A Query-Efficient Stochastic Volume Rendering Framework for Time-Varying Implicit Neural Volumes

Alper Sahistan, Haichao Miao, Zhimin Li, Peer-Timo Bremer, Joshua A. Levine, Valerio Pascucci

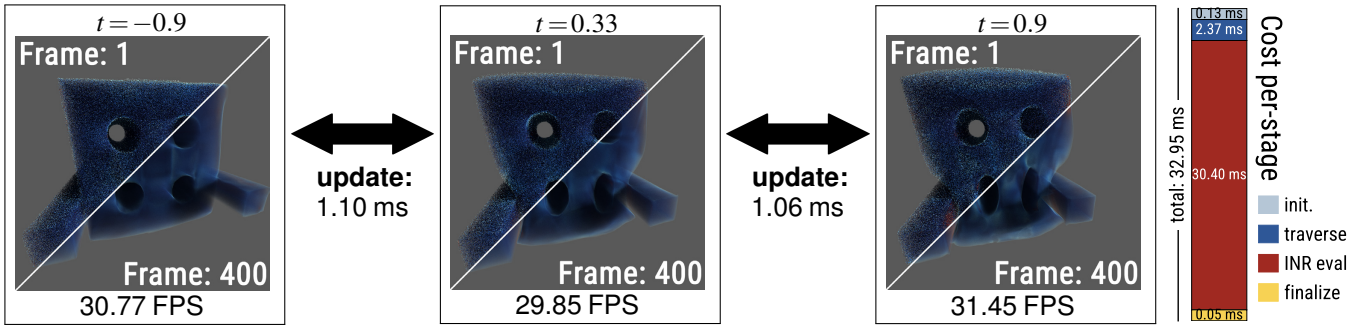

Figure 1: Volume-rendered frames of a time-varying implicit neural representation (INR) encoding X-ray tomography of a physically deforming object, in which Fourier-feature-encoded coordinates (x,y,z,t) are evaluated by a six-layer fully connected network to produce the scalar field. Despite neural inference dominating runtime (as shown in the bar chart on the right), our query-efficient ray tracing framework renders the original network directly, without retraining or modification, at ~ 30.7 FPS and 1024^2 resolution with ray-traced shadows, and enables interactive animation between timesteps ($\sim$1 ms) by leveraging the continuous nature of the INR.

Abstract—Time-varying implicit neural representations (INRs) provide a compact representation of scientific volumes and, for modalities such as dynamic X-ray computed tomography (CT), are often the only practical way to represent the data. However, interactive volume rendering of INRs is challenging, as cheap memory lookups are replaced by expensive neural inferences, hindering the performance. Therefore, conventional volume rendering methods such as ray marching with dense sampling are often impractical. While resampling, caching, and retraining can mitigate this cost, they compromise convenience and accuracy and become impractical for time-varying data. We tackle these challenges using a query-efficient stochastic volume rendering framework based on delta tracking. Our system employs a four-stage pipeline that exploits heterogeneous parallelism, using ray tracing cores for traversal and tensor cores for batched neural evaluation. Furthermore, we present strategies to reduce INR queries via ray budgeting and query pruning, thereby increasing per-frame performance. Using our renderer, many time-varying INRs can be rendered directly from their original representation. The system achieves $\sim$30–40 FPS at 1024^2 resolution on an RTX 4090 GPU and converges to high-fidelity images. Moreover, the system enables interactive temporal exploration of the continuous domain, with timestep updates taking approximately 1–2 ms.

Index Terms—Volume rendering, ray tracing, implicit neural representations, time series data, scientific visualization.

1 Introduction

Beyond spatial scale, many scientific datasets are time-varying. Rather than analyzing a single static volume, scientists increasingly seek to understand how complex phenomena evolve, such as fluid dynamics, material deformation, or energy transport. Furthermore, INRs can encode dynamic phenomena when explicit reconstruction is not feasible [32]. Interactive exploration of these datasets, therefore, requires not only rendering large static volumes but also supporting temporal navigation to analyze continuous changes. However, explicit data representations such as meshes restrict navigation to discrete timesteps and require additional processing to interpolate or transition between them.

These challenges have motivated the adoption of implicit neural representations (INRs), which encode volumetric data as continuous neural functions, thereby significantly reducing storage requirements from terabytes to megabytes. Beyond compression, INRs are also well suited for data representation in computed tomography of dynamic or deforming objects, where measurements acquired at different projection angles do not correspond to a single static volume due to deformations during acquisition [32]. Moreover, by selecting a time parameter, a single model can generate arbitrary timesteps without loading, caching, or managing separate timestep volumes, enabling direct access to any point in continuous time. Furthermore, the continuous domain also lends itself to analysis, making the data format valuable beyond purely visualization.

Despite these advantages, interactive visualization of INRs remains challenging, as sampling a point's value requires evaluating multiple layers of neural network inference, which is substantially more expensive than the memory accesses used in conventional volume rendering. Resampling an INR onto a grid is problematic: coarse grids may miss features, fine grids could exceed memory limits, and selecting an optimal resolution is nontrivial. More broadly, such approaches introduce additional complexity and approximation. Rendering through intermediate grids or retrained models adds extra preprocessing steps, reduces fidelity due to resampling, and becomes increasingly impractical in the temporal domain. While caching is effective for static volumes, extending it to time-varying data requires storing separate caches for each timestep, leading to frequent invalidations during temporal navigation.

In recent volume visualization literature, Monte Carlo–based rendering techniques have gained popularity due to their ability to reduce computational cost by taking fewer but more important samples. While this approach introduces variance into individual frames, the resulting images converge over time as additional samples are accumulated. Among these techniques, delta tracking [13, 20, 30, 64]—also known as Woodcock tracking—has emerged as one of the most widely used methods for stochastic volume rendering. In this work, we reinterpret delta tracking as a query-reduction mechanism for implicit neural fields, where the dominant cost is network inference rather than texture fetch.

• *Alper Sahistan and Valerio Pascucci are with the University of Utah.*
• *Haichao Miao and Peer-Timo Bremer are with the Lawrence Livermore National Laboratory.*
• *Zhimin Li is with Vanderbilt University.*
• *Joshua A. Levine is with the University of Arizona.*

Manuscript received xx xxx. 202x; accepted xx xxx. 202x. Date of Publication xx xxx. 202x; date of current version xx xxx. 202x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.202x.xxxxxxx

Building on this insight, we present a query-efficient stochastic volume rendering framework that operates directly on continuous neural representations, minimizing and amortizing neural field evaluations to achieve interactive frame rates and low-latency temporal interaction while preserving visual quality (Figure 1). Our contributions are:

- A four-stage rendering pipeline that decouples traversal from neural inference and coordinates ray tracing cores, tensor cores, and general compute units to maximize GPU utilization while operating directly on the original INRs.
- Three adaptive ray budgeting schemes that prioritize sampling in temporally and perceptually changing regions, using inter-frame color variation and high-frequency cues to allocate computation efficiently while preserving full coverage over time.
- A homogeneity-based query pruning strategy that detects near-uniform regions and probabilistically avoids unnecessary neural evaluations while minimizing visual artifacts.

2 Related Work

This work builds on two important bodies of literature: volume rendering and implicit neural representations.

Volume Rendering. Ray marching has been the de facto standard approach for volume rendering, as it is capable of producing images with virtually no variance by deterministically accumulating many (typically equally spaced) partial samples along each ray [15, 24, 43, 51]. Because these deterministic methods require dense sampling patterns, substantial effort has focused on reducing traversal cost through space skipping, bricking, and multi-resolution acceleration structures [28, 34, 62]. However, even with adaptive sampling strategies, achieving real-time rates proves challenging with this family of methods.

Another popular school of methods for volume rendering is tracking-based Monte Carlo estimators, which provide an alternative to dense integration by importance sampling the medium. A well-known tracking method, delta tracking [64], performs stochastic sampling of the medium based on an upper bound of its density [10, 57, 68]. There are several works that improve efficiency through adaptive traversal schemes [13, 20, 30, 71] over multiple data formats, but these works are designed for non-neural volumes. Similar to our work, Sahistan et al. [53, 54] use multiple coarse grids for multi-channel rendering rather than for time-varying data. Other tracking variants, such as Novak et al.'s residual-ratio tracking [39] or Kutz et al.'s spectral/decomposition tracking [25], split *extinction* across independently parameterized media, assumptions impractical for scientific visualization, where color is derived from a transfer function over a single scalar field.

Beyond algorithmic improvements, recent work has explored leveraging the dedicated, specialized hardware in modern GPUs for scientific volume rendering. Ray tracing (RT) cores have been used to accelerate point-location queries in unstructured meshes [35, 52, 61, 70], enabling efficient traversal and containment tests directly on hardware designed for ray tracing. Building on this capability, Quick Clusters demonstrate GPU-parallel partitioning to support efficient ray tracing of unstructured volumetric grids [33].

Human perception is not uniform over a rendered frame. Exploiting this fact to improve performance, researchers used various perception-optimized sampling strategies [59], such as foveated rendering [12, 23] and blue-noise masks [11], to distribute rendering effort where it matters most visually. Spatio-temporal blue-noise sampling has been used to spread work progressively over space and time while preserving high-frequency noise characteristics [63]. Although several applications of these ideas live in the scientific visualization domain [5, 36], most research effort has primarily targeted virtual reality applications [42, 67].

Neural Representations. Implicit neural representations (INRs) have gained significant traction across both graphics and scientific visualization. In graphics, neural radiance fields (NeRFs) [31] model view-dependent image synthesis with pre-integrated color, whereas in scientific visualization, INRs represent scalar fields with color defined by a transfer function during integration. Although they are similar in principle, INRs in SciVis are also considered a form of compact neural encoding for lossy compression [17, 29]. By enabling random access to arbitrary spatial regions of a volume, INRs eliminate the need to decompress entire volumes compared with previous state-of-the-art lossy compression techniques [8, 27].

The NeRF literature in computer graphics has explored model- and system-level optimizations. KiloNeRF [48] replaces a large and deep MLP with thousands of tiny MLPs, each modeling a small local region of volume, to reduce per-query inference cost. MINER [55] leverages a Laplacian pyramid for high-resolution and large-scale signals modeling, enabling a coarse-to-fine scale query. Alternative approaches improve inference performance by optimizing computational memory usage. For example, MERF [49] replaces dense 3D feature grid with high-resolution 2D planes and sparse feature grid. However, these methods achieve efficiency by modifying the underlying representation, which requires retraining or restructuring the base model.

