DOI: 10.1111/cgf.70413
EUROGRAPHICS 2026 / B. Masia and J. Thies
(Guest Editors)

COMPUTER GRAPHICS *forum*
*Volume 45 (2026), Number 2*

# NAADF: Globally Illuminated Voxel Worlds Accelerated with Nested Axis-Aligned Distance Fields

A. Ulschmid[1] and M. Ott[1] and J. Macho[1] and M. Wimmer[1] and S. Ohrhallinger[1]

[1]TU Wien, Institute of Visual Computing & Human-Centered Technology

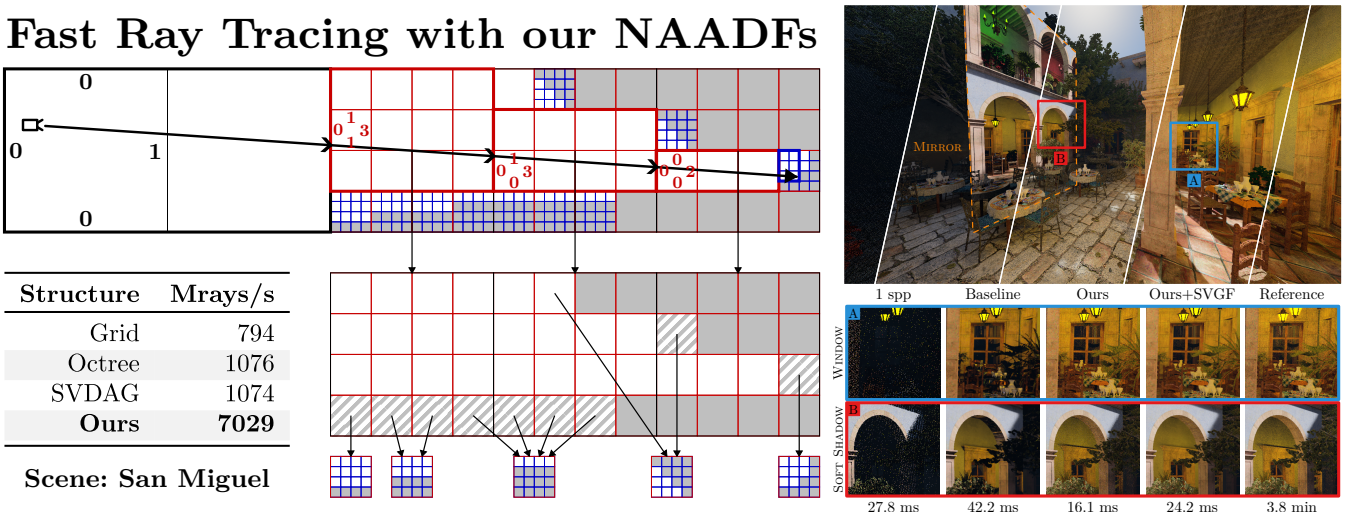


**Figure 1:** *Left: Our multilayered cell hierarchy with Nested Axis-Aligned Distance Fields (NAADFs) accelerates ray tracing by an order of magnitude while allowing editing and non-aligned dynamic entities. Right: Our NAADFs also enable significant improvements of our real-time global illumination approach in runtime and image clarity, especially when moving the camera, compared to state-of-the-art (Baseline: ReSTIR GI+SVGF+TAA with color clamping, reference: 8192 spp). The mirror wall (orange outline) and window in A are reflective.*

**Abstract**

*Achieving realistic rendering of 3D scenes in real time using path tracing is challenging due to the high sample count required, with ray tracing as the bottleneck. Focusing on voxels as a geometry representation offers significant opportunities for optimizations, especially for tracing the rays, but also for computing the samples. We propose a novel multilayered spatial structure augmented with in-cell axis-aligned distance fields (AADF) operating as caches. Our nested cell structure already accelerates ray tracing 3-5x compared to the state-of-the-art dense spatial structures, such as variants of directed acyclic graphs (DAG). Using the AADFs (constructed while rendering) inside the cells, we can double the ray throughput again (total 10x). As an application, exploiting nested AADFs (NAADFs) also allows us to double the speed of global illumination computations while significantly reducing artifacts from camera motion, such as flickering, blurring, ghosting, and aliasing, all of which are especially important in voxel worlds with sharp edges. We achieve this by adapting temporal antialiasing (TAA) to retain the last 32 frames rather than a single history buffer to create the final antialiased image, since the discretized voxel structure requires much less memory to store the quantized positions and normals of ray bounces. The sample accumulation for global illumination is optimized by compressing and separating lit/unlit samples, and we apply 8x8 window spatial resampling based on a reservoir-based spatiotemporal importance resampling (ReSTIR) method. Our proposed NAADFs support editing with quick updates to the acceleration in the background, overlays of non-aligned dynamic geometry, and can be easily extended to support transform-aware compression or to represent huge real-world scans. Retaining many past frames rather than just combining them opens up new opportunities to remove spatial and temporal artifacts in path tracing for global illumination.*

**CCS Concepts**

• ***Computing methodologies*** → ***Ray tracing; Volumetric models;***

## 1. Introduction

Voxel-based environments are becoming popular in modern games, as their discrete structure enables not only highly editable and dynamic worlds but also permits quick procedural generation or model creation, which can easily be crowd-sourced. Their appeal lies especially in their inherent support of volumetric operations, which easily permits complex physical interactions such as real-time terrain editing, fluid simulations, and other dynamic object interactions. However, when rendering geometric structures, e.g., using path tracing to simulate global illumination, voxel engines are constrained by either high memory usage or slow ray traversal, limiting their scalability and responsiveness. Various methods have been proposed to balance memory usage and the traversal efficiency of a simple grid. Sparse Voxel Octrees (SVO) and Directed Acyclic Graphs (DAG) exploit the redundancy of similar geometry to compress the data, while signed distance functions (SDF) store the nearest distance to the surface as a precalculation to accelerate ray marching. Both approaches have limitations: while SVOs and DAGs require complex traversal logic that does not perform well on the GPU, e.g., by very frequently loading pointers randomly from global memory, SDFs lack the directional information that can drastically accelerate ray tracing because it describes the extent of empty space in the respective directions.

In our approach, we propose applying axis-aligned distance fields (AADF) within nested (NAADF) cells to accelerate ray traversal while efficiently compressing the geometry in a multilayered cell hierarchy, further reducing redundancy through hashing. Contrary to SDFs, our AADFs store the distances for the six axis-aligned directions, limited to their cell bounding box, so only a few bits are required. These axis-aligned distances, perfectly suited for the nature of voxel worlds, together with cache-efficient storage and the spatial hierarchy, enable ultra-fast ray traversal with just a small memory usage trade-off. Compared to SVOs or DAGs, NAADFs can be edited much faster because they have only 3 layers. Their separate buffers mean no restructuring is required, and recomputing only the affected AADFs adds minimal overhead that can be hidden during rendering. Furthermore, NAADFs permit overlaying non-aligned dynamic geometry with little overhead.

Independent of the scene representation, accurately computing global illumination in real time remains a challenging problem. Traditional methods have approached this using a variety of approximations, including baked lighting, screen-space techniques, and simplified light-transport models [RDGK12]. Path tracing, on the other hand, offers a unified framework that naturally includes effects such as reflections, soft shadows, and caustics. However, real-time rendering of these global illumination effects with path tracing is slow and thus currently limited to only a few (1-4) samples per pixel (spp). Due to these constraints on the sampling rate, various artifacts arise, such as aliasing and noise. Over time, multiple spatiotemporal approaches for antialiasing, denoising, and resampling have been developed to mitigate these artifacts. For temporal reuse, data from previous frames is accumulated and stored in so-called history buffers. Most previous approaches only used a single history buffer due to their high memory usage. We reduce the memory footprint of each sample, enabling us to store the past 32 frames, while further optimizing spatial sample reuse based on the axis-aligned voxel structure.

Additionally, discarding all samples at the end of each frame would be inefficient, as light transport typically exhibits coherence across neighboring pixels and frames. Resampling both spatially and temporally exploits this coherence by enabling the selection and reweighting of samples based on their contributions to the final image. This can significantly reduce noise while staying within the strict performance constraints of real-time rendering. Recent reservoir-based spatiotemporal importance resampling (ReSTIR) approaches, however, do not consider the specific properties and challenges of rendering voxel worlds. When rendering voxel worlds, separately storing light sources incurs high overhead, especially with dynamic entities and editing. Not only would larger light structures consist of many voxels, but many small emitters could also be used to model partially glowing phenomena, such as lava. Thus, we optimized a recent ReSTIR approach, considering compression opportunities enabled by the voxel structure and performing material-based sampling only.

Our principal contributions are as follows:

- A multilayered distance cache accelerating ray tracing $\approx$ 10-fold
- 2x faster global illumination compared to state-of-the-art approaches, while also improving image clarity
- Real-time editing and lighting voxel worlds with non-aligned moving entities
- A voxel-engine implementing the described methods is available on GitHub [UOM*26]

## 2. Related Work

In this section, we describe the state of the art in voxel-based geometry structures and distance fields, as well as techniques for reducing sample-based artifacts in path tracing for global illumination, such as antialiasing, denoising, and resampling.

### 2.1. Voxel-based Geometry Structures

Voxels are regularly discretized elements of a volumetric space; their attributes can range from a single occupancy flag to appearance, such as color, material, density, or other properties. One of the first approaches to ray tracing voxels was to step through a uniform grid [AW87], which is fast; however, its cubic storage requirement is largely redundant for surface representations.

