CEIG - Spanish Computer Graphics Conference (2014)
Pere-Pau Vázquez and Adolfo Muñoz (Editors)

# GPU Visualization and Voxelization of Yarn-Level Cloth

Jorge Lopez-Moreno[1], Gabriel Cirio[1], David Miraut[1] and Miguel Angel Otaduy[1]

[1]Universidad Rey Juan Carlos, Madrid (Spain)

**Abstract**

*Most popular methods in cloth rendering rely on volumetric data in order to model complex optical phenomena such as sub-surface scattering. Previous work represents yarns as a sequence of identical but rotated cross-sections. While these approaches are able to produce very realistic illumination models, the required volumetric representation is difficult to compute and render, forfeiting any interactive feedback. In this paper, we introduce a method based on the GPU for simultaneous visualization and voxelization, suitable for both interactive and offline rendering.*

*Our method can interactively voxelize millions of polygons into a 3D texture, generating a volume with sub-voxel accuracy which is suitable even for high-density weaving 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 decade we have seen how the level of simulation has gone from simulating spring-based simple models on the edges of triangular meshes [BHW94, Pro95] or more accurate methods based on finite elements [EKS03], to the computation of highly detailed physical interactions at the yarn level [KJM08, MBCN09, KJM10].

This level of detail has created a demand for render methods which can leverage the data available and deal with a volumetric representation of the yarns and their constituent fibers. While the complexity of the cloth reflectance representation has increased over the years, even including microCT-captured threedimensional models of the fibers [ZJMB12], these approaches are limited to a surface-based representation of the cloth objects and/or stochastic replication of texture tiles.

Yarns, as a rendering target, were introduced by Xu et al. [XCL*01]. They included the GPU in the process, obtaining very compelling volumetric renderings in less than an hour of computation. Their representation was used as input by Jacob et al. [JAM*10] for more accurate and realistic (offline) rendering with volumetric path tracing and the micro-flakes model.

## 2. Related Work

Our method relies on a similar representation to Lumislice, introduced by Xu et al. [XCL*01]. A single yarn is modelled by a set of cross-sections. At each cross-section, fine level phenomena such as occlusion, shadowing and multiple scattering is described at fiber level. For Xu et al., the advantage of working with slices as atomic unit is that, instead of ray tracing a full volumetric model, they compute all the interactions for each cross-section, store the resulting colors and transparency values and then rely on hardware-based transparency to do the visibility rendering for them, overlaying all the slices in alpha-blending mode. We use the same atomic structure, the slice, but with a different data structure. In their work, Xu et al. extend the volumetric render approach of Meyer and Neyret [MMN98], in which slices have access to all the data, and only transparency rasterization is left to hardware. In order to leverage the parallelism capabilities of the GPU, we precompute data for each slice, allowing for out-of-order rendering and geometry instancing. At the same time, we take advantage of the recent Shader Model 5 (SM5) which allows random access and 3D-addressing from the GLSL shader pipeline in order to map these slices to a voxel grid. The potential of such access for voxelization was recently shown by Crassin and Green [CG12]. In their work they focus on two problems: not missing parts of geometry due to rasterization and augmenting the stor-



DOI: 10.2312/ceig.20141115

age capabilities of voxel grids with sparse octree representations. The present work addresses additional rasterization problems (see Section 5).

## 3. Our Method

### 3.1. Overview

In order to define a yarn, we will rely on a slice representing the distribution of fiber density (See Figure 1), which is replicated and rotated along a path following the 3D shape of the yarn. The size of the slice is equal to the diameter of the yarn and the separation between consecutive slices will depend on requirements of the scene as we will see in the next section.

Our method adapts previous slice models to a GPU friendly structure in order to handle a larger amount of slices, performing illumination computations and voxelization on per-pixel shaders. This will prove to be tricky as a large set of slices (millions) requires instantiation and asynchronous rendering and thus each slice will not share any information with their neighbouring slices. Moreover, our shader-based approach allows for real time voxelization of each slice at a pixel level. At every time, we keep a coherent volumetric representation of the yarns with density and 3D orientation of each fiber in a 3D texture. This texture is visualized with a real-time (GPU-based) ray casting method but it can also be stored in a volumetric file format for posterior offline rendering with more complex methods, such as volumetric path tracing and the micro-flakes model. We show such results in Section 4.

