Mechanics-Aware Deformation of Yarn Pattern Geometry

GEORG SPERL, IST Austria, Austria
RAHUL NARAIN, Indian Institute of Technology, India
CHRIS WOJTAN, IST Austria, Austria

Triangle mesh-based simulations are able to produce satisfying animations of knitted and woven cloth; however, they lack the rich geometric detail of yarn-level simulations. Naive texturing approaches do not consider yarn-level physics, while full yarn-level simulations may become prohibitively expensive for large garments. We propose a method to animate yarn-level cloth geometry on top of an underlying deforming mesh in a mechanics-aware fashion. Using triangle strains to interpolate precomputed yarn geometry, we are able to reproduce effects such as knit loops tightening under stretching. In combination with precomputed mesh animation or real-time mesh simulation, our method is able to animate yarn-level cloth in real-time at large scales.


Additional Key Words and Phrases: mechanics-aware, yarn-level cloth, real-time, knitted, woven, cloth simulation

ACM Reference Format:

1 INTRODUCTION

The intricate geometry of knitted and woven fabric gives rise to both visual and physical complexity. For example, knit loops tighten when the fabric is being stretched (Figure 2 right), or fabric might curl depending on the yarn pattern [Cirio et al. 2016; Sperl et al. 2020].

Existing research focuses either on the level of individual yarns, or on overall cloth behavior. Direct simulation of individual yarns [Bergou et al. 2010; Cirio et al. 2016; Kaldor et al. 2008; Sánchez-Banderas et al. 2020] produces detailed yarn behaviors, which can be scaled up to accurately simulate large-scale cloth behaviors albeit at great computational expense. On the other hand, spring- or mesh-based cloth simulation [Baraff and Witkin 1998; Müller et al. 2007; Sperl et al. 2020; Terzopoulos et al. 1987] efficiently approximate the behavior of fabric at the large scale, but they ignore the behavior of individual yarns.

This work proposes to bridge this gap by adding yarn-level deformations to mesh-based cloth simulation. We first precompute the behaviors of a periodic yarn pattern based on the large-scale deformation of the underlying cloth. We then interpolate these deformed yarn patterns at runtime based on the deformation state of the cloth mesh, resulting in richly detailed yarn-level geometry which rearranges in accordance with yarn-level mechanics in real-time. Figure 1 highlights some examples achieved with our method.
We also introduce an efficient way of linearizing the bending response of yarn patterns in terms of stretching, and we propose a way to clamp compression, allowing the user to tune the extent of yarn buckling. We implement the yarn deformation procedure as compute shaders on the GPU.

Overview. We start by discussing related work in Section 2. Then we detail the two main stages of our method: data generation and real-time displacement. Section 3 explains the data generation step shown in Figure 3, where we optimize for the rest configuration of yarn patterns under a range of large-scale deformations, and compute local displacements relative to these deformations. Section 4 details the real-time displacement phase illustrated in Figure 4, where we map the precomputed yarn geometry onto a deforming triangle mesh. Section 5 explores our method’s results. Section 6 discusses limitations and advantages, and concludes with a summary and future work.

2 RELATED WORK


Researchers recently combined yarn-level and mesh-level cloth simulators. Casarfranca et al. [2020] enriched a mesh-based cloth simulation with yarn-level simulation in regions of interest, and Sperl et al. [2020] used precomputed yarn patterns to develop a homogenized continuum material model. The latter use texture maps to visualize yarn geometry, so the individual strands do not react to the deformation of the cloth. Our work enhances these ideas by adding mechanics-aware yarn-level geometry to any mesh animation at negligible cost.

Example-Based & Mechanics-Aware Deformation. Our work animates yarn physics in real-time by interpolating from precomputed examples. Researchers also use example geometry to bias physics simulators toward preferred results [Gao et al. 2019; Koyama et al. 2012; Martin et al. 2011; Schumacher et al. 2012; Wampler 2016]. Similar to Ma et al. [2008]’s interpolation of displacement textures based on the deformation of a face mesh, we interpolate yarn geometry based on the deformation of a cloth mesh. Montazeri et al. [2019] deform yarn cross-sections by learning a model from precomputed simulations; our approach operates at one scale higher, by animating the reconfiguration of yarn patterns based on a database of precomputed examples.