**Octrees** To address this, trees [Mea82] were applied to ray tracing [Gla84], removing large unused volumes by hierarchically subdividing for relevant detail. To avoid slow explicit neighbor-finding for ray marching, a top-down traversal strategy [RUL00] was introduced that uses previously computed parameter values derived from the ray's parametric representation to compute the ray-voxel intersections. With the advent of the GPU, sparse octree traversal was combined with aggregated voxel storage in the leaf nodes in the octree [CNLE09], represented as a small 3D grid and thus can be efficiently sampled with hardware acceleration; however, this is memory intensive, so large scene data have to be loaded on demand from the CPU. Sparse Voxel Octrees (SVO) [LK10] subdivide octree nodes only if they contain geometry, and for the required child

pointers, compress them where possible. The leaf voxels are not filled but contoured by intersecting surface planes, which improves visual quality. SVOs are inherently read-optimized and thus are not well-suited for editing, as that would require at least partial rebuilding; also, following lots of pointers is not cache-effective. Smooth Mixed-Resolution GPU Volume Rendering [BHMF08] enables seamless ray casting across volume bricks of differing resolutions by warping texture coordinates, with a focus on transparent volume rendering. They store a multi-resolution pyramid that is, however, similarly to SVOs, read-optimized, prohibiting live edits.

**Sparse Voxel DAGs** Sparse Voxel Directed Acyclic Graphs (SVDAG) [KSA13] extend SVOs by merging identical subtrees, resulting in a directed acyclic graph that reduces memory usage in redundant scenes, by a factor 5 in one example. However, this is not very suitable for dynamic scenes, as editing a node would affect all its references. Two approaches further improve static geometry reuse: The Symmetry-aware SSVDAG [VMG16] detects subtrees mirrored about any axis, adding a 3-bit tag, and applies variable-length pointers from the SVO [LK10]. The recent Transform-aware TSVDAG [ME25] further adds axis permutations and translations, allowing shifts beyond the boundary, and the large number of permutations requires a Huffman-compressed transformation ID. While more transformations result in increased compression, the subtree identification becomes overproportionally slower due to the large search space. To allow editing without decompressing and re-compressing the entire SVDAG, the HashDAG [CBE20] adds modified nodes to the tree instead, using a hashmap, but slows performance by a factor of 1.5-2. However, this inspired us to use hashing to reduce redundancy in our method.

**Out-of-Core Approaches** OpenVDB [Mus13] is widely used to represent sparse volumetric data for offline rendering. While it is very efficient at that, its out-of-core structure requires significant CPU-GPU synchronization, resulting in much worse ray traversal performance. Brickmaps [vWin15] are somewhat similar to our lowest layer of nodes in that they store a uniform grid of $8^3$ voxels in 512-bit bricks. However, these bricks are organized in a compressed, out-of-core grid, so a complex levels-of-detail mechanism is required to stream bricks to the GPU based on occlusion.

**Distance Fields** Distance fields, whether unsigned or signed (SDF), are a well-established representation for surfaces or level sets in computer graphics. They store the distance to the surface for points in space, often on a uniform grid. However, in their usual form, they do not store directional information and are therefore not suitable for efficient ray traversal. Directed Distance Fields [KBSS01] store a distance value for each axis direction per grid point. While at first this seems similar to our method, it represents only the distance to the surface along a single axis. Our approach uses six distance field values instead to create an empty bounding box around a cell. Adaptively Sampled Distance Fields (ADF) [FPRJ00] use an octree to store distances at cell corners, which are adaptively subdivided based on geometry. The hierarchy of distances in the ADFs accelerates ray tracing but does not use directional distances, which reduces its effectiveness, especially in voxel worlds. NanoVDB [Mus21] offers GPU support for rendering and simulation, improving performance by linearizing the OpenVDB data structure, resulting in a pointer-less tree and switching to in-core processing when possible. While using SDFs for collision detection and enabling to render SDFs, NanoVDB itself focuses on general voxel processing. Brixelizer [Adv24] is a sparse distance field technique on the GPU that can be used to trace rays against both static and dynamic geometry. It does, however, not use directional information; therefore, it is comparable to our structure with just using SDFs instead of NAADFs. Additionally, the Brixelizer pipeline is optimized for triangle geometry as input and not voxels.

### 2.2. Artifact Reduction in Path Tracing

The basics of physically-based light transport are given by Kajiya [Kaj86], with an extensive path tracing implementation, PBRT [PJH23]. While they aim for physical correctness, we prioritize real-time feedback and thus introduce bias (color inaccuracies in lighting effects) into the original rendering equation. Many rendering approaches compute global illumination using voxel discretizations but project the results onto triangle meshes for rendering [CPP24; YS20]. For example, VXGI [Pan15] is a real-time global illumination implementation using voxel cone tracing, which approximates indirect lighting by integrating radiance over multiple levels of a voxel octree. We instead directly display the discretized world, opting for the voxel aesthetic instead of triangles. Those triangle-based approaches could, however, benefit from our fast ray traversal using NAADFs. Both voxel- and triangle-based approaches suffer from artifacts due to undersampling, and multiple approaches to mitigate these have been developed over time.

**Antialiasing** Most spatial antialiasing techniques, such as supersampling (SSAA) [HA90], multisampling (MSAA)[Ake93], or morphological antialiasing (MLAA) [Res09] require the computation of additional samples or overblur details. Temporal antialiasing (TAA), on the other hand, reuses results from previous frames, while various history-rejection techniques prevent ghosting and disocclusion artifacts. A recent literature survey on TAA techniques is given by Yang et al. [YLS20]. Typically, TAA relies on reprojection of previous samples using a single history buffer and motion vectors. In contrast, we reproject and then accumulate multiple previous samples, accounting for the voxel structure. Building upon these approaches, there are proprietary machine learning-based variants (NVIDIA DLSS [NVI25a], AMD FSR [Adv25]), which perform joint super-sampling and antialiasing to boost frame rates.

Stable ray tracing techniques, which are closely related to TAA, have been proposed to improve temporal coherence in interactive ray tracing [CSK*17]. These methods reproject individual samples across frames and use dedicated hole-filling or visibility heuristics to reduce flickering, ghosting, and blurring artifacts. In addition, sample-caching approaches extend temporal reuse beyond image-space accumulation [NSL*07]. These methods decouple visibility from shading and cache samples in screen space, barycentric space, or world space using different data structures, such as shading atlases, voxel hierarchies, or hash-based representations [PJF25].

**Denoising** Real-time denoising is often performed via spatio-temporal variance-guided filtering (SVGF) [SKW*17; SPD18], or

Nvidia Real-Time Denoisers [NVI25b; Zhd21], which are used in video games such as Cyberpunk. Typical spatial image denoisers (bilateral [PKTD09], non-local means [RKZ12], median [Wei06]) and their geometry-aware variants [BEM11] also perform in real time. Denoisers designed for offline rendering include approaches based on regression [BRM*16], deep learning-based filters [CKS*17; Áfr25], joint neural denoising, and supersampling [TLP*22], as well as recent statistically-based approaches [SFWH25]. However, these often require more samples (≈32 spp) to perform well, consume more computational time and memory, and are thus not suited to a real-time context.

**Resampling** Bitterli et al. [BWP*20] introduced Reservoir-based Spatio-Temporal Importance Resampling (ReSTIR) based on earlier work by Talbot et al. [TCE05]. Since then, there have been many improvements to the original ReSTIR, the most important ones for this work being extensions to global illumination (ReSTIR GI) [OLK*21], enabling indirect lighting, and a generalized approach allowing correlated samples in long paths with arbitrary bounces [LKB*22], which, however, aims at unbiased rendering at interactive framerates. ReSTIR GI performs temporal resampling by using a history buffer similar to TAA, where reservoirs from previous frames are reprojected and combined with newly generated samples. These temporally accumulated reservoirs are used as candidates for spatial resampling, which randomly selects and merges a few neighboring reservoirs to further increase sample reuse. We extend the biased ReSTIR GI approach to benefit from just material-based sampling, as well as the reduced memory footprint of samples due to our optimizations for voxel worlds.

## 3. Method

The goal of our method is to structure a scene of voxels with color and material attributes as a hierarchical data structure, enabling efficient querying, editing, and ray tracing. First, we describe our multilayered data structure (Section 3.1), how we construct it from the raw voxels (Section 3.2) and then augment it with the distance fields (Section 3.3) that operate as a cache to accelerate ray tracing (Section 3.4). Finally, we describe how we achieve fast scene editing (Section 3.5) and how we extend it to overlay the scene with non-aligned dynamic entities (Section 3.6). An implementation of our methods can be found on GitHub [UOM*26].

### 3.1. The Multi-Layered Cell Data Structure

Since grids use a lot of memory and octrees or directed acyclic graphs (DAGs) are prone to pointer chasing, resulting in very many and, furthermore, slow global memory accesses, we design an in-between data structure as a shallow hierarchy with large aggregated nodes and organize the raw voxels in this multilayered cell structure. Then, we augment these cells with axis-aligned distance fields (AADFs) to accelerate ray marching and, further, we partially compress the data using hashing to eliminate redundancy, inspired by DAG variants (see Figure 2). Each voxel is described as a 15-bit type pointer to a material buffer that stores albedo color along with emissivity/reflectivity and roughness in 16 bits per entry. We group $4^3$ voxels into a *block*, and again $4^3$ blocks into a *chunk* since their respective sizes of 64 are well-suited for parallelization on the GPU. We generalize the three layers of chunks, blocks, and voxels to call them *cells*, and each layer is stored in its own buffer. Note that we also considered a simpler two-layer hierarchy with $8^3$ voxel blocks, but found that for larger scenes, the third layer improves compression without degrading performance.