Instead of learning a function mapping from a coordinate index to density and color, as in NeRF, Gaussian splatting [22] learns *explicit* Gaussian blobs to represent the data. Sewell et al. [9] highlight the importance of robust point cloud initialization using Gaussian splatting for scientific data visualization. Bauer et al. [4] apply Gaussian splatting as a cache mechanism to improve path tracing performance in scientific volume rendering. Han et al. [19] extend 3D Gaussian splatting over multiple GPU training for scientific data and demonstrate significant performance improvement over a single GPU run. While Gaussian splatting-based methods achieve high rendering performance and compression, their representations are primarily optimized for image synthesis and do not readily support non-visualization analysis tasks that require direct access to the underlying scalar field.

When applied to interactive scientific visualization, INRs fundamentally shift the performance bottleneck from memory access to neural inference. While previous works [16–18, 58] primarily focus on improving the compression ratio and reconstruction quality, inference cost remains the dominant factor affecting rendering performance.Among the few works addressing interactive INR rendering, Zavorotny et al. [69] reduce the query cost through advanced caching mechanisms.

3 Background on Delta Tracking

Our framework builds on *delta tracking* [64]. We briefly summarize its formulation and common design practices [33, 53], using notation similar to that of [10]. As this is not a comprehensive guide, we point readers to [44] for further details.

Delta tracking simulates interactions between rays and a participating medium via Monte Carlo sampling. For a homogeneous medium with constant extinction σ_t, the *free-flight* distance (the distance a ray travels before hitting a particle) follows an exponential distribution:

$$p(t) = \sigma_t \exp(-\sigma_t t), \tag{1}$$

where σ_t denotes the extinction coefficient.

The caveat with scientific volumes is that the extinction coefficient $\sigma_t(x)$ varies spatially. Typically, a user-adjusted transfer function maps scalar values to optical properties, including emission $L_e(x,\omega)$ and extinction $\sigma_t(x)$. Because $\sigma_t(x)$ is not constant, the exponential form in Equation 1 cannot be sampled directly. Delta tracking instead introduces a constant *majorant* $\bar{\sigma} \geq \sigma_t(x)$ and samples free-flight distances from the homogeneous distribution obtained by replacing σ_t in Equation 1 with $\bar{\sigma}$. Importance sampling this distribution with a uniform random variable $\xi \sim \mathscr{U}(0,1)$ yields

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

In practice, Equation 2 reduces to a single exponential inversion (Listing 1), producing the next free-flight step.

Listing 1: Free-flight sampling under a majorant extinction coefficient.

```
function FreeFlightStep(majorant)
    return -log(1 - UniformRandom(0, 1)) / majorant;
end
```

At each sampled location $x_t = x + t\omega$, the interaction is classified as either a *real* or *null* collision, occurring with the following probabilities respectively:

$$P_{\text{real}}(x_t) = \frac{\sigma_t(x_t)}{\bar{\sigma}}, \qquad P_{\text{null}}(x_t) = 1 - \frac{\sigma_t(x_t)}{\bar{\sigma}}. \tag{3}$$

This could be represented as a Bernoulli test with acceptance probability $\sigma_t(x_t)/\bar{\sigma}$ (Listing 2).

Listing 2: Bernoulli acceptance test for real collisions.

```
function AcceptCollision(majorant, localExtinction)
```

```
    if UniformRandom(0, 1) < localExtinction / majorant then
        return true;
    else
        return false;
    end if
end
```

If the candidate collision is accepted, emission is accumulated, and the ray terminates (emission-absorption model). If a *null collision* occurs, another free-flight distance is drawn, and sampling continues from Equation 2.

This formulation can be interpreted as rewriting the volume rendering equation in terms of interactions sampled from the majorant medium. The expected radiance along a ray starting at x in direction ω can then be expressed as

$$L(x,\omega) = \mathbb{E}\left[\sum_k T(t_k)\frac{\sigma_t(x_{t_k})}{\bar{\sigma}} L_e(x_{t_k},\omega)\right]. \tag{4}$$

Here, t_k is the k-th free-flight distance sampled under the majorant $\bar{\sigma}$, x is the ray origin, and ω is the ray direction, so $x_{t_k} = x + t_k\omega$ denotes the sampled point along the ray. The term $T(t_k)$ is the accumulated transmittance up to t_k, and $\sigma_t(x_{t_k})/\bar{\sigma}$ is the probability that the interaction at x_{t_k} is a real collision rather than a null event.

In more intuitive terms, delta tracking treats a heterogeneous medium as a homogeneous one with density $\bar{\sigma}$ and corrects the discrepancy through rejection sampling: each sampled interaction is accepted with probability $\sigma_t(x_t)/\bar{\sigma}$ and otherwise discarded as a null event. This mechanism enables traversal of spatially varying media without explicitly evaluating the heterogeneous transmittance. The core stepping logic is summarized in Listing 3.

Listing 3: Core delta tracking loop for scientific volume visualization.

```
function DeltaTracking(volume, rayOrigin, rayDirection,
                       tMin, tMax, majorant, transferFunction)
    t = tMin;
    while t < tMax do
        t = t + FreeFlightStep(majorant);
        if t >= tMax then
            break;
        end if
        position = rayOrigin + t * rayDirection;
        scalar = SampleVolume(volume, position);
        sampleColor = transferFunction(scalar);

        if AcceptCollision(majorant,
            ↪ sampleColor.opacity) then
            return sampleColor;
        end if
    end while
    return backgroundColor;
end
```

A single global majorant can induce excessive null collisions, particularly when small regions exhibit high density. A common remedy is to subdivide the volume and assign each cell a local majorant $\bar{\sigma}(x)$. A traversal scheme such as a 3D digital differential analyzer (DDA) is then used to march between cells, applying the delta-tracking loop (Listing 3) within each region using its corresponding local majorant.

In scientific visualization (sciVis), the transfer function determines the extinction and therefore the local majorants. To compute these efficiently, we precompute the scalar minimum and maximum within each subvolume, commonly referred to as *macrocells*. The majorants can be updated in parallel on the GPU by scanning each macrocell's scalar minimum and maximum against the transfer function and recording the maximum resulting opacity. This enables dynamic, interactive majorant updates [33].

Unlike slicers or ray marching, where many partial samples are over-composited [21], only a single sample per ray is accepted. This produces variance, yet accumulating more samples over multiple rays lets this noise converge. A common practice is to implement an *accumulation buffer* next to a frame buffer. This buffer accumulates frames from a still scene and averages the results over time before copying the results back to the framebuffer.

Our motivation for adopting this method for INR rendering mirrors its original appeal in scientific volume rendering: fixed-step integration can quickly become non-interactive, whereas stochastic sampling remains efficient, extends naturally to secondary effects such as shadows, and converges in an unbiased manner. In the INR setting, the benefit is amplified, as each neural field query is substantially more expensive than a 3D texture fetch or finite-element interpolation, making query reduction essential for interactivity.

4 Framework Design

We design our framework around four constraints: (1) enabling interactive rendering of INRs, (2) allowing low-latency temporal exploration of time-varying INRs, (3) operating directly on INRs as-is, and (4) maintaining estimator correctness under practical approximations.

Under these constraints, INR queries become the dominant bottleneck. Achieving interactivity, therefore, requires minimizing neural queries and efficiently executing the unavoidable ones. Dense sampling strategies, such as ray marching, entail excessive neural evaluations, and caching schemes [69] break down for time-varying data, as temporal navigation would invalidate cached values. Allowing users to work directly with existing INRs, without added preprocessing or workflow overhead (as in established sciVis solutions), rules out approaches that require retraining or modifying the original network.

Overall, delta tracking has been shown to greatly reduce the number of samples, but naïvely embedding neural inference into a ray tracing loop results in poor hardware utilization as neural inference and ray traversal exhibit fundamentally different parallelism patterns. Therefore, coordinated heterogeneous parallelism is required to tackle this problem efficiently. The following sections describe the pipeline (Section 4.1) and query-reduction strategies (Section 4.2), together enabling interactive visualization of time-varying INRs.

4.1 Wavefront Pipeline Overview

A naïve implementation of delta tracking over a neural volume would evaluate the INR inside the innermost ray-marching loop, issuing one network inference per sample point. On modern GPUs, this leads to severe under-utilization: delta tracking is inherently serial per ray, whereas neural inference benefits from large, uniform batches that can saturate tensor cores. Our pipeline decouples the two workloads by splitting each frame into alternating *traverse* and *evaluate* stages that communicate through shared `INR query buffers`, following the wavefront ray tracing paradigm [26]. Moreover, this design allows us to leverage dedicated hardware for each computation — tensor cores for INR evaluations and ray tracing (RT) cores for traversal.

Figure 2 provides an overview of our rendering framework, which is organized into four stages: ray initialization, traversal, INR evaluation, and finalization. After ray initialization (subsubsection 4.1.2), a host-controlled traverse-evaluate loop alternates between RT-core macrocell traversal (subsubsection 4.1.3), which advances rays and produces candidate collisions, and batched tensor-core INR evaluation (subsubsection 4.1.4), which evaluates those candidates and updates ray state. The loop repeats until all rays terminate, typically after 18–22 iterations for a 64^3 macrocell grid. Finalization (subsubsection 4.1.5) writes the results to the framebuffer and updates per-pixel statistics; persistent cross-stage data structures are described in subsubsection 4.1.1.

4.1.1 Data Structures

The pipeline first constructs the spatial structures based on the macrocells outlined in section 3. Since INRs are continuous, exact per-cell extrema are not analytically tractable, so we approximate scalar bounds using GPU-based *lattice* sampling. The macrocell grid is processed in 4^3-cell tiles, where threads cooperatively evaluate the INR in shared memory over a 9^3 lattice at cell boundaries and half-cell offsets. Each macrocell reduces its bounds from the local 3^3 stencil covering the cell, storing per-cell minima and maxima for majorant construction. Overlapping lattices allow the reuse of many INR evaluations across adjacent macrocells. This process is illustrated in Figure 3).

