Fast Voxelization and Level of Detail for Microgeometry Rendering

Javier Fabre · Carlos Castillo · Carlos Rodriguez-Pardo · Jorge Lopez-Moreno

the date of receipt and acceptance should be inserted later

Abstract Many materials show anisotropic light scattering patterns due to the shape and local alignment of their underlying micro structures: surfaces with small elements such as fibers, or the ridges of a brushed metal, are very sparse and require a high spatial resolution to be properly represented as a volume. The acquisition of voxel data from such objects is a time and memory-intensive task, and most rendering approaches require an additional Level-of-Detail (LoD) data structure to aggregate the visual appearance, as observed from multiple distances, in order to reduce the number of samples computed per pixel (E.g.: MIP mapping). In this work we introduce first, an efficient parallel voxelization method designed to facilitate fast data aggregation at multiple resolution levels, and second, a novel representation based on hierarchical SGGX clustering that provides better accuracy than baseline methods. We validate our approach with a CUDA-based implementation of the voxelizer, tested both on triangle meshes and volumetric fabrics modeled with explicit fibers. Finally, we show the results generated with a path tracer based on the proposed LoD rendering model.

Keywords Ray tracing · Level of Detail · Volumetric Models · Voxelization

Javier Fabre
Universidad Rey Juan Carlos
E-mail: fjavifabre@gmail.com

Carlos Castillo
Universidad Rey Juan Carlos
E-mail: carlos.castillo@urjc.es

Carlos Rodriguez-Pardo
Politecnico di Milano
Euro-Mediterranean Center on Climate Change (CMCC)
RFF-CMCC European Institute on Economics and the Environment (EIEE)
E-mail: carlos.rodriguezpardo.jimenez@gmail.com

Jorge Lopez-Moreno
Universidad Rey Juan Carlos
E-mail: jorge.lopez@urjc.es

1 Introduction

Voxel-based data representations have become fundamental tools across numerous domains in computer graphics, as well as medical and scientific data. Their structured nature provides an efficient discretization of 3D-spaces which enables complex algorithms to process geometric data systematically. Voxels have been used in rendering for fast ray-tracing [1], volumetric path-tracing [2], shadow computation and visibility [3] among other applications.

Recent advances in neural scene representations, particularly Neural Radiance Fields [4, 5] (*NeRF*), have renewed the interest in efficient volume representation and rendering. While learning-based approaches may offer impressive results, they often come with significant computational complexity and a lack of desirable properties like editability. More recent works, like *Gaussian Splatting* (GS) [6, 7] and its ray-tracing counterparts [8, 9], offer different trade-offs between quality, speed and memory use. Building upon these works, approaches like *Radiant Foam* [10], or sparse voxels [11] have shown comparable or superior performance. The combination of simpler analytic representations with sparse voxel flexibility, as in SplatVoxel [12], is promising, especially for temporal coherence. While RAW data has its uses, complex representations via distributions enable higher information storage per voxel and easier level-of-detail (LoD), also performing well in optimization, as demonstrated by GS.

Despite these advances, a significant challenge remains in efficiently representing and rendering materials with anisotropic microstructures—such as fabrics, hair, and brushed metals. These materials exhibit high sparsity (occupying a small fraction of the volume), require high spatial resolution to capture fine details, and exhibit complex directional properties that define their appearance.

To utilize voxels in rendering, vector-based primitives such as points, lines, splines, triangles, surfaces, and solids must undergo a voxelization process. This transformation can become computationally prohibitive, especially for complex geometry. Consequently, previous research has developed approaches that optimize voxelization for specific primitive types [13] or scenarios, such as high-resolution voxel grids [14], complex models [15], or image-based data [16]. While voxelized volumes for rendering are inherently sparse [17], microgeometry exhibits particular sparsity. Materials with fine-scale structures—such as fabrics, hair, and brushed metals—present two challenges: they occupy a minimal fraction of the volume while simultaneously requiring high resolution to capture their detail. This combination creates significant computational and memory complexity that existing approaches struggle to address efficiently.

To our knowledge, no prior method specifically addresses the GPU-optimized voxelization of complex data in the highly sparse scenarios characteristic of microgeometry, despite the prevalence of such materials in real-world applications. To fill this gap, we present the following contributions:

- **A novel voxelization method** optimized for sparse microgeometry. Our approach handles various input primitives, including splines and sources with textured orientation data, while maintaining memory efficiency at high resolutions.
- **A GPU-CPU implementation** that maximizes voxelization speed and memory efficiency through strategic workload distribution, enabling processing of highly detailed microgeometry that would exceed memory limits with traditional approaches.
- **SGGX-H**: A novel hierarchical volumetric data model for LoD rendering based on Symmetrical GGX (SGGX) [18]. Unlike previous approaches that lose critical directional information through simple averaging, our method clusters and preserves directional distributions across resolution levels, maintaining the characteristic anisotropic appearance of complex materials.

We validate our approach through a CUDA-based implementation tested on both triangle meshes and volumetric fabrics with explicit fibers, demonstrating significant improvements compared to baseline approaches. Upon publication, we will release the source code and render scenes.

2 Previous Work

In the following, we discuss current voxelization methods, the problem of representing and voxelizing microgeometry, and the state of the art in level-of-detail filtering.

2.1 Voxelization methods

Before volumetric data can be used, it must be stored in a format allowing both efficient storage and fast access. Data compression is essential for high-resolution volumes, making sparse representations the standard approach. OpenVDB [19, 20] and its GPU counterpart NanoVDB [21, 22] are the industry standard for voxel data management, offering excellent performance for both reading and writing operations. However, neither framework focuses on minimizing voxelization time or optimizing for specific primitives. Instead, they provide raster-based voxelization methods for meshes that prioritize storage density over processing speed, primarily using CPU parallelization rather than fully exploiting GPU capabilities for the voxelization process itself.

Instead, researchers have developed specialized approaches for different 3D primitive types, including lines [23], triangles [24], polygons [25], and solids [26]. For more computationally intensive scenarios, Crassin et al. [27, 28] pioneered GPU-based algorithms for volumetric data generation. For a comprehensive review of voxelization methods and recent advances, we refer readers to the survey by Aleksandrov et al. [13]. Beyond simple voxelization, accurately storing complex properties such as optical density, orientation distributions, or statistical data presents additional challenges. While OpenVDB and NanoVDB support custom data types, operations like down-sampling and interpolation become problematic for these complex properties. Most existing voxelization techniques focus primarily on density values, requiring significant modifications to effectively capture and store directional or statistical information that is crucial for accurately representing anisotropic materials.

2.2 Voxelizing and rendering micro structures

Rendering materials with micro-geometric details using explicit geometry or high-resolution orientation and dis-

placement maps presents significant computational and memory challenges. These representations not only demand substantial resources but also frequently suffer from aliasing artifacts, making the efficient representation and rendering of micro-geometry an ongoing challenge in computer graphics.

Volumetric approaches offer a promising alternative, with various specialized scattering models developed for different material types, including microflakes and scattering models for fabrics. Khungurn et al. [29] demonstrated that accurately representing fiber-like materials requires dedicated scattering models tailored to their unique properties. We refer the reader to the survey by Castillo and colleagues [30] for further information about the different techniques used for cloth rendering.

Lopez-Moreno et al. [31] adapted the *Lumislice* [32] algorithm for volumetric cloth representation in GPU. Their approach deals with some of the problems of fiber-like materials; however, it is not suitable for LoD generation, as samples are generated directly at the finer level, with simple averaging of multiple fragments per voxel. This operation fails to preserve critical orientation data, making accurate scattering simulation impossible at coarser detail levels.

Our voxelization method addresses this limitation, preserving orientation distributions across multiple detail levels. By maintaining accurate directional information even at lower resolution levels, fibers and textured surfaces can be rendered with less samples, using various anisotropic scattering models while preserving their characteristic appearance across different viewing distances.

2.3 Filtering and Level-of-Detail

MIP-mapping [33] has been widely use since it was presented for textures LoD. Usually, the MIP-map data is generated by linearly interpolating pyramid levels. This approach works for some parameters like *albedo* or *roughness*, but fails for *alpha* or orientation data (i.e. *normal* or *tangent maps*).