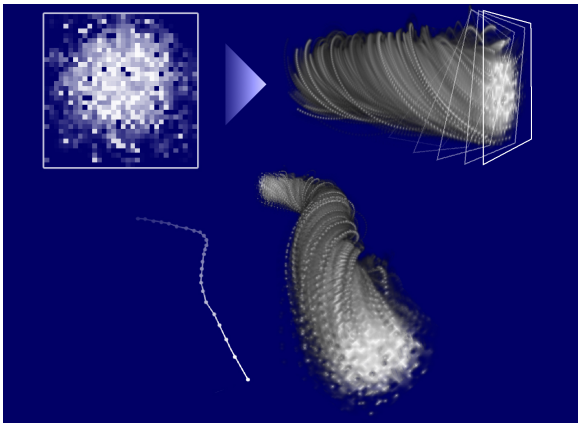

**Figure 1:** *Top left: Density distribution of fibers at a given slice, stored as a texture. Top Right: A stack of rotated slices conform a single thread section. Bottom: A path defines the position of these slices*

### 3.2. Input data and smooth interpolation

Most natural fibers are only a few centimeters long. In order to obtain longer yarns, fibers are interlocked through spinning, with the resulting threads exhibiting desirable mechanical properties such as improved resistance and flexibility. Therefore, threads do not bend sharply when they touch other threads; instead, their shape follows a smooth curve proper to weaving patterns.Our compact data model only provides the thread location in contact points and model boundaries. So, intermediate positions must be devised in order to reproduce the inherited physical behavior from spinning process. Countless books and published papers in Computer Graphics and Industrial Design offer a wide range of strategies to trace curves through a set of knots. We have chosen classic Catmull-Rom splines [CR74] because of its algorithmic simplicity, low computational cost (usually below 2-3 seconds for a million crossings) and good results with our initial sampling.

Contact points are located at non uniform distances, alternating to one and another side of the fabric. All singular points are defined as knots in the curve, and we may consider that our sampling rate is similar to Nyquist one in a certain sense; since they capture points with maximum variability and no aliasing artifacts are produced. Catmull-Rom curves are cubic Hermite splines, so each portion of the curve traces third-degree polynomial specified by its values and first derivatives at the end points of the corresponding domain interval. So, sampling points $\{P_1 P_2 \dots P_n\}$ are easily organized in sequential subsets that are drawn independently. It ensures that yarn trajectory is only influenced by simulation, not by interpolation side effects; ie. if one of the control points is moved, it just affects the curve locally. 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 estimate the curve tangent in the knots (see Figure 2). Each initial sample is shared in four overlapping subsets, where the sample changes its role (as a clue for the first knot tangent, the first knot, the second knot, and a clue for the second knot tangent). The tangent to the curve at each control point is defined to be:

$$\frac{P_{i+1} - P_{i-1}}{2}, \frac{P_{i+2} - P_i}{2}$$

respectively. As -usually- sequential samples are located in opposite sides of the fabric, previous definition approximates the tangent of each surface of the cloth.

The next matrix formulation defines the cubic curve [Fol96], which represents a piece of the total curve between two successive control points. It can be applied to all segments of the curve except for the first and last segments where tangents must be estimated separately.

$$P(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -5 & 4 & 1 \\ -1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix}$$

The detail in the segment can be tuned by evaluating more

or less $t \in [0,1]$ parameter values in the function. New interpolated samples will lead to a smooth curve, that will be used as a guide to place billboards. 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 with the classic algorithm.