For time-varying INRs, we precompute T temporal macrocell grids at start-up, forming a $N_x \times N_y \times N_z \times T$ array of per-cell bounds. When the timestep changes, a *working grid* is reconstructed from the two nearest temporal slices via linear interpolation. However, this interpolation may miss intermediate extrema. Increasing T reduces these issues at the cost

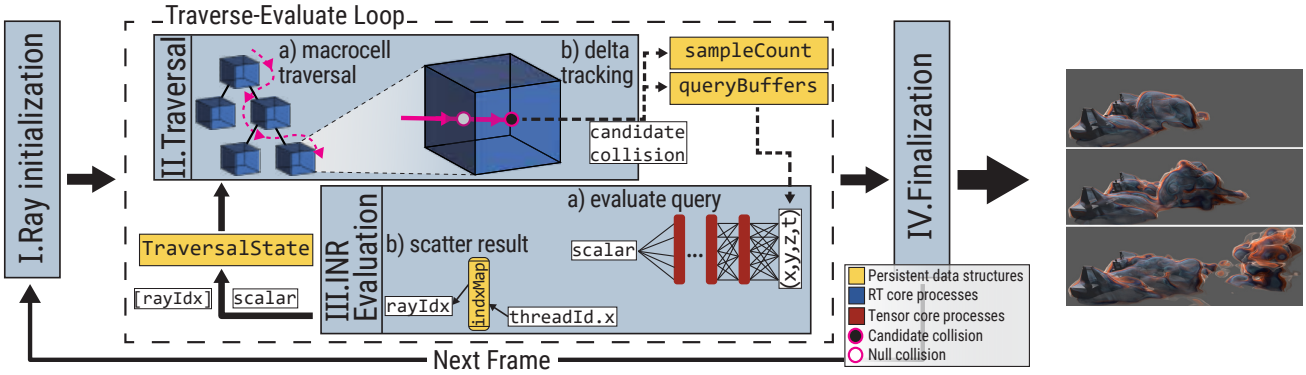

Figure 2: Overview of our INR visualization framework: Each frame begins with **I. Ray initialization** (cf. subsubsection 4.1.2), where rays and all other persistent data structures are initialized. Control then enters the *traverse–evaluate* loop, which consists of two stages. In **II. Traversal** (subsubsection 4.1.3), a) Ray tracing (RT) cores locate the current macrocell, and b) delta tracking within that macrocell generates a *candidate collision*. In **III. INR evaluation** (subsubsection 4.1.4), a) the recorded candidate queries are batched and evaluated on the GPU's tensor cores, and b) the resulting scalars are scattered back to the corresponding rays' `TraversalState`. Ultimately, **IV. Finalization** (cf. subsubsection 4.1.5) is invoked once all rays either find a collision or exit the volume, writing results to the frame buffer and collecting statistics.

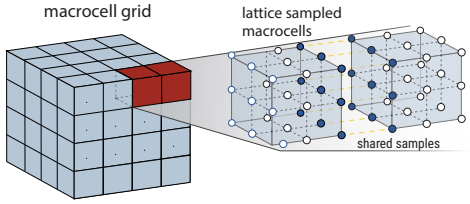

Figure 3: Illustration of the lattice-shaped sampling process for the macrocell grids. Blue colored (on the right) samples are shared between the two example macrocells, and all samples within a macrocell are used to find a minimum and a maximum of the scalars.

of memory and start-up time. One can add padding to the min and max values during interpolation to allow for error, but this would result in looser majorants and more null collisions.

The main problem with these approximate macrocells is underestimation, which leads to visual artifacts resembling boxes. After the working grid is constructed, we can allow the renderer to progressively refine these underestimated bounds as the user explores the data over time. Yet this would only refine some of the macrocells if the user never changes the transfer function (i.e., majorants). As the free-flight distance (Listing 1) is inversely proportional to the majorant, it causes some macrocells to be falsely mapped to a low majorant, causing rays to skip over that macrocell and never update. To address this, we introduce *ghost passes*, in which the volume is rendered with a set of cosine-shaped transfer functions that sweep the scalar range. These passes are not displayed, but ensure that visible macrocells are probabilistically visited, enabling consistent refinement of macrocell bounds and mitigating persistent underestimation artifacts. Figure 4 depicts these artifacts and effects of the ghost pass refinement.

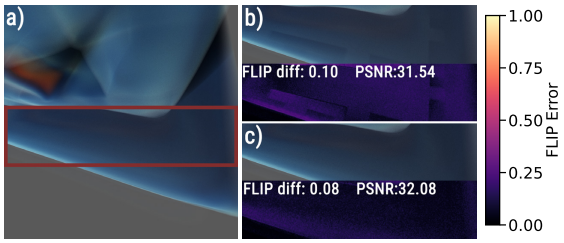

Figure 4: Comparison of ghost-pass macrocell refinement against ground truth on the S03_001 dataset: a) ground truth rendered with a loosely allocated 1^3 macrocell grid (2.80 FPS); b) crop of the red box from a) rendered using a 64^3 macrocell grid without ghost-pass refinement (21.04 FPS); c) same as b) with ghost-pass refinement (18.65 FPS). Under b) and c), we report the FLIP [2] difference relative to the ground truth image, along with mean FLIP and Peak Signal-to-Noise Ratio (PSNR).

The majorants are converted into a sparse set of custom AABB primitives within the *OptiX* ray tracing pipeline, emitting one primitive per macrocell whose majorant exceeds a small threshold, along with an `int3` array mapping primitive indices to cell coordinates. These primitives form a single Geometry Acceleration Structure (GAS) built with `OPTIX_BUILD_FLAG_PREFER_FAST_TRACE` and subsequently compacted. Because only non-empty cells generate primitives, the resulting acceleration structure is sparse and cache-friendly. The GAS is quickly rebuilt within 2-3 ms whenever the working grid changes due to transfer-function edits or timestep updates.

In addition to these spatial structures, the pipeline allocates its persistent runtime buffers. The primary structure is a per-ray `TraversalState` record (as described in Listing 4), which stores all state required to suspend and resume a ray across kernel launches.

Listing 4: Per-ray persistent state that is preserved across traverse-evaluate iterations.

```
struct TraversalState {
    Ray ray;
    unsigned seed; // RNG state
    uint8_t flags; // bools: active, hit, shadow_hit
    float sample; // scalar from INR (NaN = pending)
    float collision_product; // randFloat * majorant
};
```

The pipeline also maintains the aforementioned INR query buffers: arrays for query positions (`float4`), and ray indices (`int`), each with a fixed capacity. An atomic counter keeps track of the number of samples generated per iteration. Results are written back to the `sample` in `TraversalState` without indirections.

4.1.2 Ray Initialization

Ray initialization is performed by a lightweight 2D CUDA kernel that launches one thread per pixel and prepares the persistent `TraversalState`. Each thread generates a jittered camera ray, intersects it with the volume bounding box, and stores the valid ray segment together with the per-ray random-number state. Rays that miss the volume are marked not active immediately; rays that intersect the volume are marked active and store the valid ray segment for subsequent traversal passes. After ray initialization, the host enters a loop that alternates between traversal and batched INR evaluation.

4.1.3 RT-Core Macrocell Traversal and Delta Tracking

Unlike standard delta tracking implementations based on 3D DDA traversal, our method leverages ray tracing hardware. Macrocells with non-zero majorants are encoded as AABB primitives in a sparse *OptiX* GAS. At the beginning of each traversal pass, a global sample counter is reset, and the *OptiX ray-generation program* is launched for all active rays.

Each ray either resumes from its stored `TraversalState` or starts a new one. If a candidate collision is pending resolution from the previous iteration, the ray first performs the acceptance test (Listing 2) using the

stored sample obtained from the INR query. If there is no pending candidate collision or the candidate yields a null collision, traversal continues within the macrocell. An accepted collision terminates the traversal.

A ray visits intersected macrocell bounding boxes in front-to-back order using the GAS. Within each macrocell, the custom *OptiX intersection program* executes delta tracking (see Listing 3), sampling candidate free-flight distances from the exponential distribution using the local majorant $\bar{\sigma}$. Traversal proceeds directly to the next macrocell primitive without host intervention if the tracking step occurs outside the macrocell.

When a candidate collision is reported, the *closest-hit program* records the world-space sample position and stores the precomputed quantity `collision_product` $= \xi\bar{\sigma}$ in the persistent `TraversalState`. Because the acceptance test depends only on this product, neither ξ nor $\bar{\sigma}$ is stored separately, reducing per-ray state size. The `sample` field remains `NaN` until the INR evaluation completes. If INR evaluation is required for the sample, the program appends a query to the INR query buffers using `atomicAdd` on the global sample counter. Consequently, each ray generates at most one unresolved INR query per outer wavefront iteration.

After traversal, the host copies back only the sample counter. If it is zero, no unresolved samples remain and the primary traversal phase terminates. Otherwise, the collected sample positions are forwarded to the batched INR evaluation stage.

4.1.4 Batched Tensor-Core INR Evaluation

The evaluation stage consumes the dense batch of sample coordinates produced by traversal and executes INR inference on tensor cores. Each warp processes a fixed-size tile of queries using WMMA (Warp Matrix Multiply-Accumulate) primitives, propagating them through the network's fully connected layers in a single fused kernel. To maximize tensor-core utilization, weights are stored in 16-bit precision, and each warp maintains a private shared-memory workspace for staged inputs, weight tiles, and intermediate activations. The resulting scalars are written directly back into the corresponding `TraversalState` records via the ray-index array, eliminating a separate gather pass and completing the traverse–evaluate iteration in-place (see Listing 5).

Listing 5: Representative fused INR evaluation + scatter kernel. Each warp evaluates a tile of samples and writes scalar results directly into the per-ray `TraversalState` to avoid a separate gather pass.

```
__global__ void batchedEvalFused(
    const float4* coords, const int* rayIndices,
    TraversalState* states, int sampleCount)
{
    const int lane = threadIdx.x & 31; // warp lane
    const int warp = (blockIdx.x
        ↪ * blockDim.x + threadIdx.x) / 32;
    const int batch
        ↪ = warp * TILE; // TILE: queries per warp
    if (batch >= sampleCount) return;
    const int n = min(TILE, sampleCount - batch);
    // Evaluate tile (WMMA inference)
    float values[TILE];
    evaluateTile(coords + batch, values, n, model);
    //scatter each lane's
        ↪ result into the corresponding ray state
    if (lane < n) {
        const int rayIdx = rayIndices[batch + lane];
        states[rayIdx].sample = values[lane];
    }
}
```

When the evaluation kernel returns, every ray that requested a sample holds a resolved scalar in its `sample` field. The host resets the atomic counter and relaunches traversal (subsubsection 4.1.3). Resumed rays apply the transfer function, perform a collision test against the stored `collision_product`, and either terminate on a real event or clear the sample and continue stepping past a null collision.

4.1.5 Finalization

Once the traverse–evaluate loop terminates (sample count reaches zero), a finalize kernel is called. The finalize kernel is responsible for maintaining the accumulation buffer and computing the per-pixel residual signal r_i (Equation 5) used by the adaptive ray budgeting schemes described in subsection 4.2.

Ray-traced shadows can be achieved by using an analogous traverse-evaluate loop that is executed after the primary rays have found a collision, and the finalize kernel incorporates the resulting visibility term.

