that allow for interacting with the mouse without significant lag is acceptable; and for computer games there is a strict budget of 30 ms or less that must accommodate all sorts of computation, *including* volumetric rendering.

We focus on real-time volume graphics used in sci-vis. With that we follow in the footsteps of the 2004 real-time volume graphics course by Engel *et al.* [Engel et al. 2004]. This focus has several implications:

- Performance requirements are often within tens of milliseconds or more. Applications are considered interactive if there is no significant lag when the user interacts with the mouse.
- Possibly, there is a demand for user interaction with special devices, such as head mounted displays—this can impact the aforementioned performance requirements.
- Volume rendering is not limited to using graphic APIs such as Vulkan, DirectX, or OpenGL, as opposed to games that usually implement their low-level graphics within one of those frameworks.
- Transfer functions are used for color and opacity mapping, which is in contrast to VFX rendering where volumes are considered (multi-spectral) densities. This has numerous implications for the algorithms that can be used and which optimizations are possible. This discussion of how transfer functions impact algorithmic decision is a common theme that we will come back to throughout this course.
- Depending on the application, rendering preview or convergence frames (possibly containing Monte Carlo noise) may be acceptable.

Regarding the final point, one could argue that our course does not focus on real-time but on *interactive* rendering, as we advocate taking a Monte Carlo approach to volume rendering in sci-vis in general, so there will always be some noise in the final output. In that sense we deviate from the premise of Engel’s course. We assume that the user will take measures such as using a denoising framework, such as the ones that come with OptiX and other ray tracing libraries. A discussion of those is beyond the scope of this course though.

2.2 Some theory

The central equation for direct volume rendering is the radiative transfer equation (RTE) for participating media, also referred to as volume scattering equation by the literature. The RTE has been derived and presented in literature

extensively; to avoid being repetitive, we refer the reader to the seminal work by Max [Max 1995] or the SIGGRAPH course by Novák *et al.* [Novák et al. 2018] for the details and derivation. However, remember that the model presented by those authors is based on light/particle interactions. This is a simplification of the general radiative transfer model in that light particles (photons) can *not* interact with each other but only with other particles forming solids, gaseous media, liquids, etc. When photons interact with particles, energy is either lost or gained, subject to the following phenomena. Loss of energy results from *absorption* (energy dissipates as heat, resulting in radiation in the non-visible spectrum) or *out-scattering* (photons change direction based on the *phase function*). Energy is gained by *emission* (radiation within the visible spectrum) or by means of *in-scattering* (photons scattered *out* from other locations intersect the path of our photon). Given these phenomena, the characteristic differential equation of that model becomes

$$\frac{dL(x, \omega)}{dz} = \underbrace{-\mu_a(x)L(x, \omega)}_{\text{absorption}} - \underbrace{\mu_s(x)L(x, \omega)}_{\text{out-scattering}} + \underbrace{\mu_a(x)L_e(x, \omega)}_{\text{emission}} + \underbrace{\mu_s(x)L_s(x, \omega)}_{\text{in-scattering}}. \quad (1)$$

Here the terms μ_a and μ_s are position-dependent absorption and scattering coefficients (sometimes combined to become the extinction coefficient). While the literature sometimes omits the positional dependence, this is crucial in sci-vis as this is the point where sampling the volume (and potentially the transfer function) happens.

For clarity we also omit wavelength dependence; the various coefficients and quantities, when computed with pure physics in mind, also depend on the wavelength λ and would be represented by spectral power distributions (SPD). This will later become important again when discussing RGBA color transfer functions; the VFX field often treats volumes as multi-spectral and then also (importance) samples the SPD dimension.

The central quantity $L(x, \omega)$ is called radiance and is calculated for a position in space x , from a differential cone of directions ω , and differential solid angle. Subscripts indicate if radiance is associated with light emission (L_e) or results from in-scattering (L_s). In its integral form, to obtain radiance arriving at x but computed over a finite segment along a ray traveling through space, this equation becomes the *volume rendering integral* (VRI):

$$L(x, \omega) = \int_0^z T(x, y) [\mu_a(y)L_e(y, \omega) + \mu_s(y)L_s(y, \omega)] dy, \quad (2)$$

where

$$T(x, y) = \exp \left(- \int_0^y \mu_t(s) ds \right) \quad (3)$$

is called *transmittance*.

Transmittance is a central term of the VRI and represents the average amount of particles that travel from y to x without energy loss. The exponential term in that equation is a consequence of solving the first order differential equation associated with absorption in Eq. (1) and is also known as Beer-Lambert’s law. Beer-Lambert’s law in other contexts is usually presented in simpler form though, namely $T = \exp(-\tau(t))$, i.e., the quantity inside the exponential (the optical thickness) does not vary in space and the medium is homogenous¹. In sci-vis we assume that volumes vary in space though. In fact, the volume density and its boundary are often quite involved and are represented by different topological elements ranging from voxels to finite elements, and non-trivial boundary surfaces.

In many cases, evaluating Eq. (4) with all the phenomena involved is not desirable in sci-vis. This has to do with faithfulness of the visualization. Excessive multi-scattering, e.g., can result in pastel colors in the final output. Since sci-vis volume data usually gets color-mapped, more involved lighting models increase the chance that colors end up in the final image that are not present in the color map, by that impacting the faithfulness of the visualization. At the

¹This formulation is, for example, relevant when evaluating simple bidirectional scattering distribution functions (BSDF) at surface interactions, e.g., when rays transition between vacuum and glass.

same time, depth cues such as shadows are important for exploration, demanding a trade-off between using simplified lighting models versus more advanced effects. Using simpler lighting models also has the side-effect that they are less compute intensive.

The in-scattering term $L_s(y, \omega)$ in particular gives rise to simplifications. In its unaltered form this term, when evaluating the equation numerically, requires an infinite recursion that stops when the photon either exits the medium by accident, or through Russian roulette based on the light intensity. By reformulating Eq. (4) so that the term L_s only contributes radiance by its emissive term L'_e but no additional, recursive, in-scattering L'_s we, e.g., obtain volume rendering with single scattering. As opposed to multi scattering, this lighting model supports volumetric shadows, but no global illumination.

Another common simplification to the model is to only account for absorption and emission by removing the whole scattering term. In this case the model simply becomes

$$L(x, \omega) = \int_0^z T(x, y) [\mu_a(y) L_e(y, \omega)], \quad (4)$$

which does not require recursion at all. By dividing the segment $[0, z]$ into n equally sized segments Δit (see Max [Max 1995] for a detailed derivation) one can compute the absorption and emission model using a Riemann sum, by n times stepping through the volume from back to front:

$$L(x, \omega) \approx I_0 \prod_{i=1}^n t_i + \sum_{i=1}^n g_i \prod_{j=i+1}^n t_j \quad (5)$$

with $t_i = \exp(\Delta it)$ and $g_i = \mu_a(y) L_e(y, \omega)$. This formulation adds an ambient light term I_0 , so that absorption has two effects: it attenuates the ambient (background) light, and emission from the medium itself. The integral equation can be evaluated using ray marching and front-to-back compositing² as presented by Levoy [Levoy 1990]. Ensuring that the steps of the nested sum and product visit the same positions (i.e., $x(i) = x(j)$), this is an $O(n)$ operation so that absorption affects both radiance from the ambient light source as well as the emissive radiance at positions $x(i)$. The two terms do not have to be evaluated separately, which is crucial to this model.

In this form, transmittance is computed as a side effect. We however note that ray marching can also be used to compute transmittance in isolation, which can come in handy when picking a light source to compute single-scattered volumetric shadows. Production and VFX rendering has come up with unbiased ways to compute transmittance that are generally superior to ray marching and are discussed in later chapters. Also note that the complexity of marching rays becomes $O(n^2)$ when sending a shadow ray from each position $x(i)$ towards a light source, which is unpractical for real-time rendering. Therefore, ray marching is mostly used in sci-vis to compute absorption and emission, but due to those practical as well as numerical limitations is usually not applied to lighting models using single or multi scattering. Absorption plus emission is still a very common lighting model in sci-vis though, and we will therefore revisit ray marching later in Section 2.4.

2.3 How real-time volume rendering was achieved twenty years ago

Volume graphics received a lot of attention when GPUs with 3D texture support became available. For techniques reaching even further back (register combiners, 2D texture-based volume graphics) we refer the reader to Engel *et al.*, but devote this section to a brief historical overview to explain where certain techniques and optimizations (that to some capacity still live on in software such as VTK or others) were coming from.

²Flipping the traversal direction is a matter of simple reformulation of Eq. (5). Both variants have their pros and cons.

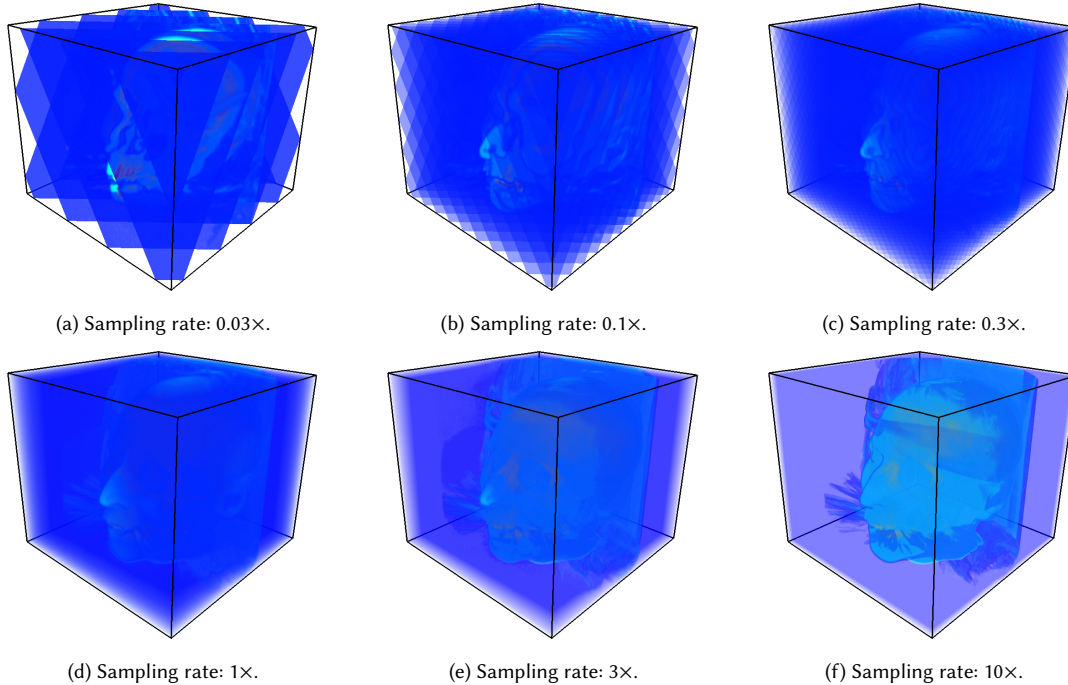


Fig. 3. 3D slicing rendering as done in the early volume graphics days. Two types of artifacts are predominant: visible polygons (top row), and artifacts from low compositing precision, (bottom row). The renderer uses an exponential opacity correction term; the blue color, e.g., should be nearly identical when rendered into a typical 24-bit color buffer. Differences in lightness come from limited precision during compositing; using a higher-precision render target fixes this, but incurs higher costs.

