Volume xx (200y), Number z, pp. 1–14

Sparse GPU Voxelization of Yarn-Level Cloth

Jorge Lopez-Moreno, David Miraut, Gabriel Cirio and Miguel A. Otaduy

Universidad Rey Juan Carlos, Madrid (Spain)

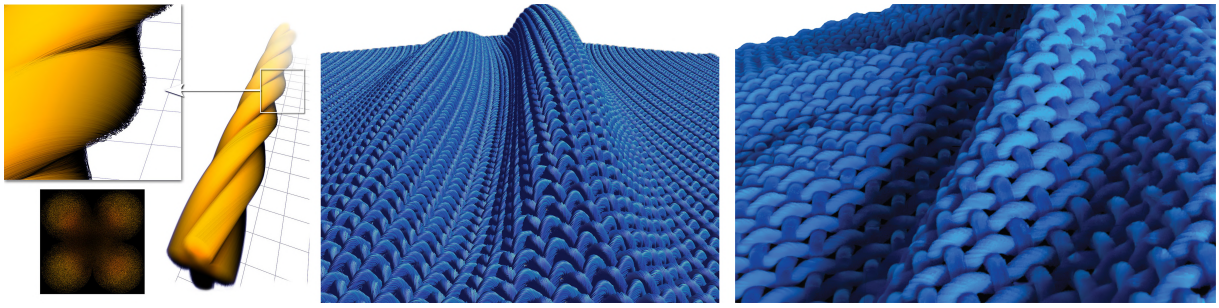

Figure 1: *Left: Our voxelization algorithm uses as input a cross-section slice of yarns, which stores fiber albedo and density. The voxelization works by rasterizing slice instances to a sparse, high-resolution 3D texture. Middle: With small changes to the voxelization pipeline, we obtain visualization for interactive preview and parameter setting. Right: The output of our voxelization is used as input for high-quality rendering with volumetric path tracing.*

Abstract

Most popular methods in cloth rendering rely on volumetric data in order to model complex optical phenomena such as sub-surface scattering. These approaches are able to produce very realistic illumination results, but their volumetric representations are costly to compute and render, forfeiting any interactive feedback. In this paper, we introduce a method based on the GPU for voxelization and visualization, suitable for both interactive and offline rendering.

Recent features in the OpenGL model, like the ability to dynamically address arbitrary buffers and allocate bindless textures, are combined into our pipeline to interactively voxelize millions of polygons into a set of large 3D textures ($> 10^9$ elements), generating a volume with sub-voxel accuracy which is suitable even for high-density woven cloth such as linen.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

1. Introduction

Cloth simulation and rendering is an active research field. In the last two decades we have witnessed how simulation models have evolved from mass-spring networks on triangle meshes [BHW94, Pro95] or finite-element discretizations of continuum models [EKS03], to the computation of highly detailed physical interactions at the yarn level [KJM08, MBCN09, KJM10, CLMMO14, CLMO15].

This level of detail has created a demand for render methods which can leverage the available data and deal with a volumetric representation of yarns and their constituent fibers. While the complexity of the cloth reflectance representation has increased over the years, even including

microCT-captured three-dimensional models of the fibers [ZJMB12], these approaches are limited to surface-based representations of cloth objects and/or stochastic replication of texture tiles.

Yarns, as rendering primitives, were introduced by Xu et al. [XCL*01]. By partially relying on OpenGL rasterization, they obtained very compelling volumetric renderings in less than an hour of computation. Their representation was used as input by Jakob et al. [JAM*10] for more accurate and realistic rendering with volumetric path tracing and the micro-flakes model. Volumetric yarn-based models offer high realism, but they come with additional difficulties. One is that simply selecting the rendering settings is a very time-consuming task. The artist must iterate several times the operations of generating the volumetric representation and rendering the scene, both of which are costly.

In this paper, we present a fast, massively parallel, sparse, high-resolution voxelization algorithm. It generates the volumetric datasets for full animations in a short time, thereby enabling agile adjustment of rendering settings, prior to the actual full-quality offline rendering. Thanks to the sparse volumetric representation, it also optimizes offline rendering costs. Our voxelization algorithm performs a fast 3D rasterization of millions of yarn slices, as shown in Figure 1-left, by efficiently managing instantiation and asynchronous rendering. Our algorithm also exploits modern GPU features, in particular a direct combination of image unit access and bindless images to handle the maximum memory available in the card at any time.

In addition, and as a complementary tool to voxelization, we present an interactive visualization algorithm. This algorithm is designed with just minor modifications to the voxelization pipeline, as shown in Figure 2. The visualization pipeline rasterizes the output to a view-dependent 2D buffer instead of 3D, by means of fast, parallel sorting of slices. The visualization algorithm also incorporates Kajiya-Kay shading and shadow mapping. Thanks to interactive visualization of the voxelization results, the artist may quickly adjust geometric and visual parameters of yarns. Figure 1 compares output results of interactive preview visualization and offline high-quality volumetric path tracing.

This paper is an extension of our recently published work [LMCMO14]. Here, we add the sparse representation, the use of bindless image textures, more efficient GPU slice sorting, and higher quality rendering for interactive visualization.

2. Related Work

2.1. Yarn Rendering

Our method relies on a similar representation to Lumislice, introduced by Xu et al. [XCL*01], which modeled a single yarn by a set of cross-section slices. At each slice, fine-scale phenomena such as occlusion, shadowing and multiple scattering were described at the fiber level. The use of slices as atomic units brought two major advantages: efficient visibility computation and data re-use. Instead of ray tracing a full volumetric model, they pre-computed a 4D BRDF per voxel (VRF) for a one-voxel-wide slice. They stored the precomputed result in a table, and at run-time they used hardware-based transparency to handle visibility by overlaying all the slices in alpha-blending mode. In their work, Xu et al. extended the idea of using repetitive structures in volumetric rendering presented by Meyer and Neyret [MMN98]. In their approach, slices have access to nearby geometry (yarn segments) both for VRF pre-computation and slice rendering on the CPU, and only transparency rasterization is executed on GPU hardware for the final yarn composition. However, there are some limitations even in the Lumislice model: first, the VRF is limited to diffuse reflectance, and second, while intra-slice light interactions can be pre-computed and stored, inter-slice interactions are computed by considering a whole (undefined) stack of surrounding slices in order to account for attenuation and scattering from all directions. This approximation is inaccurate at the fine scale of yarns, as the actual yarns may not obey the shape used for pre-computation.

More recently, Jakob et al. [JAM*10] introduced a radiative transfer framework to render anisotropic scattering materials (such as yarns). This model, called micro-flakes, is composed of two-sided specularly reflecting flakes oriented according to a known directional distribution (hence requiring per-voxel orientation and density values). The output of our voxelization algorithm fits the representation used by Jakob et al. for high-quality rendering. For voxelization, and inspired by the work of Xu et al. [XCL*01], we use slices as atomic primitives, but with a different data structure. Our yarn representation avoids assumptions regarding the environment of the slice, and limits the slice pre-computation to locally available data: albedo, occlusion in the slice plane and derivative of fiber orientation (see Section 3.3). This information is used for both real-time shading and offline rendering.

Hair fibers can be regarded as rough dielectric cylinders, and present a structure similar to yarn fibers. The seminal work by Kajiya and Kay [KK89] was further improved by Goldmand [Gol97] to include translucency. Furthermore, the physically-based model by Marschner et al. [MJC*03] and its energy-conserving version by d'Eon et al. [dFH*11], set the basis for the current state of the art in hair and fur rendering.

