Real-time Fiber-level Cloth Rendering

Kui Wu*
University of Utah

Cem Yuksel†
University of Utah

Figure 1: ***Examples of rendering fiber-level cloth at real-time frame rates:*** *A sweater model that consists of 356K yarn curve control points and over 20M fiber curves, rendered using different yarn types with different fiber-level geometry. Notice the difference in appearance.*

Abstract

Modeling cloth with fiber-level geometry can produce highly realistic details. However, rendering fiber-level cloth models not only has a high memory cost but it also has a high computation cost even for offline rendering applications. In this paper we present a real-time fiber-level cloth rendering method for current GPUs. Our method procedurally generates fiber-level geometric details on-the-fly using yarn-level control points for minimizing the data transfer to the GPU. We also reduce the rasterization operations by collectively representing the fibers near the center of each ply that form the yarn structure. Moreover, we employ a level-of-detail strategy to minimize or completely eliminate the generation of fiber-level geometry that would have little or no impact on the final rendered image. Furthermore, we introduce a simple yarn-level ambient occlusion approximation and self-shadow computation method that allows lighting with self-shadows using relatively low-resolution shadow maps. We demonstrate the effectiveness of our approach by comparing our simplified fiber geometry to procedurally generated references and display knitwear containing more than a hundred million individual fiber curves at real-time frame rates with shadows and ambient occlusion.

Keywords: Cloth rendering, procedural geometry, textile

Concepts: •**Computing methodologies** → **Rendering;**

*e-mail: kwu@cs.utah.edu

†e-mail: cem@cemyuksel.com

I3D 2017, February 25-27, 2017, San Francisco, CA
ISBN: 978-1-4503-4886-7/17/03
DOI: http://dx.doi.org/10.1145/3023368.3023372

1 Introduction

In computer graphics cloth is typically represented as an infinitely thin (polygonal) surface. However, cloth is actually made up of a multitude of yarn pieces interlocked together, often knitted or woven. Yarn itself is also made up of a few plies, each of which can contain hundreds of fibers. Recently, researchers have shown that this yarn-level structure of cloth is important for simulation of cloth motion and deformation as well as realistic cloth rendering [Kaldor et al. 2008; Kaldor et al. 2010; Yuksel et al. 2012; Cirio et al. 2014; Cirio et al. 2015; Cirio et al. 2016; Zhao et al. 2016b]. Nonetheless, yarn-level representation of cloth not only consumes a considerable amount of memory for storage but it also involves handling a vast amount of geometry data for rendering, which makes it considerably expensive even for offline applications.

In this paper we present a real-time cloth rendering method with fiber-level details. Utilizing a procedural yarn model, our method is capable of rasterizing full garment models containing more than a hundred million individual fiber curves at real-time frame rates on current GPUs. We achieve this by generating simplified fiber-level geometry on the GPU using yarn-level control points and we provide an extra performance boost via a level-of-detail approach. We also introduce a yarn-level self-shadow computation method and a simple ambient occlusion approximation for high-quality lighting with limited resources.

We compare our simplified fiber-level models with reference models generated by a recent procedural yarn model [Zhao et al. 2016b]

Figure 2: ***Yarn structure:*** *Yarn typically consists of multiple plies, each of which is made up of tens to hundreds of micron-diameter fibers, depending on the yarn type.*

using parameters acquired via fitting CT-scan data. Our comparisons show that we can qualitatively reproduce the fiber-level geometric appearance of yarn. We also provide examples of full garment models rendered using our method (Figure 1). Since our method does not rely on any precomputation of the yarn-level control points, it is suitable for yarn-level cloth animation. Furthermore, our approach allows interactively changing the parameters of the procedural yarn model, thereby providing a new visualization mechanism for yarn-level cloth modeling and appearance editing.

While our method can support physically-based shading models, we use a simple shading model in our tests. Computing multiple scattering of light or global illumination is beyond the scope of this paper. Therefore, the methods we describe in this paper cover only a portion of a full cloth appearance modeling process with fiber-level details. This paper describes methods for efficiently handling the vast amount of geometric data of fiber-level cloth models with self-shadows and ambient occlusion at real-time frame rates on current GPUs.

2 Background

Fabric appearance has been an active research area in computer graphics. The geometric complexity combined with the optical complexity of light interaction make fabric appearance difficult to predict. Fabrics are constructed by interlocking multiple yarns pieces and they are often generated using knitting or weaving. A yarn itself also has a complex structure shown in Figure 2, formed by twisting a few sub-strands that are called plies. Each ply has a similar construction, formed by twisting tens to hundreds of individual fibers. The variations and imperfections in fiber geometry impact the overall appearance of the fabric.

2.1 Related Work

Most work on fabric appearance treat cloth as thin sheets with textures and use a specialized Bidirectional Reflectance Distribution Function (BRDF). Far-field BRDF models were introduced for approximating fabric appearance without an explicit yarn-level model [Ashikmin et al. 2000; Wang et al. 2008; Sadeghi et al. 2013]. For woven fabrics procedural patterns [Adabala et al. 2003; Kang 2010] were used for approximating fabric appearance with limited yarn-level detail, and the far-field appearance was improved using mip-maps [Yuen et al. 2012]. Fitting measured data to a detailed procedural model was used for capturing the anisotropic specular reflections for woven fabric [Irawan and Marschner 2012]. Recently, Schröder et al. [2015] proposed a pipeline for estimating the structure of a woven fabric from a single image. While most of these methods can produce realistic fabric appearance from a distance and some of them can even be used for real-time rendering [Adabala et al. 2003; Kang 2010; Yuen et al. 2012; Velinov and Hullin 2016], they can only handle woven cloth and cannot reproduce fiber-level details.