GPUs from the early 2000s were not programmable using general compute APIs (GPGPU). Instead, during the advent of GPU-based real-time volume graphics the common programming models used rasterization APIs with programmable vertex and fragment stages. In the years leading to the course material by Engel, unified shader models were on the horizon but not yet common-place (implying load imbalances between vertex and fragment stage). Not so long ago, geometry computations were conducted solely on the CPU.

Rasterization output (ROP) had (and still has) support for blending fragments, ensuring that eventually (but usually not yet at the fragment stage) would be retired in order. This implied that blending operations (such as the ones used to compute the absorption plus emission integral using Riemann sums, cf. Section 2.2) was possible using hardware operations only—if the hardware abstraction (ROP units) of the pipeline were used for that. This made it particularly appealing to perform the compositing operations using the framebuffer render target directly attached to and responsible for output to the display.

The typical formats for direct rendering targets were (and still are) **RGBA8** (i.e., 8-bit per color and alpha channel). This gave rise to an interesting trade-off: the two options were a) to implement ray marchers as the one by Levoy [Levoy 1990] in a fragment shader by rasterizing the front-side of the volume's bounding box and performing all operations (including compositing) in software (but with floating point precision), or b) to use a proxy geometry slicing through the volume, to implement only the sampling stage (texture lookup, post-classification using the transfer function) in the shader, and for compositing use the ROP units of the GPU. Renderers implemented that way, using, e.g., a view-aligned, planar proxy

geometry clipped against the volume bounding box, would generally be faster than their software counterparts, but have the obvious limitation of reduced precision for the compositing operations (the exact precision for the calculations would not be specified by the vendors, but using 8-bit render targets implied that temporary results, including alpha, during compositing would be converted to 8-bit). See Fig. 3 for the various artifacts that occur with this kind of renderer.

While using 8-bit compositing these days may seem ludicrous, it is important to note the restrictions of those times. GPUs from the early 2000s (e.g., NVIDIA GeForce 6800 Ultra, 256 MB) would come with a couple hundred MB of memory at most, where a good portion of that would however be devoted to render targets, geometry, or other content. So the sizes of typical volumes were heavily limited—but consequently so were the sampling rates required to reconstruct those volumes. This is one reason for that in those early days of real-time volume graphics, 3D slicing-based volume renderers were extremely popular, especially when targeting real-time applications like VR.

Within this framework, supporting more advanced lighting effects was quite challenging. To compute scattering, e.g., one would need to spawn extra samples in different directions from the current sample, but using a fixed proxy geometry would not easily allow for that; e.g., in order to emulate single scattering from a point light source, an extra stack of slicing polygons was necessary that was oriented towards the light (see Kniss *et al.* [Kniss et al. 2002]). This also sparks an interesting discussion, in how important advanced global illumination effects even are for sci-vis. For example, shadows or ambient occlusion can often provide important depth cues, but from a strict data analysis perspective and if the data is somewhat abstract, can also masquerade as features that are not in the data. The same is even more true for multi-scattering effects, which leads to colors mixing in a way that often do not represent the color mapping from the transfer function anymore. As a consequence, those simple shading models based on A+E were (and still are) very important to sci-vis, and arguably, frameworks that would optimize for speed rather than image quality were even more appealing at those times, given the overall hardware limitations.

In the last twenty years, the odds have shifted significantly though. GPGPU programming models have become omnipresent (and even interoperate easily with raster-based programming models). Compute power in general has increased, and so has GPU memory. The size of volumes that are produced these days (and their topology) has changed significantly. At the same time, individual GPU threads have become more powerful, allowing for thread-divergent operation to be much more common-place than it was twenty years ago. What is the status quo in real-time volume graphics has consequently changed as well—and so have the possibilities in terms of what *can* be done but is not yet common-place or the default used by industry standards such as VTK. We therefore in the following explain how the status quo from twenty years ago has embraced different programming models, just to (in later sections) present what the possibilities are for sci-vis style volume rendering that are currently only used by a very bespoke set of frameworks focused on high-quality ray tracing.

2.4 Status quo: ray marching in a GPGPU kernel

Today, GPUs have enough main memory to support much larger volumes. It has since become the de facto standard to compute absorption and emission for sci-vis volume rendering using ray marchers implemented in a compute kernel. While rays are effectively traced in parallel, the programming model allows us to describe the control flow and data path taken to render a single pixel in isolation, which gives access to a much more extensible framework by tracing rays into the volume. Extending this framework to support scattering, for example, is much more intuitive than with a framework based on view-aligned slices.

Listing Fig. 4 describes the most simple ray marching loop that will serve as a common baseline for all the extensions we describe later in this course. This procedure is a straightforward implementation of the Riemannian summation to

```
1 vec3f color = 0.f;
2 float alpha = 0.f;
3 float transmittance = 1.f;
4
5 for (float t=ray.tmin;t<ray.tmax&&alpha<0.99f;t+=dt) {
6     vec3f P = ray.org+ray.dir*t;
7
8     float value{0.f};
9     if (!sampleVolume(volume, P, value))
10         continue;
11
12     vec4f sample = postClassify(transfunc, value);
13
14     float stepTransmittance = powf(1.f - sample.w, dt / unitDistance);
15
16     vec3f shadedColor = vec3f(sample.x,sample.y,sample.z) * ambientColor * ambientRadiance;
17
18     color += transmittance * (1.f - stepTransmittance) * shadedColor;
19     alpha += transmittance * (1.f - stepTransmittance);
20     transmittance *= stepTransmittance;
21 }
```

Fig. 4. Ray marching baseline for absorption plus emission rendering. The corresponding code can be found in sample app 00 “Hello DVR course” of our sample code framework.

A+E from Section 2.2, evaluated from front to back in-between the ray’s $[t_{min} : t_{max}]$ range. Things worth noting are the early ray termination constant of 0.99 (in the loop’s header in line 5), and the weighting of the sample by means of Beer-Lambert’s law in line 14. This has been referred to by the literature as “opacity correction”, but effectively just ensures that the transmittance *between* samples is correctly weighted based on Beer-Lambert’s law, which is the model we use for transmittance along any distance t inside the volume. This causes the appearance to be similar in terms of brightness of colors, etc., regardless of the sampling rate used.

We also point the reader to the `sampleVolume()` function in line 9, which provides an abstraction for random access into the spatial field associated with the volume. Without loss of generality, we assume that this function, given any real-valued coordinate in 3D space, either returns **false** (if the volume density was missed), or **true** (if not), and in the latter case, the reconstructed density value at said position is returned in the variable `value`. Given an abstraction like this, it is easy to accommodate any spatial field type, under the condition that it is possible to sample the volume at random positions in space. The operation also does not imply any asymptotic runtime behavior—sampling a voxel grid may incur $O(1)$ asymptotic runtime for constant-time offset calculations to access memory, whereas more complex volume types may require recursion or other more expensive operations. It is also easy to conceive that for simpler volume types such as grids, this function may always return **true** if the sample positions are carefully chosen to fall within the volume’s bounds, whereas more complex volumes (e.g., those comprised of finite elements) may not find a valid sample everywhere inside their bounding box and then also return **false**.

Finally, we note that this formulation uses post-classification, so generally, the sample returned is a scalar value that is mapped to an RGBA color using a transfer function lookup (line 12). Classification happens post-interpolatively, i.e., after the spatial field was sampled. We also assume this to be an intrinsic function and that transfer functions are pretabulated; in this course, and without loss of generality we assume that their values are represented by small 1D arrays (e.g., 256–512 entries), so it is relatively inexpensive to even iterate over those values upon lookup, if necessary.

This will become relevant in later sections when looking at space skipping techniques, and is a precondition that is compatible with most sci-vis applications we are familiar with.

We consider this algorithm to be the baseline technique for rendering volumes of all kinds with absorption and emission on modern computers. The algorithm is easy to embed in a parallel program, either through thread parallelization on the CPU, or by using a GPGPU API such as CUDA or HIP. Extensions such as shading using central difference gradients, ambient occlusion in voxel space, etc., would be easy to add; at the same time, the algorithm in its simplicity makes it easy to reason about and compare with later extensions, so it will serve as a baseline not only quality-wise but also for the sake of the arguments we make in this course.

2.5 Introducing Voxels

So far we have not talked about the specifics of the function `sampleVolume()` from Fig. 4 yet, but only about its interface. We proactively made that as general as possible, since sampling volume densities at random positions in space is a core intrinsic of any volume renderer, regardless of the integration algorithm and volumetric field type. We note though that volume renderers back in the days were more restrictive as of which field types were compatible with what kind of renderer—the motivation once again having to do with performance considerations.

It is easy to conceive that random access-style sampling into the volume density is the primary bottleneck of this algorithm (assuming that a reasonable amount of samples are taken, as is usually the case in sci-vis); regardless of the target platform, today, memory accesses come at a significant cost compared to, e.g., arithmetic operations even in the simplest case of structured-regular volumetric field types. However, with structured-regular data, and in the presence of hardware-accelerated 3D texture access routines, sampling the volume with random access is still comparably cheap as the operation takes—at least theoretically—constant time. In comparison, sampling routines using tree traversal for example can be quite expensive if the tree traversal happens per sample taken.