2.2. Voxelization

Pantaleoni [Pan11] showed the potential of parallel GPU computation to voxelize complex 3D models in a few milliseconds with voxelPipe, a CUDA-based voxelization pipeline. Furthermore, Crassin and Green [CG12] improved these results by relying exclusively on the OpenGL pipeline,

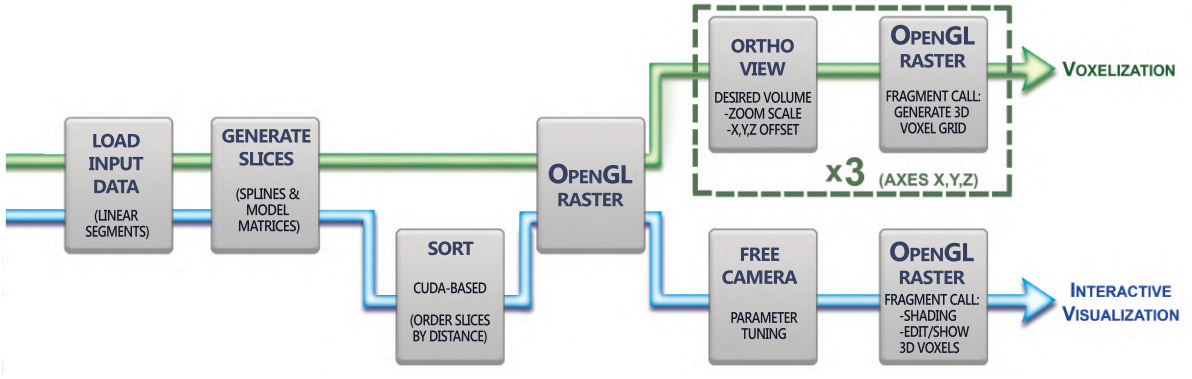

Figure 2: *Overview of our yarn-level cloth voxelization and visualization pipeline. In green, the voxelization workflow, which instantiates and rasterizes yarn slices to a sparse 3D grid. Rasterization is executed using orthographic projection from three orthogonal views to eliminate sampling artifacts. The voxelization workflow is fully automatic and generates the files required for offline rendering. In blue, the interactive visualization workflow, which rasterizes the slices to the frame buffer and incorporates an efficient slice sorting step for correct view-dependent transparency. The interactive visualization may display shaded yarns or additional slice and voxel information, to allow artists to interactively adjust geometric and visual settings.*

thanks to the recent Shader Model 5 which allows direct video memory access and storage from shaders. In their work they addresed two problems: the accuracy of geometry representation due to rasterization issues, and augmenting the storage capabilities of voxel grids with sparse octree representations. Our work addresses additional rasterization problems and targets models and textures at least one order of magnitude larger. Moreover, although we rely on the same OpenGL pipeline, our approach follows a different path in the GPU architecture. Instead of vertex shaders and buffer objects, we use a direct combination of image units access (*imageLoad-Store*) and bindless images to handle the maximum memory available in the card at any time. To our knowledge, this is a novel (if not the first) example of such coupling. Furthermore, the recent release of OpenGL 4.5 points towards a future of bindless textures and arbitrary memory access from shaders.

Sparse volume voxelization for data such as fluids, fog, clouds, or cloth has become a rising trend in the industry. Dreamworks has recently realeased openVDB, a format which has been quickly adopted by all the major rendering companies. With openVDB, Museth proposes a sparse hierarchical grid structure for the discretization of sparse, dynamic volumes [Mus13]. In a similar spirit, we have implemented a two-level hierarchical grid within our GPU framework which can be easily extended to more complex structures.

3. Sparse Voxelization Pipeline

The voxelization pipeline, shown in green in Figure 2, takes as input for each yarn a list of 3D points representing the centerline of the yarn, plus one cross-section slice storing the distribution of fiber density. To model the full yarn, we first compute a smooth centerline using Hermite splines, as described in detail in Section 3.1. Then, we sweep and twist instances of the slice along this smooth centerline, as shown in Figure 3 and described in detail in Section 3.2. The size of the slice is given by the diameter of the yarn, and the separation between consecutive slices is set based on requirements of the scene.

Our yarn voxelization and visualization approach adapts previous slice models to a massively parallel and GPU-friendly algorithm, which allows handling millions of slices efficiently. This is achieved by instantiating and rendering asynchronously the slices, and the challenge is to rasterize each slice independently, without sharing any information with its neighbors, as described in Section 3.3.

The output of the voxelization pipeline is a 3D grid where each voxel stores the density and orientation of yarn fibers. To exploit the 2D nature of cloth and avoid using a memory-intensive 3D regular grid, we perform a sparse voxelization using a two-level grid as described in Section 3.4. In addition, we exploit modern OpenGL features to enjoy fast and arbitrary access to texture memory from fragment shaders.

3.1. Input Data and Smooth Interpolation

Most natural fibers are only a few centimeters long. In order to obtain longer yarns, fibers are interlocked through torsion, with the resulting threads exhibiting different appearances depending on the fiber distribution and torsion degree. Threads do not bend sharply when they touch other threads; instead, their shape follows a smooth curve. Most yarn simulation models [KJM08, CLMMO14] discretize thread positions only at contact points and model boundaries. Our volu-

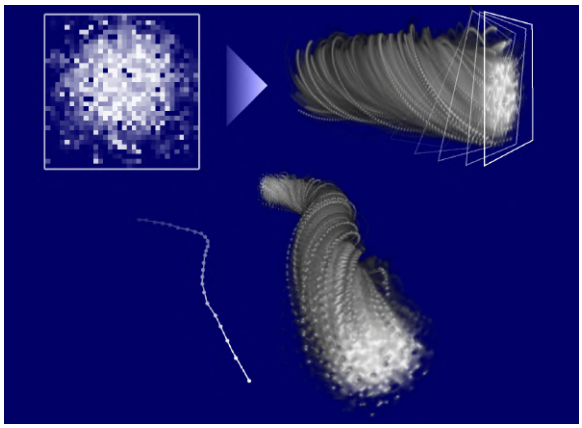

Figure 3: *Top left: Density distribution of fibers at a given slice, stored as a texture. Top Right: A stack of rotated slices conform a single thread section. Bottom: A path defines the position of these slices*

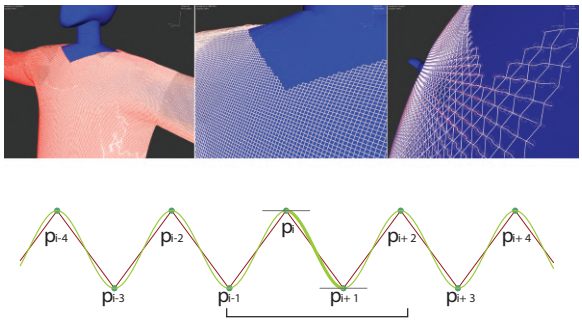

Figure 4: *Top Row: From left to right, increasing zoom levels of the input representation. Yarns are sampled at crossing points and represented as piecewise linear curves. Bottom row: A classic Catmull-Rom spline (green). The control points are interpolated to produce a smooth spline, which passes through each knot in a direction parallel to the line between the adjacent points (grey). The jagged brown line represents the input piecewise linear yarn.*

metric representation requires a dense and smooth sampling of the yarn's centerline, and the resulting samples are used as locations for the placement of slice billboards. Therefore, given the simulation output, intermediate positions must be devised in order to reproduce a detailed yarn model.