The importance of using a yarn-level representation for cloth was demonstrated in recent work on modeling [Yuksel et al. 2012] and simulation of knitted [Kaldor et al. 2008; Kaldor et al. 2010; Cirio et al. 2015] and woven [Cirio et al. 2014] cloth. Rendering such yarn-level models, however, has been a challenge. Though it is possible to explicitly render each fiber forming the yarn structure, the geometric complexity of this approach lead to volumetric approximations that convert the entire cloth model into volume data [Groller et al. 1995; Xu et al. 2001]. The volume data is generated by sweeping an image representing a cross-sectional distribution of yarn fibers along each yarn curve. Obviously, this creates a vast amount of volume data to be rendered. Lopez-Moreno et al. [2015] employed a similar approach for generating sparse volume data on the GPU, which allows rendering relatively small models interactively, but with limited fiber-level detail. Jakob et al. [2010] proposed a framework for volumetric modeling and rendering of materials with anisotropic micro-structure. Micro CT imaging was used for repeated fabric patterns for volumetric fabric modeling [Zhao et al. 2011] and explicitly modeling the interaction of light with micro-geometry [Khungurn et al. 2015]. For reducing the extensive storage requirements of volumetric fabric rendering, Zhao et al. [2016a] used the SGGX microflake distribution [Heitz et al. 2015] to represent volumetric yarn data and approximate the distant appearance by a down-sampling approach. Even though these methods can provide a remarkable level of realism, they are highly expensive in both storage and computation.

2.2 Procedural Yarn Model

Recently, Zhao et al. [2016b] described a procedural representation of fiber geometry forming the yarn structure, and provided the parameters of their method for real world yarn samples captured using micro CT imaging. The procedural fiber generation method we describe in this paper is based on this work, though we use a slightly different notation. The fiber geometry is defined in the ply-space, where the ply is aligned with the z-axis. The center of the i^{th} fiber $\mathbf{c}_i$ is defined parametrically using

$$\mathbf{c}_i(\theta) = [\, R\cos(\theta_i + \theta), R\sin(\theta_i + \theta), \alpha\,\theta/2\pi \,]^T \,, \qquad (1)$$

where θ is the polar angle that parameterizes the fiber helix, θ_i is initial polar angle for the fiber, R is the distance from the ply center, and α is a constant determining the twist of the fiber. The centers of plies $\mathbf{c}^{\text{ply}}$ twisting around the yarn are represented similarly in yarn-space, where the yarn is aligned with the z-axis.

Fibers can be classified into three types: migration, loop, and hair. Migration fibers are the most common fibers that twist around the ply regularly. Their distances to the ply center R, however, change continuously between two parameters R_{min} and R_{max} using

$$R(\theta) = \frac{R_i}{2}\left(R_{\text{max}} + R_{\text{min}} + (R_{\text{max}} - R_{\text{min}})\cos(\theta_i + s\theta)\right), \qquad (2)$$

where R_i is the distance of the i^{th} fiber to the ply center line, and s is a parameter that controls the length of the rotation. Loop fibers do not strictly follow this regular structure. They represent fibers that have been (accidentally) pulled out during the manufacturing process. They are handled by simply replacing R_{max} in Equation 2 with a larger value $R^{\text{loop}}_{\text{max}}$. Finally, hair fibers are fibers that have open endpoints sticking outside of the their plies. They significantly contribute to the fuzzy appearance of yarn.

3 Yarn Rendering with Fiber-level Detail

In our real-time fiber-level cloth rendering method we explicitly render fiber curves, as opposed to using a volumetric representation. Since a full garment model can easily have more than a hundred million fiber curves, we employ a number of simplifications to

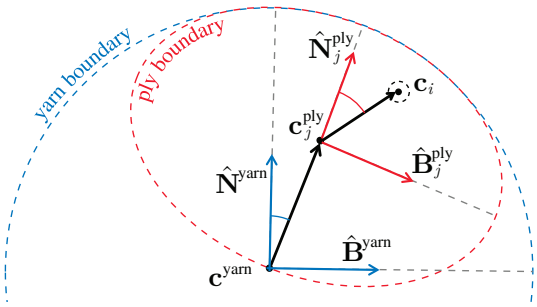

Figure 3: ***Placing fibers around the yarn:*** *The computation of fiber positions takes place on the cross-section plane perpendicular to the yarn curve.*

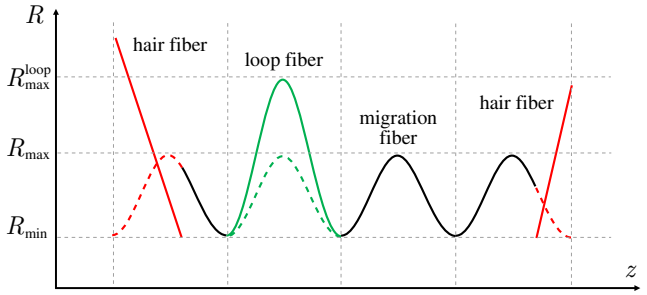

Figure 4: ***Fiber types:*** *Black curves are migration fibers with R values changing between R_{min} and R_{max}. The green curve is a loop fiber and the red curves are hair fibers.*

minimize the data stored and sent to the GPU, fiber segments actually drawn on the screen, and lighting computations needed for self-shadows and ambient occlusion.

3.1 Fiber Generation

We use a procedural fiber generation method based on the model of Zhao et al. [2016b]. For minimizing the data storage and the data transfer to the GPU, we generate the fiber curves on the GPU using control points that define the center of the yarn curves and a small number of parameters used by the procedural model. Thus, the cloth model we render is composed of a number of curves (cubic Bézier or Catmull-Rom), each of which is represented by four control points. For generating the individual fiber curves from the yarn curve we must compute the displacements from the yarn to each ply and then from each ply to its fibers.