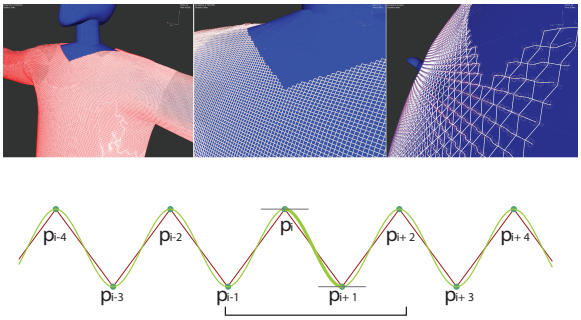


**Figure 2:** *Top Row: From left to right, increasing zoom levels at input data from visualization. Yarns are represented by crossing linear segments. Bottom row: A classic Catmull-Rom spline (green). Control points are interpolated by the spline, which passes through each knot in a direction parallel to the line between the adjacent points (grey). The jagged brown line represents a less natural way to trace yarns.*

### 3.3. Placing the yarn slices

In our system the slice will be represented by a quad composed by two triangles and a texture with the albedo values and the fiber density distribution stored in the alpha channel (See bottom-left image in Figure 3). In order to reduce computation costs, those quads are instances with the same geometry and texture but different attributes such as position and orientation (albedo and fiber density distribution can also be attributes, as textures, but this is left for future work).

For each stop in the spline path we compute the model matrix for the corresponding slice. We set as forward-vector the difference between the position of the current slice and the following one, the up-vector is set as the perpendicular to the forward vector at the beginning of each yarn. Then, the up-vector is rotated w.r.t. the forward-vector incrementally along the yarn in order to simulate the characteristic twist of the thread (input parameter to the system). 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*).

Additionally, for a correct alpha blending rendering, all the slices are sorted according to their distance to the camera (their global positions are stored as a column in the corresponding model matrices).

### 3.4. Computing position and orientation at fiber level

At this point we want to have orientation and position in world coordinates at pixel level which corresponds to a single fiber of the yarn cross-section stored in a texture. Each vertex shader has access to the model matrix of the slice, passing down to each fragment shader its interpolated position and the global orientation of the slice.

The fragment shader has then the position in world coordinates and it can access the slice density distribution stored in a texture, therefore we know the density of a 3D position in the world. Orientation is a bit more complicated, as we have only the orientation of the whole slice. However, the orientation of each pixel is equal to the tangent of the corresponding fiber. To compute this tangent for a point $X$ we would need to know its next position $X + \partial X$ between two consecutive slices (as is done in previous work [XCL*01]), however the instances are executed out of order and we cannot access this neighboring information.

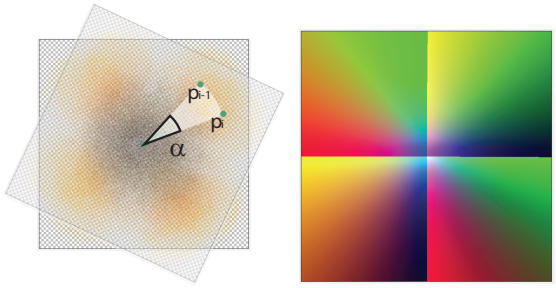


**Figure 4:** *Left: Computing the orientation tangent as a difference of positions between consecutive slices. Right: Texture with the XY coordinates of the orientation vector in tangent space.*

We leverage the fact that we know a priori the exact amount of rotation between two consecutive slices ($\partial X, \partial Y$ plane in tangent space), and the differential separation between them along the path ($\partial Z$ in tangent space) to store this data and share it among all the instances (See Figure 4). We compute $\partial X$ and $\partial Y$ for each pixel of the cross-section for a given rotating angle $\alpha$:

$$\begin{aligned} P_i &= (X,Y) P_{i+1} = (X',Y') \\ X' &= X\cos(-\alpha) - Y\sin(-\alpha) \\ Y' &= X\sin(-\alpha) + Y\cos(-\alpha) \\ (\partial X, \partial Y) &= (P_i - P_{i+1}) = (X - X', Y - Y') \end{aligned} \tag{1}$$

### 3.5. Shading

For visualization purposes we compute a simple lambertian shading based on the albedo colors stored in the texture and