4.2 Query Reduction Strategies

Our key observation is that, under a limited ray budget, image quality improves more by selectively allocating rays than by uniformly tracing every pixel. In still or slowly changing regions, previously accumulated estimates are often sufficient, whereas unstable regions benefit most from new samples. We therefore treat ray allocation as a budgeting problem: decide which pixels to re-render, which to defer, and how to distribute those decisions so that skipped work produces acceptable noise and staleness patterns.

The strategies in subsubsection 4.2.1 control *where* the ray budget is spent at the pixel level. The techniques in subsubsection 4.2.2 reduce the cost of *each* traced ray by avoiding unnecessary INR queries inside near-uniform regions.

4.2.1 Adaptive Ray Budgeting

Progressive refinement is already common in scientific visualization [1, 14, 50, 60, 65]. Since not all rays contribute equally, we use progressive refinement as a budgeting process: prioritizing unstable pixels, deferring stable ones, and avoiding staleness and artifacts in the resulting image.

We define a per-pixel *convergence residual* signal r_i for pixel i at accumulation frame n as the mean absolute channel difference between the newly rendered color $C_i^{(n)}$ and the running accumulation average $\bar{C}_i^{(n-1)}$ from the previous frame. To guide these decisions, we estimate the change in each pixel relative to the current accumulation.

$$r_i = \frac{1}{3}\sum_{c\in\{R,G,B\}} \left| C_i^{(n)}[c] - \bar{C}_i^{(n-1)}[c] \right|, \quad r_i \in [0,1], \tag{5}$$

where $C_i^{(n)}[c]$ is channel $c \in \{R,G,B\}$ of the color obtained by tracing pixel i at frame n, and $\bar{C}_i^{(n-1)}[c]$ is the corresponding channel of the accumulated average from frame $n-1$. This residual is computed during finalization at negligible cost and drives most of the budgeting schemes. At ray initialization, each scheme decides whether pixel i is *budget-skipped*; budget-skipped pixels reuse their accumulated color and cast no rays or query the INR. Over these schemes, we explore different trade-offs between adaptivity and sampling pattern quality.

Residual-Histogram Thresholding. This scheme allocates the budget most aggressively to unstable pixels by ranking them by residual and skipping increasingly stable ones across multiple frames. Every rendered pixel contributes its residual to a 256-bin histogram, from which we derive a threshold retaining the top ρ fraction of pixels for immediate rendering. After the frame, the host scans the histogram to build a cumulative distribution function (CDF) and finds the bin b^* such that the cumulative fraction reaches $1-\rho$, where ρ is the user-specified re-render percentile (e.g. ρ=0.6 keeps the top 60% most divergent pixels). The threshold $\tau = (b^*+0.5)/256$ is uploaded for the next frame. During initialization, pixels whose residual r_i meets or exceeds the threshold τ render immediately ($\text{skip}_i = 0$); pixels below receive a skip countdown proportional to their stability:

$$\text{skip}_i = \left\lfloor \left(1 - \frac{r_i}{\tau}\right) \cdot S_{\max} + 0.5 \right\rfloor, \quad S_{\max} = \min\left(K, \lfloor n/4 \rfloor\right), \tag{6}$$

where skip_i is the number of consecutive frames pixel i will be skipped before re-entering the render set, τ is the residual threshold derived from the histogram CDF, K is a user-specified maximum skip duration, and n is the number of accumulated frames. The ratio $(1-r_i/\tau)$ ranges from 0 (at the threshold) to 1 (fully converged), so the most stable pixels receive the longest skip. The $\lfloor n/4 \rfloor$ ramp ensures that $S_{\max}$ grows gradually with n, preventing aggressive skipping during the first few frames when the accumulation buffer has not yet converged. Each skipped frame decrements the counter by one; when it reaches zero, the pixel re-enters the render set. This guarantees that every pixel is rendered at least once every $S_{\max}+1$ frames, bounding worst-case staleness.

Among the three schemes, this one is the most residual-driven and spatially concentrated, but also the most dependent on a reliable residuals.

STBN Thresholding. This scheme distributes a fixed render budget using spatiotemporal blue noise without using the residual signal. It

therefore serves as the simplest baseline and tends to produce more favorable sparse-update patterns.

We preload N spatiotemporal blue-noise (STBN) textures [63] of resolution 128×128 and tile them across the screen. For a pixel at screen coordinate x and frame index n, we fetch

$$u(x,n) = \frac{\texttt{stbn}(x, n \bmod N)}{256}, \tag{7}$$

where $u(x,n) \in [0,1)$. A pixel renders if $u(x,n) < \rho$, where again ρ denotes a user-defined render fraction. Over N frames, each pixel cycles through the STBN sequence, producing approximately ρN render events per cycle with blue-noise spatial distribution.

Rendered pixels update a running average, while skipped pixels reuse the accumulated value. Because it is active from the first frame, it avoids the warm-up and stale-signal issues of residual-based schemes. The trade-off is that the budget is distributed independently of image instability, so updates are not concentrated on regions that would benefit most.

Residual-Weighted STBN. This scheme combines STBN-based allocation with residual-driven steering by modulating each pixel's render probability according to its residual. Instead of a fixed fraction ρ, each pixel computes an effective render probability:

$$\rho_i^{\text{eff}} = \text{clamp}\big(\rho + \alpha(r_i - \rho), \tfrac{2}{256}, 1\big). \tag{8}$$

Here $\alpha \in [0,1]$ controls the strength of residual steering. When $\alpha = 0$, the scheme reduces to pure STBN; as α increases, more budget is directed toward unstable pixels, while the blue-noise mask still regularizes the spatial update pattern. The floor at $2/256$ prevents permanent starvation.

A ray is cast when $u_i < \rho_i^{\text{eff}}$. Because decisions remain tied to the blue-noise texture, the pattern retains blue-noise characteristics for small α, while larger α shifts it toward a more residual-driven allocation. The scheme thus offers an adjustable middle ground between stochastic allocation and fully residual-driven budgeting.

Residual reprojection. Camera motion resets the accumulation buffer, leaving the residual signal r_i undefined. To preserve informed budgeting after interaction, we reproject residuals using stored world-space hit positions [38, 66], alongside pixel-value reprojection. Valid reprojected hits are used to set the histogram and skip state, while invalid or occluded pixels default to high residual and are re-rendered.

This mainly benefits the residual-driven schemes by restoring adaptive budgets immediately after a view change; pure STBN thresholding does not benefit because its decisions do not depend on residuals, but that also means the pure STBN mode does not lose performance when r_i is undefined. The approximation degrades under large camera rotations, transfer-function edits, or timestep changes, so reprojected budgets should be viewed as heuristics rather than exact estimates under the new view.

4.2.2 Homogeneity-Based Query Pruning

If a macrocell's scalar range is sufficiently small, the INR queries could be avoided, as these macrocells will mostly behave as a uniform block. We exploit this observation to skip the INR query entirely for such cells, replacing it with an expected scalar $\hat{s}_c$.

Uniformity criterion. Each macrocell c stores a precomputed scalar range $\Delta_c = s_c^{\max} - s_c^{\min}$ from its min–max bounds. Given a user-controlled threshold ε, a candidate collision point in cell c is pruned (i.e., answered without an INR query) whenever $\Delta_c < \varepsilon$. To soften the transition between pruned and evaluated cells, a stochastic falloff band of width δ linearly ramps the pruning probability from 1 to 0 over the interval $[\varepsilon, \varepsilon+\delta]$:

$$p_{\text{skip}} = \begin{cases} 1 & \Delta_c < \varepsilon, \\ 1 - \frac{\Delta_c - \varepsilon}{\delta} & \varepsilon \le \Delta_c < \varepsilon + \delta, \\ 0 & \Delta_c \ge \varepsilon + \delta. \end{cases} \tag{9}$$

Because the substituted scalar is a per-cell constant rather than the true field value at the sample point, this approximation introduces bias: the collision test sees the cell's expected value instead of the local densities, which can produce blocky artifacts at cell boundaries, particularly when the user-controlled threshold ε is set aggressively. Figure 5 illustrates the effect of these user-controlled parameters on an example volume.

The stochastic falloff band mitigates hard transitions but does not eliminate the underlying bias. In practice, this technique is best applied conservatively to cells with very small scalar ranges, where the substitution error is negligible relative to the transfer-function response. Notably, exploiting this fact makes sense only when sampling volumes is expensive, as is the case with the INRs.

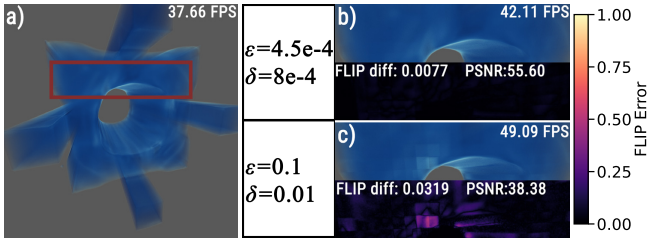

Figure 5: Homogeneity-based query pruning on the S05_700 dataset. a) Ground truth without pruning. b) Crop (red box in a) using low threshold and falloff width. c) Same crop with a higher threshold and falloff width. FLIP and PSNR differences are reported relative to the full ground truth image.

Expected-value refinement. A naïve choice for $\hat{s}_c$ is the midpoint $(s_c^{\min} + s_c^{\max})/2$. However, by introducing another float per macrocell, we can refine this value. The idea is to maintain a running estimate $\hat{s}_c$ that is refined at runtime via an exponential moving average (EMA). Whenever a real INR evaluation returns a scalar s inside cell c, the estimate is updated as

$$\hat{s}_c \leftarrow \hat{s}_c + \alpha(s - \hat{s}_c), \tag{10}$$

where $\alpha = 0.1 \max(w_c, 0.01)$ and $w_c \in [0,1]$ is a center weight that attenuates updates from samples near cell boundaries, which may not be representative of the cell interior. The initial estimate is set to the mean of the collected samples used during macrocell construction. The EMA refinement reduces the magnitude of the substitution error over successive frames, but does not eliminate it, as the pruned scalar remains spatially constant within each cell. While the stochastic falloff has a limited impact on image quality when the threshold ε is present, it is necessary to ensure sufficient sampling coverage for stable EMA updates. In our experiments, $\delta = 0.006$ provides a robust balance across all tested datasets.