Among the wide range of strategies for curve parametrization, we have chosen classic Catmull-Rom splines [CR74] because of their algorithmic simplicity, low computational cost, and suitability for our input control points. We resample the smooth, curved yarn representation each time the model geometry changes, but not under camera or illumination modifications. It takes less than two seconds on the CPU to resample our largest models (with up to 35 million slices). See Table 1 for initialization times, including memory copy to the GPU and voxel volume initialization.

Catmull-Rom curves are cubic Hermite splines, so each portion of the curve traces a third-degree polynomial specified by its values and first derivatives at the end points of the corresponding domain interval. For a given yarn, the corresponding sampling points $\{P_1 P_2 \dots P_n\}$ are organized in sequential subsets, each composed of a pair of contact points (which are the result of the yarn simulation) . These subsets contain two knots (P_i and P_{i+1}) and two extra samples (P_{i-1} and P_{i+2}), which are used as a key to compute the curve tangents at knots (see Figure 4). By setting the consecutive yarn simulation points as control knots, we ensure that the yarn trajectory will be exact and consistent with the simulation at those points, while locally interpolating a smooth curve in between.

The detail within each segment can be tuned by evaluating more or less $t \in [0,1]$ parameter values in the curve function [Fol96]. For interactive preview visualization, the sampling density can be adjusted to meet the desired frame rate and/or visual quality. For offline high-quality visualization, we discuss sampling criteria in Section 5. We do not apply more sophisticated Catmull-Rom schemes, like chordal or centripetal implementations [YSK09], because our sampling does not produce loops or self-intersections which would be problematic for the classic algorithm.

3.2. Placing Yarn Slices

In our method, the slice is represented by a quad composed of two triangles and a texture with the albedo values and the fiber density distribution stored in the alpha channel. In order to reduce computation costs during voxelization, the different slices are implemented as instances of a standard slice, with the same geometry and textures but different attributes such as position and orientation. Albedo and fiber density distribution could be dynamic attributes, but this is left for future work.

For each sample in the spline centerline, we must compute the model matrix for the corresponding slice. We define the forward vector as the difference between the position of the current slice and the following one. To define the up vector, we initialize a vector perpendicular to the forward vector at the beginning of each yarn, and rotate this vector w.r.t. the forward vector incrementally along the yarn. In this way, we capture the characteristic twist of the thread, which is set as an input parameter to the algorithm. We map all the model matrices to a buffer (GL_ARRAY_BUFFER) so they can be included in a vertex attribute array for rendering (GL function *glDrawElementsInstanced*). In order to rasterize (and thus voxelize) the yarns, these model matrices are combined with view and projection matrices to project each slice quad onto the screen.

3.3. Voxelization of Density and Orientation

In order to fill with density and orientation values a voxel at a given position in world coordinates, one option is to access the point corresponding to those world coordinates in the geometric model of the yarn, which is composed of multiple textured quads as seen before. In our method, based on rasterization, the order is reversed: each fragment-pixel computes its world coordinates and stores the associated texel data at the corresponding voxel.

With our model, density is voxelized in the following way: The vertex shader has access to the modelview matrix of the slice, and it passes down to each fragment shader its interpolated position and the global orientation of the slice. The fragment shader receives subsequently the position in world coordinates, it can access the fiber density distribution stored in a texture, and thus writes the density value at the appropriate position.

In addition to fiber density, we also voxelize fiber orientation. Given the position of a fiber in two consecutive slices, P_{i-1} and P_i, fiber orientation can be computed by normalizing the difference vector between these two positions. In previous CPU-based yarn voxelization methods [XCL*01], it was possible to access adjacent slices to query for fiber positions, but in our parallel GPU-based approach slices are processed out of order. Instead, we compute fiber orientation procedurally based on the inter-slice twist angle α. Furthermore, assuming a constant twist angle for all yarns, we precompute fiber position differences in the local reference system of the standard slice, and use this information during voxelization to evaluate fiber orientation.

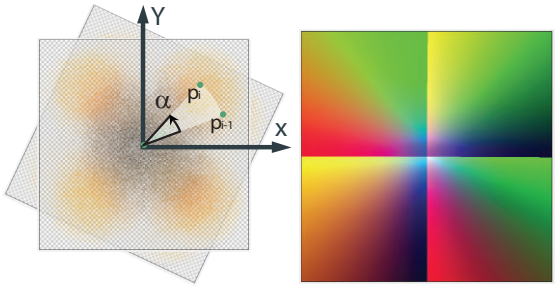

Figure 5: *Left: The orientation of fibers is defined by the position difference $P_i - P_{i-1}$ of the same fiber in two consecutive slices. If slices are sampled densely, they can be considered parallel, and we express these positions in the local reference frame of the slice. The position difference is a function of the inter-yarn twist α. Right: For a constant inter-yarn twist, we precompute fiber position differences for the standard slice and store them in a texture. Discontinuities are due to color codification of the orientation vector.*

If slices are placed densely along the yarn, we can assume that the current and previous slices are almost parallel to each other. Then, we can express fiber positions on the plane orthogonal to the yarn using a common reference frame as shown in Figure 5-left. We define as $P_i = (X_i, Y_i)$ the position of the current voxel in the local reference frame of the slice. And we define as $P_{i-1} = (X_{i-1}, Y_{i-1})$ the position in the previous slice of the same fiber that passes through P_i. The difference between these two positions, together with the centerline of the yarn, defines fiber orientation. The position in the previous slice, P_{i-1}, can be computed by rotating P_i an angle of $-\alpha$, therefore their difference is based solely on the yarn twist α, and can be computed as follows:

$$\begin{pmatrix} \Delta X \\ \Delta Y \end{pmatrix} = P_i - P_{i-1} = P_i - \begin{pmatrix} \cos(-\alpha) & -\sin(-\alpha) \\ \sin(-\alpha) & \cos(-\alpha) \end{pmatrix} P_i$$
$$\begin{pmatrix} \Delta X \\ \Delta Y \end{pmatrix} = \begin{pmatrix} 1-\cos\alpha & -\sin\alpha \\ \sin\alpha & 1-\cos\alpha \end{pmatrix} \begin{pmatrix} X \\ Y \end{pmatrix}. \quad (1)$$

Together with the (constant) ΔZ inter-slice difference along the yarn's centerline, we form a 3D vector, normalize it, and store this precomputation result as a texture, as shown in Figure 5-right. During voxelization, at each voxel location we simply query this texture and multiply the vector by the modelview matrix of the slice to obtain the fiber orientation in world coordinates. When multiple fragments contribute to the same voxel, we average their orientations.

3.4. Volume Generation on OpenGL

We base the volume generation on the OpenGL rasterization pipeline. Since version 4.3, GLSL provides direct access to images at arbitrary positions from any shader by means of *image_load_store* instructions. Moreover, since version 4.4, we can also use bindless textures to hold image data (also known as surface type) in memory. Bindless textures avoid the communication bottleneck produced by *bind* OpenGL calls and remove the limitation in the number of simultaneous textures available to the fragment shaders.