Of course, all the considerations about memory access still hold today: random access sampling in a sci-vis volume renderer is still by far the most expensive operation. However, overall compute power has increased significantly due to hardware advancements. Yet another factor that made random access on more complex field types in a ray marcher become feasible was the ability to perform adaptive sampling. Now that ray marchers are common place, it is easy to adapt the sampling step size locally given some heuristic, as we will discuss in Section 4.2. For now, suffice it to say that advancements in both hardware *and* software led to random access point location becoming feasible even for relatively complex field types such as unstructured meshes and other hierarchical representations.

Before discussing such field types in Section 4.1 and presenting methods to sample and traverse them in an optimal way (Section 4.2), we suggest voxel grids as the first non-trivial volumetric representation. Traditional (and even modern) volume renderers usually rely on 3D textures; we suggest to use a more adaptive in-memory representation for voxels that also allows to express sparseness (if present in the data). A representation that fulfills these properties is OpenVDB [Museth 2013]. Recent work [Zellmann et al. 2025] has shown that even if the data is dense, the memory as well as sampling overhead are low compared to 3D textures, whereas the memory savings can be significant if the data itself (not just the transfer function) is sparse. The OpenVDB API comes with tools for importing dense representations. For rendering on the GPU we convert from OpenVDB to NanoVDB. An example implementation of `sampleVolume()` is shown in Fig. 5. We note that the function still always returns `true`, as the field is structured and when sampling within its bounds we will always retrieve a value. For simplicity’s sake our implementation is limited to floating point voxels. One word of advice is that if one wants to support other data types (e.g., byte-sized voxels) natively to not implement that as a `switch-case` construct in `sampleVolume()` directly, but to rather call different (template) instantiations of


```
1 inline __device__ bool sampleVolume(const Volume &vol, vec3f pos, float &value)
2 {
3     auto acc = vol.asNanoVDB->getAccessor();
4     nanovdb::math::Vec3<float> nvdbPos(pos.x, pos.y, pos.z);
5     if (!vol.filterLinear) {
6         auto smp = nanovdb::math::createSampler<0>(acc);
7         value = smp(vol.handle->worldToIndexF(nvdbPos));
8     } else {
9         auto smp = nanovdb::math::createSampler<1>(acc);
10        value = smp(vol.handle->worldToIndexF(nvdbPos));
11    }
12    return true;
13 }
```

Fig. 5. Point location sampling using NanoVDB. The corresponding implementation can be found in sample app 01 “let there be voxels” of our sample code framework.

the ray-marching device function itself, to avoid costly branching in the inner loop. NanoVDB is not only accessible through CUDA, but also in C++ host code or Vulkan and other miscellaneous shading languages.

3 Adopting Monte Carlo Rendering for Real-Time Volume Graphics in Sci-Vis

To understand why ray marching is so popular in scientific visualization, let us recapitulate the objectives in this field. One major objective is interactivity. Although the acceptable latency is somewhat higher than in computer games, interactive exploration is only possible if images are available within a fraction of a second. Another important objective is the faithfulness of the generated images. As we learned in the previous chapter, volumetric data can consist of billions of volume elements, and without knowledge of what is important and what is not, in an explorative setting, every volume element should be considered. Ray marchers are so popular because they allow to adapt the sampling rate along the viewing ray, either by setting up a constant marching step size, or even by adapting to local frequency as discussed in later chapters. By adapting the sampling rate we can make sure to eventually “catch” every volume element within the raster given by the sample spacing, and we are also provided with a trade-off between quality and rendering performance that we can use as a tunable parameter.

Using ray marching to compute absorption and emission comes with a couple of limitations, though. As we are computing not only transmittance, but a more complex integral, distances have no definitive meaning, and ray marchers usually do not output that. Meaningful distances along the ray may exist for edge cases (e.g., high gradients that correspond to surfaces) but not in general; whereas when only integrating transmittance, the *free-flight distance* where a stochastic event causes the photon to eventually get extinct does have meaning. As a consequence, interoperability with surfaces requires special handling when using ray marching. In the case of opaque surfaces it requires these to be rendered first, which also places a burden on the application’s side, which must provide facilities to traverse the render graph accordingly. In the case of transparent surfaces it becomes even more complex and requires using techniques such as depth peeling [Mammen 1989]; which, if we only had a distance for the volume contribution we just computed, could be avoided altogether by compositing all those entities by their depth.

Ray marching becomes even more limiting in the context of multiple overlapping volumes. A quite common modality in sci-vis are so called “multi-channel” volumes where a voxel has not just one but multiple values, possibly representing a vector or multiple scalars that are logically bound to the same position in space. Ray marching requires these to be evaluated exhaustively in a loop, as each sample position will only be visited once. If volumes overlap but have their

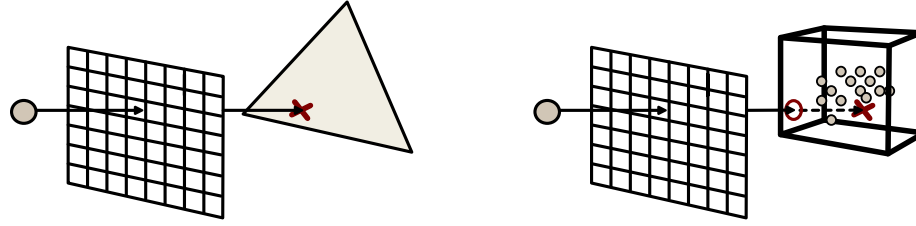


Fig. 6. Relationship between surface intersections and free-flight distances. While a surface intersection occurs at a definitive position in space, a collision event also has a definitive position, but the result is subject to variance that depends on the local density.

own model transform, possibly with rotation, etc., this becomes even more complicated. A possible implementation for volumes in a scene graph with non-static parent transforms would be computing ray segments with pointers to the volumes that overlap, and then integrate over those segments from front to back. Since this has to happen in device or shader code, the number of overlapping volumes would need to be capped so the segments fit into memory local to the shader cores.

Finally, how to compute scattering within the framework given by ray marchers is not as obvious. The naïve approach would be to cast and march along additional rays towards the light source(s) from any given sample position along the viewing ray; that would, however, result in quadratic complexity in the simple case of single scattering, and lead to combinatorial explosion when taking more than one scattering event per sample into account. We therefore find it worthwhile to take a look at techniques from VFX and production rendering, which solve these issues using methods that favor random walks, implying that depth-first traversal of ray trees instead of breadth-first traversal as in ray marching is used.

The VFX industry has since adopted volumetric path tracing for offline rendering of volumetric effects. Path tracing in general concentrates on computing paths connecting the camera and light sources; this can also be understood as following a single branch down to the leaf, of the widely branching tree given by a (hypothetical) multi scattering implementation based on a ray marcher (depth-first vs. breadth-first). In contrast to ray marching, *volumetric* path tracing uses free-flight distance tracking for the density estimate. In this section, we make an argument that tracking is also useful, if not in many cases superior to, ray marching for interactive volume rendering in sci-vis. Due to the specific nature of how volumes are rendered in sci-vis—especially the presence of RGBA transfer functions—the concepts from volume path tracing need to be adapted though. It is the goal of this chapter to derive a Monte Carlo framework for interactive volume rendering that meets the needs of scientific visualization.

3.1 Free-flight distance tracking

Volumetric path tracing computes light paths through the density connecting samples on the lens of the camera to a light source. We refer the reader to the standard literature for path tracing in general [Pharr et al. 2023]. There are numerous advantages to using this technique instead of ray marching, even if only computing absorption and emission. Indeed, elements of path tracing can also be used (and turn out to be quite useful) even if not the full light path with all its scattering events is computed, but only absorption and emission.

For that, let us recapitulate how a path tracer would proceed to compute the particle interactions of a photon along a straight path through the volume density. With surfaces only, one would track this photon from the camera into the scene along a *viewing ray*, and there would either be a definitive ray/surface intersection, or the ray misses the scene.

With volumetric path tracing, there are no surfaces, and we instead have to compute the *free-flight distance*, which is the stochastic distance a photon can travel without all its radiant energy being absorbed (Fig. 6).

To compute the stochastic distance a photon travels without colliding with a particle from the medium we use a Monte Carlo method, sampling that distance using random events. If the random event tells us the photon hit a particle, we denote the distance the photon traveled thus far the free-flight distance, otherwise the free-flight distance is just the distance traveled through the medium from beginning to end. Given that distance (and assuming we also have a way to obtain an estimate for albedo/color for shading operations at that position), variance is something we can handle, by sampling and averaging an appropriate number of times, e.g., using an accumulation buffer—so the question is: how exactly do we compute the free-flight distance.

3.1.1 Heterogeneous media and null collisions. The representative quantity in volumetric path tracing is *transmittance* (see Eq. (3)), and is related to the free-flight distance. If the medium were homogeneous (and the formula for transmittance simplifies to $\exp(-\tau(t))$), we could just analytically compute transmittance; yet if the volume density varies depending on the position in space a numerical solution is required. First, let us establish the correspondence between transmittance and distance: transmittance is the amount of light that passes through the medium along a certain distance, so transmittance and (that particular) distance are inverse functions of each other. Transmittance decreases monotonically: at distance 0, 100% of the light is transmitted, and while the transmitted light energy decreases with distance it can never become more when only considering absorption.

A (viable) numerical solution to compute the distance where a particle is fully absorbed would be to ray march and by that compute transmittance by integrating all the $\mu_t(x)$ we encounter at each position x (again refer to Eq. (3) here) until that integral becomes 1. In contrast to the ray marching algorithm from Section 2.4 we integrate over a different quantity, but the procedure in general is subject to the same limitations. One problem is to find an initial marching step size; whatever step size we choose, we will either overestimate, resulting in undersampling, or underestimate, resulting in poor performance. Another issue is systematic bias, as no matter the chosen step size, by quantizing the integral that way, there is a remainder that is unaccounted for (or accounted for, but overestimated). This is generally more pronounced when computing full light transport, including multi-scattering and emissive sources, where biases equate to loss or gain of energy. Among others, this results in image darkening or brightening, which is problematic for artistic control in VFX. Systematic biases can also affect sci-vis visualizations though, only that here it has more of an effect on faithfulness than on artistic control. Other estimators for heterogeneous media exist, such as regular tracking, but are applicable only in edge cases where it is possible to decompose the volume into homogeneous subregions.