Modifying additional maps can solve this issue. LEAN-Mapping [34] has been used for real-time scenarios to overcome the problems that arise from normal-map interpolation. Gauthier and colleagues [35] addressed the issue of normal MipMapping with a neural approach, also showing that straight-forward down-sampling of normal-maps requires changes in *roughness* for each LoD. These techniques do not generate correct input data, instead relying in the material to visually fix the wrong appearance that the data generates in the final render. It would be beneficial to obtain a representation of the material properties that is agnostic to the material model, so different models could be used to render such data without any modifications.

Mesh decimation has been used to generate different LoD meshes that can be used to save space when rendering complex. Commonly, these new meshes for LoD are generated using either *edge contraction* [36], *vertex decimation* [37] or vertex clustering [38] techniques driven by some metric to measure quality of the resulting mesh for a given LoD. However, these metrics often failed to accurately represent all factors that influence LoD appearances. Recently, there has been major advances in inverse rendering techniques (please refer to Laine et al. [39] for an overview of current approaches) that help generate new metrics that can account for final appearance to drive mesh optimization, such as the proposed by Hasselgren and colleagues [40].

Previous work in LoD generation has followed two main approaches: either optimizing each level to minimize memory consumption [41], thereby reducing overall asset size, or optimizing the rendered appearance across the entire LoD chain [42], which often sacrifices coherent internal data representation. For fibrous materials specifically, recent approaches have focused on down-sampling volumetric data [43, 44] using exclusively Symmetric GGX microflake (SGGX) distributions [18]. This limitation prevents the use of advanced fiber scattering models [29], as the original directional data (fiber tangents, density) becomes inseparably mixed with scattering information at each voxel. Zhou and colleagues [45] follow a similar approach to tackle the problem of LoD in big scenes using an Aggregated Bidirectional Scattering Distribution Function (ABSDF), aggregating different appearances for multiples objects in a single voxel. Such mixing restricts material editability and constrains rendering to SGGX-compatible shading models.

A special case for LoD generation are scenes that include elements at both macroscopic and microscopic level. Hybrid mesh–volumes methods have been proposed to tackle this problem [46]; however, a mixed representation introduces its own challenges for some rendering techniques and require special-case handling. Vicini et al. [47] proposed a non-exponential transmittance model that handles this issue while relaying just in the usage of volumes.

Neural methods have also proven to be useful in LoD generation. Nevertheless, they either focus in geometry explicit representations, such as Signed Distance Functions (SDFs) [48] or fail to properly capture high-frequency appearances [42].

Our approach overcomes these limitations by maintaining a clear separation between geometric/directional data and material properties throughout

the LoD hierarchy. We preserve orientation distribution, thereby enabling the use of various anisotropic scattering models while maintaining visual consistency across detail levels. This separation also facilitates material editing at any level of detail without requiring regeneration of the entire hierarchy. Furthermore, our method efficiently handles the extreme sparsity of microgeometry through a parallel GPU implementation that processes only occupied regions, enabling high-resolution representation with manageable memory requirements.

3 Voxelization for data aggregation

Traditional voxelization algorithms (Section 2.1) require multiple passes for the generation of LoD structures, specially for geometries with high variability in local orientations, or surfaces with microscale data (e.g.: tangent maps for anisotropic reflections). Aggregating efficiently fine sparse details into coarser blocks is a complex task that usually implies rasterization and fixed-size memory allocation for sub-block grids.

In this section, we explain our voxelization model, suitable for high frequency sparse geometric objects such as fibers, hair or thin planes with texture data, and designed to process volumetric data into multiple Levels of Detail (LoD), preserving as much appearance information as possible.

The voxelization pipeline is shown in Figure 1. Our model first samples the input 3D primitives, generating voxels only if they are filled by, at least one sample, thus avoiding unnecessary computation for non-occupied voxels. For each sample, we store as much data as necessary for future rendering purposes (Section 3.1). The generation of samples is different for curves and surfaces (Section 3.2), but the data per node is stored and indexed in the same manner. Once the samples are generated at the maximum sampling rate and detail desired and stored as leaf nodes, they are gathered and aggregated into coarser bounding voxel blocks, in a multilevel hierarchical structure that is then used for LoD generation. In our examples, we will use a three-level multi-grid, with resolutions optimized for each scenario and limited automatically by the samples generated at the geometry and the GPU memory available, but additional levels can be considered depending on the complexity of the scene. This gathering is computed in parallel on the GPU, composing histograms of orientation data (Section 3.4) and optical density for densities (Section 3.5) from the data stored at the leaf nodes. Furthermore, since our memory consumption does not depend of the grid resolution, it is specially suitable for high resolution voxelization of sparse, detailed geometry with a low percentage of volume occupancy: thin surfaces, hair, fibers, etc.

In Section 4, we propose a novel LoD hierarchical representation based on SGGX for rendering, that leverages the data gathered at this stage. The results are shown in Section 5.

3.1 Vertex map generation

The first stage of our algorithm is executed once per input model (triangle mesh or spline collection in our examples 1).

In this step, we generate a map to keep the relationship between each sample of the primitive that we receive as input (triangle or spline), and the voxelization nodes sharing same space. We store multiple indices per node, referencing the geometric and material properties of the object at that point, and the multilevel blocks where the node is spatially registered. In practice, we can map this identifier to different materials (BSDF or Phase Function), for rendering purposes, or even parametric data to use in additional voxelization sub-steps to expand the original sample in the node to finer voxel resolutions. For instance, we can store the nodes of a spline representing the central axis of a yarn and a cross section displaying the distribution of secondary fibers at each node.

For each input model, we compute a global bounding box ($B(b_{min}, b_{max})$) and maximum generation radius δ (taking into account factors like maximum triangle edge length, radius, or displacement distance). Then, we move node data to GPU memory, where we use the bounding box information to transform our samples from global space to normalized coordinates in range $[0, Resolution)$ based in our input resolution for first and second levels, and also our selected resolution for the third level (8 voxels per axis in the examples of this paper):

$$Resolution = \prod_{i=1}^{3} Resolution_i \tag{1}$$

$$\mathbf{P}_{normalized} = \frac{\mathbf{P} - b_{min}}{b_{max} - b_{min}} \tag{2}$$

$$\mathbf{P}_{volume} = \mathbf{P}_{normalized} * Resolution \tag{3}$$

After this step, each node holds the following information:

- **Translation**: The position of the node represented by three floating-point values.

$$p = (x, y, z) \in \mathbb{R}^3 \tag{4}$$

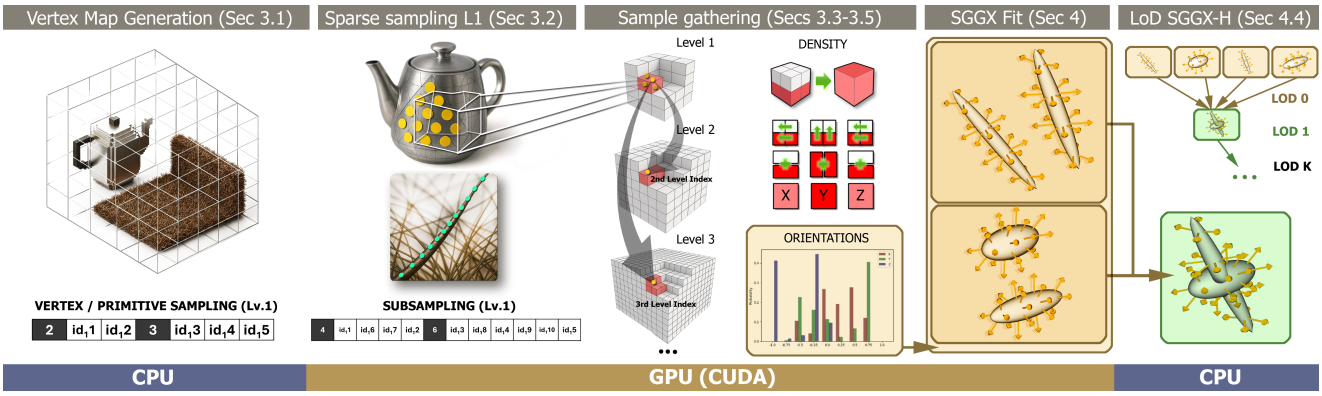

Fig. 1 Our pipeline process from sparse 3D vertices to LoD of voxels in representing 3D generated model. We generate new *vertices* subdividing our original primitives, each of these vertices with all information needed to be processes in the GPU. Our 3rd step gathers all valid samples already arranged in the corresponding voxel. Last, we process each generated histogram per voxel (for orientation data) and generate our whole hierarchical structure for rendering.