Our volume generation takes as input the positions of the slices computed in Section 3.2, and executes the following steps. These steps are also outlined in Figure 6, along with their corresponding implementation in the voxel fragment shader:

1. We create a regular grid of blocks at a coarse resolution determined by the user. This grid is stored as a 2D texture of unsigned integer pairs (*uvec2*) bound to a buffer object (denoted as *tex_handles* in the shader code sample). This buffer constitutes the global linear memory available to all the shaders.
2. We traverse all the slices (on the CPU) to determine which blocks are occupied and which are left empty. For each block containing slices, we create a 3D texture of fine resolution, store the handle in the buffer and mark it as resident in memory (or leave a zero if empty). In a

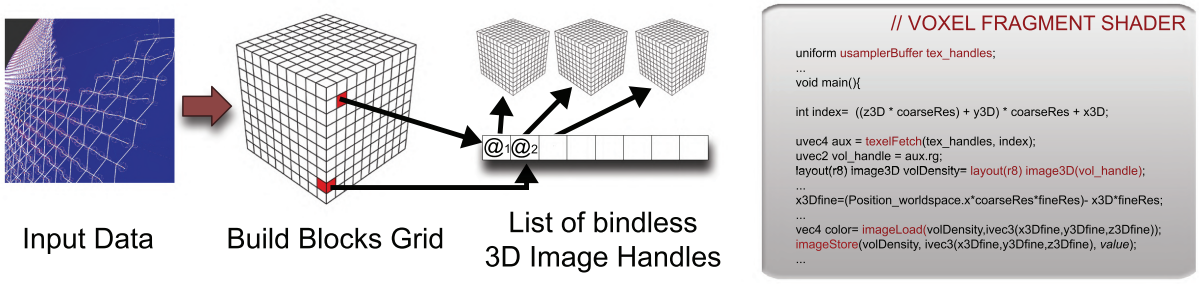

Figure 6: *Overview of the OpenGL implementation of the rasterization pipeline, and related commands on the fragment shader.*

supplementary document, we provide the full implementation details.

The fine resolution may be either defined by the user or computed to maximize memory occupancy given the number of blocks intersected by the geometry. For instance, the shirt model in Figure 13 has only 9023 occupied blocks in a 128x128x128 coarse grid, and we sample each of those blocks using a 32x32x32 fine grid, for a total memory occupancy of 845MB (instead of the 163 GB needed for the same resolution in a single block) .

3. We rasterize the slices. On the fragment shader, we read the associated block handle from the uniform buffer, cast it as $layout(r8)image3D$ texture, and fetch the texel corresponding to the absolute 3D position of the fragment in space.

To avoid sampling artifacts due to rasterization, we rasterize every slice three times along three orthogonal projection axes, with Z-rejection disabled. When a slice is parallel to a projection axis, the fragment shader may incur in large errors in the interpolation of the world coordinates provided by the vertex shader, thus placing voxels at wrong locations (See Figure 7). We avoid such artifacts thanks to GPU antialiasing (x8 FXAA in our case), with negligible cost (5%). Our approach differs from the recent voxelization method of Crassin and Green [CG12], who rasterize each triangle only once, after selecting on a geometry shader the axis that produces the maximum projected area. We avoid the geometry shader at the cost of three rasterization passes, which can be executed on a hidden frame buffer and interleaved with the visualization pass for interactive visualization of the volumetric data.

In our method, the user retains full control of both rasterization and voxelization resolution, and may change them dynamically to obtain a desired result. Fragment coordinates are decoupled from the resolution of the target 3D texture, and the 3D position of the fragment in its corresponding texture block is computed as described above. Thanks to this decoupling, the user may change on the fly the voxel resolution and the hierarchical structures. This allows for voxelization planning via our interactive interface: if the user desires to focus in a given area of the cloth model, the voxelization area is adapted to the viewport volume, processing a particular set of yarn nodes at a higher resolution (See the snag example in Figure 15).

In general, the combination of *image load/store* operations and simultaneous voxel and geometry processing opens a new range of rendering possibilities. For instance, in the bottom of Figure 14, we have modified the albedo of several slices depending on the value stored at the corresponding voxel (e.g., coloring in white the boundaries of the 3D voxel block). Other applications, such as global illumination by voxel cone tracing, might benefit from the flexibility of this GPU approach.

3.5. Fragment Composition

When several fragments hit the same output voxel, we have designed heuristics to compose the output density and orientation. Based on our observations, simply averaging density values yields an output volume that is excessively transparent. Figure 8 shows an example where a single slice is downsampled to generate a mipmap. The coarsest slice texture appears more transparent, and we have found that this transparency reduces the quality of final renderings. Instead, we have opted to favor actual fiber hits, following a strategy similar to conservative rasterization [AMA05]. When multiple fragments hit the same output voxel, we simply keep the last hit with a positive density. We rely on hardware antialiasing (x8) and high-resolution rasterization to guarantee multiple samples per output voxel.

Our approach could be improved by adopting a custom composition method, but this is a non-trivial task, as the composition policy should account for the visualization algorithm. For instance, Zhao et al. [ZJMB12] capture voxel densities with a micro-CT scanner and optimize the optical properties for a particular rendering model (microflakes) in order to match the pixels observed in real photographs. Xu et al. [XCL*01] compute density from a function which can be

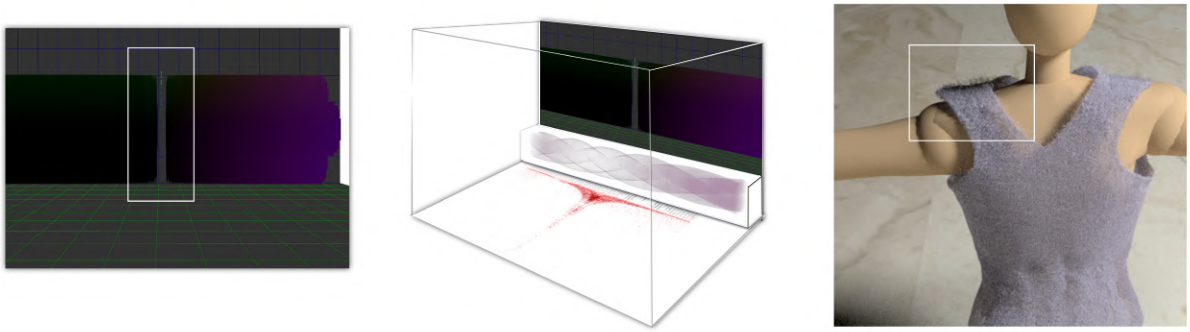

Figure 7: *Rasterization of perpendicular triangles is problematic both for visualization and voxelization. Left: OpenGL view of a single yarn sampled for voxelization. The green fragments on near-perpendicular planes yield wrong 3D world coordinates. Middle: Volumetric view of the same yarn and XZ projection of the 3D world coordinates computed for each voxel, shown in red. Position error is most evident for the fragments on near-perpendicular slices. Right: The resulting volumes are noisy as shown on the path-traced render of the shoulder.*

analitically averaged to be represented at each level of detail with custom volumetric mipmaps. Recent work by Heitz et al. [HDCD15] on volumetric density mipmapping presents a solution for the automatic creation of multi-level textures for microflake distributions, and might be a direction for future research.

For the composition of output orientations, we average incoming orientations in the order in which they are rasterized. Correct averaging would require atomic operations and waiting for all fragment hits before executing the average operation. However, following this approach we would incur in high processing and memory costs. We use single byte images, while atomic math for images is limited to GL_R32I and GL_R32UI integer types. Our policy for the composition of orientations might introduce some temporal or inter-frame incoherence, but we have shown the feasibility of our approach in animations [CLMMO14, CLMO15].