We compute the displacement vector $\Delta\mathbf{c}_j^{\text{ply}}$ from the yarn center $\mathbf{c}^{\text{yarn}}$ to the center of the j^{th} ply $\mathbf{c}_j^{\text{ply}}$ at any given point along the curve (determined by θ) on the cross-section plane perpendicular to the yarn curve, as shown in Figure 3. Let $\hat{\mathbf{T}}^{\text{yarn}}$ be the unit tangent vector at a point along the yarn curve and $\hat{\mathbf{N}}^{\text{yarn}}$ be a perpendicular unit normal vector defining the orientation of the yarn. The displacement is calculated using

$$\begin{aligned}\Delta\mathbf{c}_j^{\text{ply}}(\theta) &= \mathbf{c}_j^{\text{ply}} - \mathbf{c}^{\text{yarn}} \\ &= \frac{1}{2}R^{\text{ply}}\left(\cos(\theta_j^{\text{ply}}+\theta)\,\hat{\mathbf{N}}^{\text{yarn}} + \sin(\theta_j^{\text{ply}}+\theta)\,\hat{\mathbf{B}}^{\text{yarn}}\right),\end{aligned}$$

where R^{ply} is the radius parameter of the ply, $\hat{\mathbf{B}}^{\text{yarn}} = \hat{\mathbf{T}}^{\text{yarn}} \times \hat{\mathbf{N}}^{\text{yarn}}$ is a perpendicular direction (forming an orthonormal basis with $\hat{\mathbf{T}}^{\text{yarn}}$ and $\hat{\mathbf{N}}^{\text{yarn}}$), and $\theta_j^{\text{ply}} = 2\pi j/n^{\text{ply}}$ is the initial polar angle of the j^{th} ply, and n^{ply} is the number of plies.

We compute the displacement vector $\Delta\mathbf{c}_i$ from the ply center $\mathbf{c}_j^{\text{ply}}$ to center of the i^{th} fiber $\mathbf{c}_i$ similarly. However, unlike yarn, the cross-section of a ply is not a circle, but it is elongated in the perpendicular direction to its displacement $\Delta\mathbf{c}_j^{\text{ply}}$ forming an ellipse. Let $\hat{\mathbf{T}}_j^{\text{ply}}$ be the unit tangent vector of the ply curve computed using the derivative $\partial\mathbf{c}_j^{\text{ply}}/\partial\theta$. The displacement vector is given by

$$\begin{aligned}\Delta\mathbf{c}_i(\theta) &= \mathbf{c}_i - \mathbf{c}_j^{\text{ply}} \\ &= R\left(\cos(\theta_i+\theta)\,\hat{\mathbf{N}}_j^{\text{ply}} e_{\mathbf{N}} + \sin(\theta_i+\theta)\,\hat{\mathbf{B}}_j^{\text{ply}} e_{\mathbf{B}}\right), \quad (3)\end{aligned}$$

where $\hat{\mathbf{N}}_j^{\text{ply}} = \Delta\mathbf{c}_j^{\text{ply}}/||\Delta\mathbf{c}_j^{\text{ply}}||$, $\hat{\mathbf{B}}_j^{\text{ply}} = \hat{\mathbf{T}}^{\text{yarn}} \times \hat{\mathbf{N}}_j^{\text{ply}}$, and $e_{\mathbf{N}}$ and $e_{\mathbf{B}}$ are the scaling factors for the ellipse.

Finally, $\mathbf{c}_i = \mathbf{c}^{\text{yarn}} + \Delta\mathbf{c}_j^{\text{ply}} + \Delta\mathbf{c}_i$ is computed using the two displacements and the yarn center. This way, given the control points of the yarn curve, we can compute any point on any fiber curve.

In this model, the radius of a fiber R changes periodically (using Equation 2) along the z-axis of the ply with a period of α (see Equation 1). We consider each full period of R separately and assign a fiber type. Figure 4 shows an example that includes four periods and their fiber types. The migration fibers simply follow Equation 2 and loop fibers merely use a different maximum radius $R_{\text{max}}^{\text{loop}}$ with the same equation, similar to the method of Zhao et al. [2016b]. These two fiber types span their entire periods. The hair fibers, however, are handled differently in our method for minimizing the number of fibers we generate. Instead of generating separate hair fibers, we effectively break parts of migration fibers and convert them into hair fibers. If a period is chosen to be a hair fiber, only a portion of the period is used for generating the hair fiber and the remaining portion is used as a migration fiber. Our hair fibers either start from a random radius $R_{\text{max}}^{\text{hair}}$ and linearly decrease to R_{min} or they extend in the opposite direction. We also use a different twist parameter α^{hair} for hair fibers, so that their rotations of a hair fiber around the ply is distinctly different from other fiber types. The similarities in the way that these three fiber types are handled allow generating these fibers on the GPU with minimal conditional branching in execution.

3.2 Core Fibers

While we can generate each individual fiber by computing its displacement from the yarn curve as explained above, considering that a ply can have hundreds of fibers, this can quickly lead to a large number of fiber curves to be rasterized. Reducing the number of fibers generated per yarn would not only reduce the geometry count, but it can also minimize the number of draw calls, since the current tessellation shaders can generate up to 64 curves (isolines) from a single curve. Fortunately, a portion of these fibers that are closer to the ply center are often occluded by other fibers closer to the surface of the ply. However, since these fibers near the center of the ply are not completely invisible, eliminating them altogether changes the appearance of the ply. Instead, we can use a lower-resolution representation for these fibers. We achieve this by collectively representing all fibers near the center of the ply using a single thick fiber that we call the *core fiber*. Thus, in our method each ply has a single core fiber that represents all fibers near the ply center. The number of fibers a core fiber represents depends on the parameters of the procedural model and it can consist of a significant portion of the fibers forming the ply.