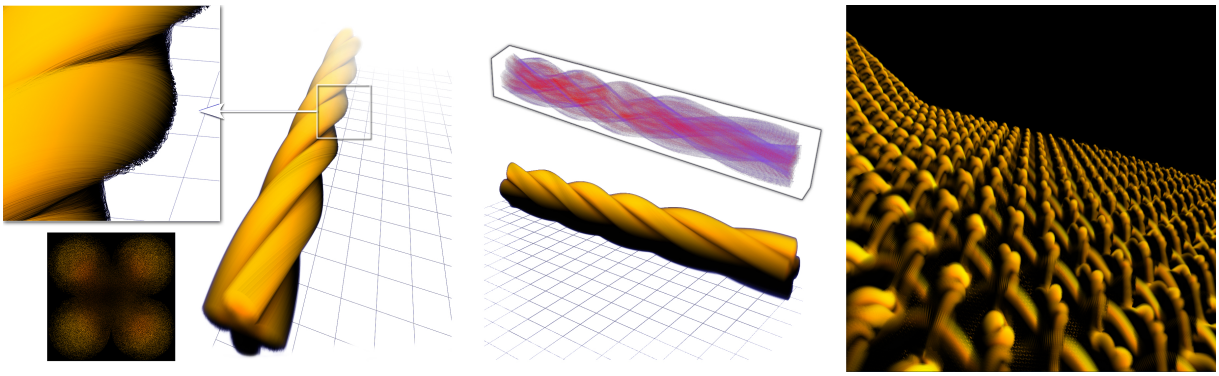

**Figure 3:** *Zoom at a single yarn in our interactive UI. Left: Close up view of fibers and fiber distribution slice used in this rendering. Middle: Simultaneous visualization of yarn based on alpha blending (bottom), and the volumetric density 3D texture (top). The 3D texture is ray-casted on the fly from the fragment shader. Right: Close up interactive view of the yarns at the dense cloth shown in Figure 6.*

the orientation computed at each fragment pixel. We assume parallel light (given the radius of a microfilmer is a safe assumption). We also include pre-computed ambient oclussion already baked in the albedo texture to increase volume perception.

Additionally, we provide a view of the current voxelization (for verification purposes) by means of GPU-based ray casting on the 3D texture [SSKE05]. This single pass shader has little impact on the GPU performance, encoding values with a 1D color map (see the white cube in the middle image in Figure 3).

### 3.6. Volume Generation

We base the volume generation in 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. In the absence of visualization, our approach is similar to the recent voxelization method proposed by Crassin and Green [CG12]. They raster each triangle to a 3D texture in three steps:

- First, they choose the maximum projection axis (X,Y or Z) for the triangle (this operation is done in a geometric shader).
- Second, they create a viewport matching the dimensions of the 3D texture lateral view and disable Z rejection.
- Third, they expand and raster the triangle. Each fragment coordinates are mapped to the 3D texture coordinates (s,t,d).

In our case, we want to take advantage of the user visualization of the mesh, substituting the first two steps of Crassin and Green's method with the rastering already computed by the visualization itself. Therefore, we will render the fragments directly called by the viewport of the user camera. This means that when the user is far away from the object, the triangles will be small for the viewport resolution and the texture details will be lost. However, as the user zooms into the object, additional details (at the texture level) are visible to the camera, generating fragment calls with resolution above the voxel accuracy.

In Crassin and Green's method, fragment coordinates are used for the mapping of the slices to a 3D volume. In our method, the position in world coordinates of each slice is passed from the vertex shader and interpolated at each fragment shader in order to map the raster fragment to the corresponding grid position. This decouples the fragment coordinates from the target 3D texture allowing for *on the fly* changes in the voxel resolution and hierarchical structures like octrees. If the user desires to focus in an area, the 3D texture resolution can be adapted to the viewport volume, storing a particular cloth area at a higher resolution.

## 4. Results

In Figure 6 we show screen captures of our interactive visualization. The method can process interactively very dense models (around a million yarns), yielding the volumetric data used to render the vest shown in Figure 7 and 5, and the shirt in Figure 8. All the offline render results shown in this paper are generated with the Mitsuba raytracing engine [Jak10], using a volumetric path tracer and the microflakes model [JAM*10]. Our volumetric textures are translated into the offline format with one byte for density and two bytes for orientation (discrete polar coordinates $\theta$ and $\phi$).