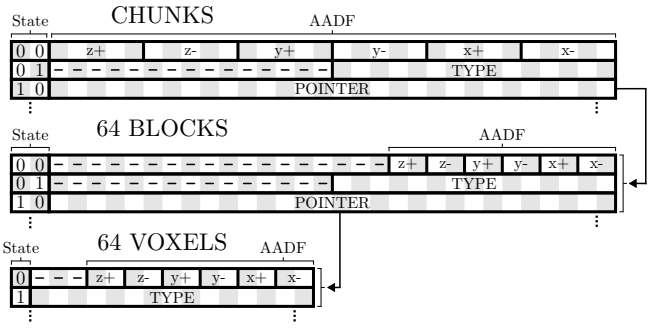


**Figure 2:** *The three layers of cells and their potential states: if empty (00 or 0), it contains the AADF, if fully uniformly filled (01 or 1), the type, pointing to a material buffer, if mixed (10), a chunk contains a pointer to 64 blocks, respectively a block to 64 voxels.*

Cells can have three states (empty, uniformly full, or mixed): if they are empty, we augment them with AADFs; if they are completely full with uniform voxel types, we just need to store that single type; and if they are mixed types or partially filled, we store a reference to a cell in their lower layer. A chunk is represented by 32 bits, of which 2 bits are for the state. If empty, the remaining 30 bits are used by the AADF with 5 bits for each of the six directions to indicate how far its empty bounding box extends in that direction, used for fast ray skipping; if uniformly full, for the 15-bit voxel type; or else a child pointer to a consecutive array of 64 blocks of the chunk in the block buffer. The layout of a block is similar to a chunk, except that a mixed block points to a hashed group of 64 voxels in the voxel buffer, permitting deduplication of blocks with equal voxels, and the AADF has only 2 bits per direction, as this is sufficient to represent the maximum distance to the bounding box of the chunk for each block of its $4^3$ blocks. Finally, a voxel requires only 16 bits since it only has 1 bit of state (full or empty) and uses the remaining 15 bits for the voxel type or the AADF (also 2 bits per direction).

### 3.2. Constructing the Multilayered Structure

When constructing our hierarchy, we simultaneously compress the data by determining equivalent blocks and uniformly filled chunks. We process the raw input voxels in chunks on the GPU using groups of 64 threads, since this matches their $4^3$ blocks. First, we test the voxels in each block in that chunk to see whether they are all equal; if so, the block is set to that voxel type. Otherwise, the voxels of the block are hashed using this function:

$$H = c_0 + \sum_{i=0}^{63} c_i \cdot v_i \quad \text{where} \quad v_i \text{ is the voxel type ID at index } i$$

with the coefficients $c_i = 31^{(64-i)} \mod 2^{32}$ precomputed on the CPU. As collision prevention, we use open addressing with linear probing [Knu98]. In an atomic operation, we check the hash slot: If

**Algorithm 1:** *Multilayered structure generation on GPU*

**Input:** *blockID*, *chunkID*, *threadIndex*
**Output:** *chunkBuffer*, *blockBuffer*, *voxelBuffer*

```
// Each thread processes one block
// Each group processes one chunk
voxels ← voxelInput[blockID];
hash ← getHash(voxels);
if voxels are of uniform type then
    block ← uniform type;
else
    block ← storeOrReuse(hash, voxels);
end
SyncThreadsOfGroup();
if threadIndex is 0 then
    if blocks are of uniform type then
        chunkBuffer[chunkID] ← uniform type;
    else
        blocksPointer ← getPointerToBlocks();
        chunkBuffer[chunkID] ← blocksPointer;
    end
end
SyncThreadsOfGroup();
if blocks are not of uniform type then
    blockBuffer[blocksPointer + threadIndex] ← block;
end
```

it is occupied, we check all voxels for equivalence and reuse the slot if true; else, we check up to 100 subsequent slots for an empty one to insert it; otherwise, we add a new block to the buffer. If the buffer reaches 75% occupancy, we resize it by 100%. Once all blocks have been hashed, the thread group is synchronized in order to determine whether all blocks of the chunk are identical and uniformly contain the same voxel type, which we assign in that case. If the chunk is entirely empty, we assign it the empty state. If partially filled, the first thread reserves 64 slots in the block buffer, stores its address as the child pointer in the chunk, and all threads write the block in parallel into their respective subsequent slots. The pseudo-code is shown in Algorithm 1.

### 3.3. Augment the Cells with AADFs

Our multilayered hierarchy can also be traversed with rays without computing the AADFs and is already much faster than the state-of-the-art structures (see Section 5). When loading or editing a voxel world, the modified cells are queued and the AADFs are computed (see Figure 3) in the background during rendering, with separate queues per layer, further speeding up ray tracing.

We define the AADF of a cell as a cuboid bounding box empty of geometry extending around it by the respective number of cells in each of the six axis-aligned directions (x+, x-, y+, y-, z+, and z-). To compute the AADF, we start with a cuboid equivalent to the cell (all zero values) and expand it iteratively, alternating between the three dimensions and concurrently expanding in both positive and negative directions by a single cell, bounded by either the maximum AADF size or the containing upper-layer cell. If the cells that

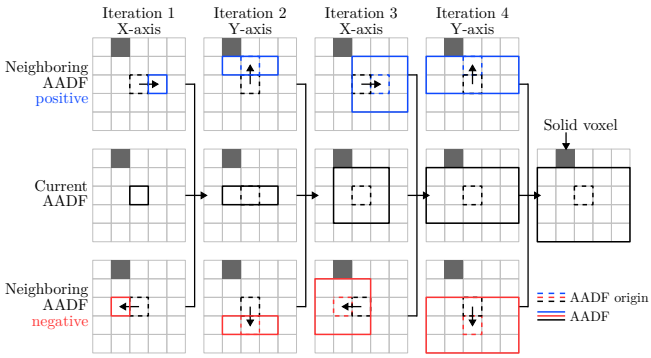


**Figure 3:** *The process of building the AADF for the center cell of an upper layer is illustrated in 2D. Each column corresponds to an iteration, expanding alternatingly by one cell into the x and y dimensions. As the AADFs are similarly expanded for each other cell as well in each iteration, from the third iteration, the AADFs of neighbor voxels are merged, resulting in linear construction time.*

would be added in the other two dimensions are all empty (and thus extending the empty cuboid by a slice), we increase the direction value. As an optimization to avoid complexity from becoming cubic, we synchronize the iterations and dimension expansions for all cells, which allows us to just merge the cuboid with the respective cuboid of the neighbor cell instead, resulting in $O(3dn)$ linear time complexity for $n$ cells and the distance $d$ to the cell boundary.

The value of $d$ is small because, for block or voxel cells, the size is $4^3$, so $d$ is at most 3, resulting in $3*3$ iterations in total across the three axes per cell. The maximum distance $d = 3$ is stored in 2 bits of the AADF. For chunks, the distance can be much larger, since it is only limited by the size of the voxel world. However, since we store only 5 bits per direction, the AADF can store values up to 31 (chunks), corresponding to 496 voxel distances. This results in $3*31$ total iterations per chunk and a good trade-off between skipping enough voxels in ray marching and a fast AADF construction time.

Furthermore, construction time is not critical, as the modified cells are queued separately for each cell type and their AADF is computed in the background. For each queue, the iterations+axes computations are synchronized to allow the reuse of the already computed neighbor cells' AADFs. Since chunks have more iterations than blocks or voxels, we will create separate $3*31$ queues, one per iteration. In our implementation, we handle one queue per frame, but this can be easily adjusted. Changing the axis order during construction may result in different cuboid regions, but this is rare, e.g., when AADF edges align with edges in the voxel world. This change would also average out over many AADFs. Therefore, we have not measured the performance impact of changing the axis-order computation.

### 3.4. Ray traversal

For ray marching, we use a method based on Digital Differential Analyzer (DDA), which advances along one axis and computes the increments for the other dimensions [AW87]. To take advantage of the empty bounding box spanned by the AADF, we improve on

DDA to advance many voxels along multiple axes (see Figure 1) in a single iteration to jump to the intersection point by computing which axis the next intersection point occurs on. We then advance the ray to this position, which lies on the boundary of two voxels, and use the previously intersected axis to determine whether we have hit the next voxel; otherwise, we continue with the next iteration. We start with the highest layer, the chunk that contains the ray origin. If the cell is empty, we use the AADF to intersect the empty volume with the ray, advancing its position. If the cell is mixed, we continue on the lower layer, and if a cell is full, we compute the intersection properties of the ray and terminate the traversal.

### 3.5. Editing the Data Structure

To simplify editing operations, we keep a copy of the world (voxels, blocks, and chunks) on the CPU. These data are also typically needed for other purposes on the CPU (e.g., path finding, collision detection, event triggers). After the initial world generation on the GPU, the data is copied to the CPU. From then on, any voxel, block, or chunk change is synced from CPU to GPU. When a chunk changes from empty to non-empty or vice versa, any AADFs in the surrounding volume of $63^3$ chunks need to be reset. Otherwise, ray tracing could skip regions with voxels, leading to undesired artifacts. If multiple chunks change, their surrounding volume can easily overlap. This is why we use a flood fill algorithm to mark each chunk for recomputation only once. As a further optimization, we perform the flood fill in groups of $4^3$ chunks. The marked chunk groups are then reset and gradually recomputed in the background.

### 3.6. Adding Dynamic Entities to the Voxel World

Using our data structure makes it easy to add non-aligned dynamic entities to the voxel world. We do this by extending each chunk with an additional 32-bit value that consists of a 24-bit pointer to an extra buffer and an 8-bit counter of the number of entities overlapping a chunk. The extra buffer contains the entity instance data for all the chunks, where each entry consists of the entity ID, position, rotation, and a pointer into a separate entity voxel buffer. Each voxel of an entity occupies 32 bits and also contains AADF values to improve ray traversal performance. Because many chunks may contain the same entities, the same pointer is used across chunks to eliminate redundancy. This computation is performed on the CPU by first adding the entities to the chunks in which they overlap. Afterwards, we use hashing to deduplicate chunk entity instances. Lastly, the entity instance buffer and the affected chunks are synced with the GPU.

During ray traversal, when a chunk contains such entities, that chunk index is saved. After the ray traversal of the main scene is completed, all entity instances for the saved chunks are iterated and checked for a ray hit. This is first done with their bounding box and if needed, the voxel volume of the entity is traversed using its AADF values. Note that AADFs of chunks must not contain chunks with entities, as these could be skipped during ray traversal. We address this by tracking chunks that change from empty to non-empty (as we do with edited chunks) and vice versa, triggering the reset of AADFs around chunks that change due to entity movement. We measured the basic performance penalty of handling entities at about 10%.