5 Results and Discussion

We evaluate the memory overhead and end-to-end performance of our system without query reduction, and then assess the trade-offs of our query-reduction strategies. As datasets, we use six INRs from three architectures, each mapping spatiotemporal coordinates (x,y,z,t) to a single scalar field value. Three INRs from Mohan et al. [32, 47] use a Fourier-feature network (FFN), which encodes coordinates with Fourier features before a six-layer fully connected network with Swish activations; they are reconstructed from dynamic X-ray CT scans of deforming objects and represent energy-averaged linear attenuation coefficients. These datasets follow the *S0X_XXX* naming convention. Two are sinusoidal representation networks (SIRENs) [56] generated from CFD time series [3, 46] produced with the Gerris Flow Solver [45]; they use five width-256 sine layers followed by a linear layer and represent simulated flow quantities over time. These are named *cylinder* and *tangaroa*. The last is a coordinate-based network (CoordNet) [17], a residual sinusoidal MLP that grows the input to width 192 over three residual blocks, applies six width-192 residual blocks, and ends with a residual block projecting to the scalar output; each block sums two ω_0=30 sine layers through a skip connection. It represents a vortical flow field and is named *vorts*.

All experiments run at 1024^2 resolution on an NVIDIA RTX 4090 using CUDA 13 [40] and *OptiX* 9 [41]. Unless stated otherwise, results use 400-frame converged renderings.

5.1 Data Structure Sizes and Updates

We measure the memory footprint and update cost of the acceleration structures used for ray tracing and delta tracking.

One of the many advantages of an INR is its compact size, and the same neural architecture occupies the same amount of memory—unlike adaptive formats like unstructured meshes. This is the case for our datasets; each occupies 1–2 MB of GPU memory in isolation. In Table 1, we chose to demonstrate two 4D macrocell grid sizes for three different INR architectures that also occupy constant memory. From those macrocell grids, we build a working grid: the interpolated and corrected

Table 1: Data structure sizes for three example datasets.* indicates an average over 10 evenly spaced timesteps, as *OptiX* GAS size depends on transfer function and timestep due to empty-cell culling.

data & macrocell res. \ Size (MB)		INR	macrocell grid	working grid	GAS	Total
S03_001 (FFN)	$64^3 \times 20$	1.26	40.0	3.0	0.49*	44.76
	$128^3 \times 20$		320.0	24.0	3.59*	347.86
Tangaroa (SIREN)	$64^3 \times 30$	1.01	60.0	3.0	0.21*	64.22
	$128^3 \times 30$		480.0	24.0	1.24*	506.25
Vorts (CoordNet)	$64^3 \times 20$	2.06	40.0	3.0	4.38*	49.44
	$128^3 \times 20$		320.0	24.0	35.15*	381.21

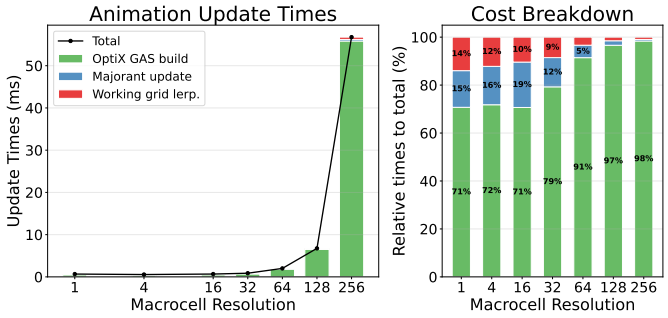

Figure 6: Data structure update time (lower is better) versus macrocell grid resolution. Times are averaged over 10 timestep interactions for S03_001 and S04_018. Left: absolute timings per update stage; right: timings normalized by total update time. The x-axis denotes per-dimension resolution, i.e., value$^3 \times T$ with $T = 20$.

3D macrocell grid for the current timestep, along with the 3D majorant grid—again, much smaller, constant-size. Finally, a sparse *OptiX* acceleration structure is built over those majorants, which is the only variable component of this memory usage. One thing we do not depict in this table is the INR query and `TraversalState`, yet their sizes are again trivial to calculate, since one of each is allocated per pixel (shadow and primary rays share the `TraversalState`). At 1024^2 resolution, the `TraversalState` buffer occupies 42 MB, and query buffers collectively consume 20 MB.

During animation, changing t requires updating the macrocell-derived structures. Figure 6 reports these costs as macrocell resolution increases; finer grids increase the number of cells and therefore the cost of interpolation, updates, and construction. Transfer-function edits follow the same update path, except that working-grid interpolation is skipped because cell extrema are unchanged.

As Table 1 shows, with a smaller macrocell resolution, rendering-related structures are much larger than the INR's base size. Even with the relatively larger, $128^3 \times 20$, grid, the acceleration structure memory remains within a 40-400 MB range. For the smaller macrocell grid resolution, measurements obtained using `nvidia-smi` indicate a total GPU memory usage of approximately 600 MB. The application explicitly accounts for ∼150 MB, including the trivial allocations for frame and accumulation buffers. We attribute the remaining memory to *OptiX* and CUDA contexts, and their associated programs. This is substantially lower than typical scientific visualization systems [1, 7, 60].

Despite the *OptiX* acceleration structure build time dominating the animation updates, the total update time falls under 55 milliseconds (ms). With smaller macrocell grid resolutions, this time decreases to ∼2 ms. As a result, users can animate the continuous temporal deformations of the INR in real time.

5.2 Framework Performance

We measure and compare the base efficacy of our framework without the query reduction strategies. In Figure 7, we show various volume rendering implementations rendering the same scenes over six datasets. For each method, the frames per second (FPS) performance is measured against increasing macrocell grid resolution, as shown in Figure 6.

To independently assess which components and hardware resources contribute to our framework's performance, we evaluate four GPU-based renderer implementations: (1) a simple ray marcher; (2) a naïve delta tracker based on implementations designed for conventional explicit data formats (e.g., voxels and meshes); (3) a wavefront delta tracker operating in the same four stages and using tensor cores, but without RT-core-based traversal (instead using 3D DDA); (4) our fully realized four stage framework, which utilizes RT cores for traversal (see Figure 7). The results draw a consistent picture across different INR architectures, data shapes, and physical quantities represented, including CT- and CFD-derived scalars.

Ray marching is by far the slowest approach; even with sub-Shannon–Nyquist step sizes, it fails to exceed 1 FPS. As discussed earlier, such dense sampling strategies require a prohibitively large number of samples to achieve interactive performance on INR data. Since ray marching does not operate over macrocells, it is depicted as flat lines in Figure 7.

A simple delta tracker without specialized hardware or query batching also performs below 1 FPS. Designed for conventional volume rendering with fast memory lookups, this approach underperforms when each sample instead requires a GPU thread to execute a six-layer neural inference independently per ray, failing to amortize the cost of neural evaluation across samples and leading to significant thread divergence and inefficient parallelism for its most expensive procedure.

In contrast, our four-stage framework applies more suitable parallelization for neural evaluation and leverages tensor cores, enabling delta tracking over INRs to reach approximately ∼25 FPS, making it ∼125× faster than the simple delta tracker.

Furthermore, replacing DDA traversal with RT-core-based traversal yields more than a 2× performance improvement for most tests, achieving over 40 FPS for emission–absorption rendering and 30–40 FPS with shadows, which require approximately twice as many samples.

Across these results, CoordNet is the most challenging representation for our framework, reaching only 19–24 FPS compared to roughly 30–40 FPS for the FFN and SIREN models. This stems from CoordNet's deeper residual MLP, which introduces longer per-query dependency chains, reducing effective tensor-core and warp occupancy on the hardware.

Despite INR queries being the dominant bottleneck, using RT cores for traversal still provides a significant speed-up. We attribute this to two factors: (1) the sparse acceleration structure enables more effective empty-space skipping, with $O(n \log n)$ traversal rather than DDA's $O(\text{cells-along-ray})$; and (2) DDA traversal must be suspended and resumed across kernel launches, requiring a larger `TraversalState` that persists per-ray DDA stepping state (current cell, axis distances, cell boundaries), increasing register pressure and memory traffic.

Table 1, Figure 6, and Figure 7 suggest that a macrocell grid resolution of 64^3–128^3 provides a reasonable trade-off between end-to-end performance, memory consumption, and animation updates. For the following experiments, we use $64^3 \times 20$ for the FFN datasets and $64^3 \times 30$ for the SIREN datasets, using a slightly finer temporal discretization for the latter to better match their higher temporal variation while keeping the spatial resolution fixed.

5.3 Query Reduction Tradeoffs

We examine the trade-offs of the query-reduction schemes proposed in subsubsection 4.2.1 and subsubsection 4.2.2. We explore the parameter space and the practicality of each scheme.

In Figure 8, we evaluate three ray-budgeting strategies against a 400-frame ground truth image. Each plot shows Peak Signal-to-Noise Ratio (PSNR) relative to the ground truth over time for 3–6 representative parameter settings, while the heatmaps visualize the average number of rays cast per pixel, and the close-ups highlight early-frame noise patterns at ∼1.2 seconds.

Early plot points of the residual-histogram thresholding scheme exhibit closely clustered trends across parameter settings. Because it ranks pixels by residual and skips stable ones for multiple frames, it concentrates rays most aggressively on unstable regions, as reflected in the heatmaps. This yields lower error than the no-budget baseline in the first few seconds, but after approximately 3 seconds, the benefit diminishes as repeatedly skipped pixels reduce opportunities for further error correction. The initial frames retain the "white-noise" appearance of standard stochastic rendering.

In contrast, STBN thresholding does not use residuals and distributes a fixed ray budget according to a high-frequency noise pattern. Its cost-effectiveness, therefore, depends mainly on the render fraction (ρ), trading reduced ray counts for increased error. Higher ρ reduces early error,

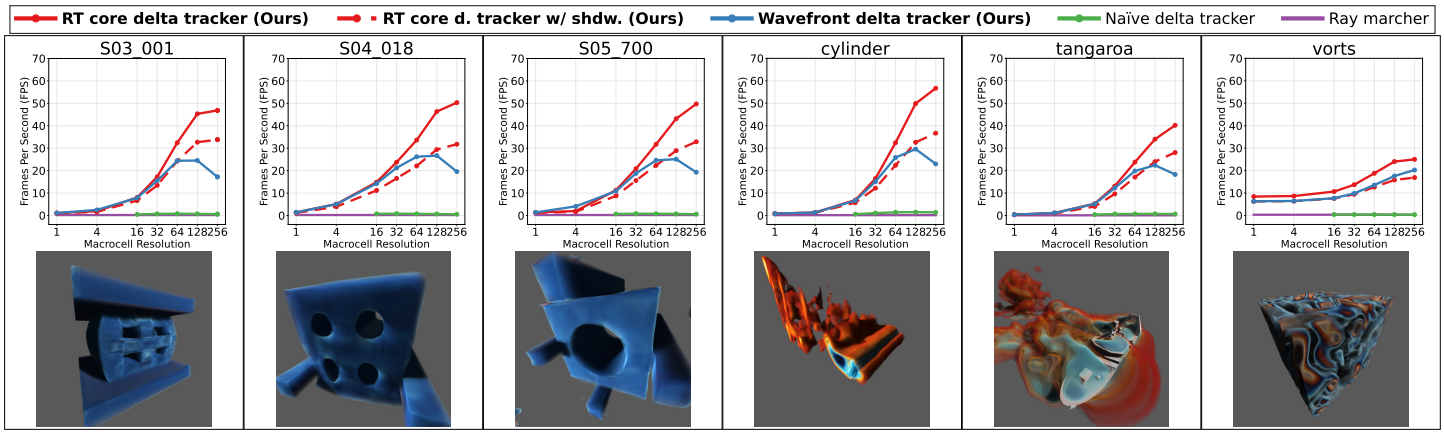

Figure 7: Frames per second performance (higher is better) of various volume rendering methods against increasing macrocell grid resolution for six INR datasets. The methods are ray marching (which does not use macrocells and therefore remains nearly constant at ~ 0.2 FPS), a naïve delta tracker (closely following ray marcher), wavefront delta tracker (ours), and ray tracing (RT) core accelerated wavefront delta tracker (ours). We present the performance of the RT core wavefront d. tracker with only emission+absorption and ray-traced shadows (as "shdw."). The 400-frame-converged images for each dataset are given below the plots.