Our voxelization approach relies on the rastering pipeline to generate fragment calls at the pixels inside each slice. Crassin and Green [CG12] proposed to raster each triangle

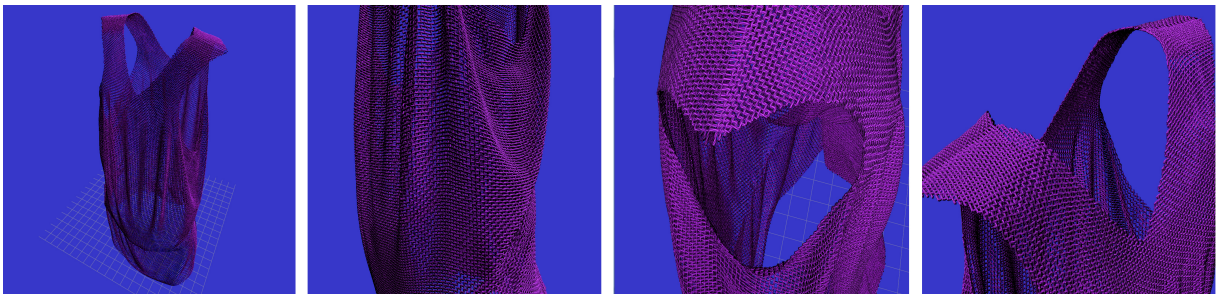

**Figure 6:** *From left to right: Screenshots of our user interface at different zoom levels for a shirt model. Fibers are noticeable only ant the closest distances, but their local radiance affects the overall aspect of the cloth.*

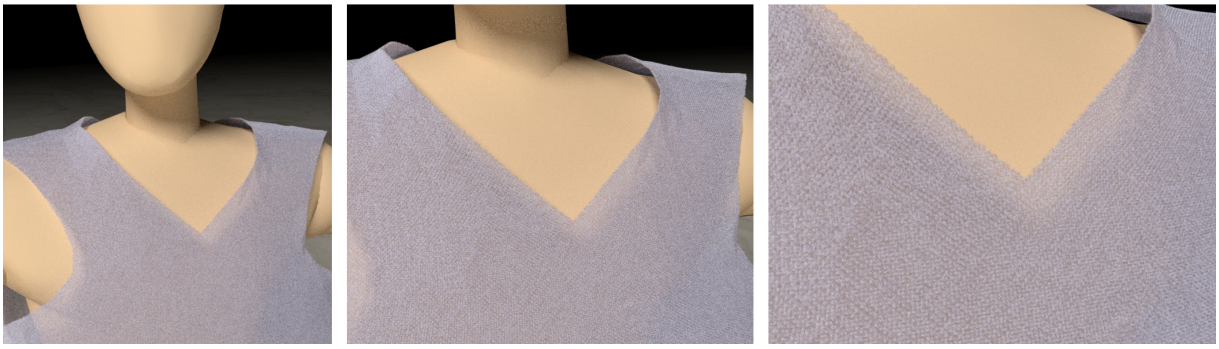

**Figure 7:** *Offline renderings pathtraced with the micro-flakes model (stdev = 0.3) for a dense cloth model (410k crossings). Our slice representation is rendered with 82 millions of triangles (41 million slices) and a 1857 x 1127 x 952 voxel grid*

from their maximum projection axis (constrained to x,y and z axes), that is, the plane where the projection of the triangle covers the maximum area. This maximizes the probability of generating fragments, however in our approach we allow the user to navigate freely around the mesh, generating fragments (and thus voxels) on the fly. In several cases this translates in tilted planes with a worst case: slices perpendicular to the camera plane. In this situation, the fragment computation produces wrong data when interpolating the world coordinates from the vertex shader, thus placing the voxels in wrong positions (see Figure 9). To avoid such problems, it is recommended to use antialiasing methods (a x32 filtering on the GPU is enough in our case) which have a negligible impact on the method.

One of the limitations of the current implementation is the depth sorting of slice quads, based on single-CPU (*std::sort*). For dense models this step could imply sorting an array with millions of model matrices, which takes around two seconds per frame in our case. While it is not required for volume generation, it would improve the user interactive navigation (we deactivate it during camera movements, computing it when the user stops at a given view) and it is good candidate for a multi-thread or GPU parallel implementation.