The solution that the VFX community came up with is to use importance sampling. For that we first note that if the distance estimate is unbiased, a *stochastic* estimate of where the light gets fully absorbed is good enough³, if we repeat and average the result often enough. We call that stochastic estimate the *free-flight distance* or also the *collision-free distance*. In that framework, we assume that photons, once they collide with a particle, are fully absorbed, and the probability of absorption depends on the local density. In this way, distance sampling becomes a random event.

We then recall the monotony of the transmittance function, and recall from the Monte Carlo literature a technique called *inversion sampling*. We are not going to derive this technique here, but refer the reader to the literature [Morrical and Zellmann 2021]. Remember that if we want to sample a probability density of a function that is numerically hard to integrate, we can also inversely sample its *cumulative density function* (CDF), and the projection is more likely to

³One implication of this is that we do not have a notion of a medium being “fully opaque” anymore, which in particle physics is not well defined anyway, but can be of importance to certain simplifying optimizations often used in computer graphics.

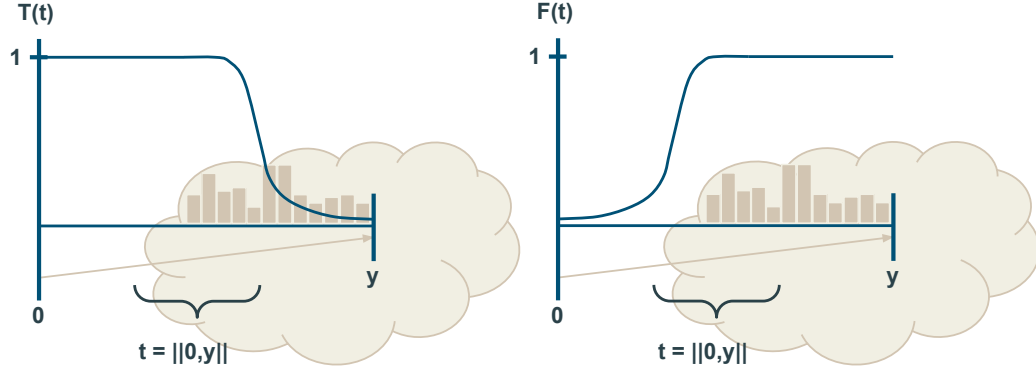


Fig. 7. Left: transmittance T . Right: absorbance—defined as $F = 1 - T$ —has the shape of a cumulative probability density function that we can sample into using inversion sampling.

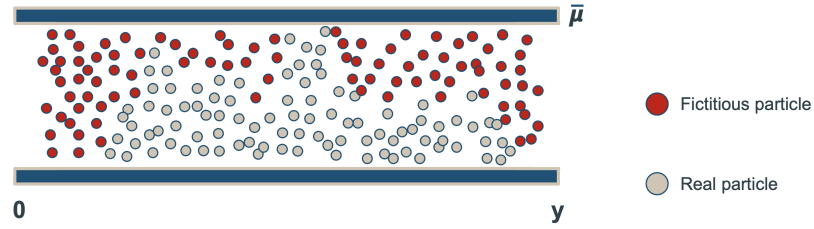


Fig. 8. Virtually homogenizing the volume with fictitious particles. Photons interacting with a fictitious particle will continue integration without changing direction or losing or gaining energy.

fall into a range where the PDF is neither 0 nor 1. Note that the function $1 - T(t)$ (for a given distance t) has all the properties that a (continuous) CDF fulfills, namely that it increases monotonically, starting from 0 to 1. We also note that this function is the absorbance, and can be interpreted as the probability that a photon does *not* travel beyond t (Fig. 7). Given this importance sampling technique, we could obtain distance estimates that fall within a range that is meaningful for computing transmittance. However, the issue with this approach is that transmittance for heterogeneous media (i.e., Eq. (3)) is not invertible.

There is a trick, though, to artificially *homogenize* the medium [Novák et al. 2014]. By virtually filling up the medium with *fictitious particles*, the density becomes homogeneous. Fig. 8 We have two types of particles now: one that, upon collision, causes the photon's energy to be absorbed, and another one that does not (cf. Fig. 8). If we interact with a fictitious particle, we just assume that no collision occurred and the photon just continues along its original path without losing (or gaining) energy. This gives rise to a simple algorithm. We generate a random event to probe the particle density. Upon reaching a point in space, we look up the real density, and compute the difference of that and the maximum density of the whole volume. This gives us a percentage of how dense the region is that we are in, and we can generate another random event that we weight by that percentage. If the region is very dense, we interpret that as being more likely to have collided with a real particle (and stop tracking), while if the density is low compared to the maximum we are more likely to have encountered a fictitious particle, resulting in what we call a *null collision*. Then we continue tracking the distance. This gives rise to the delta / Woodcock tracking algorithm for light transport

Algorithm 1 Woodcock tracking. We modify the algorithm to compute albedo $\rho_{LUT}(t)$ and extinction coefficients μ_{LUT} that are obtained as lookups into an RGBA transfer function (RGB color becomes albedo, α is interpreted as extinction).

```

1: function DELTATRACKING( $o, \omega, t_{min}, t_{max}$ )
2:    $t = t_{min}$ 
3:    $Tr = 1$ 
4:   while true do
5:      $\zeta = \text{RAND}()$ 
6:      $t = t - \frac{\log(1-\zeta)}{\mu_i}$ 
7:     if  $t > t_{max}$  then
8:       break
9:     end if
10:     $\xi = \text{RAND}()$ 
11:    if  $\xi \geq \frac{\mu_{LUT}(o+t*\omega)}{\mu_i}$  then
12:       $Tr = 0$ 
13:       $\rho = \rho_{LUT}(o + t * \omega)$ 
14:    end if
15:  end while
16:  return ( $t, Tr, \rho$ )
17: end function

```

in heterogeneous media shown in Algorithm 1. The implementation details of this algorithm are the subject of the following section.

Once a sample is calculated, and light transport properly integrated using the desired scattering model, it is converted to RGBA and then averaged using an accumulation buffer. That way we decoupled the tightly coupled ray marching loop from Section 2.4 into a number of independent samples that on average converge to an unbiased result with low variance. Placing those samples in a clever way, by using acceleration structures as described in Section 4.2, we can overcome all the disadvantages of the ray marching framework. The price that we pay for this is noise, as the individual RGBA pixel sample can come with high variance.

3.2 Woodcock tracking with RGBA transfer functions

Woodcock tracking (Algorithm 1) is a popular method for computing free-flight distances in heterogeneous participating media. The algorithm—as well as the other tracking algorithms—assumes that extinction coefficients are wavelength-dependent and consequently whenever we track a distance we do that for one wavelength in particular. In sci-vis, RGBA transfer functions tabulate color and opacity, but tracking algorithms are incompatible with that concept and require a real extinction coefficient to be present. The first assumption we make is that the extinction coefficient is proportional to α from the RGBA map, i.e., instead of the volume tabulating the extinction directly, extinction is obtained through an indirection, first sampling the volume and then accessing the transfer function (the term μ_{LUT} in Algorithm 1).

A crucial factor to this tracking method is that an upper-bound extinction is known (or can be estimated) that is called the *majorant extinction* $\bar{\mu}$. The majorant corresponds to the upper horizontal bar in Fig. 8. Geometrically and from Algorithm 1, it is obvious why a non-conservative estimate of the majorant is critical for performance. The higher the majorant, the more likely it is for the tracked photon to encounter a fictitious particle. While not affecting correctness—the photon will just continue traveling along the path undisturbed—this incurs unnecessary sampling operations and memory accesses. Gladly, interpreting α as extinction, we already have an upper bound to the otherwise

```

1 float transmittance = 1.f;
2 vec3f albedo;
3 float t=ray.tmin;
4
5 while (1) {
6     if (majorant <= 0.f)
7         break;
8
9     t -= (logf(1.f - rnd()) / (majorant / unitDistance));
10
11     if (t > ray.tmax)
12         break;
13
14     vec3f P = ray.eval(t);
15
16     float value{0.f};
17     if (!sampleVolume(volume, P, value))
18         continue;
19
20     vec4f sample = postClassify(transfunc, value);
21     float u = rnd();
22     if (sample.w >= u * majorant) {
23         albedo = vec3f(sample.x, sample.y, sample.z);
24         transmittance = 0.f;
25         return {t, transmittance, albedo};
26     }
27 }
28
29 return {INFINITY, transmittance, albedo};

```

Fig. 9. Woodcock tracking in C/C++ pseudocode. The corresponding implementation can also be found in our sample app 02 “woodcock” from the sample code framework.

unbounded quantity, namely 1.0. We note that $\alpha = 1.0$ by no means implies no particle can ever pass through that density, only that the probability is very low; as even a very dense medium such as metal has a non-zero probability of letting any particles through, even if this is a very rare event. Extinction and opacity are different concepts. This also manifests itself in Algorithm 1, in the comparison in line 11. The implication is that if majorant and actual extinction differ significantly (e.g., through a sudden peak due to a surface-like structure), sampling can become extremely inefficient. In Section 4.2 we discuss how to build accelerators with locally varying majorant densities to address this.

Finally, we have to deal with the color contribution from the RGBA transfer function. The VFX way of doing so would be to interpret the volume as multi-spectral and convert the color component to a spectral power distribution (SPD). However, that is not compatible with post-classification, which is required so the transfer function can be changed interactively. With post-classification it is not known in advance which wavelength we are sampling, as the lookup happens during distance tracking. We could also use pre-classification, but that would require us to apply the transfer function to the voxel, vertex, or cell values in advance, and by that replicating the volume per each RGB component. While this is obviously not feasible given the volume sizes in sci-vis, pre-classification also has other disadvantages, once again including transfer function updates not being interactive.