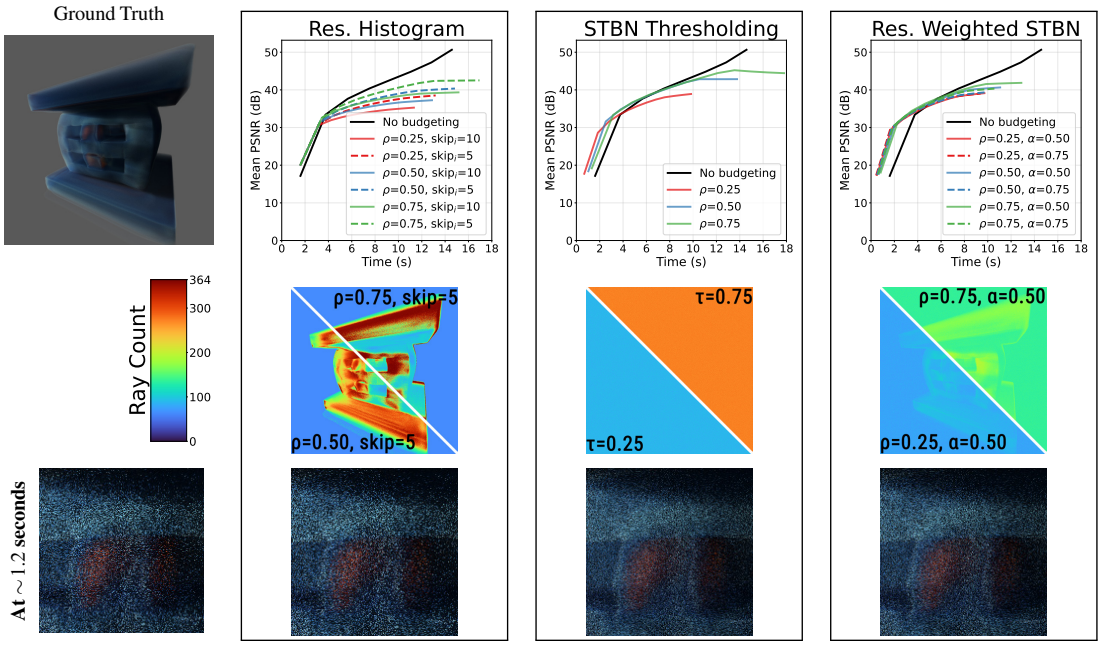

Figure 8: Evaluation of ray budgeting strategies. The top row shows image quality (higher is better) as a function of elapsed time (seconds) for the three schemes described in subsubsection 4.2.1 for various parameter configurations. Experiments are conducted on the S03_001 dataset with a $64^3 \times 20$ macrocell grid resolution. Image quality is measured using the Peak Signal-to-Noise Ratio (PSNR) and the mean difference relative to a 400-frame ground-truth rendering without budgeting. The second row presents heatmaps of the average number of rays cast per pixel for two representative parameter settings per strategy. The final row shows close-ups of unconverged images at ~ 1.2 seconds, highlighting noise patterns (using the same parameters as the top-right heatmaps in the second row).

while lower ρ saves more rays and increases effective frame rate by rendering only a fraction of pixels per frame. The curves reconverge after approximately 5 seconds as the image stabilizes. Among the tested settings, $\rho = 0.75$ provides the best trade-off, maintaining effective amortization up to ~ 6 seconds. Additionally, early frames perceptually benefit from the high-frequency structure of blue noise, as shown in the bottom row.

In the hybrid method (residual-weighted STBN), the base render fraction still controls the overall cost, while the residual weighting redistributes a portion of that budget toward unstable pixels. The resulting heatmaps show a concentration of rays on changing structures while retaining a well-distributed high-frequency pattern, making the scheme a more cost-effective compromise between pure STBN and residual-histogram thresholding. Its curves remain closely clustered, indicating the method is relatively insensitive to parameter choice. We observe it reaching less than 4% error slightly faster than pure STBN in the first two seconds, while preserving similar early-frame noise characteristics.

The first set of frames is particularly relevant for interactive use, where rapid visual feedback influences user decisions. In this regime, STBN-based methods offer the clearest practical advantage, as blue-noise sub-sampling produces more favorable sparse-update patterns. Residual-weighted STBN is best viewed as a modest refinement: steering part of the budget toward unstable pixels makes it slightly more cost-effective in the first few seconds, but the gain over pure STBN is small and fades as the image stabilizes. The fully residual-driven histogram scheme is more aggressive, but on this dataset, its benefit is short-lived, and its early-frame noise patterns resemble "white-noise," making it less attractive for perceptual quality. Nevertheless, it remains useful as a simple adaptive baseline and as an incremental step toward the hybrid methods, suggesting that residual guidance is better realized when combined with blue-noise allocation than when used alone. Residuals also remain valuable because they can be reprojected across frames during interaction, a benefit not captured by these static-view plots. All strategies produce images that converge to within 2% of the ground truth.

Homogeneity-based query pruning trades image fidelity for FPS through ε as seen in Figure 9. Larger values skip more homogeneous macrocells, but replace true samples with per-cell estimates. The

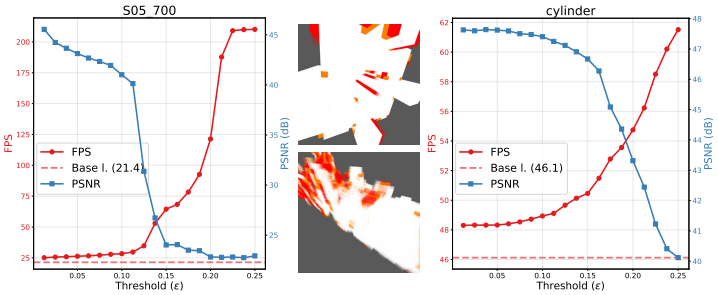

Figure 9: Homogeneity-based query pruning on S05_700 and cylinder data. Dual-axis plots show performance (FPS, red) and image quality (Peak Signal-to-Noise Ratio, PSNR) versus increasing ε from subsubsection 4.2.2; higher is better on both y-axes. Middle visualizations mark pruned cells in red and stochastically pruned cells in orange for S05_700 (top) and cylinder (bottom).

main caveat we observe is a dataset-dependent critical threshold, ε_c, after which quality drops sharply. We find $\varepsilon_c = 0.12$ for S05_700 and $\varepsilon_c = 0.16$ for cylinder. At this point, the pruned fraction catches up with the empty-space fraction, already flattening $\sim$58% of S05_700's and $\sim$33% of cylinder's visible cells. Further pruning then erodes the more heterogeneous macrocells, and rapidly degrades the image quality.

The degradation in quality is gradual at first, indicating that small ε values successfully prune near-uniform regions with minimal visual impact. This effect is particularly beneficial in scientific datasets, which often contain extended low-variance regions or zero-valued outer layers, allowing rays to traverse large portions of the volume without incurring additional INR queries. Beyond a critical threshold ($\varepsilon \approx 0.15$), however, the error increases more noticeably as larger and more structurally significant regions are approximated. The visualizations on the right illustrate this behavior: deterministically pruned cells (red) and stochastically pruned regions (orange) concentrate in low-variance areas, while higher ε values begin to affect perceptually important structures. In practice, moderate thresholds strike a favorable balance, while larger thresholds may be more cost-effective when rendering more expensive neural models or when the rendering task is more demanding, such as path tracing.

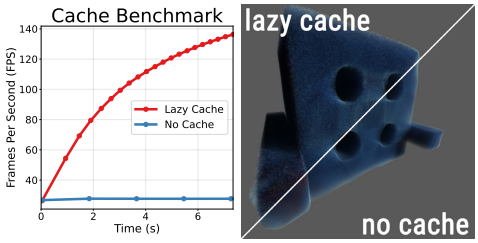

Figure 10: Frames per second performance over time with and without a lazy caching scheme (500 MB cache). The corresponding renderings after 400 frames are shown on the right.

A vital discussion concerns the use of cache-based acceleration (such as [69]) as a complementary technique rather than an alternative to our framework. As a proof of concept, we implement a simple *lazy caching* scheme on top of our pipeline; as shown in Figure 10, it already provides performance gains that accumulate over time at a fixed timestep.

However, the temporal limitations noted in section 4 still apply: repeated navigation through time can invalidate cached samples and reduce their reuse for time-varying INRs. Moreover, caching introduces a spatial approximation trade-off, since finite cache resolution can bias the reconstruction. As visible in Figure 10, artifacts remain even with a 500 MB cache (about 397$\times$ the INR size). At that point, one could also consider other adaptive surrogate representations such as PruningAMR [72]. Thus, caching can be beneficial under sufficient coherence, but it does not fully resolve the challenges of interactive INR rendering.

6 Limitations and Future Work

Our current implementation is tightly coupled to NVIDIA's CUDA, *OptiX*, and RT-core stack. While the pipeline is portable in principle, achieving comparable performance on other vendors would require careful mapping to their ray tracing and matrix-compute units; in such cases, the wavefront delta tracker in Figure 7 provides a viable alternative and starting point in the absence of RT cores. Our proof-of-concept uses a thin, templated CUDA evaluator with the INR architecture explicitly implemented; following this work with additional architectures would require adding the corresponding layers or relying on a library [37].