One problem which arises from rastering sub-pixel polygons with alpha blending mode is that at mid-range distances, only the highest mip maps of their corresponding textures will be accessed. At those levels, the interpolated alpha values are so blurred that the polygon will be almost transparent (see Figure 10). For that reason, we deactivate mip-mapping. In the Figure 6 we can observe how conservative rasterization at level 0 of the texture produces yarns which look thinner as they are further away from the camera (specially visible in the the leftmost image). This is due to the small probability of rastering a fragment pixel which is fully opaque in the fiber density distribution texture of the slice. This could be avoided with customized mipmapping (manually creating more suitable density texture levels, removing transparency at the highest levels), however in that case the 3D volume voxelization will not be coherent anymore, requiring a special treatment (e.g.: updating only when the fragment shader is accessing level 0 textures).

In terms of memory management, the 3D textures are the main limitation. We handle up to 6GB in three textures, for a

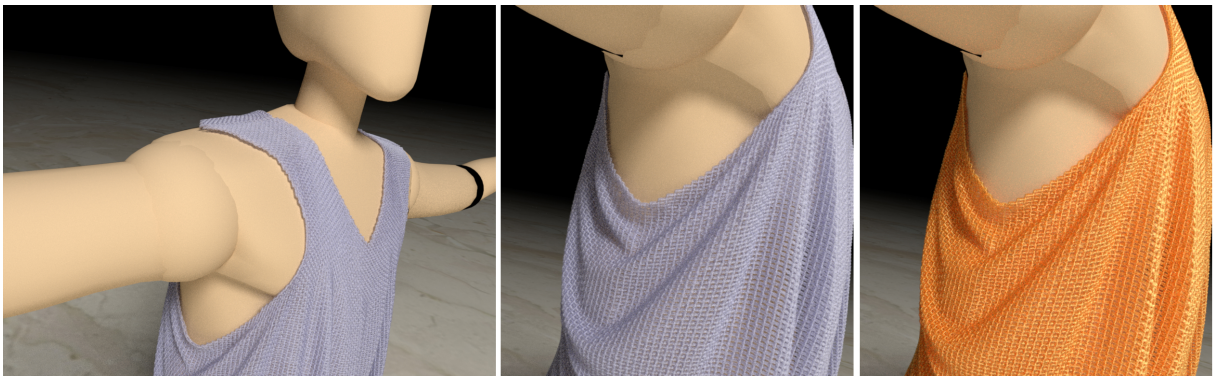

**Figure 8:** *Offline renderings of our volumetric data. Left, middle: two views of a low yarn count shirt (5.006 million slices). Right: Same model with different micro-flake distribution to simulate shinier fibers (stdev = 0.1)*

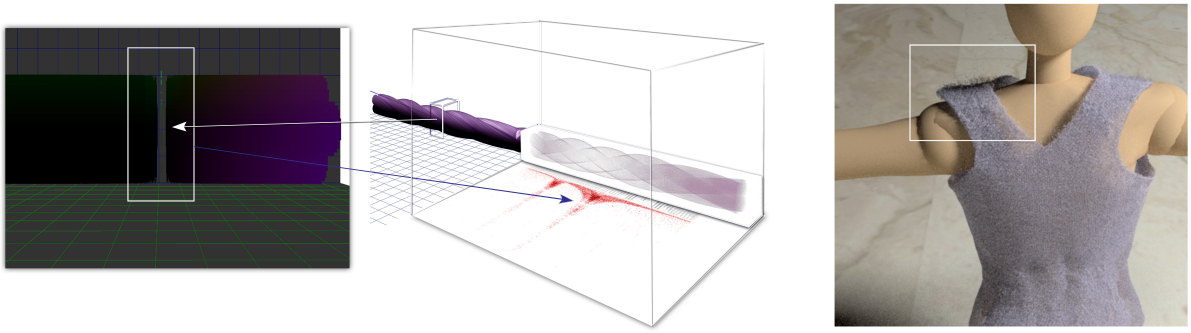

**Figure 9:** *The problem of aliasing. Left: Lateral view of a single yarn in OpenGL. In green, fragments produced by aliasing which store at wrong 3D coordinates. Right: Volumetric view of the yarn and projection of the voxels created in XZ plane. In red, wrong voxel positions produced by aliasing. The resulting volumes are noisy as shown on the shoulder in the right image (pathtraced render).*

maximum size of 2GB per texture and 2048 voxels of maximum size for any axis (limited by the graphic card, a Geforce GTX 670). In memory we keep 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 for the biggest models, but no impact afterwards, as it is only read for visualization.