We instead choose to interpret RGB from the transfer function as *albedo* ρ_{LUT} that is also applied post-interpolatively, when evaluating local lighting models. This is a practical and non-physically inspired choice. We could also have interpreted the color contribution as emissive (L_e in Eq. (4)), or require that there be separate transfer functions for emissive and reflective components. We decided to stick with the albedo interpretation for several reasons. User

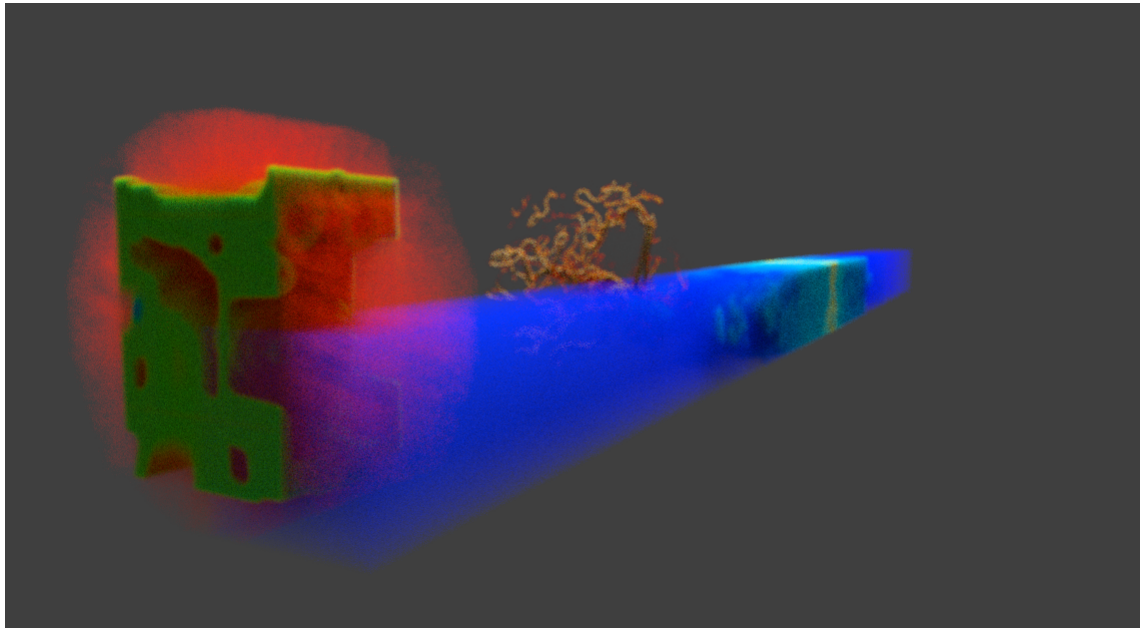


Fig. 10. Example of multiple volumes with different field types overlapping. This is elegantly realized using Monte Carlo free-flight distance tracking.

interfaces in sci-vis usually have one transfer function, and we find it advantageous that our framework is plug-and-play compatible with applications like, e.g., ParaView. The color components are usually bounded by the interval $[0, 1]$ while a physically based emission term would not have an upper bound, whereas albedo is defined in $[0, 1]$. We also observe that from a purely aesthetic perspective, using RGB as albedo gets us quite close to the appearance of traditional renderers; the interpretation is that there is an ambient term that factors in albedo and ambient light; and when using gradient-based shading, interpreting the local gradient as a normal vector to evaluate a local shading model, using RGB as albedo for that model works quite well and also results in similar appearance as the status quo in other applications.

This approximation of interpreting RGB as albedo, though practical, has several implications for us. One implication is that the volume now technically becomes inhomogeneous not just in density but also in reflectivity / the medium's tendency to scatter. Fictitious particles and null collisions still work, but residual methods, as in [Novák et al. 2014] that decompose the volume into control and residual extinction, do not. The sampling method would base this decision on μ only, but the question of what is the homogeneous residual (the minorant) now also depends on ρ . We must also obtain a definitive collision position from our tracking algorithm, and not just transmittance, as we need to evaluate the transfer function at that exact point; i.e., if we are interested in the RGB component of the transfer function, we must resort to tracking methods that also return the actual distance. Woodcock tracking meets this requirement, but since the transmission estimate is either 0 or 1, i.e., it comes with high variance. We discuss transmittance and free-flight distance estimators in more detail in Section 3.4. A C++ implementation of the Woodcock tracking algorithm can be found in Fig. 9.

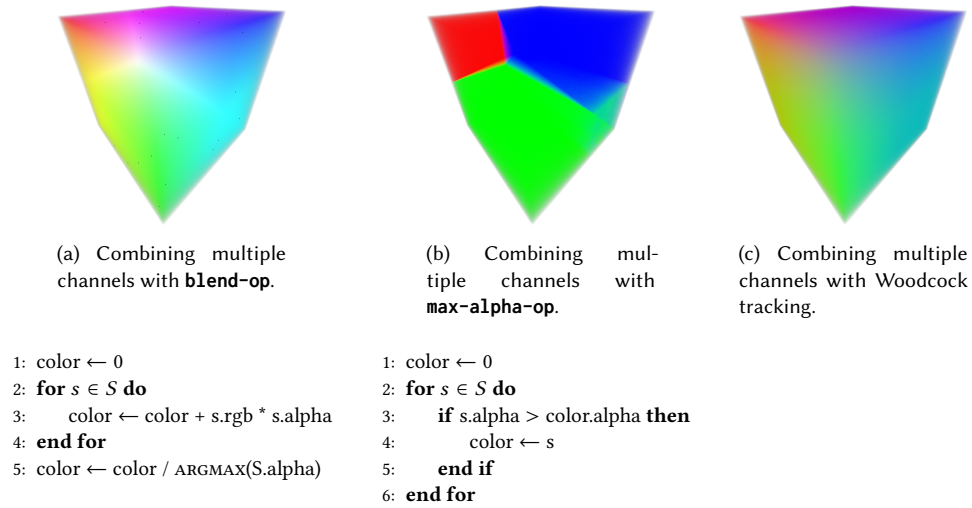


Fig. 11. Different strategies to combine samples from multiple volume channels with separate RGBA transfer functions. **blend-op** and **max-alpha-op** are commonly used in sci-vis and combine the contribution of the individual samples $s \in S$ into a single color sample. Woodcock decouples this and combines the samples in the accumulation buffer. The appearance is similar and the technique much simpler. The downside is variance manifesting itself as image noise.

3.3 Multi-field and multi-channel volumes

Rendering multiple, even overlapping volumes immediately becomes much simpler with Woodcock tracking, as we now have a visibility estimate with variance that we can compare, e.g., in a **closestHit()** query. An example is shown in Fig. 10. Arguably, what is more common in sci-vis than multiple volumes overlapping due to general transforms are volumes with multiple channels. The term channel here reminisces of color channels in images and texture interpolation units on GPUs and indicates the number of such channels is rather limited. In sci-vis one would often cram up to four channels into one RGBA texel and perform that lookup at once. Adopting a free-flight distance based framework we now have the freedom to implement this operation in different ways. One is to sample the volume and apply an operator that combines the n channels into one RGBA color using n transfer functions. Operators that are common in sci-vis are **blend-op** or **max-alpha-op**. We now have another way of combining samples at our avail, by simply computing separate Woodcock estimates into each density and compositing the resulting samples using the accumulation buffer. We show examples of these operators and image comparisons in Fig. 11. We observe that their appearance is quite similar and that with large volumes differences are barely noticeable (the example shows compositing one single voxel). Woodcock allows us to draw samples independently, which has several advantages and provides us with a simpler method, yet again, at the cost of variance that results in noise in the final image. For further reading on multi-channel volume visualization, we refer the reader to the paper by Sahistan *et al.* [Sahistan et al. 2026].

3.4 Advanced lighting

While in VFX volumetric pathtracing typically implies using multi-scattering to evaluate the full radiative transfer equation (Eq. (1)), in sci-vis this decision is more nuanced. We show several examples of volumetric rendering of a fluid dynamics data set using different scattering models in Fig. 12. We observe that more realistic scattering models can result in colors not even present in the RGBA transfer function. This may not be an issue per se—the color black, e.g., is

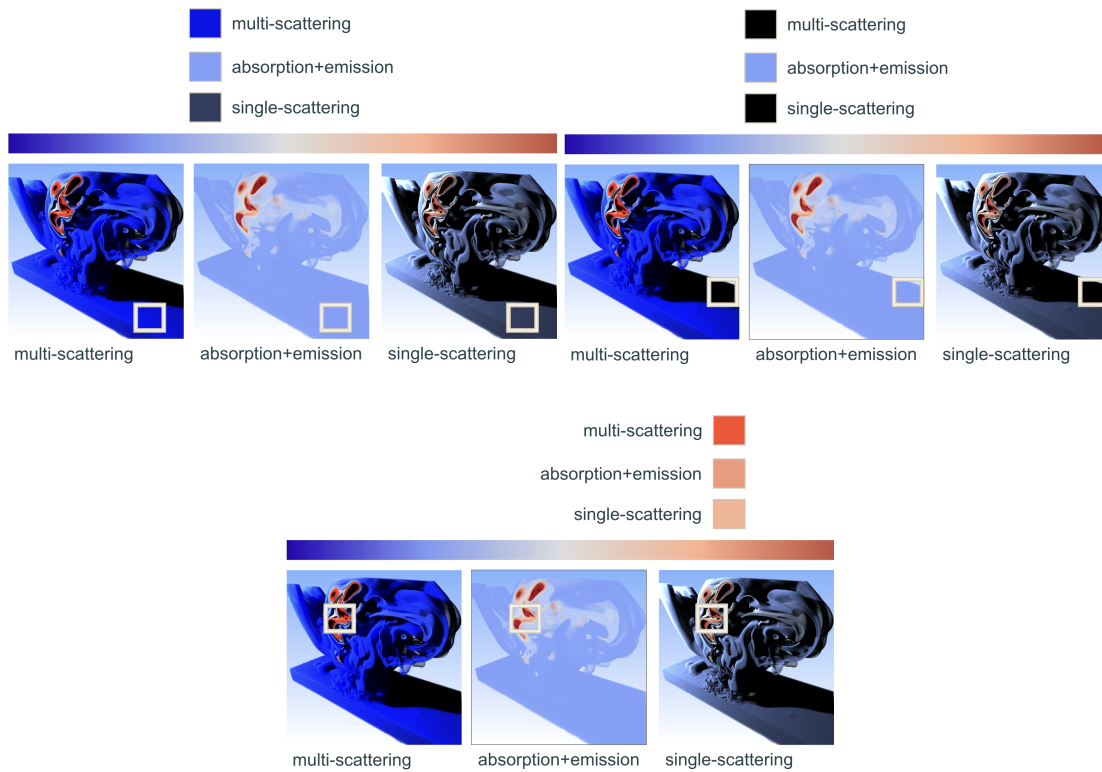


Fig. 12. CFD data set rendered with cool-to-warm color map, with multi-scattering, absorption plus emission, and single-scattering. The cutouts show that the scattering-based models are prone to exhibiting colors not present in the color map.

easy to associate with shadows that give additional depth cues and help with exploration. The more realistic the light transport calculation, the more pronounced the issue becomes, though. There is an obvious trade-off between realism that facilitates interaction, and faithfulness to the color map, potentially resulting in misinterpreting the data. We do not intend to give any additional advice beyond that, because the effect depends on different factors, including the color map and light scene, the data in general, but also the message the scientist wants to convey with the visualization. We observe though that simpler scattering models are often preferred in sci-vis, and that one reason may be that colors should correctly represent the data.