The thickness of a core fiber is determined by the maximum distance of all fibers it represents to the ply center (i.e. the distance of the farthest fiber). Therefore, depending on the parameters of the

Figure 5: *The height channel of an example core fiber texture.*

procedural model, a core fiber can be considerably thicker than regular fibers. To make the core fiber appear like a collection of fibers, we use a precomputed texture on the core fiber. This texture is generated by rendering all fibers that correspond to the core fiber going through one full twist, as shown in Figure 5, and it is updated only when the parameters of the procedural model are modified. Storing a height-map, 2D surface normal, and an alpha channel in this texture we can reproduce the appearance of all fibers represented by the core fiber. However, these fibers only represent migration fibers and we use $s = 1$ for them, so that their radius period is the same as one full twist around the ply, thus the texture for one full twist tiles seamlessly. The loop and hair fibers are represented by other fibers that are explicitly generated.

When using this texture, the v (vertical) coordinate of this texture corresponds to the position along the thickness of the core fiber. The u (horizontal) coordinate, however, depends on both ply and fiber rotations, as well as the view direction, such that

$$u = \frac{1}{2\pi}\left(\frac{\theta\,(\alpha^{\text{ply}})^2\,\alpha}{\alpha^{\text{ply}} - \alpha} + \frac{\psi_{\text{view}}}{2}\right), \tag{4}$$

where ψ_{view} is the angle between the view direction and $\hat{\mathbf{N}}^{\text{yarn}}$.

3.3 Level-of-detail

Since we are generating the fibers on the GPU, we can employ a level-of-detail (LoD) strategy to minimize the number of fibers to be generated when the cloth model is viewed from afar. We use the width of a fiber in screen space to determine the number of fibers and subdivisions along the curve segments.

As the view point moves away from cloth and the width of a fiber gets smaller in screen space, its contribution to the final image gets smaller as well. In fact, as the fiber gets thin, its probability of intersecting with any of the shading sample points decreases. Thus, the expected visual contribution of a fiber is proportional to its screen-space area.

Therefore, in our method we adjust the number of fibers generated for a ply based on the screen-space width of a fiber ω^{fiber} placed at the center of the yarn curve. If the fiber width is smaller than a user-defined threshold ω^{LoD} (typically smaller than a pixel), we simply generate fewer fibers. To maintain a similar overall appearance for the plies, we in turn increase the width of the core fiber to compensate. We reach the maximum LoD level when the width of a ply ω^{ply} placed at the yarn center is smaller than the same threshold ω^{LoD}, in which case only core fibers are generated for the plies. Note that $\omega^{\text{ply}}/\omega^{\text{fiber}}$ is a constant determined by the parameters of the procedural yarn model. We compute our LoD scale factor λ^2 once for each yarn curve segment using

$$\lambda = \begin{cases} 1, & \text{if } \omega^{\text{LoD}} \le \omega^{\text{fiber}} \\ 0, & \text{if } \omega^{\text{ply}} \le \omega^{\text{LoD}} \\ \frac{\omega^{\text{fiber}}}{\omega^{\text{LoD}}}\left(\frac{\omega^{\text{ply}} - \omega^{\text{LoD}}}{\omega^{\text{ply}} - \omega^{\text{fiber}}}\right), & \text{otherwise,} \end{cases} \tag{5}$$

where $n^{\text{fiber}}_{\text{max}}$ is the maximum number of fibers to be generated. The thickness of the core fiber is scaled between its original thickness and the thickness of the ply using the same scaling factor λ^2. This scaling factor λ^2 is also used for adjusting the number of subdivisions along each fiber curve.

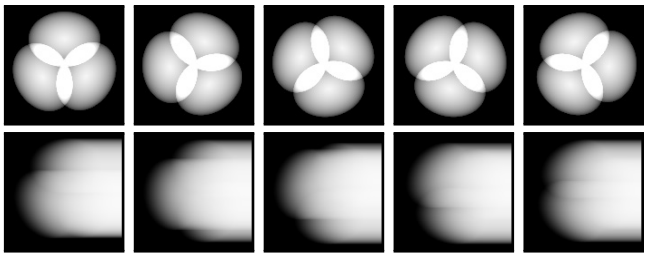

Figure 6: ***Precomputed self-shadows:*** *(Top row) density slices for different orientations of yarn and (bottom row) their corresponding shadow densities with light coming from the left sides of the images.*

We can also check if a yarn piece is in the view frustum using the control points of the yarn curve. If we detect that the yarn piece is outside of the view frustum, we simply discard the yarn piece without generating any fibers.

3.4 Self-Shadows

Self-shadows are extremely important for realistic fabric appearance. On the other hand, it is expensive to compute the fiber-level shadows of a yarn model. In particular, using shadow maps for individual fiber shadows would require an extremely high-resolution shadow map, since individual fibers are typically orders of magnitude thinner than the size of the cloth model. Therefore, we separate the fiber-level self-shadows within the yarn and the shadows between yarn pieces. The former is approximated using a precomputed self-shadow texture and the latter is handled via shadow mapping.

To simplify the self-shadow precomputation and substantially reduce its dimensionality, we do not consider individual plies for the self-shadow computation. Instead, we rely on the density function that we use for placing the fibers around a ply (the same density function as Zhao et al. [2016b]). Since the density function is circularly symmetric, we do not need to consider the twist of fibers around the ply center. We must, however, consider the twist of plies around the yarn direction, since the cumulative density function for the yarn is not circularly symmetric.

For further simplification we precompute self-shadows on the 2D cross-section plane of the yarn. Thus, for the purposes of self-shadow computation, the light direction is assumed to be perpendicular to the yarn direction. This allows parameterizing self-shadows using the relative orientation of the perpendicular light direction and the yarn twist.