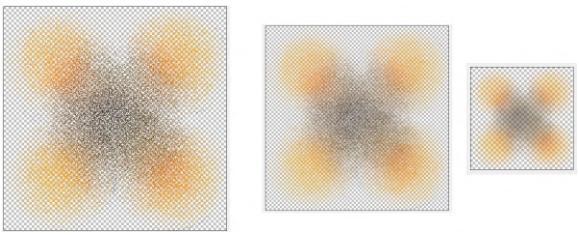

Figure 8: *From left to right: GPU automatic mipmapping of a fiber density distribution texture, starting at level 0 (no filtering). Notice how the last map is almost transparent.*

3.6. Volume Ray Casting

One possible way to visualize the result of voxelization is to apply GPU-based ray casting of the 3D texture [SSKE05]. We have implemented this ray casting as a single-pass shader, encoding density values with a 1D color map. The ray-casting shader has minimal cost compared to voxelization, and can be used, e.g., for the purpose of verifying the correctness of the voxelization. Volume ray casting is not to be confused with our preview visualization algorithm to be described in the next section. Figure 10 compares a ray-casting visualization of a voxelized yarn (top), with our preview visualization of the slice-based model (bottom). We use volume ray casting only to check the stored voxel data, whereas we use interactive visualization to inspect the output of the slice-based representation and refine the voxelization process. Both visualization methods allow for interactive rates and could be combined into more sophisticated visualization schemes.

4. Interactive Visualization Pipeline

Our interactive visualization pipeline shares many steps with the voxelization algorithm described in the previous section, as shown in Figure 2. However, it has two major differences. One is, of course, that volume rasterization is replaced with frame-buffer rasterization. We accumulate the effect of multiple semi-transparent slices through alpha-blending, and this blending calls for the second major difference. Correct alpha-blending requires sorting of slices, which are then rasterized back-to-front according to the camera view.

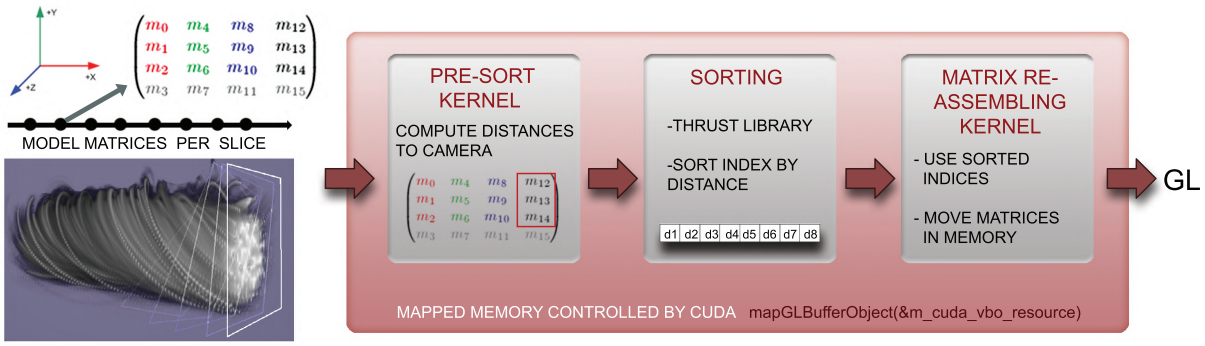

Figure 9: *Overview of our CUDA pipeline for sorting textured quads for real time visualization. The modelview matrices are stored in GPU memory as an array in a buffer object.*

4.1. GPU-Based Slice Sorting

In the first version of our pipeline [LMCMO14] we relied on single-CPU sorting (*std::sort*). For dense models this step implies sorting a vector with millions of elements, and may take up to three or four seconds per frame on our examples. While sorting is not required for voxelization, for interactive visualization it becomes a bottleneck that might hinder the user experience. In the present work we introduce a CUDA-based sorting step in our pipeline, keeping all operations in the GPU and avoiding memory transfer between the host (CPU) and the device (GPU). As depicted in Figure 9, the sorting is divided into four sub-steps:

1. We take the modelview matrices, computed and stored in GPU memory as described in Section 3.2, and we map this memory area for CUDA processing. Then, we give control to the CUDA context to perform the actual sorting (For a code example, see the supplementary document).
2. A first kernel goes through the array of matrices, reads the translation vector of each slice and computes the squared distance to the camera view (passed as a global variable). A list of corresponding incremental indices is generated too.
3. With the indices and distances, we call the *sort-by-key* method of the *Thrust* library to re-order the list of indices, using the distances as keys.
4. A third kernel takes these re-ordered indices and copies the data of the modelview matrix array to a new (re-ordered) array. This implies a single *device-to-device* copy operation; the CPU is never used and the process finishes when the control is returned to the OpenGL pipeline via the $unmapGLBufferObject(m\_cuda\_vbo\_resource)$ command.

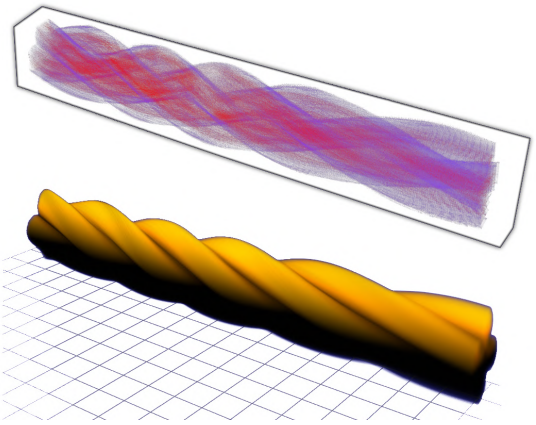

Figure 10: *Top: Ray-casting of a voxelized yarn, as described in Section 3.6. Bottom: Interactive visualization of the slice-based model for the same yarn, with the method described in Section 4.*

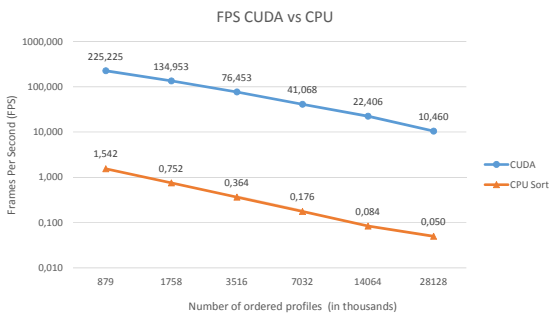

Figure 11: *Frame rate comparison between a sorting operation on the CPU (std::sort) and our CUDA-based method.*

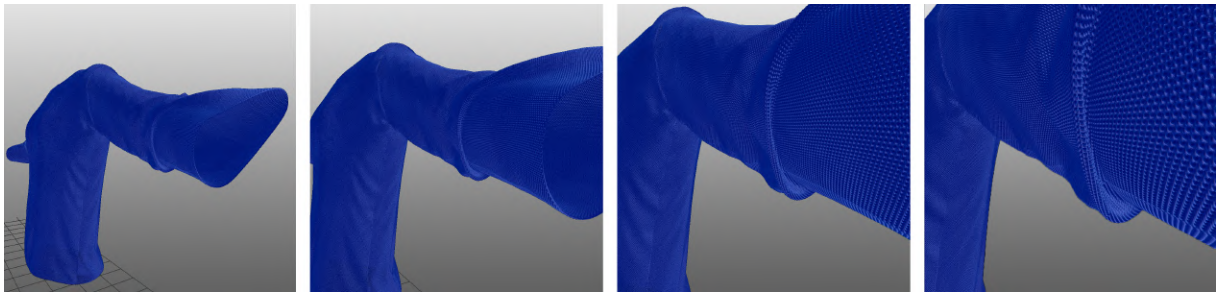

Figure 12: *Real-time visualization of a shirt model at different zoom levels. Fibers are noticeable only at the closest distances, but their local radiance affects the overall aspect of the cloth.*

To summarize, in this fashion there is no memory transfer between the GPU device and the CPU (modelview matrices are already stored in GPU) and, while sorting, we avoid moving matrices until the last step, using a distance-based index sorting instead. We achieve a speed-up of two orders of magnitude on the sorting operation alone, as shown in Figure 11. Thanks to this fast sorting method, the impact of sorting on the frame rate of our visualization becomes negligible; the millions of fragment executions are the actual bottleneck of the pipeline.