– **Normal**: The main direction of crimp offset displacement (splines) or face normal (triangles) represented by three floating-point values.

$$n = (n_x, n_y, n_z) \in \mathbb{R}^3, \quad \|\mathrm{n}\| = 1 \tag{5}$$

– **Tangent**: The direction to the following node (splines) or face tangent (triangles) represented by three floating-point values.

$$t = (t_x, t_y, t_z) \in \mathbb{R}^3, \quad \|\mathrm{t}\| = 1 \tag{6}$$

– **Properties ID**: An identifier for the corresponding properties, represented by a 32-bit integer: $id \in Z_{32}$

Once the data is uploaded to the GPU in our desired coordinate space, we run a parallel kernel to generate the maximum number of nodes by first level block. During this process, we can also identify (and discard) the blocks containing no data (invalid blocks). For this, we check each control node using a distance function:

$$\mathbf{P}_{block} = S_{block} * I_{block} \tag{7}$$

$$C_{block} = \mathbf{P}_{block} + \frac{S_{block}}{2} \tag{8}$$

$$\mathbf{P}_{relative} = \|\mathbf{P}_{volume} - C_{block}\| \tag{9}$$

$$\mathbf{Q} = P_{relative} - \frac{S_{block}}{2} \tag{10}$$

$$\mathbf{D} = \|\max(\mathbf{Q}, \mathbf{0})\|^2 + \min\left(\max(\mathbf{Q}_x, \mathbf{Q}_y, \mathbf{Q}_z), 0\right) \tag{11}$$

where we can use block size (S_{block}) and 3D index (I_{block}) to compute the node position relative to a given block $\mathbf{P}_{relative}$ based on the position of the block itself $\mathbf{P}_{block}$. $\mathbf{P}_{relative}$ can then be used to compute the distance D to consider nodes that, even if they fall outside of a block, could still generate new samples that fall within it (e.g., for fiber generation or triangle point generation). Therefore, we consider any node that ensures $D < \delta^2$, where δ is the maximum distance where data can be generated (e.g: maximum yarn radius, maximum triangle edge length). Using this equation, we can discard those nodes with distance lower than zero.

Once we know the maximum number of nodes per block (level 1), we use that information to initialize a CPU flat array with space for this number per block and an extra counter (Figure 1 left). Then, we use a kernel to write for each block the number of nodes that contain data (valid nodes) and the ID of the nodes. At this point, we copy this mask to CPU to know which first level blocks are empty and generate an axis–aligned stencil, using the shape of the filled blocks for later usage during volumetric path tracing.

3.2 Sparse data sampling

After computing the array of valid nodes per block, we can proceed to generate our voxel data. For this we run, only for non empty blocks, three different kernels that will do the following:

1. Prepare initial nodes to interpolate.
2. Subsample the data.
3. Generate voxel nodes from new samples

The first step prepares nodes for easier computation of later steps. This step may reorder, duplicate or generate additional data so later is easier to access consecutive (connected) samples. Our initial nodes may not be enough to cover all space continuously at the desired voxelization resolution. To avoid discontinuities we introduce a sub-sampling step. Notice that, for higher output resolutions, a higher number of sub-samples will be required if the input is really sparse. The last step generates samples from the interpolated new vertices and stores them in a given voxel based on their 3D position. Additionally, this step can be used to further in-

crease detail of the samples, adding texture based data (3D or 2D) to modify properties of the nodes before storing them in a given voxel (eg: Normals, tangents, twist).

Per sample, we compute: a second and third level indices, the node orientation (tangent and normal data) and a single material index we can use to later apply different spatially varying properties such as Phase-functions or BSDFs for each node (Table 1).

Even though all steps may differ somewhat for each input primitive, the second step is particularly dependent on it. As an example, for splines defined using *Catmull-Rom* interpolation [49] we want to first define the whole spline from the 4 required points that we store in memory, and then generate additional samples along the shape of the spline using linear spline interpolation. In case we use triangles as our input primitive, our first step is easier since each triangle is defined in our data-model by points that are not shared between different triangles. During the second step, we also generate additional points in the surface of each triangle. We use barycentric coordinates for this sampling.

Since all of these steps are computed in the GPU, relying on actual random sampling is not feasible. Instead, we use equally distributed samples based in kernel indices. We distribute the amount of samples chosen as input along t for later usual interpolation when voxelizing spline primitives, and the low–distortion mapping proposed by Heitz [50], modified for GPU to avoid branching. Additionally, we weight the amount of final samples to generate in each single triangle based on their relative area to the biggest triangle in the model. We show a comparison between the Naive method and this approach in the Supplementary Material.

3.3 Sample gathering

Once the samples are generated, we move this data to a grid-like structure, as done by previous methods [17, 21]. By saving just the last sample generated at each voxel, in an asynchronous fashion, we avoid the implicit ordering of draw calls that previous algorithms suffer (multiple calls per axis) since we can order the nodes in the GPU. Such ordering also facilitates data aggregation, histogram computation, and further processing.

Once our samples are ordered by second level block ID, we use a kernel to detect the vector positions where the ID changes, so we know where the samples for a given block start and how many of them we computed. Note that, since samples are the main data, the amount of memory that we need to prepare for a worst-case scenario depends on the amount of samples generated instead of the voxel resolution. Even if the resolution scales dramatically, we do not waste any memory in empty voxels.

Parameters	Type	Bytes
2nd level index	integer	4
3rd level index	integer	4
Tangent data	u–integer	3x1
Normal data	u–integer	3x1
Material ID	u–integer	4

Table 1 Required space for our initial information per node. This data is used to compute the information that we stored in the final leaf nodes (in our examples, the voxels at the 2nd resolution level).

3.4 Directional data

Thanks to the per-voxel sample gathering described in Section 3.3, we can compute a distribution histogram for any directional data, such as normals or tangents, at every voxel with $density > 0$.

While this histogram gives us a higher detail information for the underlying data inside a voxel than a single direction, using this information for rendering can be difficult. Aside from the space needed to store these histograms for each voxel, sampling a direction from it (needed if we want to use the volume for volumetric rendering) is also computationally expensive.

To overcome this, we propose to fit each computed histogram to a SGGX function [18], which is considered a representation with a good trade-off between compression and quality for 3D orientation distributions. Since SGGXs represent a spheroid–like shape, and can be easily sampled, we can use them to encode the distribution of directions in the histogram, using just 6 values. Furthermore, they are easily aggregated into new SGGX distributions, which we leverage in the algorithm presented in Section 4.

3.5 Density occupation and view dependent data

Many real-world materials can only be accurately represented using correlated media [51, 52]. Thus, storing a single value for density data without any consideration for the view direction, usually produces inaccurate rendering results.

As a solution, we compute the probability of an event depending of the view direction, using the finer resolution level in the voxelization (level 3), in a similar fashion to Crassin et al. [28] we project the 3D voxel block grid to a 2D grid. Samples are projected in the

XY, XZ and YZ planes using GPU shared memory, and we store the projected position of every sample by block, so we can obtain a value per axis, producing an anisotropic voxel (Figure 1).