Our precomputed self-shadow texture contains multiple slices of 2D self-shadow textures, each of which correspond to a different relative angle between the light direction and the yarn twist. In our implementation the light direction for all slices is aligned with the texture-space u-coordinate and each slice uses a different rotation of the yarn cross-section, as shown in Figure 6. Using the rotational symmetry of the yarn density, we only need to sample relative angles in the range $[0, 2\pi/n^{\text{ply}}]$. The shadow computation begins with computing the cumulative density function for each slice (Figure 6 top row). Then, for each pixel of each slice, we accumulate the total density on the left side of the pixel (in the opposite direction to the light direction). This value determines the total expected fiber density the light would have to go through to reach the pixel position.

We store the entire collection of these slices in a 3D texture, so that we can use trilinear filtering to lookup the total expected fiber density that light must go through to reach any point on the yarn cross-section for a given relative orientation of the yarn twist and the light direction. This total density value can be used with an

Figure 7: *An example of distance-based ambient occlusion.*

exponential decay function to estimate the light reaching any point within the yarn volume during shading.

While computing the shadow map for handling the shadows between yarn pieces, we simply render each yarn curve as a thick tube, completely disregarding the fiber-level structure of yarn. Nonetheless, since the shadow map is not used for computing the self-shadows within the yarn, this provides an acceptable and computationally efficient approximation of yarn geometry for shadow map generation.

3.5 Ambient Occlusion

The subtle impact of ambient occlusion can substantially improve the image quality, but it can also have a considerable computation cost. We opted to use a very simple ambient occlusion approximation that merely uses the distance to the yarn center to linearly scale the ambient light.

Even though this simple distance-based ambient occlusion approximation does not correspond to a traditional ambient occlusion formulation, it can provide the necessary visual queues for visualizing yarn with fiber-level detail. An example is shown in Figure 7. Notice that our distance-based ambient occlusion approximation is effective in accentuating the spaces between plies as well as individual fibers, providing an acceptable approximation for ambient occlusion.

4 Implementation Details

The input for our system are the parameters of the procedural yarn model and a set of curve control points for the yarn center. The fiber-level geometry is generated on the GPU using tessellation shaders, which also handle LoD and discard curve segments outside of the view frustum. Each fiber is generated as a collection of line segments (i.e. isolines), which are converted to camera-facing strips in the geometry shader.

We use a 1D texture for storing the fiber coordinates R_i and θ_i, accessed using fiber index i. These coordinates are ordered based on their distances to the ply center R_i, so that when LoD is used for reducing the number of fibers generated during rendering, the fibers that are skipped first are the ones closer to the ply center.

One important limitation of current GPUs for fiber-level cloth rendering with our method is that the tessellation shaders can only generate up to 64 isolines. This means that when using a typical yarn model that contains 3 plies, we can only have up to 21 fibers per ply, which is highly limited for yarn types that contain hundreds of fibers in each ply. Yet, this limitation is remedied by our core fibers, each of which can represent all fibers near the center of a ply using only a single isoline. This way, we can devote the remaining 20 isolines per ply to loop and hair fibers, as well as migration fibers near the surface of the ply for providing high-quality geometric detail.

Because we must generate different types of fibers (core, migration, loop, and hair) within the same tessellation shader, branching is unavoidable. Yet, the similarities between the fiber types help minimize branching in the tessellation shader. Moreover, we can completely eliminate the need for branching in the geometry and fragment shaders for handling different fiber types. Since the geometry shader merely converts lines to strips, sending it the fiber

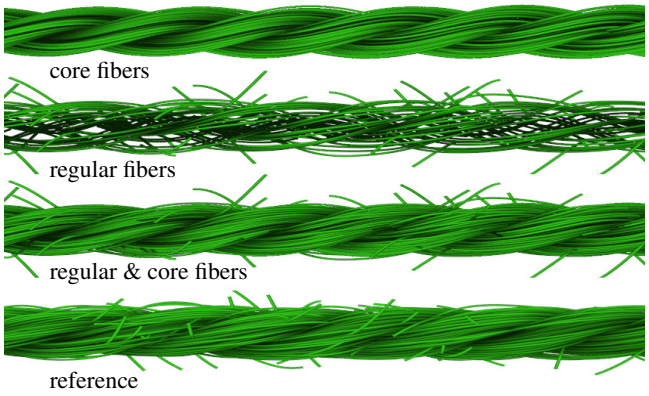

Figure 8: *An example yarn model with our core fibers and regular fibers, and the comparison of our full model to a reference model generated using the method of Zhao et al. [2016b].*

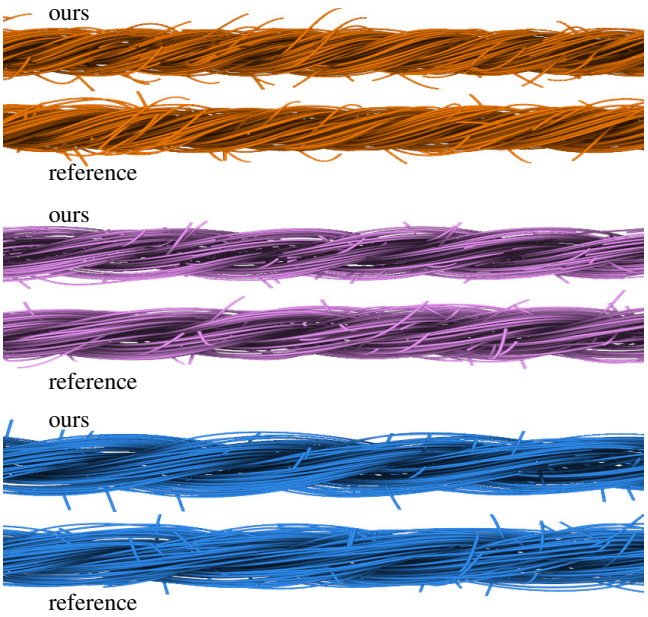

Figure 9: *Comparison of our fiber generation method to reference models generated using the method of Zhao et al. [2016b].*

thickness alone makes it indifferent to the fiber type. The fragment shader, on the other hand, must use a texture on the core fibers to account for the missing geometric detail. To unify the fragment shader operations for all fiber types, we use a different part of the same texture for regular (migration, loop, and hair) fibers as well. The only difference between a regular fiber and a core fiber in the fragment shader is their texture coordinates, which are computed in the tessellation shader.