4.2. Shading and Blending

For preview visualization purposes, we compute local shading based on the Kajiya-Kay model [KK89], which was originally created for rendering of hair but equally approximates the scattering of light in a fiber inside a yarn. As input we take the albedo colors stored in the texture, the orientation computed at each fragment pixel and both light source and camera directions. Since the radii of fibers are small, we can safely assume a parallel light model. We also include pre-computed inter-fiber ambient occlusion already baked in the albedo texture to improve volume perception.

In Figure 12 we show screen captures of our interactive visualization. Cast shadows, obtained by shadow mapping, are shown in the left image of Figure 13. Please see the accompanying video for additional screen recordings of our interactive visualization.

In Figure 14 we show an example of visualization based on a combination of data stored on the voxel grid and the triangle-based slices. This shading approach can be used to verify the quality and resolution of the stored volumetric data in comparison with the highly detailed slice model (e.g.,

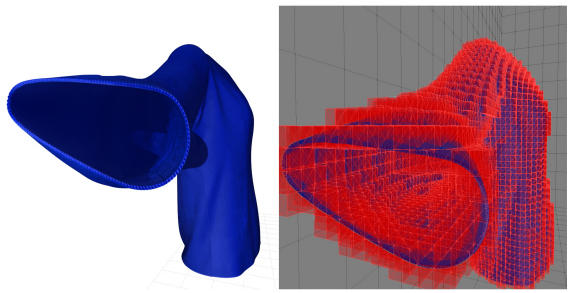

Figure 13: *Left: Interactive rendering (1-2 FPS) of a shirt model (45.6 million triangles) with Kajiya-Kay shading and shadow mapping. Right: Visualization of the underlying grid structure (64x64x64 blocks).*

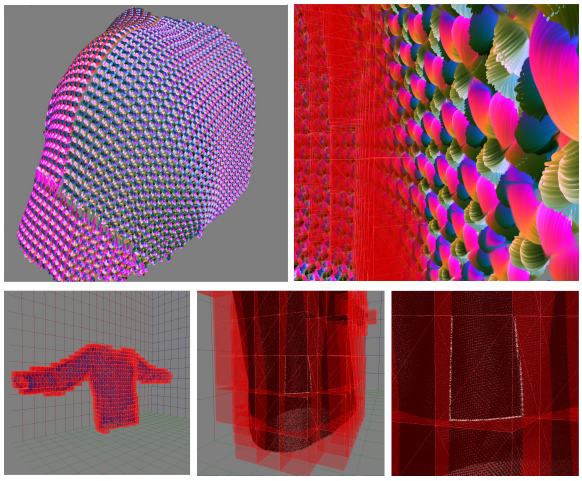

Figure 14: *Shading example, combining voxel data and triangle rastering. Top row: The slices are colored according to the orientations stored on the voxel grid. Bottom Row: Different zoom levels focusing on a particular block (blocks are overlaid in red color), where a set of slices has been colored in white to verify the resolution of a single voxel row.*

	Low density cloth	High density cloth	Low density Close-up	High density Close-up
Number of slices	2.5M	17.6M	4.08M	35.2M
CPU memory	1.8GB	7.2GB	9GB	12GB
GPU memory	2.06GB	3GB	4GB+2GB(shared)	4GB+2GB(shared)
Slice generation (yarn sampling)	1.43s (0.12s)	8.67s (0.726)	2.09s (0.18s)	17.32s (1.47s)
Single visualization/voxelization pass	71ms	150ms	120ms	400ms

Table 1: *Statistics and voxelization performance using a single-block grid for two models at two different zoom levels. Slice generation consists of yarn spline sampling, volumetric texture initialization, and OpenGL quad instantiation, and is required only if the geometry of the cloth is changed. The last row reports times for one rasterization pass, either as part of voxelization or visualization. This is the time needed for preview visualization when camera or illumination, but not geometry, are modified.*

the orientations stored on the slices), or even to improve realism by incorporating volumetric illumination effects.

To speed up preview visualization in a view-dependent manner, we have considered using mipmapping of the slice textures. However, with standard mipmapping, we suffer the same transparency problems as discussed for density composition in Section 3.5 and shown in Figure 8. We have disabled mipmapping and we leverage hardware anti-aliasing and high-resolution rasterization to obtain high-quality results. As also discussed in Section 3.5, custom visualization-aware mipmapping methods are a possible direction for improvement.

5. Experiments and Performance

In this section, we describe statistics of the benchmarks used for testing, as well as the performance results. We first discuss performance of the voxelization algorithm, and then the impact of the sparse voxelization on the performance of offline rendering. In all our tests, we have used an eight-core i7-3770S processor with 16 GB of memory and a NVIDIA GeForce GTX TITAN Black graphics card with 6 GB of memory.

5.1. Voxelization and Interactive Visualization

In terms of memory management, the 3D textures are the main limitation. We handle up to 6GB in three textures, for a maximum size of 2GB per texture and 2048 voxels of maximum size for any axis (limited by the graphic card). We also keep in memory an array with a 4x4 model view matrix per slice (up to 2.62GB for 41 million slices). Setting up both structures implies a warm-up of a few seconds per frame of voxelization for the biggest models, but it has no impact on subsequent visualization operations for the geometry.

In terms of computation times, we have compared performance with a light and a dense model, both at two different zoom levels. For the far view, we use 25 slices between yarn crossings (1mm, 1/40 of the pixel width), and we increase it to 50 for the close view. We also increase the 3D texture resolution to the maximum available(6GB). In Table 1, we show computation times for the two models and zoom levels. First, we show the cost of sampling the yarn splines and setting up the slice data structures on the GPU. These operations are carried out once per voxelized frame, and they dominate the cost of voxelization. Second, we show the cost of executing one rasterization pass, either as part of the three passes in voxelization, or as part of visualization. Once the slice data structures are set up, our preview visualization is interactive.

The data in the table was obtained using a single-block texture representation. With a sparse representation consisting of a 32x32 two-level grid, the cost of slice generation grew only by a factor of 4.6% on average. In addition, we have compared performance and quality using cross-section slices of two different sizes, 1024x1024 and 32x32, with no noticeable differences. As discussed earlier, mipmapping was disabled in both cases.

In comparison to previous voxelization methods, we have observed similar performance to the one achieved by Crassin and Green [CG12] on models of the same size. They reported computation times around 1-2ms for direct voxelization on models with less than one million polygons. Xu et al. [XCL*01], on the other hand, reported over 31 minutes for the rendering of 368k slices.

5.2. Offline Rendering

Our method can process very dense models (around a million yarns), producing the volumetric data used to render the vest and the sheet shown in Figure 16, or the shirt shown in Figure 17. Notice how the sheet exhibits a transparency effect in the top part due to the stretching of yarns, which would not be possible without a full yarn-level model.

All the offline render results shown in this paper have been generated with the raytracing engine Mitsuba [Jak10]; in particular, with its volumetric path tracer and the microflakes model [JAM*10]. Our volumetric textures are translated into an offline format with one byte for density and two bytes for orientation (discrete polar coordinates θ and ϕ). The animations and videos of our cloth simulation works [CLMMO14, CLMO15] were also generated in the same fashion, demonstrating that our method is suitable for the generation of temporally coherent results.