To compute occupancy $\mathbf{O}$, instead of projecting the data to the axis, we analyze the third level data, obtaining a percentage by dividing the number of different identifiers by the amount of voxels in our third level of voxelization, he nce computing actual density for any given voxel of second level:

$$\mathbf{O} = \frac{\sum_{v \in \text{block3}} \text{hit}(v)}{Resolution_3^3} \tag{12}$$

4 Data post-processing and fitting

Our voxelization process ends with different samples per voxel, that we are able to use to compute an histogram of orientations. To generate the final volume, we could store randomly one of them as the final value, hence arriving at the same data that can be generated with classic voxelization approaches. However, we can use all this samples per voxel to perform an aggregation step and arrive to a better representation for voxel values compared to a single random value/sample, also easier to handle than RAW histograms.

For our pipeline, we focus in fitting this data using a single SGGX distribution [18] at leaf-voxel level. Our approach could be used to fit more complex distribution instead, or even perform sampling using the raw data from the histogram. We use the original distribution presented by Heitz and colleagues without its phase-function interpretation, since our only requirement is to store directional data such as tangents and directional density information for later evaluation, with a different phase function model. By just storing orientation directly by using SGGX instead, we are free to use any material model later, using the frame defined by our sampled orientation data during render.

4.1 Fitting to SGGX distribution

To generate our single distribution for the leaf-level voxel we use the approach proposed by Heitz [18] to fit arbitrary distributions to their SGGX distribution. We compute the covariance matrix $\mathcal{E}$ for our orientation data X:

$$X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}, \mathbf{x}_i \in \mathbb{R}^3 \tag{13}$$

$$\mathcal{E} = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^\top \tag{14}$$

From this covariance matrix we can later extract eigenvector $(\omega_1, \omega_2, \omega_3)$ and their corresponding eigenvalues $\{\lambda_1, \lambda_2, \lambda_3\}$, equivalent to the projected areas on each eigenvector direction [18]:

$$\lambda_1 = \sigma(\boldsymbol{\omega}_1), \quad \lambda_2 = \sigma(\boldsymbol{\omega}_2), \quad \lambda_3 = \sigma(\boldsymbol{\omega}_3) \tag{15}$$

Using this computed values we can then get the SGGX matrix S defining the distribution of the original histogram as:

$$S = (\omega_1, \omega_2, \omega_3) \begin{pmatrix} \sigma(\omega_1) & 0 & 0 \\ 0 & \sigma(\omega_2) & 0 \\ 0 & 0 & \sigma(\omega_3) \end{pmatrix} (\omega_1, \omega_2, \omega_3)^T \tag{16}$$

We can sample this SGGX distribution to obtain a direction from the inner orientation distribution allowing scattering functions that rely in tangential/normal orientation to be used for rendering. Different approaches such as microflakes/microfacets models can be used too, re-oriented using the distribution stored at each voxel.

Our sampling process uses a similar approach as the one presented by Heitz et al. [18] for the Visible Normal Distribution Function (VNDF). We modify their approach by generating uniform samples in the whole sphere, instead of the hemisphere, so our samples can be projected from the whole space to the underlying ellipsoid. This ensures that our samples get distributed in the whole Normal Distribution Function (NDF), not only in the VNDF subsection. When sampling SGGX distributions for rendering, we use the VDNF as usual. We also modify our fitting step to account for corner cases where all the input orientations align to almost the exact same orientation, hence, generating a ill-defined SGGX. We use a uniform distribution to introduce a small deviation to the original data so we obtain a noisier but unbiased SGGX as output.

4.2 Density data

For density data, it is not trivial to encode directional occupation using SGGX distributions, since occupation does not fit a set of main axes, and requires multiple evaluations of a given number of distributions that will increase exponentially with the LoD selected. Moreover, SGGX distributions are not normalized and the normalization term has a non-closed form. While this lack of normalization does not affect orientation encoding, evaluating the area of the encoded ellipsoid for density storage, requires this normalization factor, or at least the scale of the ellipsoid volume.

Taking into account these limitations, we compute LoD density levels with the same projection approach

described in Section 3.5. We project nodes at the finer resolution level to the XY, XZ and YZ planes, obtaining a value per axis for the coarser level.

4.3 Storing data

Once we have computed the SGGX orientation distributions, and LoD densities, we need to store them in the final volumes that will be used for volumetric rendering. Using GPU kernels, we store the main indices of the SGGX matrix representing orientation data (S) in the leaf nodes of the LoD pyramid (LoD 0 voxels), using the compact storage presented in the original work (Equation 17) that expresses the parameters in linear space, allowing the use of 1 byte per parameter, so each distribution takes 6 bytes of space.

$$\begin{aligned} \sigma_x &= \sqrt{S_{xx}}, & \sigma_y &= \sqrt{S_{yy}}, & \sigma_z &= \sqrt{S_{zz}}, \\ r_{xy} &= \frac{Sxy}{\sqrt{S_{xx}S_{yy}}}, & r_{xz} &= \frac{Sxz}{\sqrt{S_{xx}S_{zz}}}, & r_{yz} &= \frac{Syz}{\sqrt{S_{yy}S_{zz}}} \end{aligned} \tag{17}$$

For higher (coarser) levels we can compute new volumes of larger voxel blocks using SGGX interpolation [18] to aggregate voxels in a $2x2x2$ structure. However, joining all distributions into a single new SGGX distribution produces a loss of accuracy, specially with highly anisotropic distributions. This inaccuracy is increased with each interpolation, producing very isotropic results, that will yield rougher scattering at the rendering stage (See the naive SGGX results in our Supplementary Material).

For better quality, we can use more than a single SGGX distribution to represent this anisotropy, by referencing lower level distributions and using them together to sample the mixture of orientations. However, such approach increases exponentially the number of memory operations and samples in rendering time for coarse levels. Thus, we propose an aggregation method on top of the previous approach, merging similar distributions so we control the storage needed to a maximum amount of distributions per LoD block.

Our method retains the most relevant distributions, even if they encode perpendicular orientations. In our experiments, we obtained good results with a maximum of three (3) distributions per level, which retain information in three main axes, even if the data we want to fit is evenly distributed. For density and occupancy, we store the values computed as explained in Section 3.5.

4.4 LoD computation

Accurately representing a complex mix of orientations may require more SGGX functions than we expect to store in a volume. A simple workaround can be designed to reduce their number to a single, representative, SGGX, by sampling multiple times each original SGGX, concatenate their samples and generate the end SGGX using the fitting algorithm described in Section 4.1. However, as discussed previously, these distributions tend to be highly anisotropic, and such sampling produces a drastic information loss.

To overcome these problems, and to allow for a more fine-grained LoD calculation, we propose a simple aggregation method that progressively reduces the number of SGGXs used to represent the needed directional data, whilst preserving more high-frequency details than what could be accomplished using a simpler aggregation method. We draw inspiration from classical unsupervised learning algorithms in machine learning and pose this problem as a *hierarchical clustering* method [53]. In particular, we build a *dendrogram* of LoDs, by progressively reducing the original set of distributions, as we illustrate on rightmost part of Figure 1. Given a set of distributions $D_L = \{S_1, S_2, \dots, S_L\}$ for LoD L, we compute the next LoD $L-1$ simply by aggregating the two most similar SGGXs at L. This aggregation method preserves all the information present in the rest of SGGX distributions and loses the least amount of information, as the two most redundant SGGX are substituted by a single, representative one.

Instead of computing the similarity between two SGGX distributions by comparing their defining parameters (see equation 17), we compare the histograms of their samples. This way, we compute distance between SGGXs in terms of the underlying probability distributions they encode. More specifically, for each SGGX S_i at each L, we draw N samples, which we bin into a histogram $H(S_i)$. Using a probability distribution distance function $dist(d_1, d_2)$, we then compute the distance between each pair of SGGX at L. We obtain the two most similar distributions searching for the smallest distance between each pair of histograms, as described on equation 18:

$$\arg\min_{i,j} \; dist(H(S_i), H(S_j)), \quad where\, i \neq j; S_i, S_j \in D_L \tag{18}$$

As we use a symmetric distance function ($dist(d_1, d_2) = dist(d_2, d_1)$), this distance can be computed only once for each pair of distributions (eg

$\forall i > j; S_i, S_j \in D_L$), resulting in a strictly triangular matrix of distances, for which this minimum is more easily found. We then compute a new distribution by aggregating the samples of S_i and S_j, and fit their resulting histogram into a new SGGX S_n. The next LoD D_{L-1} will simply be D_L without S_i and S_j and with the new S_n, as in: $D_{L-1} = (D_L \setminus \{S_i, S_j\}) \cup S_n$. Running this algorithm until $L = 1$, we can build a hierarchical aggregation of LoDs, which provides fine-grained control over the trade-off between computational efficiency and information preservation.