As for yarns, the 2D textures for fiber distribution have no impact in performance, given that they are instantiated and shared among all the fibers. We have tested both 1024x1024 and 32x32 sizes in the present work with no influence in the final render, although mipmapping, if activated, does have a visible effect making the cloth almost transparent for 1024x1024 textures when observed from a long distance. In the table below (Table 1) we show computation times for both a light and dense model and two different zoom levels. For the first level we use 25 slices between yarn crossings (1mm, 1/40 of the pixel width) and we increase it to 50 for the closest view. We also increase the 3D texture resolution to the maximum available(6GB).

There is no available data for previous methods for comparison purposes. The only exception is the method by Xu et al. [XCL*01], however their numbers are outdated (over 31 minutes for the rendering of 368k slices).

## 5. Discussion and Future Work

The real time shading of fibers in our system is currently lambertian, however our system is a good candidate for techniques such as screen space subsurface scattering [JG10] orany volumetric illumination model [JSYR14]. Also phenomena like translucency could 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 limitation of memory space. In our renderings we store regular voxel grids of arbitrary size up to 2GB of storage size. In order to save space for higher resolution levels we would need both sparse, mipmapped

| | Low density cloth | High density cloth | Low density Close-up | High density Close-up |
|---|---|---|---|---|
| **Number of slices** | 2.5M | 17.6M | 4.08GB | 35.2M |
| **CPU memory** | 1.8GB | 7.2GB | 9GB | 12GB |
| **GPU memory** | 2.06GB | 3GB | 4GB+2GB(shared) | 4GB+2GB(shared) |
| **Initialization time** | 2s | 16s | 5.4s | 35s |
| **Per frame time** | 71ms | 150ms | 120ms | 400ms |

**Table 1:** *Performance and computation times of our method with two models at two different levels of detail.*

**Figure 5:** *Very dense model(17.6M slices), rendered at medium level of detail (25 slices per crossing, 1152 voxels in maximum axis). The volumetric data was generated with our method automatically in 30 seconds. Rendered offline with Mitsuba(stdev 0.1).*

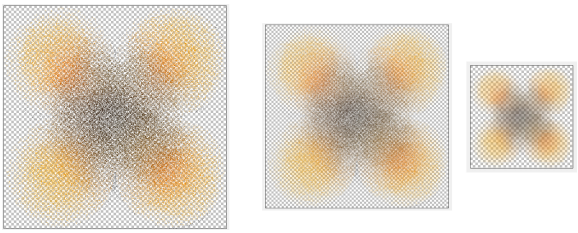

**Figure 10:** *From left to right: GPU automatic mipmapping (nearest neighbours) of a fiber density distribution texture, starting at level 0 (no filtering). Notice how the last map is almost transparent.*

representations and 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 studied for future improvements of this work.

We believe that the present work has a great potential to simulate complex visual aspects of the cloth under deformations (such as stretching). In these cases, fiber distribution could be updated (at texture level) producing a novel distribution which modifies the BSDF of the volumetric model (E.g.: under stretching, the torsion is diminished and fibers are condensed and aligned along the yarn path, increasing anisotropic reflectivity).


## Acknowledgements

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 was funded by the Spanish Ministry of Science and Education through a Juan de la Cierva Fellowship.


## References


[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., SUTER S.: State-of-the-art in compressed gpu-based direct volume rendering. *Computer Graphics Forum* (2014). 7

[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. 1, 4

[CR74] CATMULL E., ROM R.: A class of local interpolating splines. *Computer aided geometric design 74* (1974), 317–326. 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. 2

[Jak10] JAKOB W.: Mitsuba renderer, 2010. http://www.mitsuba-renderer.org. 4

[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. 1, 4

[JG10] JIMENEZ J., GUTIERREZ D.: *GPU Pro: Advanced Rendering Techniques*. AK Peters Ltd., 2010, ch. Screen-Space Subsurface Scattering, pp. 335–351. 6