The advantages of our hierarchical voxel grid structure are

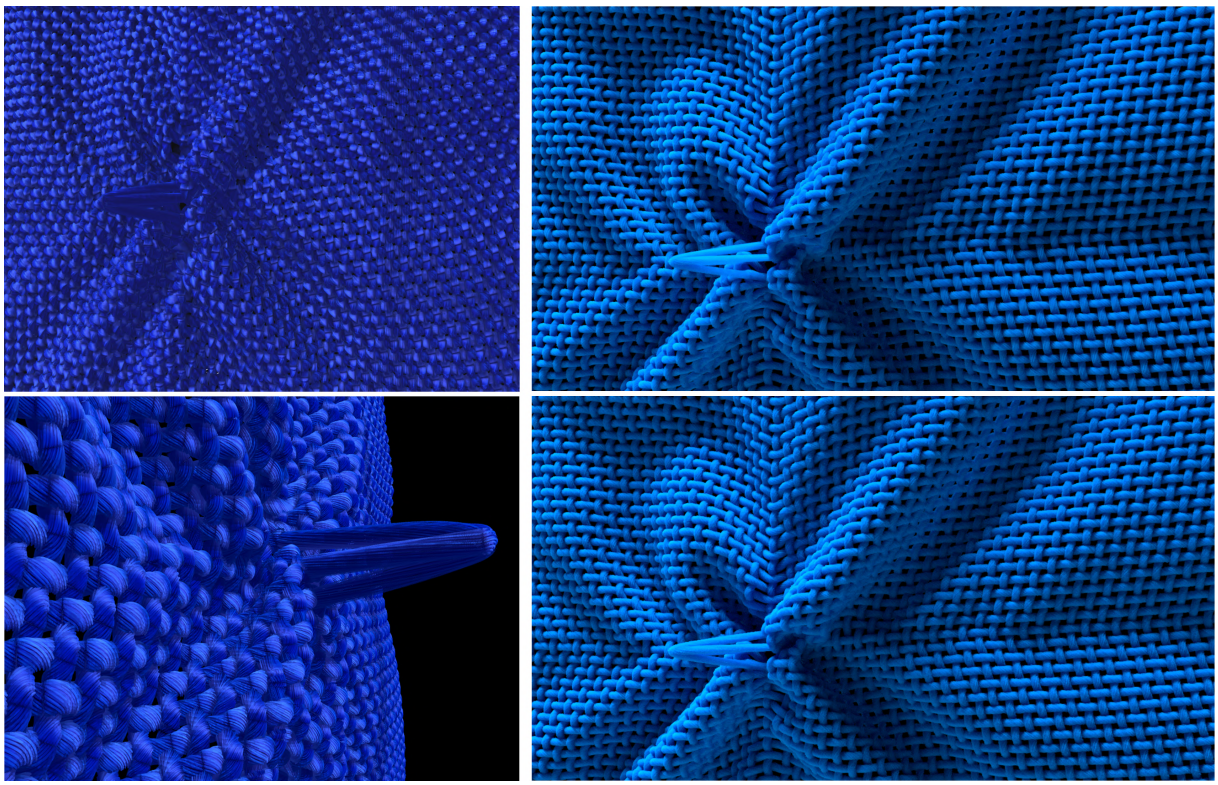

Figure 15: *Left column: Real time visualizations (Kajija-Kay model with shadow mapping) of a yarn snag. Top Right: Same model rendered in Mitsuba (stdev = 0.5) with a 32x32x32 grid of 64x64x64 voxel blocks. Bottom Right: Same model but rendered using a single-block grid (with 2048 voxels along the longest side). No differences, other than stochastic path tracing noise, are visible, while the rendering time is reduced almost to one third.*

evident both in terms of data management and path tracing costs. The sparsity of data is characteristic of cloth; the shirt model (Figure 13) has as few as 0.854% occupied blocks in a 64x64x64 grid, and only half (0.43%) in a 128x128x128 grid. In the context of volumetric path tracing, by fitting the boundaries of the volumetric data with more accuracy we can reduce the marching steps of rays (or even discard them) when sampling outside the volume. This translates into a significant reduction of rendering times for the same visual quality, as shown on the right column images of Figure 15). Furthermore in several cases, we can render at resolution levels which are not even possible with a single block; for instance, a 64x64x64 grid of 64x64x64 voxel blocks can render images of up to 4096x4096 pixels with sub-pixel voxel resolution. Such high resolution would require 68 GB in a regular single block. Table 2 summarizes the effect of hierarchical grid configurations on rendering time costs as well as storage file size, which is not negligible in production environments. Notice the differences between the first and second configurations: File size and rendering times are reduced to almost 25% of those with the single-block grid, while the voxel resolution per pixel has been doubled (from 1024 to $32x64 = 2048$ voxels). The third and fourth configurations reach up to 4096 voxels per pixel.

Model, Grid/Voxels	**Textures File Size**	**Render Time**
Shirt, 1/1024	1.5GB	5 min 10 sec
Shirt, 32/64	391MB	1 min 14.5 sec
Shirt, 64/64	1.52GB	46.1 sec
Shirt, 128/32	554MB	36.9 sec
Snag, 1/2048	6GB	1 hour 26.2 min
Snag, 32/64	1.6GB	37 min 36 sec

Table 2: *Statistics and offline rendering performance for the shirt and snag models of Figures 15 and 13, for a target image of 1024x768px.*

6. Conclusions and Future Work

In this work we have introduced a pipeline to voxelize yarn structures with several advantages over previous approaches. First, our pipeline is able to handle directly on the GPU large voxel models, while maximizing the use of video mem-

Figure 16: *Left and middle: Very dense model (17.6M slices, 410k yarn crossings), rendered at medium level of detail (25 slices per crossing, 1152 voxels along the maximum axis). Both images were pathtraced with the micro-flakes model (stdev = 0.5 and 0.1 respectively). Right: The densest yarn model voxelized by our method: one million crossings (100 yarns per inch), represented using 41.3 million slices (82.6 million triangles) and 1625 voxels along the longest axis of a single-block grid.*

ory and taking advantage of a sparse structure. This results jointly in higher voxel resolution and faster offline raytracing. Second, our simultaneous preview visualization method allows for efficient interactive parameter setting of both the voxelized yarn (type, radii, fiber density, color, torsion, etc.) and voxel structure (resolution, volume to voxelize, etc.). These parameters have a great impact on the final appearance, and we believe that any tool based on our design cycle will greatly reduce the time costs for volumetric content production.

The GPU pipeline shown in this paper (based on image unit access and bindless image textures) is very flexible, providing fast computation times and maximum GPU memory use. We believe that many applications beyond yarn voxelization can benefit from this OpenGL path.

The real-time shading of fibers in our algorithm uses currently a local Kajiya-Kay model, however our approach is a good candidate for techniques such as screen-space subsurface scattering [JG10] or any volumetric illumination model [JSYR14]. Phenomena like translucency could also be precomputed, given that the distances at each slide are previously known. In the future we would like to explore the adaptation of these methods to our framework, combining deferred shading and 3D texture direct access.

We have not addressed the compression of the 3D textures. The overhead of decoding the texture for non-coherent memory access is still a challenge for real-time applications. However, this is an active field of research [BRGIG*14], and recent advances suggest that this should be considered for future improvements of our pipeline.