Implementation details: We use $N = 5000$ samples for each S, and compute their histograms $H(S)$ using 5 bins in each spatial dimension. For comparing the histograms, we use the x *Wasserstein* distance, which is widely used in probability and machine learning algorithm [54, 55, 56].

5 Results

In this section, we show the capabilities of our method, in terms of voxelization computational cost, rendering quality, and quantitative comparisons with baselines.

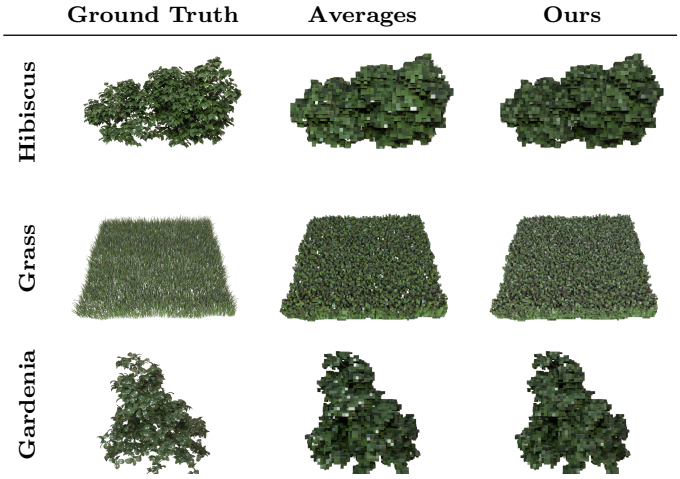

Fig. 2 Multiple comparisons for vegetation models from Hasselgen [40] dataset (Hibiscus and Gardenia) and grass model from free3d.com. We provide resolution data and ꟻLIP metrics for these scenes in our Supplementary Material.

5.1 Voxelization pipeline

We compare our voxelization pipeline to previous rasterized approaches such as Lopez-Moreno et al. [31]. We tested our algorithm with a fabric object with more than 20 M of nodes, with 60 spline subdivisions and a resolution of 65536 voxels per side. To test the voxelization pipeline separately, we down-sample the resolution to 8192 voxels by side, selecting the most probable tangent, normal, and a phase function weight with the corresponding identifiers.

We refer to Table 3 for voxelization times for three models of a fiber-level fabric patch: up to 160 million nodes were generated and voxelized within a minute in a low-end GPU card in most cases. Sample generations generation are faster in the raster approach. However, we achieve better times for the whole process. The raster approach is suitable for real time applications,

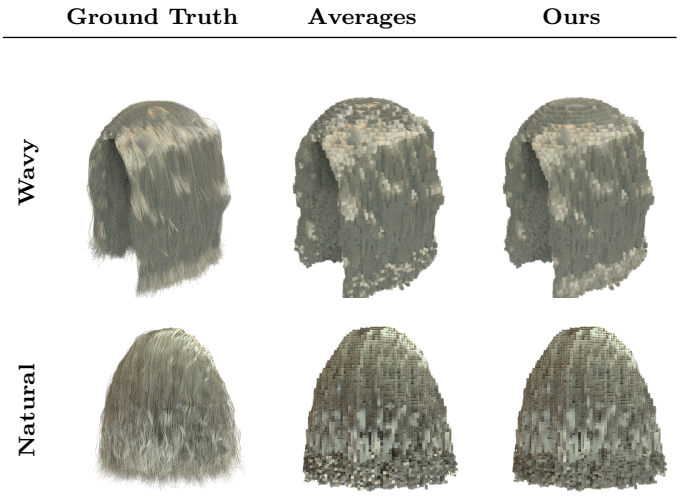

Fig. 3 Comparisons of our Level of Detail method against Naive downsampling for Yuksel's hair model. We translated the original files to splines to fit our pipeline and then proceed to voxelize them to generate a Ground Truth render from the original voxelized volume. We provide resolution data and ꟻLIP metrics for these scenes in our Supplementary Material.

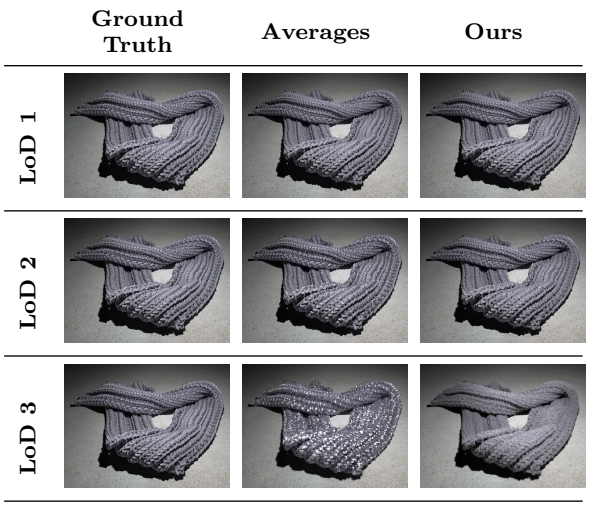

Fig. 4 Comparisons of renders at different voxelization LoDs of the Scarf volume [57] using Ground Truth data (left) and our hierarchical model (right) for tangent data.

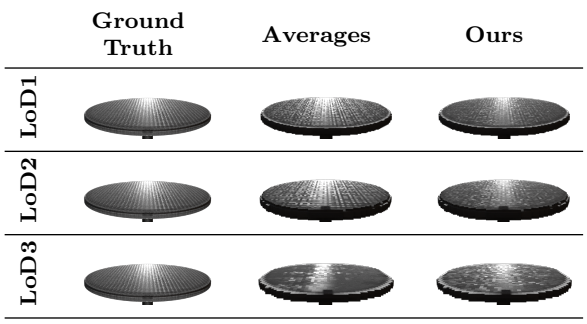

Fig. 5 Render comparison at multiple LoDs of an anisotropic steel table. Left: A three-level resolution voxelized model. Middle: LoD based on naïve average. Right: Our SGGX-H model. In the naïve approach, averages tangent orientations are collapsed into two almost perpendicular directions. Our method evaluates samples of mutliple distributions without averaging them, producing better results, even at coarse resolutions.

while this approach is better for general purposes, big models, and to improve accuracy and sampling control. The raster approach has a hardware limitation due to OpenGL architecture; it cannot render two frames at the same time, while CUDA is more flexible to parallelize the algorithm and to transfer data to and from CPU.

5.2 Rendering

To render our generated volumetric data we use a CPU-based volumetric path with Multiple Importance Sampling (MIS), testing each LoD using different techniques, and Ground Truth volume rendered at the original volume resolution. For all renders we use Woodcock's Delta Tracking technique [58] for transmittance estimation.

We show how our method behaves compared to the original data (GT) and naïve SGGX fitting for both a fiber-like material [29] using the scarf volume from Jakob et al. [57] (Figure 4); and by voxelizing mesh data along its normal data as orientations (Figures 5, 6).

We also voxelized and rendered vegetation (Figure 2) and hair (Figure 3) models, both known for their difficulty to accurately represent when doing Level of Detail.