A fundamental limitation lies in the construction of macrocells. Extrema are approximated via lattice sampling and temporal interpolation, which can misestimate extrema and lead to loose majorants, increasing null collisions, or producing structured artifacts (e.g., block-like regions as seen in Figure 11). These temporal artifacts occur when temporal macrocell bounds are too coarse or insufficiently refined, which is especially relevant for INRs with high temporal variability, such as those for CFD simulations. Ghost passes mitigate underestimation but do not guarantee uniform refinement. In addition, both ray budgeting and homogeneity-based pruning are heuristics that rely on stable residuals and well-behaved scalar ranges; during interactions or when aggressive thresholds are used, homogeneity-based pruning may introduce bias. Finally, INRs tend to favor low-frequency structure, which can produce smooth or "network-shaped" artifacts independent of the rendering method and reflects a broader limitation of the representation.

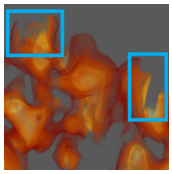

Figure 11: Artifacts in the cylinder data caused by a reduced temporal macrocell count ($T = 5$) and no ghost passes.

Future work could address both representation and rendering efficiency. On the representation side, tighter macrocell bounds, improved temporal interpolation, or learned bound estimation could reduce underestimation and null collisions. In particular, the INR training process could expose additional structure, such as conservative sub-region bounds via interval networks or auxiliary predictors for local extrema, enabling more accurate majorants. On the rendering side, replacing heuristic query-reduction schemes with learned or temporally coherent strategies could improve robustness under interaction, provided they remain lightweight. Gradient-based shading could also be incorporated by estimating local gradients using the central-difference method [6]. Additionally, neurally predicting candidate-hit regions to guide sampling could further reduce unnecessary INR evaluations. Finally, testing our framework on physics-informed neural networks (PINNs) is a promising direction for future work.

7 Conclusion

We presented a query-efficient stochastic volume rendering framework for time-varying implicit neural representations (INRs), enabling interactive visualization and low-latency exploration of the continuous time domain without requiring resampling, retraining, or caching as a prerequisite. By reinterpreting delta tracking as a query-reduction mechanism and combining it with a four-stage ray tracing pipeline, our method decouples traversal from neural evaluation and exploits heterogeneous GPU hardware, including ray tracing and tensor cores. This design minimizes and amortizes neural inference costs while maintaining estimator correctness under practical approximations, enabling direct rendering and interactive animation of continuous time-varying INRs. Our results show that the framework achieves interactive performance ($\sim$30–40 FPS at 1024^2 in most benchmarks) with ray-traced shadows while supporting animation updates within a few milliseconds.

We further demonstrated query reduction can be introduced as a cost-effective extension to our framework in two complementary ways. Adaptive ray budgeting reduces image-level costs by selectively casting rays to pixels that benefit most from new samples, exposing trade-offs among convergence behavior, spatial adaptivity, and perceptual noise characteristics. Homogeneity-based query pruning reduces cost at the sample level by avoiding unnecessary INR evaluations in near-uniform regions, providing a trade-off between rendering performance and approximation error. Used conservatively, both techniques improve efficiency with limited impact on image quality, while STBN-based and hybrid budgeting schemes further improve the perceptual quality of early frames.

Ultimately, these results show that time-varying INRs can be rendered directly and explored continuously without intermediate resampling or surrogate structures. We believe this framework broadens the practical use of neural representations in scientific visualization and enables richer exploration and higher-quality rendering of dynamic volumetric data.

Acknowledgments

This research was supported in part by NSF OAC award 2138811, 2609465 (58503144, 58503145, and 58503708), NSF OISE award 2330582 (58503534), and NASA JPL Subcontract No. 56000334. This research was also supported by the U.S. Department of Energy (DOE) under grants DE-SC-0023320 and DE-SC-0023319. This work was further supported in part by the U.S. DOE Lawrence Livermore National Laboratory Laboratory Directed Research and Development program under grant LLNL-LDRD 25-FS-002, and was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-2017782).

This work used Distributed Implicit Neural Representation (DINR), v1.0, licensed from Lawrence Livermore National Security, LLC.

Generative AI tools were used in this work in a limited capacity, including debugging, script generation for benchmarking and plotting, and grammar editing and trimming of human-written text.

The code for this work is available at `github.com/alpers-git/VIZ-INR`.

References

[1] J. Ahrens, B. Geveci, and C. Law. ParaView: An end-user tool for large-data visualization. In C. D. Hansen and C. R. Johnson, eds., *Visualization Handbook*. 2005. 5, 7

[2] P. Andersson, J. Nilsson, and T. Akenine-Möller. Visualizing and Communicating Errors in Rendered Images. In A. Marrs, P. Shirley, and I. Wald, eds., *Ray Tracing Gems II*, chap. 19, pp. 301–320. 2021. 4

[3] I. Baeza Rojo and T. Günther. Vector field topology of time-dependent flows in a steady reference frame. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE Scientific Visualization)*, 2019. 6

[4] D. Bauer, Q. Wu, H. Gadirov, and K.-L. Ma. Gscache: Real-time radiance caching for volume path tracing using 3d gaussian splatting. *arXiv preprint arXiv:2507.19718*, 2025. 2

[5] D. Bauer, Q. Wu, and K.-L. Ma. FoVolNet: Fast Volume Rendering using Foveated Deep Neural Networks . *IEEE Transactions on Visualization & Computer Graphics*, 29(01):515–525, 2023. doi: 10.1109/TVCG.2022.3209498 2

[6] C. Brownlee and D. Demarle. Fast volumetric gradient shading approximations for scientific ray tracing. In A. Marrs, P. Shirley, and I. Wald, eds., *Ray Tracing Gems II*. Apress Berkeley, CA, 2021. doi: 10.1007/978-1-4842-7185-8 9

[7] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire et al. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data, 2012. 7

[8] S. Di and F. Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 ieee international parallel and distributed processing symposium (ipdps)*, pp. 730–739. IEEE, 2016. 2

[9] L. Dyken, A. Sewell, W. Usher, N. Debardeleben, S. Petruzza, and S. Kumar. Volume encoding gaussians: Transfer function-agnostic 3d gaussians for volume rendering. *arXiv preprint arXiv:2504.13339*, 2025. 2

[10] J. Fong, M. Wrenninge, C. Kulla, and R. Habel. Production Volume Rendering: SIGGRAPH 2017 Course. In *ACM SIGGRAPH 2017 Courses*, SIGGRAPH '17, 2017. doi: 10.1145/3084873.3084907 2

[11] I. Georgiev and M. Fajardo. Blue-noise dithered sampling. In *ACM SIGGRAPH 2016 Talks*, SIGGRAPH '16, art. no. 35, 1 p. Association for Computing Machinery, 2016. doi: 10.1145/2897839.2927430 2

[12] B. Guenter, M. Finch, S. Drucker, D. Tan, and J. Snyder. Foveated 3d graphics. *ACM Trans. Graph.*, 31(6), art. no. 164, 2012. doi: 10.1145/2366145.2366183 2

[13] T. Günther, A. Kuhn, and H. Theisel. MCFTLE: Monte Carlo Rendering of Finite-Time Lyapunov Exponent Fields. *Computer Graphics Forum*, 35, 2016. doi: 10.1111/cgf.12914 1, 2

[14] T. Hachisuka, W. Jarosz, R. P. Weistroffer, K. Dale, G. Humphreys, M. Zwicker et al. Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 27(3):33:1–33:10, 2008. doi: 10/fm6c2w 5

[15] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum*, 24, 2005. doi: 10.1111/j.1467-8659.2005.00855.x 2

[16] J. Han, K. Tang, and C. Wang. Moe-inr: Implicit neural representation with mixture-of-experts for time-varying volumetric data compression. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–11, 2025. doi: 10.1109/TVCG.2025.3633893 2

[17] J. Han and C. Wang. Coordnet: Data generation and visualization generation for time-varying volumes via a coordinate-based neural network. *IEEE Transactions on Visualization and Computer Graphics*, 29(12):4951–4963, 2022. doi: 10.1109/TVCG.2022.3197203 2, 6

[18] J. Han and F. Yang. Dcinr: a divide-and-conquer implicit neural representation for compressing time-varying volumetric data in hours. *IEEE Transactions on Visualization and Computer Graphics*, 2025. doi: 10.1109/TVCG.2025.3564255 2

[19] M. Han, A. Sewell, J. Insley, J. Knowles, V. A. Mateevitsi, M. E. Papka et al. Toward distributed 3d gaussian splatting for high-resolution isosurface visualization. In *2025 IEEE International Conference on eScience (eScience)*, pp. 331–332. IEEE, 2025. 2

[20] N. Hofmann, J. Martschinke, K. Engel, and M. Stamminger. Neural Denoising for Path Tracing of Medical Volumetric Data. *ACM Transactions on Graphics (Proceedings of SIGGRAPH '20)*, 3(2), 2020. doi: 10.1145/3406181 1, 2

[21] M. Ikits, J. Kniss, A. Lefohn, and C. Hansen. Volume Rendering Techniques. In *GPU Gems*. 2004. Available at `https://developer.nvidia.com/sites/all/modules/custom/gpugems/books/GPUGems/gpugems_ch39.html`, Accessed: 24 June 2022. 3

[22] B. Kerbl, G. Kopanas, T. Leimkühler, G. Drettakis, et al. 3d gaussian splatting for real-time radiance field rendering. 2023. 2

[23] M. Koskela, K. Immonen, T. Viitanen, P. Jääskeläinen, J. Multanen, and J. Takala. Foveated instant preview for progressive rendering. In *SIGGRAPH Asia 2017 Technical Briefs*, art. no. 10. Association for Computing Machinery, 2017. doi: 10.1145/3145749.3149423 2

[24] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization*, IEEE Visualization Conference, 2003. doi: 10.1109/VISUAL.2003.1250384 2

[25] P. Kutz, R. Habel, Y. K. Li, and J. Novák. Spectral and decomposition tracking for rendering heterogeneous volumes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2017)*, 36(4), 2017. doi: 10.1145/3072959.3073665 2

[26] S. Laine, T. Karras, and T. Aila. Megakernels considered harmful: wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pp. 137–143. Association for Computing Machinery, 2013. doi: 10.1145/2492045.2492060 3

[27] P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014. doi: 10.1109/TVCG.2014.2346458 2

[28] P. Ljung, C. Lundström, and A. Ynnerman. Multiresolution Interblock Interpolation in Direct Volume Rendering. In *EUROVIS - Eurographics/IEEE VGTC Symposium on Visualization*, 2006. doi: 10.2312/VisSym/EuroVis06/259-266 2

[29] Y. Lu, K. Jiang, J. A. Levine, and M. Berger. Compressive neural representations of volumetric scalar fields. *Computer Graphics Forum*, 40(3):135–146, 6 2021. doi: 10.1111/cgf.14295 2