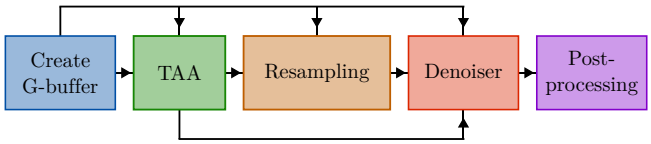


**Figure 4:** *Overview of our global illumination pipeline. Our long-term memory TAA is performed at the beginning, as its output informs global illumination and guides the denoiser. Tone mapping is applied in a post-processing step.*

## 4. Application: Accelerating Global Illumination

To enable real-time applications, such as video games, that display a voxel world with global illumination effects via path tracing, we need a fast approach tailored to the benefits and challenges of voxel rendering. Ray traversal during sample generation, as well as directional visibility checks, are time-consuming and can benefit from our fast NAADF data structure.

An overview of the global illumination pipeline is shown in Figure 4. Real-time rendering applications commonly use deferred rendering, where in the beginning, geometrical information is stored in so-called G-Buffers. We simplify the G-Buffer creation by exploiting the quantization of our voxel structure: Instead of position, depth, and normal buffers, we store each plane a ray bounced off at (3-bit normal, 14-bit distance along the normal). Specifically, the primary ray path is stored by recording up to four planes the ray bounced at until a non-specular surface is hit, including the last one. When the final position or depth is needed, we first compute the original ray with the camera transformation and pixel position. Together with the stored planes, the full path is then reconstructed by reflecting the ray along these planes. A 2D example of this can be seen in Figure 5. Material and entity properties are stored in separate buffers.

After G-Buffer creation, we perform temporal antialiasing (TAA) (Section 4.1), global illumination (GI) via path tracing with resampling (Section 4.2) and denoising (Section 4.3). Within each pixel, we jitter the position of the generated samples to further improve sampling of the distribution and reduce aliasing, thereby avoiding staircasing and Moiré pattern artifacts. While jittering and TAA consider the albedo, all other steps (resampling, denoising) work with only the indirect illumination.

### 4.1. Long-term Memory Temporal Antialiasing

Due to the discrete, grid-like structure of voxel worlds, aliasing occurs more often, particularly in the distance and at edges between voxels. The goal of anti-aliasing methods is to remove artifacts such as jaggedness and flickering. TAA addresses these issues via temporal accumulation instead of spatial filtering or super-sampling. Standard TAA uses a single history buffer to store previous samples and reprojects them into the current frame for accumulation. The reprojection is achieved via a transformation from a preceding frame for both the camera and any entities. Invalid samples are rejected based on color clamping: a previous sample is rejected if it falls outside the color range of a 3x3-pixel neighborhood of the

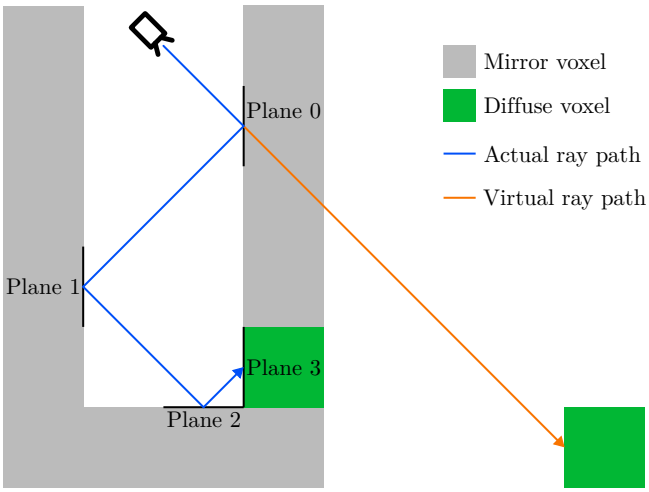


**Figure 5:** *When hitting a specular voxel, primary rays can bounce up to three times, while secondary rays can bounce up to two times. By storing the intersection planes rather than the final position directly, we can also easily compute the virtual path, which is needed for reprojections or projections.*

current sample. Color clamping is sensitive to noise and is therefore not suitable for path tracing [SKW*17].
Additionally, in the presence of moving objects or a moving camera, disocclusion artifacts and blurring can occur, as all previous frames are blended into a single history buffer. The reprojection itself is also discretized, so it never aligns perfectly, further amplifying the blurring. These errors in the history cannot be differentiated via a single accumulated value.

We perform TAA at the beginning of our pipeline (see Figure 4), right after shooting the primary rays, thus relying on depth rejection, which is not sensitive to noise in the radiance. We store everything needed to accumulate a sample in only 64 bits (See Figure 6). The color is stored with 8 bits per channel with an exponential compression using this function:

$$f(x) = 12\log_2(\tfrac{x}{100} + 2^{-\frac{255}{12}}) + 255 \quad \text{with} \quad x \in [0, 100]$$

We use a 16-bit floating-point value to store the distance to the first non-specular hit of the primary ray. Five bits are used to store the material's roughness, and 3 bits for the final normal. The remaining 16 bits serve as an additional hash for further validation during the accumulation process. The hash is created from the material type and normals of each specular reflection from primary rays. Thus, one 1440p frame requires 29 MB, compared to 59 MB for standard TAA. The lower memory requirement per frame enables us to store the past 32 frames within an acceptable amount of memory, rather than relying on a single history buffer. Accessing a long history of samples enables more faithful reprojection and rejection of invalid samples, thereby avoiding blurring, while trading off memory and achieving negligible runtime (see results, Table 4) for improved quality. A single history buffer also indirectly models an exponential falloff, generally giving less weight to older samples due to blending. We can instead assess importance on a sample-by-

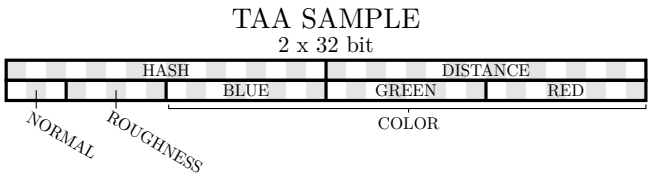


**Figure 6:** *One TAA sample requires 64 bits. The sample position is reconstructed using the distance, the pixel location in the texture, and the camera matrix from the previous frame. The hash, normal, and roughness are used for extra validation. Each color channel uses an exponential compression.*

sample basis, for example, based on changes in viewing angle for reflective materials.

For our depth-based rejection, we pre-process each pixel by calculating the minimum and maximum depth values in its 3x3 neighborhood. Next, we compute the same hash for each pixel as we did for the samples. When a sample has been reprojected from the current frame, its depth is verified against the precomputed range, and its hash must match one of the 9 neighboring pixels, otherwise, it will be rejected. Additionally, the sample is projected into the current frame to verify that the new pixel location is close to the original location (a radius of 1 pixel is used). If a sample is not fully diffuse, its roughness and changes in direction are used to reduce its contribution during movement. Reprojection is performed by selecting the closest matching pixel in the 3x3 neighborhood (not the current pixel) to avoid flickering, especially at edges where jittering can cause unstable depth variations. Any projections or reprojections are performed by using the virtual position of a sample (See Figure 5).

### 4.2. Resampling Pipeline

Separately storing light sources introduces a high overhead in voxel worlds. Thus, we can only rely on material-based sampling rather than additional light-source sampling. This adds additional challenges to resampling, as often no light source is hit and thus many samples with little radiance contribution are generated. While our approach is based on ReSTIR GI [OLK*21], we improve temporal accumulation and spatial resampling by designing a method tailored to the specific problems of material-based sampling. Additionally, ReSTIR GI produces more noise when jittering is used, as temporal pixel reservoirs are often reset during validation checks, leading to an effective 1 spp sampling rate in areas with frequent normal changes, whereas our adapted method is agnostic to jittering.
ReSTIR GI resamples in the temporal domain from pixel reservoirs, which are built from a single history buffer stored as a texture, and for efficiency, are also utilized during the spatial pass. Storing samples not within a single history buffer, but directly, would allow filtering each sample individually, removing the above problems. Normally, this would be infeasible due to huge memory requirements (48 bytes per sample, at least 30 frames when adhering to the original paper's implementation).

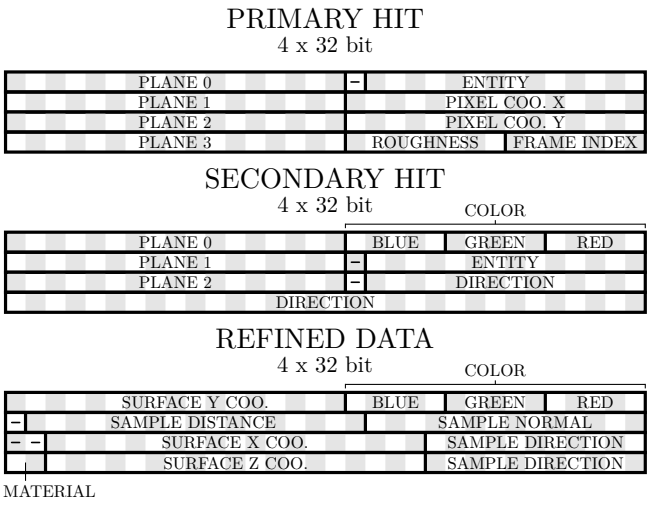


**Figure 7:** *The required sample data for resampling is split into three parts. Lit samples require both primary and secondary hit data, while unlit samples only need primary hit data. Valid lit samples from temporal resampling are converted into refined samples, which are then used in spatial resampling.*

**Sample Generation** To optimize computational resources, we leverage the accumulated sample count from our long-term memory TAA to selectively generate samples only where necessary. Newly revealed or disoccluded pixels receive priority, while previously sampled pixels are excluded as their sample count increases. If new samples are needed, secondary rays with at most 3 bounces are generated.