[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. 6

[KJM08] KALDOR J. M., JAMES D. L., MARSCHNER S.: Simulating knitted cloth at the yarn level. *ACM Trans. Graph. 27*, 3 (2008), 65:1Ű65:9. 1

[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

[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

[MMN98] MEYER A., MEYER R., NEYRET F.: Interactive volumetric textures. In *In Eurographics Rendering Workshop* (1998), Eurographics, Springer Wein. ISBN, pp. 157–168. 1

[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. 4

[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. 1, 3, 6

[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. 3

[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. 1

## Appendix A: Shader code

### Vertex Shader

```
#version 430 core
// Input vertex data, different for all executions.
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;
layout(location = 2) in mat4 model_matrix;

// Output data ; will be interpolated for each fragment.
out vec3 Position_worldspace;
out vec2 UV;
out vec4 MVleft;
out vec4 MVup;
out vec4 MVforward;
out vec4 MVtrans;

// Values that stay constant for the whole mesh.
uniform mat4 P;
uniform mat4 V;

void main(){
    // Transform position by the model-view matrix
    // and then by the projection matrix.
    gl_Position = P * ( (V * model_matrix) * vec4(↩
         vertexPosition_modelspace,1));
    // Worldspace vertex position: M * position
    Position_worldspace = (mat4(model_matrix) * vec4(↩
         vertexPosition_modelspace,1)).xyz;
    MVleft= model_matrix[0];
    MVup= model_matrix[1];
    MVforward= model_matrix[2];
    MVtrans= model_matrix[3];
    //UV coordinates for textures
    UV = vertexUV;
}
```

### Fragment Shader

```
#version 430 core
// Interpolated values from the vertex shaders
in vec3 Position_worldspace;
in vec2 UV;

in vec4 MVleft;
in vec4 MVup;
in vec4 MVforward;
in vec4 MVtrans;

// Ouput data
out vec4 color;

// Values that stay constant for the whole mesh.
uniform sampler2D DiffuseTextureSampler;
uniform sampler2D OrientationTextureSampler;
uniform int voxelDims;
layout(r8, location = 1) uniform image3D Vol3D;
layout(r8, location = 3) uniform image3D Theta3D;
layout(r8, location = 4) uniform image3D Phi3D;

void main(){
    float PI = 3.14159265358979323846264f;
    //Access Level 0 texture
    vec4 MaterialDiffuseColor = textureLod(↩
         DiffuseTextureSampler, UV,0).rgba;
     //activate this one for smoother RT visualization
    //vec4 MaterialDiffuseColor = texture2D(↩
         DiffuseTextureSampler, UV).rgba;
        int x3D;
        int y3D;
        int z3D;

        x3D= int(Position_worldspace.x*voxelDims);
        y3D= int(Position_worldspace.y*voxelDims);
        z3D= int(Position_worldspace.z*voxelDims);

        vec4 accumCol= imageLoad(Vol3D, ivec3(x3D, y3D, ↩
             z3D));
        imageStore(Vol3D, ivec3(x3D, y3D,z3D), vec4(max(↩
             MaterialDiffuseColor.a,accumCol.x)));

        //Compute ORIENTATION-Store as polar coords.
        vec3 dOrient = texture2D(↩
             OrientationTextureSampler, UV).xyz;
        vec4 gOrient= vec4( normalize((dOrient.x*MVleft +↩
              dOrient.y*MVup + dOrient.z*MVforward).xyz),↩
              1.0f);
        float theta, phi;
        theta = acos(gOrient.z);
        theta = theta/PI;  //for theta in -pi, pi

        if (gOrient.x == 0.0f){
            phi = 0.0f;
        }else{
            phi = atan(gOrient.y, gOrient.x);
            phi= (phi+PI)/(2*PI);
        }

        imageStore(Theta3D, ivec3(x3D, y3D, z3D), vec4(↩
             theta)); //theta
        imageStore(Phi3D, ivec3(x3D, y3D, z3D), vec4(phi)↩
             );//phi
        //Final COLOR
        float dotp =dot(normalize(vec3(1.0f,-1.0f,0.0f)),↩
              gOrient.xyz);
        color=vec4(1.5f*dotp*MaterialDiffuseColor.xyz,↩
             MaterialDiffuseColor.a);
}
```