To use multiple computed LoDs when rendering, different volumes could be loaded at the same time (one per LoD) and choose which volume to lookup for data at each voxel based in required LoD level. This level can be chosen using usual PathTracer MipMap selection [59] based on projected width on camera using the following metric:

$$level = MaxLevels - 1 + \log_2(width) \tag{19}$$

$$level_0 = \lceil level \rceil, \qquad p_{level_0} = level - level_0 \tag{20}$$

$$level_1 = \lfloor level \rfloor, \qquad p_{level_1} = level_1 - level \tag{21}$$

5.3 Quantitative results

In Figure 7, we show a quantitative comparison across different LoDs, on relevant metrics, including pixel-wise Mean Absolute Error (L1), as well as perceptual loss (LPIPS [60]), and render-aware metric ꟻLIP [61]. We measure the differences on the *Helmet scene* at a constant 256×256 resolution, with a mask applied to ignore the background. Across LoD levels, we plot the relative improvement from our hierarchical SGGX aggregation, with respect to the naive aggregation. As shown, our approach consistently outperforms the baseline across all levels of detail. Interestingly, these differences increase as the resolution decreases, particularly on the pixel-wise metrics, suggesting that our approach gains additional benefits as more orientations are aggregated. We show additional ꟻLIP metrics for other scenes in Table 2, as well as additional information of each LoD.

5.4 Limitations

Our pipeline exposes three main parameters to the user: **first level resolution**, **second level resolution** and **sample number**. While the former control the final resolution, choosing each one to better adjust to the sparsity of the input model while having enough GPU memory lies on the user. Specifically, combining an elevated number of samples with a high enough resolution can lead also to out of memory problems. While users can control this parameters freely, and desired final resolutions can be achieved by different combinations of resolution parameters, finding an heuristic or metric that suggest suitable combinations of input parameters for a given model would be desirable.

Additionally, for models where one dimension is almost flat (compared to the largest one), regular cubic voxel shape requires a large increase of the global resolution so the smaller dimension gets enough nodes to be voxelized properly. We refer the reader to the Supplementary Material where we discuss a fabric defined by procedural yarns and explicit fiber distributions showing this problematic.

6 Conclusions and Future Work

In this paper, we have introduced a novel voxelization method, optimized to deal with sparse microgeometry while maintaining memory efficiency. In addition to being able to efficiently handle sparse data, our method allows voxelization of complex data along density. Furthermore, it can fit distributions to generate lower resolution volumes, suitable for LoD, that accurately represent not only *ocupancy* and *optical density* but also orientation data such as normals or tangents. Our work can be extended in several directions.

In section 4.2, we described the LoD approach for density data. Due to its limited per-voxel accuracy with complex geometries, this method could be improved by exploring novel neural or statistical representations. These representations could encode aggregated density, allowing for sampling and combination in a way similar to our solution for orientation data. Additionally, such approach should help removing artifacts like the ones present when voxelizing complex fabrics.

Regarding distribution merging, method uses a hierarchical approach and aims to reduce the amount of SGGXs stored per node, however, we still need to re–fit some of them, possibly reducing the accuracy of the orientation distribution for that node.

Ideally, we could use all parent nodes to generate a mix of distributions on–the–fly that would accurately represent the full orientation distribution for a selected child node. However, this comes with the price of exploring, storing and merging all required parent nodes recursively. Future research on how to improve such approach could lead to a higher accuracy on sparse LoDs (hence rendering quality), while saving over all space when storing all LoDs together, avoiding orientation data duplication.

Also, the possible benefits of alternative GPU-CPU data flows could be analyzed. For instance, in our implementation we fit the leaf node SGGX in GPU, but then we generate an histogram using the SGGX itself in CPU to build the SGGX-H representation. While this is necessary for aggregated SGGX distributions, for leaf nodes we could use the original GPU-computed histogram instead of sampling, at the cost of moving that data from GPU to CPU. Newer GPU features could be explored in the future to try to overcome these issues.

Scene (Resolution)	Level of Detail (Resolution)	Method		
		Naive	SGGX	Ours
Table (2048)	1 (256)	0.095	0.094	0.085
	2 (128)	0.105	0.103	0.094
	3 (64)	0.111	0.118	0.101
Helmet (512)	1 (64)	0.100	0.103	0.098
	2 (32)	0.163	0.154	0.151
	3 (16)	0.201	0.181	0.179
Scarf (≈1024)	1 (≈800)	0.0642	0.091	0.0641
	2 (≈400)	0.103	0.133	0.101
	3 (≈200)	0.281	0.284	0.202

Table 2 While a straightforward SGGX fit sometimes fail to fully capture aggregated our method (SGG-H) scores improved (ꟻLIP) metrics compared to a Naive approach for all scenes, at different Levels of Detail.

Declarations

Funding

This publication is part of the project TaiLOR, CPP2021-008842 funded by MCIN/AEI/10.13039/501100011033 and the NextGenerationEU / PRTR programs.

References

1. J. Amanatides and A. Woo, "A fast voxel traversal algorithm for ray tracing," in *Proceedings of Eurographics*, pp. 3–10, Eurographics Association, 1987.
2. E. P. Lafortune and Y. D. Willems, "Rendering participating media with bidirectional path tracing," in *Rendering Techniques (Proceedings of the Eurographics Workshop on Rendering)*, (Vienna), pp. 91–100, Springer-Verlag, June 1996.
3. M. Aleksandrov, S. Zlatanova, L. Kimmel, J. Barton, and B. Gorte, "Voxel-based visibility analysis for safety assessment of urban environments," *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 4, pp. 11–17, 2019.
4. 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," in *European Conference on Computer Vision (ECCV)*, 2020.
5. K. Zhang, G. Riegler, N. Snavely, and V. Koltun, "Nerf++: Analyzing and improving neural radiance fields," *arXiv:2010.07492 [cs.CV]*, 2020.
6. B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3D gaussian splatting for real-time radiance field rendering," *ACM Transactions on Graphics*, vol. 42, July 2023.
7. G. Wu, T. Yi, J. Fang, L. Xie, X. Zhang, W. Wei, W. Liu, Q. Tian, and X. Wang, "4D gaussian splatting for real-time dynamic scene rendering," in *IEEE/CVF International Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 20310–20320, June 2024.