In VFX, much more advanced lighting effects are used and more common. This also includes emissive voxels or light sources that are placed inside the volume density. Therefore, importance sampling transmittance plays a major role in this field. One way to achieve this is using a transmittance estimator that is not based on null collisions. Null collision estimators either return 0 if a real collision occurred, or 1 if there was no collision. To get a better transmittance estimate, *ratio tracking* [Novák et al. 2014] can be used. We show pseudocode for this estimator in Algorithm 2. Ratio tracking gives us transmittance in $[0, 1]$, which results in the accumulated frames converging faster than averaging binary transmittance estimates. For us, this estimator is only useful for shadow rays (direct light source visibility, ambient occlusion, etc.) as at the sample position we evaluate a local shading model using RGB from the transfer function as albedo. The trade-off when using this technique instead of Woodcock tracking is variance vs. sampling performance. In Fig. 13 we provide the results of some measurements we took with our sample framework (sample app 04: “lights

Algorithm 2 Importance sampling transmittance with ratio tracking. This estimator does not provide distances.

```

1: function RATIOTRACKING( $o, \omega, t_{min}, t_{max}$ )
2:    $t = t_{min}$ 
3:    $Tr = 1$ 
4:   while true do
5:      $\zeta = \text{RAND}()$ 
6:      $t = t - \frac{\log(1-\zeta)}{\bar{\mu}_i}$ 
7:     if  $t > t_{max}$  then
8:       break
9:     end if
10:     $Tr = Tr * (1 - \frac{\mu_{\text{LUT}}(o+t*\omega)}{\bar{\mu}_i})$ 
11:    if  $Tr \leq 0$  then
12:      break
13:    end if
14:  end while
15:  return  $Tr$ 
16: end function

```

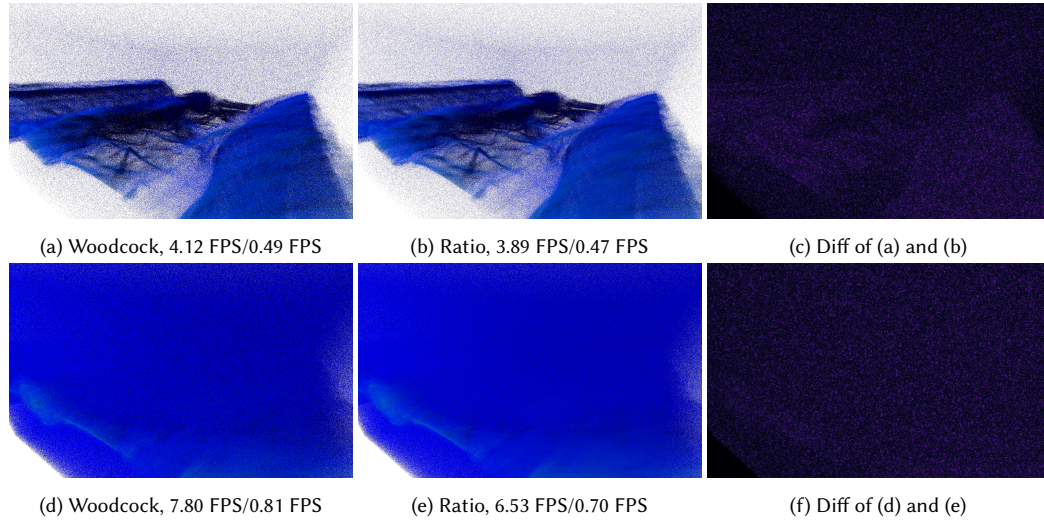


Fig. 13. Comparison Woodcock vs. ratio tracking for a typical sci-vis setup. Low overall framerates are due to oversampling to match the transfer function density. We use single scattering with a point light source and ambient occlusion with one hemisphere sample, accumulated over four frames (4 SPP). Figures (c) and (f) show difference images generated with FLIP.

on”) for setups and lighting scenarios we believe are representative of typical sci-vis applications. The evaluation is not exhaustive, but the results suggest that the differences in variance, when using simple scattering models and light sources located outside of the scene are subtle. This is not a general statement though; it is easy to conceive of scenarios in sci-vis where more advanced lighting, or scattering with more sophisticated phase functions come to use. Using this lighting model, however, is already a big step forward from the lighting model used by apps like ParaView.

4 Ray tracing and performance

In recent years, GPU vendors have started shipping graphics cards equipped with dedicated ray tracing chips. To keep things simple, in this course we will concentrate on the pioneering ray tracing implementation by NVIDIA called RTX. What all implementations have in common is that they support ray vs. object intersections accelerated by bounding volume hierarchy (BVH) traversal. We again refer the reader to the standard literature for an introduction to ray tracing with BVHs [Pharr et al. 2023].

RTX as a technology can be used via different APIs, including DirectX and Vulkan, and from C++ via NVIDIA's OptiX library [Parker et al. 2010]. Our sample code framework is based on OptiX and the OptiX wrapper library OWL [NVIDIA Corporation 2026]. We use ray tracing to implement random access into fields with unstructured and non-regular mesh topology, and to traverse majorant acceleration structures. This gives rise to a general traversal and sampling framework that looks as follows:

```
1 void traverse() {  
2     for (auto l : leaves) {  
3         majorant, t0, t1 = l.getSegment()  
4         t = woodcock(majorant, t0, t1)  
5         updateClosest(t)  
6     }  
7 }
```

The majorant acceleration structure is traversed by iterating over its leaf nodes (which can be the leaf nodes of a BVH, macrocells of a uniform grid, kd-tree leaves, etc.), giving us segments within which we perform Woodcock tracking with sampling. The details related to traversal will be discussed in Section 4.2. Note, however, that both traversal as well as sampling can potentially use ray tracing. How this is used to accelerate sampling of unstructured elements is the topic of the next section. A caveat with RTX is, however, that rays cannot be traversed recursively, i.e., once inside a BVH traversal, no additional rays can be traced against this or another BVH from within an intersection program that is executed to perform user calculations and return the closest intersection with a user-defined primitive. We therefore have to make a decision which part of the pipeline we want to implement with ray tracing. In some cases this is easy, e.g., when using voxel grids that do not require sophisticated sampling with hardware ray tracing. We discuss the pros and cons of different sampling and traversal structures and their combinations in the remainder of this section.

Another layer that potentially uses ray tracing sits on top of the volume integration mechanism and is relevant when integrating with a wider framework with surface geometry and other scene objects. We discuss this in Section 4.3.

4.1 Point-in-cell queries using BVHs

The algorithms we discussed so far deliberately used an abstract interface for volumetric field sampling. We implemented this interface for voxels using NanoVDB, but noted that replacing the volume element type is a matter of reimplementing that interface with custom functionality. In this section, we discuss how to implement the volume sampling interface for non-standard volumetric element types.

At this point we may ask what is even meant by *non-standard* element types. Historically, in sci-vis, ray marchers—and thus sampling-based architectures—used voxels, implying an $O(1)$ runtime cost for data access. We may note that NanoVDB even uses hierarchical storage for its voxel data base (we refer the reader to the course materials by Museth *et al.* for the details [Museth et al. 2025]), and though still $O(1)$ due to fixed hierarchy depth there is a potential cost overhead compared to simple table lookups, trading memory for runtime performance. The bespoke renderers from the

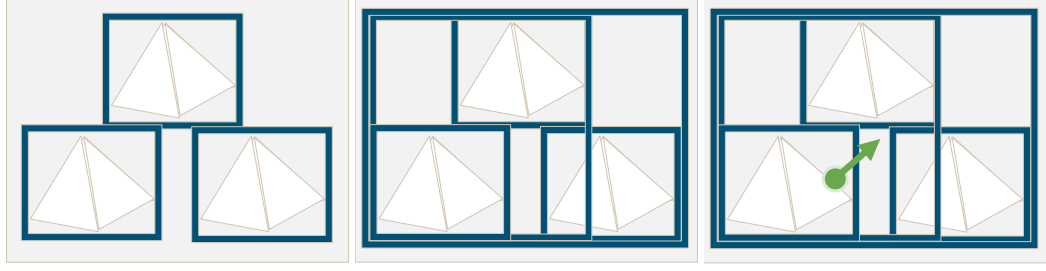


Fig. 14. Using BVHs for point-in-cell tests using OWL/OptiX user geometry. Left: the user supplies a custom bounds program so OWL can pass axis-aligned bounding boxes to OptiX that represent leaf nodes. Middle: a BVH built over user primitives. Right: we trace zero-length rays ($\mathbf{tmin}=\mathbf{tmax}=0$, $\mathbf{ray.direction} = (1,1,1)$) into the BVH and set the ray origin to the sampling position. Upon hitting a leaf we perform a custom point-in-cell test on the shader core. User geometry tests incur costly back-and-forth between shader and ray tracing cores. This test can also be implemented with triangle geometry, which on RTX GPUs uses hardware only.

2000s discussed in Section 2.3 virtually always used custom algorithms and traversal logic if volumetric elements had more complex shape and connectivity than voxels. Examples are cell projection for unstructured elements [Shirley and Tuchman 1990] or CPU-resident kd-trees for voxel hierarchies [Kähler et al. 2006].

GPUs these days have much better support for thread divergence, and another crucial addition was hardware ray tracing. Ray tracing with bounding volume hierarchies (BVH) literally means that hierarchies are traversed on the GPU by diverging threads. As a result, on modern architectures it is viable to perform traversal operations on the GPU, instead of sorting leaf nodes on the CPU and traversing them in order using full wavefronts. We now discuss how to make use of this technology to point sample into tetrahedral meshes to implement our `sampleVolume()` device interface function. Tetrahedral meshes frequently occur in CFD simulations. The implementation is easy to extend to other volume element types such as pyramids, wedges, or hexahedra.