Using a realistic shading model is important for reproducing fabric appearance. Nonetheless, in our implementation we used a simple shading model instead that has a diffuse and a single specular component. The diffuse component is based on the surface normal of a fiber, obtained from the fiber texture. For the specular component f_s we use a far-field approximation of surface reflectance for tubular shapes, similar to the hair shading model of Sadeghi et al. [2010], computed as $f_s = k_s \cos(\phi/2) \exp(-\theta_h^2/2\beta^2)$, where k_s is the specular color, ϕ is the azimuthal angle and θ_h is the longitudinal half angle between the light and view directions, and β is the longitudinal width of the specular lobe.

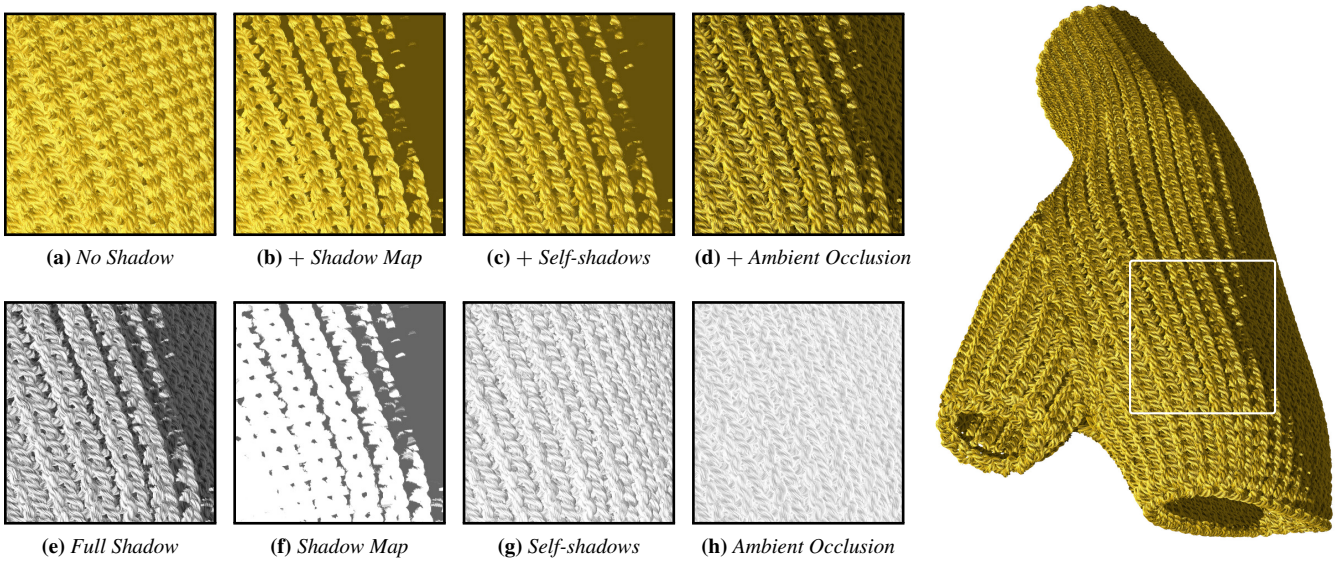

Figure 10: ***Shadow components:*** *A glove model showing the three shadow components: shadow map, self-shadows, and ambient occlusion.*

5 Results

We show the components of our fiber model in Figure 8. Notice that yarn with only core fibers looks too regular, as compared to the reference [Zhao et al. 2016b]. Our regular fibers (migration, loop, and hair fibers) provide the necessary irregularity, but the yarn model generated using only 60 regular fibers looks too sparse. Unfortunately, the tessellation limits of current GPUs prevent generating more fibers without additional draw calls. Our combined model including both regular and core fibers, however, can qualitatively match the appearance of the reference model. Other example comparisons using different yarn parameters are included in Figure 9. Note that our model cannot produce an exact match for the reference mainly because we handle hair fibers differently.

We display the contributions of different shadow components on an example glove model in Figure 10. In the case of rendering without any shadows (Figure 10a), diffuse shading and specular highlights provide some hint of the fiber-level geometry. The shadows between different yarn pieces introduced by the shadow map (Figure 10c) accentuate the knitted structure formed by the yarn curves, but they lack fiber-level details, which are introduced by the self-shadows (Figure 10b). The ambient occlusion component (Figure 10d) enhances the fiber-level details and also provides some hint of the fiber-level geometric details on fully shadowed areas, as opposed to a constant ambient component.

Figures 1 and 12 show two different sweater models rendered with different yarn types, generated using different procedural yarn parameters. In both figures, examples with different yarn types use the same yarn curve control points as input. The only difference is the fiber-level geometry generated on the GPU, but it is somewhat difficult to see the individual fiber details in these examples because of the distance of the yarn curves to the camera. Nonetheless, even though examples with different yarn types use the same shading parameters, the underlying fiber geometry still impacts the overall appearance.