A significant improvement to our method would be a GPU-based approach to the re-sampling of yarn curves, currently implemented on the CPU (Section 3.1). It would be faster and it would save memory bandwidth between device and host, thus reducing the per-frame cost for animations. Furthermore, such method could be improved with view-dependent slice allocation to render multiple levels of detail by sampling adaptively along the yarn center line. Any level-of-detail approach would also benefit from automatic customized mipmap representations which will require a non-trivial re-sampling of the fiber density texture.

Our algorithm has been successfully used to generate all the renderings in the cloth simulation work by Cirio et al. [CLMMO14, CLMO15], and from this experience we believe that it has a great potential to simulate complex visual aspects of the cloth under deformations.

Acknowledgements

We thank Jeff Bolz from NVIDIA and Wenzel Jakob (Mitsuba) for their prompt answers to our questions. This research is supported in part by the Spanish Ministry of Economy (TIN2012-35840) and by the European Research Council (ERC-2011-StG-280135 Animetrics). The work of Gabriel Cirio and Jorge Lopez-Moreno was funded by the Spanish Ministry of Science and Education through Juan de la Cierva fellowships.

References

[AMA05] AKENINE-MÖLLER T., AILA T.: Conservative and tiled rasterization using a modified triangle set-up. *Journal of Graphics, GPU, and Game Tools 10*, 3 (2005), 1–8. 6

[BHW94] BREEN D. E., HOUSE D. H., WOZNY M. J.: Predicting the drape of woven cloth using interacting particles. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1994), SIGGRAPH '94, ACM, pp. 365–372. 1

[BRGIG*14] BALSA RODRÍGUEZ M., GOBBETTI E., IGLESIAS GUITIÁN J., MAKHINYA M., MARTON F., PAJAROLA R.,

Figure 17: *Offline renderings of our volumetric data. Left and middle: Two views of a low yarn-count shirt (5.006 million slices). Right: Same model with different micro-flake distribution to simulate shinier fibers (stdev = 0.1)*

SUTER S.: State-of-the-art in compressed GPU-based direct volume rendering. *Computer Graphics Forum* (2014). 12

[CG12] CRASSIN C., GREEN S.: *OpenGL Insights*. CRC Press, Patrick Cozzi and Christophe Riccio, 2012, ch. Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer. 2, 6, 10

[CLMMO14] CIRIO G., LOPEZ-MORENO J., MIRAUT D., OTADUY M. A.: Yarn-level simulation of woven cloth. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH Asia) 33*, 6 (2014). 1, 3, 7, 10, 12

[CLMO15] CIRIO G., LOPEZ-MORENO J., OTADUY M. A.: Efficient simulation of knitted cloth using persistent contacts. In *ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (2015). 1, 7, 10, 12

[CR74] CATMULL E., ROM R.: A class of local interpolating splines. *Computer aided geometric design 74* (1974), 317–326. 4

[dFH*11] D'EON E., FRANCOIS G., HILL M., LETTERI J., AUBRY J.-M.: An energy-conserving hair reflectance model. In *Proceedings of the Twenty-second Eurographics Conference on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2011), EGSR '11, Eurographics Association, pp. 1181–1187. 2

[EKS03] ETZMUSS O., KECKEISEN M., STRASSER W.: A fast finite element solution for cloth modelling. In *Computer Graphics and Applications, 2003. Proceedings. 11th Pacific Conference on* (2003), pp. 244–251. 1

[Fol96] FOLEY J.: *Computer Graphics: Principles and Practice*. Addison-Wesley systems programming series. Addison-Wesley, 1996. 4

[Gol97] GOLDMAN D. B.: Fake fur rendering. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 127–134. 2

[HDCD15] HEITZ E., DUPUY J., CRASSIN C., DACHSBACHER C.: The SGGX microflake distribution. *ACM Trans. Graph. 34*, 4 (July 2015), 48:1–48:11. 7

[Jak10] JAKOB W.: Mitsuba renderer, 2010. http://www.mitsuba-renderer.org. 10

[JAM*10] JAKOB W., ARBREE A., MOON J. T., BALA K., MARSCHNER S.: A radiative transfer framework for rendering materials with anisotropic structure. *ACM Trans. Graph. 29*, 4 (July 2010), 53:1–53:13. 2, 10

[JG10] JIMENEZ J., GUTIERREZ D.: *GPU Pro: Advanced Rendering Techniques*. AK Peters Ltd., 2010, ch. Screen-Space Subsurface Scattering, pp. 335–351. 12

[JSYR14] JÖNSSON D., SUNDÉN E., YNNERMAN A., ROPINSKI T.: A survey of volumetric illumination techniques for interactive volume rendering. *Computer Graphics Forum 33*, 1 (2014), 27–51. 12

[KJM08] KALDOR J. M., JAMES D. L., MARSCHNER S.: Simulating knitted cloth at the yarn level. *ACM Trans. Graph. 27*, 3 (2008), 65:1ï£¡65:9. 1, 3

[KJM10] KALDOR J. M., JAMES D. L., MARSCHNER S.: Efficient yarn-based cloth with adaptive contact linearization. *ACM Transactions on Graphics 29*, 4 (July 2010), 105:1–105:10. 1

[KK89] KAJIYA J. T., KAY T. L.: Rendering fur with three dimensional textures. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1989), SIGGRAPH '89, ACM, pp. 271–280. 2, 9

[LMCMO14] LOPEZ-MORENO J., CIRIO G., MIRAUT D., OTADUY M. A.: GPU visualization and voxelization of yarn-level cloth. In *Proc. of CEIG, Spanish Computer Graphics Conference* (2014), Eurographics. 2, 8

[MBCN09] METAAPHANON N., BANDO Y., CHEN B.-Y., NISHITA T.: Simulation of tearing cloth with frayed edges. *Comput. Graph. Forum*, 7 (2009), 1837–1844. 1

[MJC*03] MARSCHNER S. R., JENSEN H. W., CAMMARANO M., WORLEY S., HANRAHAN P.: Light scattering from human hair fibers. *ACM Trans. Graph. 22*, 3 (July 2003), 780–791. 2

[MMN98] MEYER A., MEYER R., NEYRET F.: Interactive volumetric textures. In *In Eurographics Rendering Workshop* (1998), Eurographics, Springer Wein. ISBN, pp. 157–168. 2

[Mus13] MUSETH K.: Vdb: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph. 32*, 3 (2013), 27:1–27:22. 3

[Pan11] PANTALEONI J.: Voxelpipe: A programmable pipeline for 3d voxelization. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG '11, ACM, pp. 99–106. 2

[Pro95] PROVOT X.: Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *In Graphics Interface* (1995), pp. 147–154. 1

[SSKE05] STEGMAIER S., STRENGERT M., KLEIN T., ERTL T.: A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Proceedings of the Fourth Eurographics / IEEE VGTC Conference on Volume Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2005), VG'05, Eurographics Association, pp. 187–195. 7

[XCL*01] XU Y.-Q., CHEN Y., LIN S., ZHONG H., WU E., GUO B., SHUM H.-Y.: Photorealistic rendering of knitwear using the lumislice. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 391–398. 2, 5, 6, 10

[YSK09] YUKSEL C., SCHAEFER S., KEYSER J.: On the parameterization of catmull-rom curves. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling* (2009), ACM, pp. 47–53. 4

[ZJMB12] ZHAO S., JAKOB W., MARSCHNER S., BALA K.: Structure-aware synthesis for predictive woven fabric appearance. *ACM Trans. Graph. 31*, 4 (July 2012), 75:1–75:10. 2, 6