There are two ways to implement point sampling on RTX that were originally published by Wald *et al.* [Wald et al. 2022]. The first version of the point-in-cell test uses OptiX’s axis-aligned bounding box (AABB) geometry, which performs BVH traversal on the hardware but calls a user-provided intersection program when encountering leaves. This is illustrated in Fig. 14. The user supplies AABBs for each tet, uses OptiX and OWL to build a BVH, and then traces zero-length rays into the BVH. Those rays have their origin at the sampling position, and an arbitrary, yet non-zero, direction vector. Zero length is achieved by setting both of the ray’s `tmin` and `tmax` parameters to 0. RTX calls a user-provided intersection program. This test is similar to the one used by rasterization hardware, but in 3D. If the ray origin is on the right-hand side of each plane the tet’s faces are embedded in, the point is also inside the cell, and otherwise it is not.

This test, however, is quite costly as it forces the hardware to perform a context switch whenever the user-provided intersection program is called. To avoid this costly back-and-forth between ray tracing and shader cores, we can also implement the same test using triangle geometry, then making full use of the hardware, since triangle intersections are executed on the RTX cores. Instead of passing bounding boxes, we instead pass the faces of the tets to OptiX/OWL as an indexed mesh and build a triangle geometry BVH. Zero-length rays would miss those faces, so we instead make sure to initialize the rays so they are guaranteed to hit one of the faces. If we hit a backface, we know that we are inside the tet, so we enable frontface culling and by that obtain a useful result. With that we have ported the test to make full use of our RTX hardware.

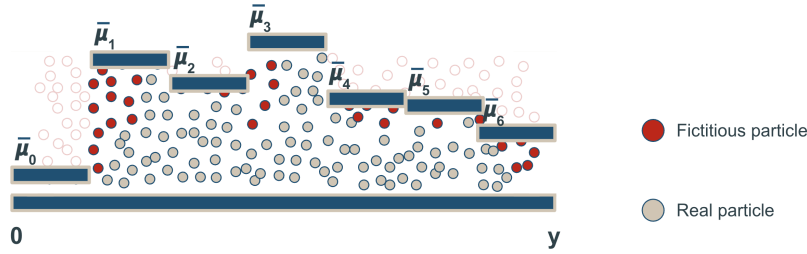


Fig. 15. Given a density with real and fictitious particles as in Fig. 8 one can build acceleration structures. The tighter the local majorants $\bar{\mu}_i$, the less samples are taken from each partition.

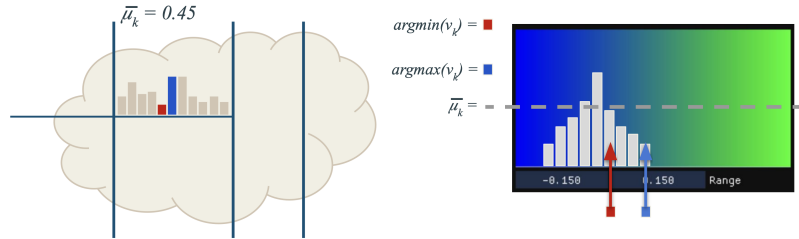


Fig. 16. To compute local majorants, we rely on a common technique to compute the maximum α from the transfer function for a given partition of the data. We determine both the minimum (red) and maximum (blue) scalar values in the partition given by v_k . We then map those to the transfer function's array index range, and iterate over those values in the transfer function that can possibly occur within the partition. The maximum α value becomes our majorant.

This procedure is illustrative of how one would implement random access into unstructured, semi-structured, or scattered field types with RTX. Many optimizations are conceivable, such as grouping tets into pairs to reduce memory overhead. We leave these optimizations for the user to contemplate and with our framework just lay the groundwork for that. Our sample code framework implements the first test based on user geometry in sample app 05 “tets n’ friends”. We also note that not all NVIDIA GPUs support hardware ray tracing—especially the ones used in data centers or high performance computers. OptiX is still available, but performance may be affected differently. The main benefit of using RTX lies in performing the point-in-cell test in hardware, without RT/shader core back-and-forth. Benchmarking on the individual platform is in order to obtain optimal results.

4.2 Traversal and acceleration structures

With Woodcock tracking there is a natural way to build acceleration structures (see Fig. 15). With a fixed step size as used by ray marchers, finding a good adaptive sampling rate is not obvious and heuristics come to use, e.g., based on the local frequency of the data subject to the transfer function. With Woodcock tracking we can, however, build acceleration structures by defining local partitions that have their own, tighter, majorants. Woodcock does not limit us to a specific acceleration structure; spatial partitioning (grids, kd-trees) is possible, as is object partitioning (BVHs). The local majorants depend on α from the RGBA transfer function. To compute majorants, we hence need to find the voxel or cell whose value is highest when the transfer function was applied. We achieve this by a simple trick well-known in sci-vis volume rendering. We initially compute for each partition the minimum and maximum *scalar* value from the field and save it for later use. When the transfer function changes, we iterate over the transfer function array, from left

to right over just those entries that map to the minimum and maximum scalar value, by that finding the maximum value for α . These values become our majorants. This procedure is illustrated in Fig. 16.

There are different dimensions we can possibly optimize for: transfer function updates should stay interactive; our technique of iterating over leaf nodes or macrocells to compute majorants from min/max values meets this requirement. The more complicated the acceleration structure though, the harder it might get to iterate over the leaf nodes. Simple grids work quite well in this regard.

Traversal performance is another dimension to optimize for, potentially resulting in the other dimensions becoming less efficient when tweaking the acceleration structure accordingly. An interesting observation here is that hierarchical acceleration structures that adapt to fine detail depend not only on the scalar data but also on the transfer function. A highly detailed BVH may, e.g., be completely ineffective if the scalar field is not visible where most of the small leaf nodes were placed. Zellmann *et al.* [Zellmann et al. 2024] have conducted an evaluation of different sampling and traversal structures for adaptive mesh refinement (AMR) data (semi-structured, hierarchical grids with adaptive voxel sizes). Their conclusion was that simple grid structures and digital differential analyzer (DDA) traversal are the most versatile combination for most scenarios. Again avoiding to give definitive suggestions and asking the reader to benchmark based on their own application and data, we also note that this combination allows us to dedicate RTX traversal to sampling—remember that recursive calls to **optixTrace** are not allowed, so we can either use RTX for traversal, or for sampling, but not for both.

4.3 Combined object models and render graphs for surfaces and volumes

Finally, traversal can also happen on the scene or render graph level. Libraries and APIs such as OSPRay [Wald et al. 2017] or ANARI [Stone et al. 2022] use render graphs to ray trace surfaces and volumes from within the same context. In OptiX, it would be natural to put all surface geometry and volumes underneath a common top-level acceleration structure (TLAS) and assign each volume its own bottom-level acceleration structure (BLAS). BLASs can have instance transforms assigned to them, which enables simple scene graph operations like rotating or moving the volume(s) in space. Other ways than using graphs to organize the scene are conceivable, yet all would potentially add an extra layer of RTX traversal that does not support recursion.

Again there are different ways to address this, and each one has its own pros and cons. One way would be to use a software BVH (or other kind of acceleration structure) as TLAS. This comes with obvious engineering costs of having to integrate additional frameworks for software traversal, and the pitfall that surface geometry ray tracing might be less efficient than when using hardware TLASs. Yet another way to address the issue is to treat surfaces and volumes differently, and not put the latter in a common TLAS at all. This is viable when only rendering a small number of (or even a single) volume—which is quite common in sci-vis. Then we can maintain a short list of volumes and instance transforms and trace rays into these directly, by linearly iterating through the list. With the combination of uniform grid for traversal and RTX for sampling, in this case we would directly start with DDA traversal and not call **optixTrace** until taking samples from the spatial field. This variant is implemented in sample app 07 “render graphs”.

5 Advanced Topics

In this section we touch upon a number of advanced topics. We avoid going into too much detail here, as topics like data parallel rendering could easily be a course on its own. Instead, we highlight the importance of those topics and how the choice of techniques facilitates their implementation, and provide references for further reading on those topics.

5.1 Data parallel volume rendering

As mentioned throughout this course, volume data from certain fields has grown significantly. The fields primarily affected are the simulating sciences, and microbiology outputting large microscopy data. Visualizing those data on one GPU is often impossible. Data parallel volume rendering therefore subdivides the volume into disjoint pieces, such as blocks, or submeshes, and assigns these to their own GPU or network computer to render an intermediary result independent of the other pieces.

With traditional volume rendering using ray marching and absorption plus emission, the standard way of parallelizing this procedure is to render and composite intermediary RGBA images using alpha compositing. Libraries like IceT [Moreland et al. 2011] provide efficient implementations for alpha compositing using the message passing interface (MPI). This comes to its limits with scattering operations: rays that need to compute secondary effects such as shadows, ambient occlusion or global illumination need to test for intersections and particle collisions not just on the GPU they originated from, but on all GPUs that hold data that is in the line of sight. Different strategies exist to implement this, usually involving queuing of rays and then passing them to other nodes based on some rules. This often includes traversing additional BVHs that are replicated on all nodes. An important observation is that Woodcock tracking greatly simplifies this. With Woodcock, we can use virtually any acceleration structure, and that extends naturally to acceleration structures on multiple devices. In fact, it does not even matter if that acceleration structure uses a spatial or object partitioning, which is in stark contrast to acceleration structures commonly used for empty space skipping and ray marching. Further readings on the topics of data parallel (volume) rendering and communication patterns are, among many others, the papers by Wald *et al.* [Wald et al. 2023] and by Zellmann *et al.* [Zellmann et al. 2022].

Another important question is the handling of local and global data, including the spatial field, but also local and global framebuffers or ray queues. If multiple GPUs in one node come to use, data can be shared using CUDA managed memory regions. When rendering on multiple nodes with a low-latency interconnect, CUDA-aware MPI [Zhang et al. 2019] becomes relevant that—in conjunction with technologies like GPUDirect—allows the GPUs to communicate directly, without going through the host and its network interface.

Finally, many sci-vis applications have to deal with data that is already distributed, e.g., if the visualization happens *in-situ*. In this case the visualization is attached to a simulation code running on a high performance computer. The simulation and visualization share or exchange local data. The visualization only has access to the data on its own compute node. While technically similar to the data parallel applications discussed above, we are now no longer in control of the data distribution, and the distribution we are given may be adversarial. Robust techniques can handle the resulting load imbalances without significantly impacting rendering performance.