**Sample Compression** Each sample typically needs to store color and material information, as well as the paths to the creation and sample points for resampling. If a sample is unlit, it does not contribute to the final radiance; we do not need to store any color or sample-path information. To still be able to reject an unlit sample, though, we cannot just store the number of unlit samples. Thus, unlike ReSTIR GI, we store lit and unlit samples separately as lists in structured buffers, rather than in textures. Unlit samples consist of only primary hit data (16 bytes), while lit samples require both primary and secondary hit data (32 bytes) as it can be seen in Figure 7. Primary hit data reuses the intersection planes from the G-buffer. The pixel position and frame index are stored explicitly as the samples are not arranged in a texture. For secondary rays, their path to the first non-specular hit is stored similarly by recording up to three planes (including the last non-specular hit) at which the ray bounced. Instead of a pixel position, this also requires storing the initial ray direction for later reconstruction.

To further conserve memory, we store only every eighth unlit sample, weighted eight times (a higher compression ratio would introduce additional noise due to the high weighting factor). Thus, with four frames worth of storage (i.e., four times the pixel count of the render target), we can accumulate at least 32 frames for unlit samples (more with fewer spp). As is typically the case, few lit samples are generated due to material-based sampling, the buffer for the lit samples can also be rather small (we chose two frames' worth of storage), while still storing information from a large number of past frames (maximum 64 frames).

**Temporal resampling** The generated and stored samples are projected into 8x8 disjoint screen-space pixel regions, rather than using reservoirs as in ReSTIR GI. During this step, up to 32 lit samples are stored for each region in an extra buffer, while unlit samples are simply incremented in a region-specific counter. This increases the probability of finding lit samples and reduces noise, especially in regions with less temporal accumulation. Of the stored lit samples, those with very low brightness contribute little to the final image. To further filter these 32 samples per region, we compare each sample to the maximum brightness in its region. Based on the brightness ratio, we remove weakly lit samples while compensating for this by increasing the brightness of the remaining samples by the removal probability. Finally, up to eight remaining lit samples are refined (see Figure 7) and stored in a separate buffer for spatial resampling. If more than eight samples remain, we discard them. While this introduces some bias, it can drastically reduce fireflies and improve the performance of the spatial pass, for example, in a scene with a dim sky during sunset and a very bright, but far-away, light source.

**Spatial Resampling** For spatial resampling, ReSTIR GI runs 3 iterations, each of which performs a visibility check. We optimize this by running 12 iterations instead, while only the final selected sample is tested for visibility, as this is the most computationally intensive task. Compared to ReSTIR GI, we also do not start with an initial sample, as this significantly reduces fireflies. Although it removes the fallback sample in case no valid neighboring sample is available, we found that this is largely mitigated by the improved efficiency in finding lit samples and being able to perform more iterations because we only shoot a visibility ray for the final selected sample, rather than for each tested one as ReSTIR GI does. Our spatial resampling method then operates in screen-space by randomly selecting an 8x8 region in each iteration within an adaptive radius for each pixel, sampling, verifying (including material and geometry checks), and weighting them before combining the results, similar to ReSTIR GI. See Algorithm 2 for pseudo-code.

### 4.3. Denoising

While antialiasing addresses artifacts already present in the noise-free albedo, our denoising step is applied exclusively to remove noise from the indirect illumination result. As a final step in our pipeline, we thus apply a sparse bilateral filter [Wei06; PKTD09] (kernel size 21, $\sigma = 10$) in screen space to the indirect illumination. The filter step is split into a successive horizontal and vertical kernel. At random, on average, every second pixel is processed, allowing a larger kernel size to filter more medium-frequency noise. A bilateral filter typically consists of two components: color and geometry. To measure color similarity, we use the result from our TAA, normalized by the material's albedo, and convert it to a luminance value. For the geometric weight, we account for differences in normal directions and depth (again stored as an intersection plane) to further reduce edge bleeding. The denoiser result is

**Algorithm 2:** *Spatial resampling for each pixel*

**Input:** *Pixel position* $\vec{p}$, *surface* $s$, *sample count* $N$
**Output:** *color*

$R_s \leftarrow 0$;
**for** $i = 0$ **to** *maxIterations* **do**
  Randomly choose a neighboring region $r_n$
  Retrieve region info $I_n$ from $r_n$
  **if** $I_n$ *invalid with* $s$ **then**
    **continue**
  **end**
  Randomly select lit sample $R_n$ from $r_n$ and extract data
  **if** *geometry of* $R_n$ *invalid* **then**
    **continue**
  **end**
  Compute and verify Jacobian $|J_{R_n \to R_s}|$
  $\vec{c_n} \leftarrow R_n.color \cdot (I_n.litCount / I_n.totalCount)$
  $\hat{p_n} \leftarrow \text{TargetFunction}(R_n, \vec{c_n}) / |J_{R_n \to R_s}|$
  $R_s \leftarrow \text{Merge}(R_s, R_n, \hat{p_n})$
**end**
**if** *sample of* $R_s$ *is not visible from* $s$ **then**
  **return** 0
**end**
**return** ConvertToColor($R_s$)

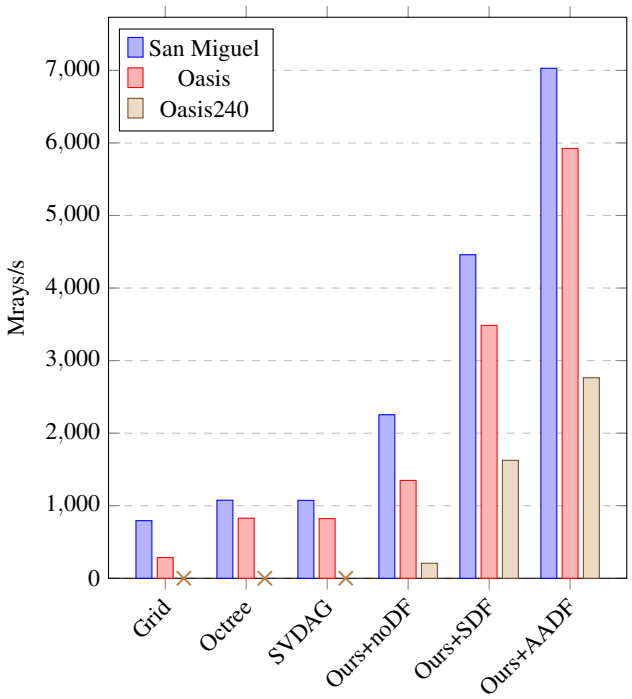


**Figure 8:** *Comparison of primary ray performance shows that our method outperforms state-of-the-art algorithms by factors 7–20x.*

then added to the computed TAA color and stored as part of the 32-frame TAA history.

Alternatively, we provide an implementation of SVGF [SKW*17]. While SVGF removes noise more aggressively, it tends to overblur details and consumes more resources (442 MB, 7.46 ms at 1440p). Our denoiser is much less resource-intensive (89 MB, 0.67 ms at 1440p) and preserves more details, albeit at the cost of increased noise. We leave the choice of denoising variant to the user, e.g., based on visual preference.

## 5. Evaluation

We evaluated our method on an AMD Ryzen 9800X3D with NVIDIA RTX 3090 Ti with 24 GB of VRAM. All computations were implemented as compute shaders using the MonoGame [Mon25] framework, which is based on DirectX 11, except for editing operations and entity logic, which are performed on the CPU; therefore, the buffers are constantly synchronized between the CPU and GPU. First, we demonstrate how our proposed data structure compares to state-of-the-art voxel-based geometry structures in terms of runtime and memory usage, providing construction times. We then detail the performance of our GI application in terms of quality, memory usage and runtime. Two representative known scenes were used for all tests, the first is SAN MIGUEL [Lla09] and the second is OASIS [Phynd]. We used a map size of 1024 x 512 x 2048 voxels. Further, we constructed an artificial test scene OASISX240 with 16384 x 8192 x 16384 (= 2 tera) voxels to show how our method performs in huge scenes. Tests for our data structure were conducted at 2160p resolution (3840x2160) to reduce potential CPU bottleneck, while the application has been tested at 1440p resolution (2560x1440). For our own implementation, we also tested it with SDFs (Chebyshev distance) as well as without distance fields to demonstrate the efficiency of our AADFs.

### 5.1. Analysis of NAADF Performance

We compare the runtime and memory usage of our proposed method with commonly used data structures, namely a simple uniform grid, an SVO [LK10], and an SVDAG [KSA13]. Our method is first used without any distance fields to demonstrate the advantage of our nested approach, then augmented with a signed distance field (SDF), and finally with our full-fledged NAADF.

#### Runtimes

Note that symmetry- and transform-aware SVDAGs are slower than the baseline SVDAG [KSA13], as they focus on compression, so we are not comparing their runtimes here, only with the baseline.

Figure 8 and Table 1 compare the performance of several state-of-the-art spatial data structures: Our nested spatial hierarchy already doubles ray performance without any distance fields. While adding SDFs only results in a significant increase for primary rays (permitting quick approaching of the surface rather than leaving it), augmenting our data structure with the NAADFs shows a total 7x speed-up for primary rays and up to 10x for secondary rays.