Figure 11 shows a knitted cable pattern rendered with and without LoD. Since our LoD approach eliminates fibers that are less likely to intersect with any screen samples, the visual impact of our

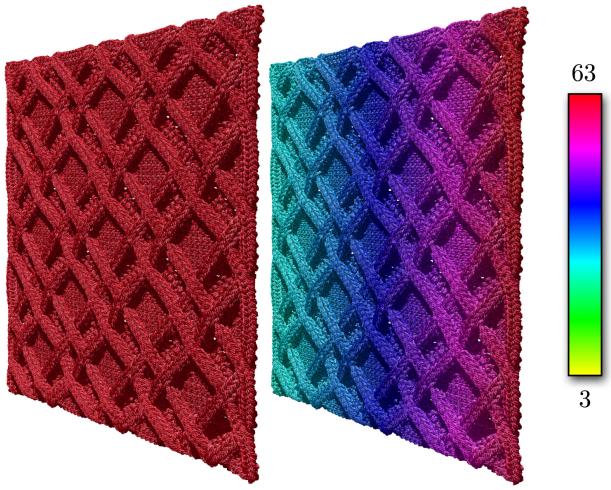

Figure 11: ***Level-of-detail:*** *(Left) disabling LoD generates* 63 *fibers everywhere, (right) enabling LoD generates varying numbers of fibers between* 3 *and* 63 *based on the fiber thickness in screen space, displayed with color coding.*

LoD is minimal, as long as it is not used aggressively (by setting a threshold value larger than one pixel size). Even with a high LoD threshold our core fibers produce high-quality results. However, since core fibers cannot represent hair fibers, a high LoD threshold is not advisable for rendering models with a high density of long hair fibers, such as the model on the right in Figure 1.

Example dress models with different yarn-level structures are shown in Figure 14. Each of them contains over a hundred million fiber curves. However, since our method generates the fibers on the GPU, we only store about two million yarn curve control points that define the yarn-level geometry. Furthermore, using our core fibers and our level-of-detail strategy, we generate and rasterize only a fraction of the total number of fiber curves.

Figure 12: ***Different yarn parameters:*** *A sweater model with 681K yarn curve control points and 1.5G fiber segments using different procedural yarn parameters, rendered using the same yarn-level control points, the same shading parameters, and the same lighting.*

Table 1: *Performance Results for Different Camera Distances*

Model	# of Control Points	# of Fiber Segments	Close View	Med. View	Full View
Glove (Fig.10)	148K	8.3M	9 ms	19 ms	9 ms
Sweater (Fig.1)	356K	20M	11 ms	23 ms	13 ms
Cables (Fig.11)	362K	20M	2 ms	23 ms	24 ms
Sweater (Fig.12)	681K	38M	11 ms	28 ms	15 ms
Dress (Fig.14-left)	1.89M	100M	9 ms	23 ms	20 ms
Dress (Fig.14-right)	1.99M	112M	11 ms	25 ms	21 ms

All performance results are obtained on an NVIDIA GeForce GTX 1080 GPU, rendering to an OpenGL viewport of size 1280×960.

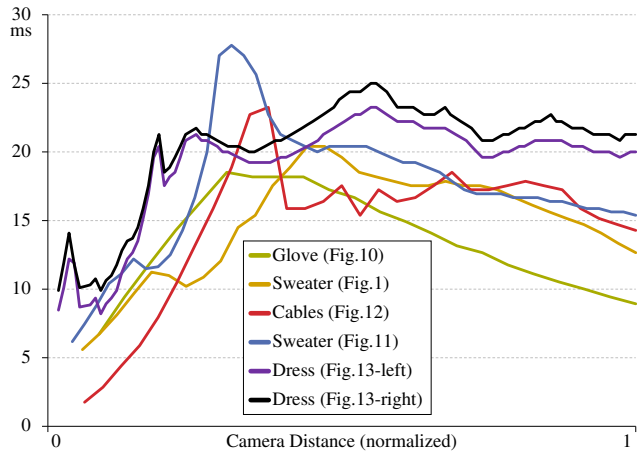

Figure 13: ***Performance Graphs:*** *The dependence of the frames per second performance values on camera distance for all examples in the paper. The camera distances are normalized such that the models are fully visible and cover the screen at their maximum distances, shown at the camera distance value of* 1.

We provide the performance results with different models in Table 1. Note that when the camera is far enough that models are fully visible, we generate fewer fibers using our level-of-detail strategy and achieve high performance (the "Far" column on the table). Close-ups achieve high performance as well, since we avoid rendering the yarn curves that are outside of the view frustum (the "Close" column on the table). Therefore, our worst-case performance appears at the medium distance, where the camera is close enough to the model to trigger the generation of most fiber curves but not close enough to cull a significant portion of the model. The "Med." column in Table 1 shows the worst performance we identified by manually adjusting the camera distance. We also provide the performance graphs for all models in the paper in Figure 13. The fluctuations in this graph are due to different parts of the models entering the view frustum as the camera moves away. Notice that our method achieves high performance at extreme close-ups and the performance begins to improve beyond a certain distance. The far distances in these graphs (around 1) and Table 1 are the closest camera distances that contain the entire model in the view frustum. As hinted by the tail end of these graphs, the performance continues improving as the camera distance increases.

6 Conclusion

We have presented a real-time fiber-level cloth rendering framework. Our method generates fiber-level geometry on the GPU during rendering. We have described a modified procedural fiber generation method for hair fibers, and we have introduced core fibers for greatly reducing the number of fibers that are generated on the GPU and allowing current GPUs with limited tessellation capabilities to render fiber-level yarn models using only yarn curve control points as input. Moreover, we have described a LoD approach for extra performance boost in distant views and close-ups. Furthermore, we have introduced an efficient self-shadow precomputation method for yarn and provided a simple ambient occlusion approximation. Our results show that our modified procedural model can produce qualitatively similar results to a state-of-the-art procedural fiber generation technique and we can render full size garment models with fiber-level detail at real-time frame rates.