5.2 Integration into real-world applications

We finally discuss integration of the proposed framework with real-world applications like VTK or ParaView [Ahrens et al. 2005]. The VTK ecosystem is the de facto standard used by sci-vis practitioners. VTK comes with its own volume renderer, which falls into the exact category of early-day renderers discussed in Section 2. Fortunately, VTK allows for a way to bypass that renderer, via the Khronos standard ANARI [Stone et al. 2022] and its rendering API. ANARI is an open standard that is inspired by the architecture of Intel’s OSPRay [Wald et al. 2017]. Many implementations targeting different hardware and rendering technology exist. ANARI also supports volume data; version 1.1 of the standard added support for unstructured volumes types, such as our tetrahedral meshes from Section 4.1. ANARI comes with an SDK that also provides software layers to build custom ANARI libraries. Our sample code framework comes with its own

ANARI library (app 09 “anari”). The library only supports volume rendering, and the only supported spatial field is the unstructured mesh type. The sample app should, however, provide a good starting point for the reader to build their own ANARI library that can be extended to become production-ready.

6 Conclusions

In this course we have made an argument for the sci-vis community to adopt volume rendering techniques common in VFX. Embracing free-flight distance tracking techniques has the disadvantage of introducing variance and noise, but once this is dealt with, provides a much more flexible framework in any way imaginable. Integration with all sorts of geometry, overlapping volumes, or mixing with semi-transparent surfaces, suddenly becomes easy to handle as techniques like Woodcock tracking give us a definitive intersection (yet one that is subject to variance) that we can compare to other intersections, and by that computing the closest hit encountered by a ray. Acceleration structures are a natural byproduct of using Woodcock and other tracking algorithms.

Volume rendering in sci-vis comes with a couple of caveats. Not all tracking algorithms are viable, due to the way transfer functions are interpreted by this framework. The transfer function indirection also has implications on how to build and update majorant acceleration structures. While in VFX it may be possible to handcraft an acceleration structure only based on the spatial arrangement given by the scalar data, including bounding the density with majorants *and* minorants, such optimizations are harder to implement in the presence of RGBA transfer functions. In fact, it is important for exploration that transfer function updates remain interactive, which places even more restrictions on the way acceleration structures can be built for scientific volumes.

The framework we present should provide a good starting point for practitioners to implement their own sci-vis volume renderers. Many of the techniques we teach in this course are already implemented by frameworks like OSPRay or several other ANARI device libraries. Our contributions are a gentle introduction to Monte Carlo volume rendering targeted at sci-vis practitioners, a thorough discussion of the different design choices and trade-offs, and how they potentially affect performance, and last but not least a well-organized sample code framework that is released under a permissive software license and comes with its own ANARI device library. By releasing all those resources and teaching the underlying fundamental techniques we hope to successfully bridge the two communities from sci-vis and VFX, and that other sci-vis apps follow suit and adopt Monte Carlo techniques in their volume renderers.

Acknowledgments

This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grant no. 456842964. Part of this work was also funded by the German Federal Ministry of Research, Technology and Space under the funding code 01LK2204A. The responsibility for the content of this publication lies with the authors.

References

- James Ahrens, Berk Geveci, and Charles Law. 2005. ParaView: An End-User Tool for Large-Data Visualization. In *Visualization Handbook*, Charles D. Hansen and Chris R. Johnson (Eds.).
 - Klaus Engel, Markus Hadwiger, Joe M. Kniss, Aaron E. Lefohn, Christof Rezk Salama, and Daniel Weiskopf. 2004. Real-time volume graphics. In *ACM SIGGRAPH 2004 Course Notes* (Los Angeles, CA) (*SIGGRAPH '04*). Association for Computing Machinery, New York, NY, USA, 29–es. doi:10.1145/1103900.1103929
 - Julian Fong, Magnus Wrenninge, Christopher Kulla, and Ralf Habel. 2017. Production volume rendering: SIGGRAPH 2017 course. In *ACM SIGGRAPH 2017 Courses* (Los Angeles, California) (*SIGGRAPH '17*). Association for Computing Machinery, New York, NY, USA, Article 2, 79 pages. doi:10.1145/3084873.3084907
 - Intel Corporation. 2023. High Performance Volume Kernels. <https://www.openvkl.org>.
- Manuscript submitted to ACM

- Ralf Kähler, John Wise, Tom Abel, and Hans-Christian Hege. 2006. GPU-Assisted Raycasting for Cosmological Adaptive Mesh Refinement Simulations. In *Volume Graphics*.
- J. Kniss, S. Premoze, C. Hansen, and D. Ebert. 2002. Interactive translucent volume rendering and procedural modeling. In *IEEE Visualization, 2002. VIS 2002*. 109–116. doi:10.1109/VISUAL.2002.1183764
- Marc Levoy. 1990. Efficient ray tracing of volume data. *ACM Trans. Graph.* 9, 3 (July 1990), 245–261. doi:10.1145/78964.78965
- A. Mammen. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications* 9, 4 (1989), 43–55. doi:10.1109/38.31463
- N. Max. 1995. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics* 1, 2 (1995), 99–108. doi:10.1109/2945.468400
- Kenneth Moreland, Wesley Kendall, Tom Peterka, and Jian Huang. 2011. An Image Compositing Solution at Scale. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*.
- Nate Morrical and Stefan Zellmann. 2021. *Inverse Transform Sampling Using Ray Tracing Hardware*. Apress, Berkeley, CA, 625–641. doi:10.1007/978-1-4842-7185-8_39
- Ken Museth. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph.* 32, 3, Article 27 (July 2013), 22 pages. doi:10.1145/2487228.2487235
- Ken Museth, Jeff Budberg, Alexandre Sirois-Vigneux, Gregory Hurst, Francis Williams, Andre Pradhana, Dan Bailey, and Nick Avramoussis. 2025. OpenVDB. In *Proceedings of the Special Interest Group on Computer Graphics and Interactive Techniques Conference Courses (SIGGRAPH Courses '25)*. Association for Computing Machinery, New York, NY, USA, Article 1, 4 pages. doi:10.1145/3721241.3733987
- Jan Novák, Iliyan Georgiev, Johannes Hanika, Jaroslav Krivánek, and Wojciech Jarosz. 2018. Monte Carlo methods for physically based volume rendering. In *ACM SIGGRAPH 2018 Courses* (Vancouver, British Columbia, Canada) (SIGGRAPH '18). Association for Computing Machinery, New York, NY, USA, Article 14, 1 pages. doi:10.1145/3214834.3214880
- Jan Novák, Andrew Selle, and Wojciech Jarosz. 2014. Residual ratio tracking for estimating attenuation in participating media. *ACM Trans. Graph.* 33, 6, Article 179 (Nov. 2014), 11 pages. doi:10.1145/2661229.2661292
- NVIDIA Corporation. 2026. OWL – A Productivity Library for OptiX. <https://github.com/NVIDIA/OWL>.
- Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics* 29, 4 (2010).
- Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2023. *Physically Based Rendering: From Theory to Implementation* (4th ed.). The MIT Press. <https://pbr-book.org/>
- Alper Sahistan, Stefan Zellmann, Haichao Miao, Nate Morrical, Ingo Wald, and Valerio Pascucci. 2026. Materializing Inter-Channel Relationships With Multi-Density Woodcock Tracking. *IEEE Transactions on Visualization and Computer Graphics* 32, 3 (2026), 2726–2740. doi:10.1109/TVCG.2026.3653310
- Will Schroeder, Ken Martin, and Bill Lorensen. 2006. *The Visualization Toolkit* (4th ed.). Kitware.
- Peter Shirley and Allan Tuchman. 1990. A polygonal approximation to direct scalar volume rendering. In *Proceedings of the 1990 Workshop on Volume Visualization* (San Diego, California, USA) (VVS '90). Association for Computing Machinery, New York, NY, USA, 63–70. doi:10.1145/99307.99322
- John E Stone, Kevin Griffin, Jefferson Amstutz, David E DeMarle, William R Sherman, and Johannes Günther. 2022. ANARI: A 3-D Rendering API Standard. *Computing in Science & Engineering* 24, 02 (2022). doi:10.1109/MCSE.2022.3163151
- Ingo Wald, Milan Jaroš, and Stefan Zellmann. 2023. Data Parallel Multi-GPU Path Tracing using Ray Queue Cycling. *Computer Graphics Forum* 42 (2023).
- Ingo Wald, Greg P. Johnson, Jefferson Amstutz, Carson Brownlee, Aaron Knoll, Jim Jeffers, Johannes Günther, and Paul Navrátil. 2017. OSPRay – A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics* (2017).
- Ingo Wald, Will Usher, Nate Morrical, Laura Lediaev, and Valerio Pascucci. 2022. RTX beyond ray tracing: exploring the use of hardware ray tracing cores for tet-mesh point location. In *Proceedings of the Conference on High-Performance Graphics* (Strasbourg, France) (HPG '19). Eurographics Association, Goslar, DEU, 7–13. doi:10.2312/hpg.20191189
- Stefan Zellmann, Milan Jaroš, Jefferson Amstutz, and Ingo Wald. 2025. GPU Volume Rendering with Hierarchical Compression Using VDB. In *Eurographics Symposium on Parallel Graphics and Visualization*, Guido Reina, Silvio Rizzi, and Charles Gueunet (Eds.). The Eurographics Association. doi:10.2312/pgv.20251152
- Stefan Zellmann, Ingo Wald, Joao Barbosa, Serkan Dermici, Alper Sahistan, and Uğur Güdükbay. 2022. Hybrid Image-/Data-Parallel Rendering Using Island Parallelism. In *2022 IEEE 12th Symposium on Large Data Analysis and Visualization (LDAV)*. 1–10. doi:10.1109/LDAV57265.2022.9966396
- Stefan Zellmann, Qi Wu, Alper Sahistan, Kwan-Liu Ma, and Ingo Wald. 2024. Beyond ExaBricks: GPU Volume Path Tracing of AMR Data. *Computer Graphics Forum* 43, 3 (2024), e15095. doi:10.1111/cgf.15095
- Jie Zhang, Xiaoyi Lu, Ching-Hsiang Chu, and Dhableswar K Panda. 2019. C-GDR: High-Performance Container-aware GPUDirect MPI Communication Schemes on RDMA Networks. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '19)*. IEEE.