While a comparison of construction time with CPU-based state-of-the-art algorithms is a bit unfair, since we construct our NAADFs on the GPU, we still report construction times for our scenes in Table 2, split into the multilayered structure and the AADFs.

| Structure | San Miguel | | Oasis | | Oasisx240 | |
|---|---|---|---|---|---|---|
| | Prim. | Sec. | Prim. | Sec. | Prim. | Sec. |
| Grid | 794 | 100 | 286 | 47 | OOM | OOM |
| Octree | 1076 | 249 | 828 | 263 | OOM | OOM |
| SVDAG | 1074 | 250 | 822 | 263 | OOM | OOM |
| Ours+noDF | 2254 | 1018 | 1349 | 1000 | 208 | 28.6 |
| Ours+SDF | 4459 | 1050 | 3485 | 1244 | 1626 | 665 |
| Ours+AADF | **7029** | **2105** | **5924** | **2821** | **2765** | **817** |

**Table 1:** *Primary and secondary ray performance (Mrays/s) at 2160p. Best results per column in bold. For some methods the performance could not be computed as they ran out of memory (OOM).*

| Data structure | San Miguel | Oasis | Oasisx240 |
|---|---|---|---|
| Grid | 350 ms | 350 ms | out-of-memory |
| Octree | 6.3 s | 6.2 s | out-of-memory |
| SVDAG | 6.6 s | 6.5 s | out-of-memory |
| Ours structure | 203 ms | 200 ms | 87.6 s |
| Ours AADF | 6.2 ms | 7.3 ms | 1.97 s |

**Table 2:** *Construction time comparison between Grid, Octree, SVDAG, and our approach. The AADF computation here is not executed in the background but rather as fast as possible.*

**Memory usage**

| Structure | San Miguel | Oasis | Oasisx240 |
|---|---|---|---|
| Grid | 2.0 GB | 2.0 GB | out-of-memory |
| Octree | 60.1 MB | 104.5 MB | out-of-memory |
| SVDAG | 38.5 MB | 36.5 MB | out-of-memory |
| Ours (total) | 75 MB | 51.9 MB | 3419 MB |
| Chunks | 1 MB | 1 MB | 2147 MB |
| Blocks | 7.9 MB | 11.3 MB | 1271 MB |
| Voxels | 66.1 MB | 39.5 MB | 39.5 MB |

**Table 3:** *Memory usage comparison between Grid, Octree, SVDAG, and our approach (split into chunks, blocks, and voxels).*

Table 3 shows a comparison of memory usage, with our NAADF compression rates competitive with octrees and requiring only slightly more memory than the SVDAG. It should be noted that for the Oasis240 scene, SVDAG runs out of memory because we use multiple trees arranged in a grid. This can significantly reduce compression but speeds up ray traversal by lowering the tree depth significantly.

### 5.2. Global Illumination

We compare our long-term memory TAA approach with standard TAA using color clamping in YCgCo space and apply a Catmull-Rom filter during reprojection. As color-clamped TAA is sensitive to noise, we compare both renderings: a pure albedo with hard shadows from the sun (Figure 9 bottom) and full GI with our denoising and resampling method applied (Figure 9 top). We do not apply SVGF as it introduces additional blurring, reducing the clarity of the comparison, and use our less noisy resampling method. Generally, color clamping is sensitive to noise, while depth rejection is less effective on edges, where a sample's normal or depth frequently changes. However, even in noise-free albedo rendering, blurring and ghosting still occur with color-clamped TAA, unlike our long-term memory TAA, which accounts for the complex geometry of the voxel structure, especially at greater distances.

| Pipeline step | | Memory | Time |
|---|---|---|---|
| Antialiasing | TAA | 59 MB | 0.16 ms |
| | Ours (16 samples) | 501 MB | 0.84 ms |
| | Ours (32 samples) | 973 MB | 1.42 ms |
| Resampling | ReSTIR GI | 590 MB | 6.83 ms |
| | Ours | 509 MB | 5.37 ms |
| Total pipeline | Baseline | 811 MB | 27.4 ms |
| | Ours (32) w/o denoising | 1645 MB | 12.8 ms |

**Table 4:** *Comparison of video memory usage and performance between our approach and baseline (ReSTIR GI+TAA) at 1440p using SAN MIGUEL as a test scene.*

For resampling, we compare with ReSTIR GI [OLK*21] (Algorithm 4) without bias correction (see Figure 10). We present both real-time and offline (32-frame accumulation, same as our TAA) renderings to demonstrate that our approach does not generate temporally correlated samples, unlike ReSTIR GI, which depends on an initial sample for each pixel. However, resampling based on a 1 spp input as the initial temporal sample results in increased fireflies and noise. In contrast, our approach avoids reliance on a noisy initial sample, thereby reducing such artifacts, but lacks a fallback sample, leading to minor color inaccuracies such as local darkening.

Our approach is faster than ReSTIR GI, as our spatial resampling is more efficient due to memory coherency and testing visibility only once per pixel (see Table 4 and 5). Our implementation also requires less memory, even though overall more samples are stored. This is due to each sample requiring less memory, and ReSTIR GI having to do double-buffering of the temporal and spatial reservoirs. Overall, our pipeline performs twice as fast, as can be seen in Figure 1 and Table 5, as we effectively only generate $0.25 - 1$ spp per frame during resampling by basing sample generation on the accumulation rate of TAA, while maintaining similar visual quality (see Figure 1 and 10).

### 5.3. Limitations

The Monogame framework we used is based on DirectX11, which restricts us to a 32-bit limit on the pointers in the buffers and a maximum resolution of 2 trillion voxels, but this is just a technical limit that can easily be eliminated by moving to another graphics API, as we have shown with a very large scene, that performance is proportional to size. A limiting factor in some scenes is that voxelization of textures results in a large number of distinct voxel types. This

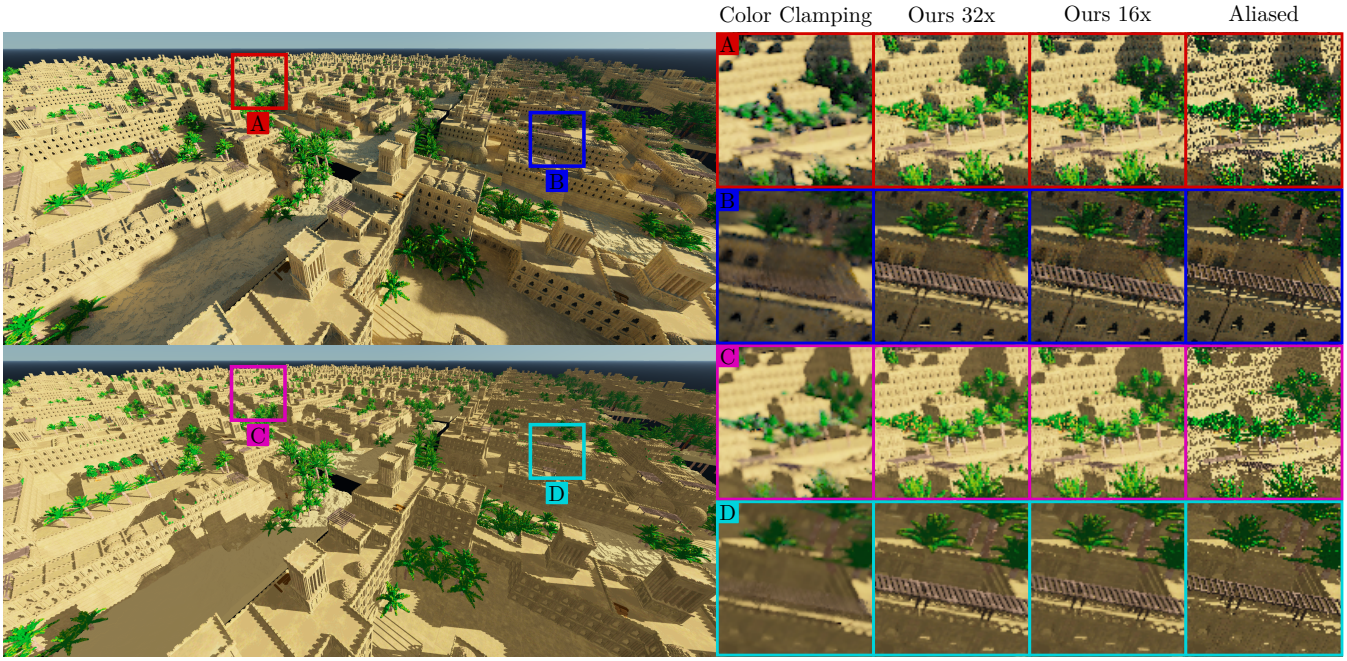


**Figure 9:** *Comparison of our long-term memory TAA approach with color-clamping TAA during movement (top: full GI with our denoiser and our resampling method, bottom: albedo plus hard shadows). An aliased version is rendered without jittering. Ours exhibits almost no blurring or ghosting. In the full GI rendering, the TAA input is still not completely noise-free after denoising. Thus, using color clamping creates blurring, flickering, ghosting, and unwanted color changes. Even for noise-free input images, as can be seen in the lower comparison, blurring still occurs. For this scene,* OASIS *is aligned into a 8x7 grid.*

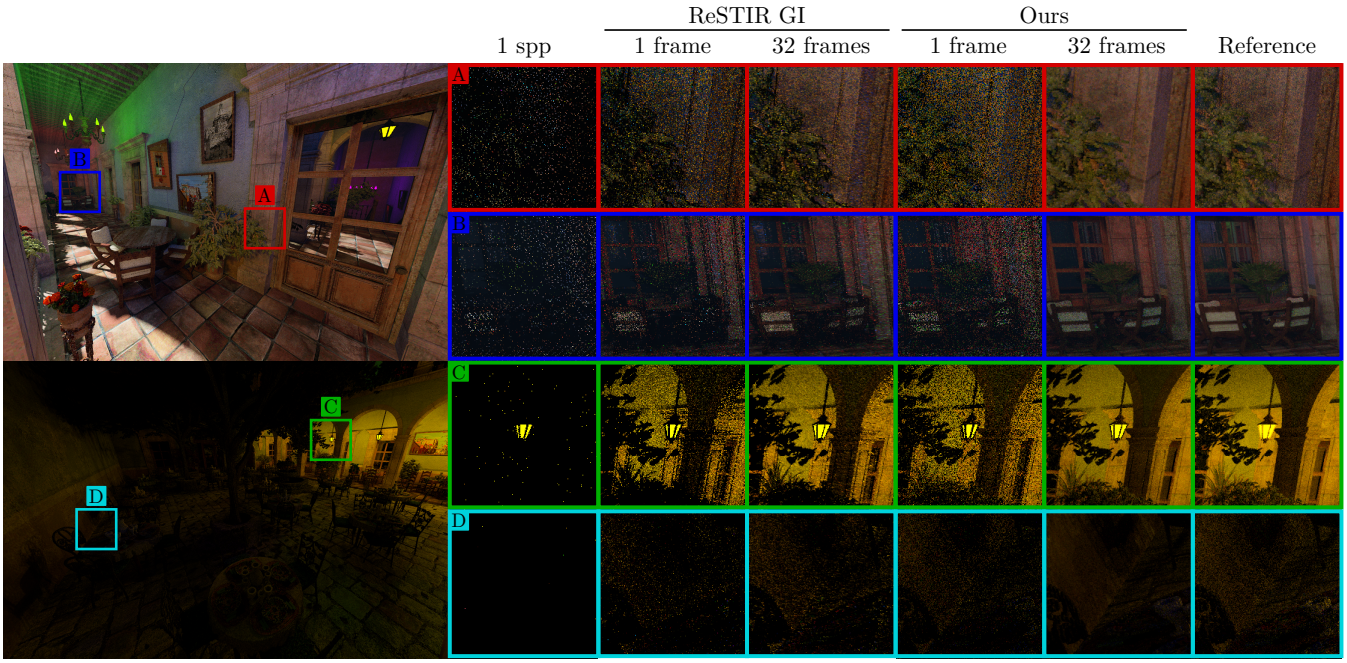


**Figure 10:** *Comparison of our resampling approach with ReSTIR GI (reference: 8192 spp without denoising). Our result contains significantly less noise and fireflies. This is especially noticeable when accumulating the result for 32 frames, as our output is not temporally correlated, unlike ReSTIR GI. Compared to the reference, some darkening can be observed in difficult areas (B, C), as we do not have a fallback sample.*