8. N. Moenne-Loccoz, A. Mirzaei, O. Perel, R. de Lutio, J. M. Esturo, G. State, S. Fidler, N. Sharp, and Z. Gojcic, "3D gaussian ray tracing: Fast tracing of particle scenes," *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 2024.
9. J. Condor, S. Speierer, L. Bode, A. Bozic, S. Green, P. Didyk, and A. Jarabo, "Don't splat your gaussians: Volumetric ray-traced primitives for modeling and rendering scattering and emissive media," *ACM Transactions on Graphics*, vol. 44, Feb. 2025.
10. S. Govindarajan, D. Rebain, K. M. Yi, and A. Tagliasacchi, "Radiant foam: Real-time differentiable ray tracing," *arXiv:2502.01157 [cs.CV]*, 2025.
11. C. Sun, J. Choe, C. Loop, W.-C. Ma, and Y.-C. F. Wang, "Sparse voxels rasterization: Real-time high-fidelity radiance field rendering," *arXiv:2412.04459 [cs.CV]*, 2024.
12. Y. Wang, L. Chai, X. Luo, M. Niemeyer, M. Lagunas, S. Lombardi, S. Tang, and T. Sun, "Splatvoxel: History-aware novel view streaming without temporal training," *arXiv:2503.14698 [cs.CV]*, 2025.
13. M. Aleksandrov, S. Zlatanova, and D. J. Heslop, "Voxelisation algorithms and data structures: A review," *Sensors*, vol. 21, no. 8241, 2021.
14. M. Schwarz and H.-P. Seidel, "Fast parallel surface and solid voxelization on GPUs," *ACM Transactions on Graphics*, vol. 29, no. 6, pp. 1–10, 2010.
15. Z. Dong, W. Chen, H. Bao, H. Zhang, and Q. Peng, "Real-time voxelization for complex polygonal models," in *Pacific Conference on Computer Graphics and Applications*, (Washington, DC, USA), pp. 43–50, IEEE Computer Society, 2004.
16. C. Loop, C. Zhang, and Z. Zhang, "Real-time high-resolution sparse voxelization with application to image-based modeling," in *Proceedings of High Performance Graphics*, (New York, NY, USA), pp. 73–79, ACM Press, July 2013.
17. K. Museth, "VDB: High-resolution sparse volumes with dynamic topology," *ACM Transactions on Graphics*, vol. 32, pp. 27:1–27:22, July 2013.
18. E. Heitz, J. Dupuy, C. Crassin, and C. Dachsbacher, "The SGGX microflake distribution," *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, vol. 34, pp. 48:1–48:11, July 2015.
19. K. Museth, J. Lait, J. Johanson, J. Budsberg, R. Henderson, M. Alden, P. Cucka, D. Hill, and A. Pearce, "OpenVDB: An open-source data structure and toolkit for high-resolution volumes," in *ACM SIGGRAPH Courses*, ACM Press, 2013.
20. Academy Software Foundation, "OpenVDB," Oct. 2024.
21. K. Museth, "NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation," in *ACM SIGGRAPH Talks*, (New York, NY, USA), ACM Press, 2021.
22. Nvidia Corporation, "NanoVDB," Oct. 2024.
23. D. Cohen-Or and A. Kaufman, "3D line voxelization and connectivity control," *IEEE Computer Graphics & Applications*, vol. 17, no. 6, pp. 80–87, 1997.
24. F. D. IX and A. Kaufman, "Incremental triangle voxelization," in *Proceedings of Graphics Interface*, pp. 205–212, 2000.
25. A. Kaufman and E. Shimony, "3D scan-conversion algorithms for voxel-based graphics," in *Proceedings of the Workshop on Interactive 3D Graphics*, pp. 45–75, 1987.
26. E. Eisemann and X. Décoret, "Fast scene voxelization and applications," in *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pp. 71–78, 2006.
27. C. Crassin, *GigaVoxels: A Voxel-Based Rendering Pipeline for Efficient Exploration of Large and Detailed Scenes*. PhD thesis, Université de Grenoble, July 2011.
28. C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, "Interactive indirect illumination using voxel cone tracing," *Computer Graphics Forum (Proceedings of Pacific Graphics)*, vol. 30, pp. 207–207, Sept. 2011.
29. P. Khungurn, D. Schroeder, S. Zhao, K. Bala, and S. Marschner, "Matching real fabrics with micro-appearance models," *ACM Transactions on Graphics*, vol. 35, Dec. 2016.
30. C. Castillo, J. López-Moreno, and C. Aliaga, "Recent advances in fabric appearance reproduction," *Computers & Graphics*, vol. 84, pp. 103–121, Nov. 2019.
31. J. Lopez-Moreno, D. Miraut, G. Cirio, and M. A. Otaduy, "Sparse GPU Voxelization of Yarn-Level Cloth: Sparse GPU Voxelization of Yarn-Level Cloth," *Computer Graphics Forum*, vol. 36, pp. 22–34, Jan. 2017.
32. Y.-Q. Xu, Y. Chen, S. Lin, H. Zhong, E. Wu, B. Guo, and H.-Y. Shum, "Photorealistic rendering of knitwear using the lumislice," in *Annual Conference Series (Proceedings of SIGGRAPH)*, (New York, NY, USA), p. 391–398, ACM Press, 2001.
33. L. Williams, "Pyramidal parametrics," *Computer Graphics (Proceedings of SIGGRAPH)*, vol. 17, pp. 1–11, July 1983.
34. M. Olano and D. Baker, "LEAN mapping," in *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pp. 181–188, 2010.

35. A. Gauthier, R. Faury, J. Levallois, T. Thonat, J.-M. Thiery, and T. Boubekeur, "MIPNet: Neural normal-to-anisotropic-roughness MIP mapping," *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, vol. 41, pp. 246:1–246:12, Nov. 2022.
36. M. Garland and P. S. Heckbert, "Surface simplification using quadric error metrics," in *Annual Conference Series (Proceedings of SIGGRAPH)*, pp. 209–216, ACM Press, 1997.
37. W. J. Schroeder, "A topology modifying progressive decimation algorithm," in *Proceedings. Visualization'97 (Cat. No. 97CB36155)*, pp. 205–212, IEEE Computer Society, 1997.
38. K.-L. Low and T.-S. Tan, "Model simplification using vertex-clustering," in *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, pp. 75–ff, 1997.
39. S. Laine, J. Hellsten, T. Karras, Y. Seol, J. Lehtinen, and T. Aila, "Modular primitives for high-performance differentiable rendering," *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, vol. 39, pp. 194:1–194:14, Nov. 2020.
40. J. Hasselgren, J. Munkberg, J. Lehtinen, M. Aittala, and S. Laine, "Appearance-driven automatic 3D model simplification.," in *Proceedings of the Eurographics Symposium on Rendering (EGSR)*, pp. 85–97, 2021.
41. J. Haydel, C. Yuksel, and L. Seiler, "Locally-adaptive level-of-detail for hardware-accelerated ray tracing," *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, vol. 42, pp. 196:1–196:15, Dec. 2023.
42. P. Weier, T. Zirr, A. Kaplanyan, L.-Q. Yan, and P. Slusallek, "Neural prefiltering for correlation-aware levels of detail," *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, vol. 42, pp. 78:1–78:16, July 2023.
43. S. Zhao, L. Wu, F. Durand, and R. Ramamoorthi, "Downsampling scattering parameters for rendering anisotropic media," *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, vol. 35, pp. 166:1–166:11, Nov. 2016.
44. G. Loubet and F. Neyret, "A new microflake model with microscopic self-shadowing for accurate volume downsampling," *Computer Graphics Forum (Proceedings of Eurographics)*, vol. 37, pp. 111–121, May 2018.
45. Y. Zhou, T. Huang, R. Ramamoorthi, P. Sen, and L.-Q. Yan, "Appearance-preserving scene aggregation for level-of-detail rendering," *ACM Transactions on Graphics*, vol. 44, Jan. 2025.
46. G. Loubet and F. Neyret, "Hybrid mesh-volume lods for all-scale pre-filtering of complex 3d assets," *Computer Graphics Forum*, vol. 36, no. 2, pp. 431–442, 2017.
47. D. Vicini, W. Jakob, and A. Kaplanyan, "A non-exponential transmittance model for volumetric scene representations," *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, vol. 40, pp. 136:1–136:16, July 2021.
48. T. Takikawa, J. Litalien, K. Yin, K. Kreis, C. Loop, D. Nowrouzezahrai, A. Jacobson, M. McGuire, and S. Fidler, "Neural geometric level of detail: Real-time rendering with implicit 3D shapes," *IEEE/CVF International Conference on Computer Vision and Pattern Recognition (CVPR)*, Jan. 2021.
49. E. Catmull and R. Rom, "A class of local interpolating splines," in *Computer Aided Geometric Design* (R. E. Barnhill and R. F. Riesenfeld, eds.), pp. 317–326, PubAP, 1974.
50. E. Heitz, "A Low-Distortion Map Between Triangle and Square." working paper or preprint, 2019.
51. B. Bitterli, S. Ravichandran, T. Müller, M. Wrenninge, J. Novák, S. Marschner, and W. Jarosz, "A radiative transfer framework for non-exponential media," *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, vol. 37, pp. 225:1–225:17, Nov. 2018.
52. J. Guo, Y. Chen, B. Hu, L.-Q. Yan, Y. Guo, and Y. Liu, "Fractional Gaussian fields for modeling and rendering of spatially-correlated media," *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, vol. 38, pp. 45:1–45:13, July 2019.
53. F. Murtagh and P. Contreras, "Algorithms for hierarchical clustering: An overview," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 2, no. 1, pp. 86–97, 2012.
54. SS. Vallender, "Calculation of the Wasserstein distance between probability distributions on the line," *Theory of Probability & Its Applications*, vol. 18, no. 4, pp. 784–786, 1974.
55. E. Heitz, K. Vanhoey, T. Chambon, and L. Belcour, "A sliced wasserstein loss for neural texture synthesis," in *IEEE/CVF International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.
56. I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved training of wasserstein gans," *arXiv:1704.00028*, 2017.
57. W. Jakob, A. Arbree, J. T. Moon, K. Bala, and S. Marschner, "A radiative transfer framework for rendering materials with anisotropic structure," *ACM Transactions on Graphics (Proceedings of*

SIGGRAPH), vol. 29, pp. 53:1–53:13, July 2010.
58. E. R. Woodcock, T. Murphy, P. J. Hemmings, and T. C. Longworth, "Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry," in *Applications of Computing Methods to Reactor Problems*, Argonne National Laboratory, 1965.
59. M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. Cambridge, MA: Morgan Kaufmann, 3 ed., 2016.
60. R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang, "The unreasonable effectiveness of deep features as a perceptual metric," in *IEEE/CVF International Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 586–595, 2018.
61. P. Andersson, J. Nilsson, T. Akenine-Möller, M. Oskarsson, K. Åström, and M. D. Fairchild, "ꟻLIP: A Difference Evaluator for Alternating Images," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 3, no. 2, pp. 15:1–15:23, 2020.