[30] J. Martschinke, S. Hartnagel, B. Keinert, K. Engel, and M. Stamminger. Adaptive Temporal Sampling for Volumetric Path Tracing of Medical Data. *Computer Graphics Forum*, 2019. doi: 10.1111/cgf.13771 1, 2

[31] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: representing scenes as neural radiance fields for view synthesis. *Commun. ACM*, 65(1):99–106, 2021. doi: 10.1145/3503250 2

[32] K. A. Mohan, M. Ferrucci, C. Divin, G. A. Stevenson, and H. Kim. Distributed stochastic optimization of a neural representation network for time-space tomography reconstruction. *IEEE Transactions on Computational Imaging*, 11:362–376, 2025. doi: 10.1109/TCI.2025.3547265 1, 6

[33] N. Morrical, A. Sahistan, U. Güdükbay, I. Wald, and V. Pascucci. Quick Clusters: A GPU-Parallel Partitioning for Efficient Path Tracing of Unstructured Volumetric Grids. *IEEE Transactions on Visualization and Computer Graphics*, 29(01), 2023. doi: 10.1109/TVCG.2022.3209418 2, 3

[34] N. Morrical, W. Usher, I. Wald, and V. Pascucci. Efficient Space Skipping and Adaptive Sampling of Unstructured Volumes Using Hardware Accelerated Ray Tracing. In *2019 IEEE Visualization Conference*, 2019. doi: 10.1109/VISUAL.2019.8933539 2

[35] N. Morrical, I. Wald, W. Usher, and V. Pascucci. Accelerating unstructured mesh point location with RT cores. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–14, In Press. doi: 10.1109/TVCG.2020.3042930 2

[36] N. Morrical, S. Zellmann, A. Sahistan, P. Shriwise, and V. Pascucci. Attribute-Aware RBFs: Interactive Visualization of Time Series Particle Volumes Using RT Core Range Queries. *IEEE Transactions on Visualization and Computer Graphics*, 30(1), 2023. doi: 10.1109/TVCG.2023.3327366 2

[37] T. Müller. tiny-cuda-nn. `https://github.com/NVlabs/tiny-cuda-nn`, 2021. Version 2.0, released April 21, 2021. 9

[38] D. Nehab, P. V. Sander, J. Lawrence, N. Tatarchuk, and J. R. Isidoro. Accelerating real-time shading with reverse reprojection caching. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pp. 25–35. Eurographics Association, 2007. 6

[39] J. Novák, A. Selle, and W. Jarosz. Residual ratio tracking for estimating attenuation in participating media. *ACM Trans. Graph.*, 33(6), 2014. doi: 10.1145/2661229.2661292 2

[40] NVIDIA, P. Vingelmann, and F. H. Fitzek. CUDA release: 13. 6

[41] NVIDIA Corporation. NVIDIA OptiX Ray Tracing Engine. Available at https://developer.nvidia.com/optix, Accessed: 27 March 2026. 6

[42] A. Patney, M. Salvi, J. Kim, A. Kaplanyan, C. Wyman, N. Benty et al. Towards foveated rendering for gaze-tracked virtual reality. *ACM Trans. Graph.*, 35(6), art. no. 179, 2016. doi: 10.1145/2980179.2980246 2

[43] K. Perlin and E. M. Hoffert. Hypertexture. *SIGGRAPH Comput. Graph.*, 23(3), 1989. doi: 10.1145/74334.74359 2

[44] M. Pharr, W. Jakob, and G. Humphreys. 11 Volume Scattering. In *Physically based rendering: From theory to implementation*, pp. 697–736. The MIT Press, 4 ed., 2023. 2

[45] S. Popinet. Free computational fluid dynamics. *ClusterWorld*, 2(6), 2004. 6

[46] S. Popinet, M. Smith, and C. Stevens. Experimental and numerical study of the turbulence characteristics of airflow around a research vessel. *Journal of Atmospheric and Oceanic Technology*, 21(10):1575–1589, 2004. doi: 10.1175/1520-0426(2004)021<1575:EANSOT>2.0.CO;2 6

[47] A. W. Reed, H. Kim, R. Anirudh, K. A. Mohan, K. Champley, J. Kang et al. Dynamic ct reconstruction from limited views with implicit neural representations and parametric motion fields, 2021. doi: 10.1109/ICCV48922.2021.00226 6

[48] C. Reiser, S. Peng, Y. Liao, and A. Geiger. Kilonerf: Speeding up neural radiance fields with thousands of tiny MLPs. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 14335–14345, 2021. doi: 10.1109/ICCV48922.2021.01407 2

[49] C. Reiser, R. Szeliski, D. Verbin, P. Srinivasan, B. Mildenhall, A. Geiger et al. Merf: Memory-efficient radiance fields for real-time view synthesis in unbounded scenes. *ACM Transactions on Graphics (ToG)*, 42(4):1–12, 2023. doi: 10.1145/3592426 2

[50] F. Rousselle, C. Knaus, and M. Zwicker. Adaptive sampling and reconstruction using greedy error minimization. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, art. no. 159, 12 pp. Association for Computing Machinery, 2011. doi: 10.1145/2024156.2024193 5

[51] S. Röttger, S. Guthe, D. Weiskopf, T. Ertl, and W. Straßer. Smart Hardware-Accelerated Volume Rendering. In *Proceedings of the Symposium on Data Visualisation 2003*, vol. 3 of *VISSYM '03*, 2003. 2

[52] A. Sahistan, S. Demirci, I. Wald, S. Zellmann, J. Barbosa, N. Morrical et al. Visualization of large non-trivially partitioned unstructured data with native distribution on high-performance computing systems. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–14, 2024. doi: 10.1109/TVCG.2024.3427335 2

[53] A. Sahistan, S. Zellmann, H. Miao, N. Morrical, I. Wald, and V. Pascucci. Materializing inter-channel relationships with multi-density woodcock tracking. *IEEE Transactions on Visualization and Computer Graphics*, 32(3):2726–2740, 2026. doi: 10.1109/TVCG.2026.3653310 2

[54] A. Sahistan, S. Zellmann, N. Morrical, V. Pascucci, and I. Wald. Multi-Density Woodcock Tracking: Efficient & High-Quality Rendering for Multi-Channel Volumes. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2025. doi: 10.2312/pgv.20251150 2

[55] V. Saragadam, J. Tan, G. Balakrishnan, R. G. Baraniuk, and A. Veeraraghavan. Miner: Multiscale implicit neural representation. In *European Conference on Computer Vision*, pp. 318–333. Springer, 2022. doi: 10.1007/978-3-031-20050-2_19 2

[56] V. Sitzmann, J. N. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein. Implicit neural representations with periodic activation functions. In *Proc. NeurIPS*, 2020. 6

[57] L. Szirmay-Kalos, B. Tóth, and M. Magdics. Free Path Sampling in High Resolution Inhomogeneous Participating Media. *Computer Graphics Forum*, 30(1), 2011. doi: 10.1111/j.1467-8659.2010.01831.x 2

[58] K. Tang and C. Wang. Ecnr: Efficient compressive neural representation of time-varying volumetric datasets. *arXiv preprint arXiv:2311.12831*, 2023. 2

[59] I. Viola, A. Kanitsar, and M. Groller. Importance-driven volume rendering. In *IEEE Visualization 2004*, pp. 139–145, 2004. doi: 10.1109/VISUAL.2004.48 2

[60] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers et al. OSPRay - A CPU ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):931–940, 2017. doi: 10.1109/TVCG.2016.2599041 5, 7

[61] I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci. RTX beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location. In *Proceedings of High-Performance Graphics - Short Papers*, HPG '19, 2019. doi: 10.2312/hpg.20191189 2

[62] I. Wald, S. Zellmann, W. Usher, N. Morrical, U. Lang, and V. Pascucci. Ray Tracing Structured AMR Data Using ExaBricks. *IEEE Transactions on Visualization and Computer Graphics*, 27(2), 2021. doi: 10.1109/TVCG.2020.3030470 2

[63] A. Wolfe, N. Morrical, T. Akenine-Möller, and R. Ramamoorthi. Spatiotemporal Blue Noise Masks. In A. Ghosh and L.-Y. Wei, eds., *Eurographics Symposium on Rendering*. The Eurographics Association, 2022. doi: 10.2312/sr.20221161 2, 6

[64] E. Woodcock, T. Murphy, H. P., and L. T.C. Techniques used in the GEM Code for Monte Carlo Neutronics Calculation in Reactors and Other Systems of Complex Geometry. Technical report, Argonne National Laboratory, 1965. 1, 2

[65] Q. Wu, W. Usher, S. Petruzza, S. Kumar, F. Wang, I. Wald et al. VisIt-OSPRay: Toward an Exascale Volume Visualization System. In H. Childs and F. Cucchietti, eds., *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2018. doi: 10.2312/pgv.20181091 5

[66] L. Yang, S. Liu, and M. Salvi. A survey of temporal antialiasing techniques. *Computer Graphics Forum*, 39(2):607–621, 2020. doi: 10.1111/cgf.140186

[67] J. Ye, A. Xie, S. Jabbireddy, Y. Li, X. Yang, and X. Meng. Rectangular Mapping-based Foveated Rendering . In *2022 IEEE on Conference Virtual Reality and 3D User Interfaces (VR)*, pp. 756–764. IEEE Computer Society, 2022. doi: 10.1109/VR51125.2022.00097 2

[68] Y. Yue, K. Iwasaki, B.-Y. Chen, Y. Dobashi, and T. Nishita. Unbiased, Adaptive Stochastic Sampling for Rendering Inhomogeneous Participating Media. *ACM Transactions on Graphics*, 29(6), 2010. doi: 10.1145/1882261.1866199 2

[69] D. Zavorotny, Q. Wu, D. Bauer, and K.-L. Ma. From Cluster to Desktop: A Cache-Accelerated INR framework for Interactive Visualization of Tera-Scale Data. In G. Reina, S. Rizzi, and C. Gueunet, eds., *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2025. doi: 10.2312/pgv.20251153 2, 3, 9

[70] S. Zellmann, D. Seifried, N. Morrical, I. Wald, W. Usher, J. Law-Smith et al. Point containment queries on ray tracing cores for AMR flow visualization. *Computing in Science Engineering*, 2022. doi: 10.1109/mcse.2022.3153677 2

[71] S. Zellmann, Q. Wu, A. Sahistan, K.-L. Ma, and I. Wald. Beyond ExaBricks: GPU Volume Path Tracing of AMR Data. *Computer Graphics Forum*, 43(3), 2024. doi: 10.1111/cgf.15095 2

[72] J. Zvonek and A. Gillette. Pruning AMR: Efficient visualization of implicit neural representations via weight matrix analysis, 2025. doi: 10.48550/arXiv.2512.02967 9