| Pipeline step | Ours | | Baseline |
|---|---|---|---|
| | 0.25 spp | 1 spp | 1 spp |
| Atmosphere | 0.12 ms | | 0.12 ms |
| Primary | 0.71 ms | | 0.71 ms |
| TAA reproject (32 samples) | 1.22 ms | | |
| Secondary select | 0.10 ms | | |
| Secondary trace | 4.91 ms | 19.4 ms | 19.4 ms |
| Resampling logic | 0.02 ms | | |
| Resampling region mask | 0.09 ms | | |
| Resampling project lit | 0.6 ms | | |
| Resampling project unlit | 0.47 ms | | |
| Resampling refine regions | 0.08 ms | | |
| Resampling temporal | **∑ 1.26 ms** | | 0.52 ms |
| Resampling spatial | 4.11 ms | | 6.31 ms |
| **Resampling total** | **5.37 ms** | | **6.83 ms** |
| Denoise | 0.67 ms | | 7.46 ms |
| TAA merge | 0.2 ms | | |
| TAA | | | 0.16 ms |
| Postprocess | 0.15 ms | | 0.15 ms |
| **TAA total** | **1.42 ms** | | **0.16 ms** |
| **Total** | **12.78 ms** | **27.16 ms** | **27.37 ms** |
| **Total (with denoise)** | **13.45 ms** | **27.83 ms** | **34.83 ms** |

**Table 5:** *Frame time breakdown of our approach and the baseline (ReSTIR GI+TAA+SVGF) at 1440p using SAN MIGUEL as a scene. Columns-spanning Cells indicate spp-independent costs.*

reduces compression, and the increased memory consumption further reduces performance due to fewer cache hits. Scenarios with a high number of entities or extensive area editing can introduce CPU limitations due to the required CPU-GPU synchronization, but large editing operations could be GPU-accelerated if needed.

As our approach for global illumination is more biased than ReSTIR GI, some areas with complex geometry exhibit noticeable darkening. However, this is offset by reduced noise, flicker, and blurring enabled by our novel long-term temporal antialiasing and resampling approaches. Note that soft shadows from the sun are not handled during resampling, resulting in slightly increased noise. Another minor drawback of our approach is increased memory usage, primarily due to storing many samples for our long-term memory TAA, which performs better on higher-end graphics hardware with large enough cache sizes and sufficient memory bandwidth. All introduced improvements exhibit around 32 frames of noticeable temporal lag in the indirect lighting (mainly when changing shadows from the sun or editing light sources), due to depth rejection during antialiasing and the failure to validate samples during resampling.

## 6. Conclusion

We propose a multilayered data structure with a carefully balanced hierarchy between depth and node size to maximize ray tracing performance for voxel worlds, and augment it with axis-aligned distance fields computed and cached on the fly, resulting in an order-of-magnitude faster ray tracing than state-of-the-art algorithms. The additional memory requirements are partially offset by hashing voxels to compress identical subsets of the voxel world. As an application, we demonstrate improvements in global illumination and temporal antialiasing (TAA), resulting in significantly reduced noise, especially with complex geometry, as our approach is agnostic to jittering. Our adapted TAA method requires more memory and is more computationally demanding, but does not introduce any blurring or flickering during movement.

In future work, we plan to explore optimizing CPU computations for editing operations and entities or entirely moving these to the GPU. We will also investigate extending our hashing of identical voxel blocks, similar to symmetry- or transform-aware SVDAGs, further reducing memory requirements. This could accommodate huge scenes, e.g., from real-world scans, while we also consider out-of-core extensions. In this regard, adding the option for voxels to reference a 3D texture instead of always embedding material properties in each voxel would drastically increase hashing efficiency in certain scenes. Furthermore, we will explore applying our approach to triangle-based worlds by tiling them into voxels for accelerated rendering, while also enabling geometry queries, collision detection with dynamic entities, and fluid simulations. Regarding our GI application, we will investigate adding sample validation to our resampling approach and implementing some approach of color clamping in TAA to help mitigate the temporal lag. We will also examine how the bias of our resampling approach can be mitigated without introducing additional noise.

## 7. Acknowledgements

This work has been partially funded by FWF project F77 (SFB *Advanced Computational Design* SP4) and by the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF) project ICT19-009. Open Access funding provided by Technische Universitat Wien/KEMÖ.

## References

[Adv24] ADVANCED MICRO DEVICES, INC. *AMD FidelityFX Super Resolution (FSR)*. https://gpuopen.com/manuals/fidelityfx_sdk/techniques/brixelizer/. 2024 3.

[Adv25] ADVANCED MICRO DEVICES, INC. *AMD FidelityFX Super Resolution (FSR)*. https://www.amd.com/en/products/graphics/technologies/fidelityfx.html. 2025 3.

[Áfr25] ÁFRA, ATTILA T. *Intel® Open Image Denoise*. https://www.openimagedenoise.org. 2025 4.

[Ake93] AKELEY, KURT. "Reality Engine graphics". *Proceedings of the 20th annual Conference on Computer Graphics and Interactive Techniques* (1993). DOI: 10.1145/166117.166131 3.

[AW87] AMANATIDES, JOHN and WOO, ANDREW. "A Fast Voxel Traversal Algorithm for Ray Tracing". *EG 1987-Technical Papers*. Eurographics Association, 1987. DOI: 10.2312/egtp.19871000 2, 5.

[BEM11] BAUSZAT, PABLO, EISEMANN, MARTIN, and MAGNOR, MARCUS. "Guided Image Filtering for Interactive High-quality Global Illumination". *Computer Graphics Forum* (2011). ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2011.01996.x 4.

[BHMF08] BEYER, JOHANNA, HADWIGER, MARKUS, MÖLLER, TORSTEN, and FRITZ, LAURA. "Smooth Mixed-Resolution GPU Volume Rendering ". *IEEE/ EG Symposium on Volume and Point-Based Graphics*. Ed. by HEGE, HANS-CHRISTIAN, LAIDLAW, DAVID, PAJAROLA, RENATO, and STAADT, OLIVER. The Eurographics Association, 2008. ISBN: 978-3-905674-12-5. DOI: 10.2312/VG/VG-PBG08/163-170 3.

[BRM*16] BITTERLI, BENEDIKT, ROUSSELLE, FABRICE, MOON, BOCHANG, et al. "Nonlinearly Weighted First-order Regression for Denoising Monte Carlo Renderings". *Computer Graphics Forum* 35.4 (2016), 107–117. ISSN: 14678659. DOI: 10.1111/cgf.12954 4.

[BWP*20] BITTERLI, BENEDIKT, WYMAN, CHRIS, PHARR, MATT, et al. "Spatiotemporal Reservoir Resampling for Real-time Ray Tracing with Dynamic Direct Lighting". *ACM Transactions on Graphics (TOG)* 39 (2020), 148:1–148:17. DOI: 10.1145/3386569.3392481 4.

[CBE20] CAREIL, VICTOR, BILLETER, MARKUS, and EISEMANN, ELMAR. "Interactively Modifying Compressed Sparse Voxel Representations". *Computer Graphics Forum*. Vol. 39. 2. Wiley Online Library. 2020, 111–119. DOI: 10.1111/cgf.13916 3.

[CKS*17] CHAITANYA, CHAKRAVARTY R. ALLA, KAPLANYAN, ANTON S., SCHIED, CHRISTOPH, et al. "Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder". *ACM Transactions on Graphics (TOG)* 36.4 (July 2017). ISSN: 0730-0301. DOI: 10.1145/3072959.3073601 4.

[CNLE09] CRASSIN, CYRIL, NEYRET, FABRICE, LEFEBVRE, SYLVAIN, and EISEMANN, ELMAR. "GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering". *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. 2009, 15–22. DOI: 10.1145/1507149.1507152 2.

[CPP24] COSIN AYERBE, ALEJANDRO, POULIN, PIERRE, and PATOW, GUSTAVO. "Dynamic Voxel-Based Global Illumination ". *Computer Graphics Forum* (2024). ISSN: 1467-8659. DOI: 10.1111/cgf.15262 3.

[CSK*17] CORSO, ALESSANDRO DAL, SALVI, MARCO, KOLB, CRAIG E., et al. "Interactive stable ray tracing". *Proceedings of High Performance Graphics* (2017). DOI: 10.1145/3105762.3105769. URL: https://doi.org/10.1145/3105762.3105769 3.

[FPRJ00] FRISKEN, SARAH F, PERRY, RONALD N, ROCKWOOD, ALYN P, and JONES, THOUIS R. "Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics". *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. 2000, 249–254. DOI: 10.1145/344779.344899 3.