Acknowledgments

We thank Charles Hansen, Yang Song, and Konstantin Shkurko for valuable discussions and suggestions. This work was supported in part by NSF grant #1538593.

Figure 14: *Two dress models with about 2M yarn curve control points and over 100M fiber curves, rendered using the same yarn type, shading parameters, and lighting conditions. The only differences are the control points of the yarn curves.*

References

ADABALA, N., MAGNENAT-THALMANN, N., AND FEI, G. 2003. Real-time rendering of woven clothes. In *Proceedings of ACM Virtual Reality Software and Technology*, 41–47.

ASHIKMIN, M., PREMOŽE, S., AND SHIRLEY, P. 2000. A microfacet-based brdf generator. In *Proceedings of ACM SIGGRAPH*, 65–74.

CIRIO, G., LOPEZ-MORENO, J., MIRAUT, D., AND OTADUY, M. A. 2014. Yarn-level simulation of woven cloth. *ACM Transactions on Graphics 33*, 6, 207:1–207:11.

CIRIO, G., LOPEZ-MORENO, J., AND OTADUY, M. A. 2015. Efficient simulation of knitted cloth using persistent contacts. In *Proceedings of SCA*, 55–61.

CIRIO, G., LOPEZ-MORENO, J., AND OTADUY, M. 2016. Yarn-level cloth simulation with sliding persistent contacts. *IEEE Transactions on Visualization and Computer Graphics PP*, 99, 1–1.

GROLLER, E., RAU, R. T., AND STRASSER, W. 1995. Modeling and visualization of knitwear. *IEEE Transactions on Visualization and Computer Graphics 1*, 4, 302–310.

HEITZ, E., DUPUY, J., CRASSIN, C., AND DACHSBACHER, C. 2015. The SGGX microflake distribution. *ACM Transactions on Graphics 34*, 4 (July), 48:1–48:11.

IRAWAN, P., AND MARSCHNER, S. 2012. Specular reflection from woven cloth. *ACM Transactions on Graphics 31*, 1, 11:1–11:20.

JAKOB, W., ARBREE, A., MOON, J. T., BALA, K., AND MARSCHNER, S. 2010. A radiative transfer framework for rendering materials with anisotropic structure. *ACM Transactions on Graphics 29*, 4, 53:1–53:13.

KALDOR, J. M., JAMES, D. L., AND MARSCHNER, S. 2008. Simulating knitted cloth at the yarn level. *ACM Transactions on Graphics 27*, 3, 65:1–65:9.

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

KANG, Y.-M. 2010. Realtime rendering of realistic fabric with alternation of deformed anisotropy. In *Proceedings of Motion in Games*, 301–312.

KHUNGURN, P., SCHROEDER, D., ZHAO, S., BALA, K., AND MARSCHNER, S. 2015. Matching real fabrics with micro-appearance models. *ACM Transactions on Graphics 35*, 1, 1:1–1:26.

LOPEZ-MORENO, J., MIRAUT, D., CIRIO, G., AND OTADUY, M. A. 2015. Sparse GPU voxelization of yarn-level cloth. *Computer Graphics Forum*, 1–13.

SADEGHI, I., PRITCHETT, H., JENSEN, H. W., AND TAMSTORF, R. 2010. An artist friendly hair shading system. *ACM Transactions on Graphics 29*, 4, 56:1–56:10.

SADEGHI, I., BISKER, O., DE DEKEN, J., AND JENSEN, H. W. 2013. A practical microcylinder appearance model for cloth rendering. *ACM Transactions on Graphics 32*, 2, 14:1–14:12.

SCHRÖDER, K., ZINKE, A., AND KLEIN, R. 2015. Image-based reverse engineering and visual prototyping of woven cloth. *IEEE Transactions on Visualization and Computer Graphics 21*, 2, 188–200.

VELINOV, Z., AND HULLIN, M. B. 2016. An interactive appearance model for microscopic fiber surfaces. In *Proceedings of Vision, Modeling, and Visualization 2016*.

WANG, J., ZHAO, S., TONG, X., SNYDER, J., AND GUO, B. 2008. Modeling anisotropic surface reflectance with example-based microfacet synthesis. *ACM Transactions on Graphics 27*, 3, 41:1–41:9.

XU, Y.-Q., CHEN, Y., LIN, S., ZHONG, H., WU, E., GUO, B., AND SHUM, H.-Y. 2001. Photorealistic rendering of knitwear using the lumislice. In *Proceedings of ACM SIGGRAPH*, 391–398.

YUEN, W., WÜNSCHE, B., AND HOLMBERG, N. 2012. An applied approach for real-time levelof-detail woven fabric rendering. *Journal of WSCG 20*, 2, 117–126.

YUKSEL, C., KALDOR, J. M., JAMES, D. L., AND MARSCHNER, S. 2012. Stitch meshes for modeling knitted clothing with yarn-level detail. *ACM Transactions on Graphics 31*, 3, 37:1–37:12.

ZHAO, S., JAKOB, W., MARSCHNER, S., AND BALA, K. 2011. Building volumetric appearance models of fabric using micro ct imaging. *ACM Transactions on Graphics 30*, 4, 44:1–44:10.

ZHAO, S., WU, L., DURAND, F., AND RAMAMOORTHI, R. 2016. Downsampling scattering parameters for rendering anisotropic media. *ACM Transactions on Graphics 35*, 6.

ZHAO, S., LUAN, F., AND BALA, K. 2016. Fitting procedural yarn models for realistic cloth rendering. *ACM Transactions on Graphics 35*, 4, 51:1–51:11.