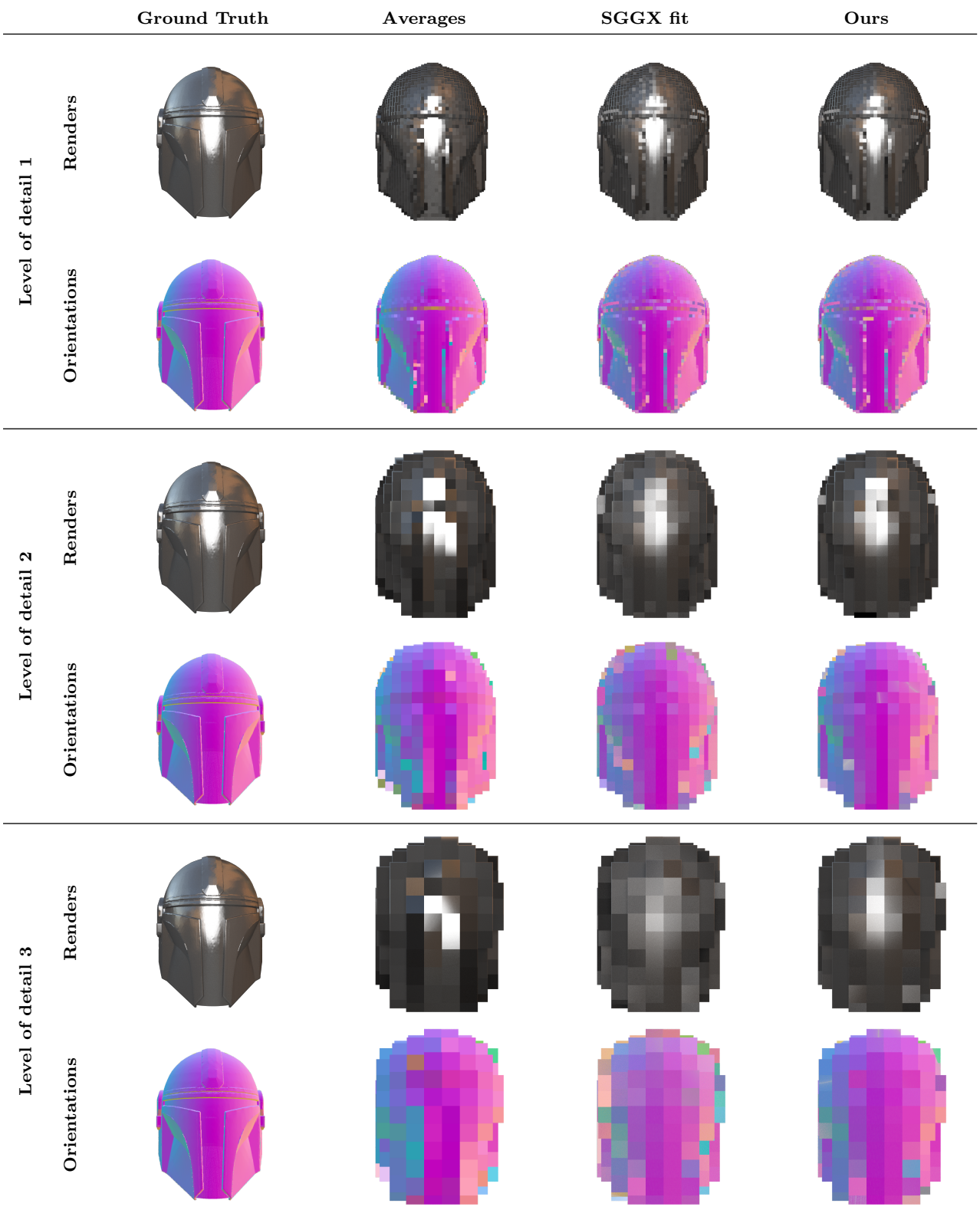

Fig. 6 Comparisons of renders at different voxelization LoDs of a helmet mesh using Ground Truth data (left), naive fit to SGGX model (center) and our hierarchical model (right) for orientation storage. We also show averaged normal per pixel. Note, however, that for both SGGX and SGGX-H we do not evaluate this averaged normal while rendering, but sample normals from the distribution for each shading evaluation.

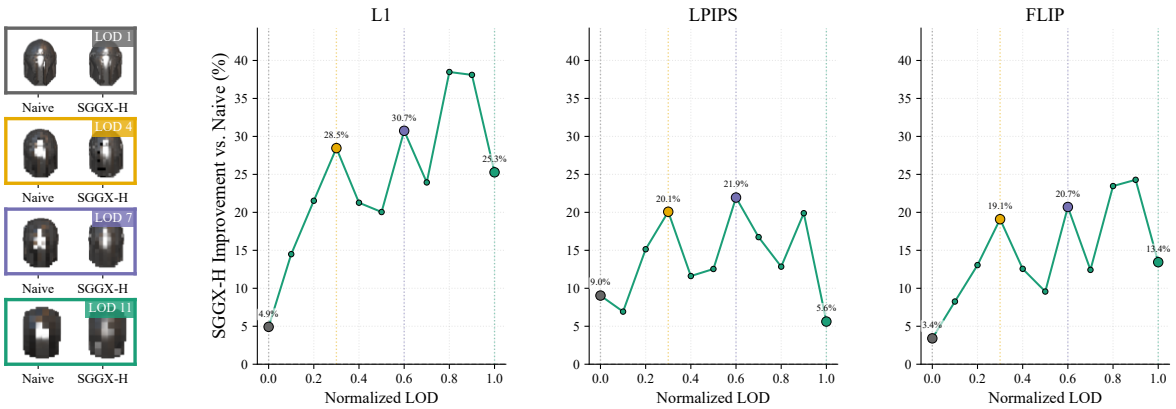

Fig. 7 Comparison of SGGX-H relative improvement over Naive rendering for the Helmet model. Left column: Side-by-side renderings for selected Levels of Detail (LoDs 1, 4, 7, 11 indicated by colored borders). Right columns: Plots showing the percentage improvement of SGGX-H compared to Naive across normalized LoD for L1, LPIPS, and ꟻLIP error metrics. The x-axis represents normalized LOD, where 0 corresponds to the lowest detail (highest LoD number) and 1 corresponds to the highest detail (lowest LoD number). Positive percentages indicate SGGX-H yields lower error than the Naive approach for the respective metric.

Scene	Parameters				Time (s)		
	Nodes	Fibers	Subdivisions	Resolution (1st x 2nd)	Raster	Ours	w/ Histogram
Fabric flat	8,359,496	150	60	32 x 32	81.117	3.450	8.813
				32 x 64	15.677	6.058	9.151
				32 x 128	41.996	22.339	9.396
Fabric folded	8,227,286			32 x 32	6.649	3.521	7.987
				32 x 64	14.502	6.156	8.2062
				32 x 128	30.796	21.152	58.188
Fabric Hanging	16,3237,089			32 x 32	Out of memory	20.485	58.188
				32 x 64		22.346	66.106
				32 x 128		36.113	48.643

Table 3 Using our voxelizer in CUDA, we measured the times to replicate the behavior of previous approaches. Measured time include loading stage, voxelization, and exportation to OpenVDB format. We compare our voxelization times with and without histogram computation against a raster based implementation using the same GPU (NVIDIA GeForce - GTX 1080 Ti). Visualization of the fabrics voxelized for these test are shown in the Supplementary Material.