[Gla84] GLASSNER, ANDREW S. "Space Subdivision for Fast Ray Tracing". *IEEE Computer Graphics and Applications* 4.10 (1984), 15–24. DOI: 10.1109/MCG.1984.6429331 2.

[HA90] HAEBERLI, PAUL and AKELEY, KURT. "The Accumulation Buffer: Hardware Support for High-quality Rendering". *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (1990). DOI: 10.1145/97879.97913 3.

[Kaj86] KAJIYA, JAMES T. "The Rendering Equation". *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902 3.

[KBSS01] KOBBELT, LEIF P., BOTSCH, MARIO, SCHWANECKE, ULRICH, and SEIDEL, HANS-PETER. "Feature Sensitive Surface Extraction from Volume Data". *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: Association for Computing Machinery, 2001, 57–66. ISBN: 158113374X. DOI: 10.1145/383259.383265 3.

[Knu98] KNUTH, DONALD ERVIN. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 1998 4.

[KSA13] KÄMPE, VIKTOR, SINTORN, ERIK, and ASSARSSON, ULF. "High Resolution Sparse Voxel DAGs". *ACM Transactions on Graphics (TOG)* 32.4 (2013), 1–13. DOI: 10.1145/2461912.2462024 3, 9.

[LK10] LAINE, SAMULI and KARRAS, TERO. "Efficient Sparse Voxel Octrees–Analysis, Extensions, and Implementation". *NVIDIA Corporation* 2.6 (2010) 2, 3, 9.

[LKB*22] LIN, DAQI, KETTUNEN, MARKUS, BITTERLI, BENEDIKT, et al. "Generalized Resampled Importance Sampling". *ACM Transactions on Graphics (TOG)* 41 (2022), 1–23. DOI: 10.1145/3528223.3530158 4.

[Lla09] LLAGUNO, GUILLERMO M. LEAL. *San Miguel Scene*. https://casual-effects.com/data. Accessed: 2025-07-12. 2009 9.

[ME25] MOLENAAR, MATHIJS and EISEMANN, ELMAR. "Transform-Aware Sparse Voxel Directed Acyclic Graphs". *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 8.1 (2025), 1–16. DOI: 10.1145/3728301 3.

[Mea82] MEAGHER, DONALD. "Geometric Modeling using Octree Encoding". *Computer Graphics and Image Processing* 19.2 (1982), 129–147. DOI: 10.1016/0146-664X(82)90104-6 2.

[Mon25] MONOGAME TEAM. *MonoGame Framework*. https://www.monogame.net. Accessed: 2025-07-12. 2025 9.

[Mus13] MUSETH, KEN. "VDB: High-Resolution Sparse Volumes with Dynamic Topology". *ACM Transactions on Graphics (TOG)* 32.3 (2013), 1–22. DOI: 10.1145/2487228.2487235 3.

[Mus21] MUSETH, KEN. "NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation". *ACM SIGGRAPH 2021 Talks*. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450383738. DOI: 10.1145/3450623.3464653. URL: https://doi.org/10.1145/3450623.3464653 3.

[NSL*07] NEHAB, DIEGO F., SANDER, PEDRO V., LAWRENCE, JASON, et al. "Accelerating real-time shading with reverse reprojection caching". *GH '07*. 2007 3.

[NVI25a] NVIDIA CORPORATION. *NVIDIA Deep Learning Super Sampling (DLSS)*. https://developer.nvidia.com/dlss. 2025 3.

[NVI25b] NVIDIA CORPORATION. *NVIDIA Real-Time Denoisers (NRD)*. GitHub repository. Version 4.15.0, accessed 14 August 2025. 2025. URL: https://github.com/NVIDIA-RTX/NRD 4.

[OLK*21] OUYANG, YAOBIN, LIU, SHIQIU, KETTUNEN, MARKUS, et al. "ReSTIR GI: Path Resampling for Real-Time Path Tracing". *Computer Graphics Forum* 40 (2021). DOI: 10.1111/cgf.14378 4, 7, 10.

[Pan15] PANTELEEV, ALEXEY. *NVIDIA VXGI: Dynamic Global Illumination for Games*. GDC Vault video, Game Developers Conference (GDC) 2015. Presented by NVIDIA, available online: https://www.gdcvault.com/play/1022392/NVIDIA-VXGI-Dynamic-Global-Illumination. 2015 3.

[Phynd] PHYRONNAZ. *Oasis Voxel Scene*. https://github.com/Phyronnaz/VoxelAssets/tree/master/Oasis. Accessed: 2025-07-12. n.d. 9.

[PJF25] PHILIPPI, HENRIK, JENSEN, HENRIK WANN, and FRISVAD, JEPPE REVALL. "Stable Sample Caching for Interactive Stereoscopic Ray Tracing ". *Pacific Graphics Conference Papers, Posters, and Demos*. Ed. by CHRISTIE, MARC, HAN, PING-HSUAN, LIN, SHIH-SYUN, et al. The Eurographics Association, 2025. ISBN: 978-3-03868-295-0. DOI: 10.2312/pg.20251294 3.

[PJH23] PHARR, MATT, JAKOB, WENZEL, and HUMPHREYS, GREG. *Physically Based Rendering: From Theory to Implementation (4th ed.)* 4th. Cambridge, MA, USA: MIT Press, May 2023, 1312. ISBN: 9780262048026 3.

[PKTD09] PARIS, SYLVAIN, KORNPROBST, PIERRE, TUMBLIN, JACK, and DURAND, FRÉDO. "Bilateral Filtering: Theory and Applications". *Foundations and Trends in Computer Graphics and Vision* 4.1 (2009), 1–73. ISSN: 15722740. DOI: 10.1561/0600000020 4, 8.

[RDGK12] RITSCHEL, TOBIAS, DACHSBACHER, CARSTEN, GROSCH, THORSTEN, and KAUTZ, JAN. "The State of the Art in Interactive Global Illumination". *Computer Graphics Forum* 31 (2012). DOI: 10.1111/j.1467-8659.2012.02093.x 2.

[Res09] RESHETOV, ALEXANDER. "Morphological Antialiasing". *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: Association for Computing Machinery, 2009, 109–116. ISBN: 9781605586038. DOI: 10.1145/1572769.1572787 3.

[RKZ12] ROUSSELLE, FABRICE, KNAUS, CLAUDE, and ZWICKER, MATTHIAS. "Adaptive Rendering with Non-Local Means Filtering". *ACM Transactions on Graphics (TOG)* 31.6 (2012), 1–12. ISSN: 07300301. DOI: 10.1145/2366145.2366214 4.

[RUL00] REVELLES, JORGE, URENA, CARLOS, and LASTRA, MIGUEL. "An Efficient Parametric Algorithm for Octree Traversal". *International Conference in Central Europe on Computer Graphics and Visualization* (2000) 2.

[SFWH25] SAKAI, HIROYUKI, FREUDE, CHRISTIAN, WIMMER, MICHAEL, and HAHN, DAVID. "Statistical Error Reduction for Monte Carlo Rendering". *Proceedings of the SIGGRAPH Asia 2025 Conference Papers*. SA Conference Papers '25. Association for Computing Machinery, 2025. ISBN: 9798400721373. DOI: 10.1145/3757377.3763995 4.

[SKW*17] SCHIED, CHRISTOPH, KAPLANYAN, ANTON, WYMAN, CHRIS, et al. "Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination". *Proceedings of High Performance Graphics* (2017). DOI: 10.1145/3105762.3105770 3, 7, 9.

[SPD18] SCHIED, CHRISTOPH, PETERS, CHRISTOPH, and DACHSBACHER, CARSTEN. "Gradient Estimation for Real-time Adaptive Temporal Filtering". *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1 (2018), 24:1–24:16. DOI: 10.1145/3233301 3.

[TCE05] TALBOT, JUSTIN, CLINE, DAVID, and EGBERT, PARRIS K. "Importance Resampling for Global Illumination". *Eurographics Symposium on Rendering*. 2005. DOI: 10.5555/2383654.2383674 4.

[TLP*22] THOMAS, MANU MATHEW, LIKTOR, GABOR, PETERS, CHRISTOPH, et al. "Temporally Stable Real-Time Joint Neural Denoising and Supersampling". *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 5.3 (July 2022). DOI: 10.1145/3543870 4.

[UOM*26] ULSCHMID, ANNALENA, OTT, MARVIN, MACHO, JONAS, et al. *NAADF: Globally Illuminated Voxel Worlds Accelerated with Nested Axis-Aligned Distance Fields*. https://github.com/cg-tuwien/NAADF. 2026 2, 4.

[VMG16] VILLANUEVA, ALBERTO JASPE, MARTON, FABIO, and GOBBETTI, ENRICO. "SSVDAGs: Symmetry-aware Sparse Voxel DAGs". *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 2016, 7–14. DOI: 10.1145/2856400.2856420 3.

[vWin15] Van WINGERDEN, TL. "Real-time Ray Tracing and Editing of Large Voxel Scenes". MA thesis. Utrecht University, 2015 3.

[Wei06] WEISST, BEN. "Fast Median and Bilateral Filtering". *ACM Transactions on Graphics (TOG)* 25.3 (2006), 519–526. ISSN: 07300301. DOI: 10.1145/1141911.1141918 4, 8.

[YLS20] YANG, LEI, LIU, SHIQIU, and SALVI, MARCO. "A Survey of Temporal Antialiasing Techniques". *Computer Graphics Forum* 39 (2020). DOI: 10.1111/cgf.14018 3.

[YS20] YALÇINER, BORA and SAHILLIOĞLU, YUSUF. "Voxel Transformation: Scalable Scene Geometry Discretization for Global Illumination". *Journal of Real-Time Image Processing* 17.5 (Oct. 2020), 1585–1596. ISSN: 1861-8219. DOI: 10.1007/s11554-019-00919-1 3.

[Zhd21] ZHDAN, DMITRY. "ReBLUR: A Hierarchical Recurrent Denoiser". *Ray Tracing Gems II*. 2021, 823–844. ISBN: 978-1-4842-7185-8. DOI: 10.1007/978-1-4842-7185-8_49 4.