Researchers use similar ideas to add fine-scale wrinkle details to a coarse cloth or flesh simulation. Procedural wrinkle methods rely on an underlying strain field [Hadap et al. 1999; Rohmer et al. 2010; Zuenko and Harders 2019] or explicit simulation of detail [Müller and Chentanez 2010], while data-driven wrinkle methods instead train local operators or pose-dependent systems [Kavan et al. 2011; Wang et al. 2010; Zurodo et al. 2012], with recent works building on recurrent or convolutional neural networks [Chentanez et al. 2020; Jin et al. 2020; Santesteban et al. 2019; Vidaurre et al. 2020]. Our work assumes that the input cloth mesh animation already contains all cloth-scale features including wrinkles, and then adds detail purely on a yarn scale.

3 DATA GENERATION

We want our embedded yarn details to deform realistically, so we prescribe their motion based on yarn-level simulations. This section describes the physics precomputation phase of our algorithm, illustrated in Figure 3.

3.1 Deformation Optimization

We use the method of Sperl et al. [2020] to simulate how each yarn pattern deforms. This method takes as input a particular yarn pattern in an undeformed configuration (e.g. the “stockinette” pattern in Figure 3), and a large-scale surface deformation encoded as the first I and second II fundamental forms. The method uses this large-scale deformation to define boundary conditions, and then optimizes for the elastostatic equilibrium configuration of the yarn pattern. Sperl et al. [2020] then reduced this geometric information down to a single scalar energy density used in a hyperelastic material model. In contrast, we will use the yarn geometry directly.

We model yarns as discrete elastic rods [Bergou et al. 2010], represented as connected lists of vertices with positions \( \mathbf{x} \) along the centerline. Each edge also stores a twist angle \( \theta \) and a reference vector \( \mathbf{d}_i \).\(^1\) We concatenate positions and twists for each vertex into a four-dimensional vector \( \mathbf{q} = (\mathbf{x}^\top, \theta^\top) \), where \( \theta \) corresponds to one incident edge. (Note that the reference directors \( \mathbf{d}_i \) are not degrees of freedom.)

\(^1\)The subscript in \( \mathbf{d}_i \) refers to its definition as one of two reference directors: one for the edge normal, and one for the edge binormal.
We recall from [Sperl et al. 2020] that during the optimization the kinematics of yarn vertex positions are defined as

\[ x(X) = \mathbf{x}(X) + \mathbf{\xi}(X), \]  
\[ \mathbf{x}(X) = \varphi(X_1, X_2) + X_3 \mathbf{n}(X_1, X_2). \]

Here, \( X = (X_1, X_2, X_3)^T \) are the material-space coordinates of the undeformed yarn pattern, with \( (X_1, X_2) \) the orthogonal and periodic directions along the pattern and \( X_3 \) the height coordinate. \( \mathbf{x} \) denotes the large-scale deformation constructed from the input fundamental forms \( \mathbf{I} \) and \( \mathbf{II} \), which encode in-plane and bending deformations respectively. From these fundamental forms, we construct the mid-surface \( \varphi \) with normal \( \mathbf{n} \), as described in [Sperl et al. 2020] and in Appendix A, such that \( \mathbf{I} = \nabla \varphi^T \nabla \varphi \) and \( \mathbf{II} = -\nabla \varphi^T \nabla \mathbf{n} \). We then solve for the yarn configuration which minimizes elastic energy. The optimization variables are the fluctuations \( \mathbf{\xi} \), which describe local displacements relative to the large-scale deformation, and the twists \( \Theta \). Using \( \Theta \) for undeformed twists, the concatenated coordinates are undeformed \( \mathbf{Q} = (\mathbf{x}^T, \Theta)^T \), large-scale-deformed \( \mathbf{\bar{q}} = (\mathbf{\bar{x}}^T, \Theta)^T \), and optimized \( \mathbf{q} = (\mathbf{x}^T, \Theta)^T \). Here, we assume that twists \( \Theta \) are unaffected by the large-scale mapping, which is true for pure in-plane deformations. In general, bending may induce local twists that depend on \( \mathbf{II} \) and the orientation of yarns relative to the curvature direction. In our experiments, we assume that this effect is small.

Crucially for our application, we found that the optimization of Sperl et al. [2020] contains a nullspace that allows yarns to slide: a periodic yarn curve \( x(s) \) parametrized by \( s \) can slide by a parametric shift \( \Delta s \) without changing its elastic energy \( E \), i.e. \( E(x(s)) = E(x(s + \Delta s)) \). Geometrically, such a shift corresponds to tangential sliding of yarn while maintaining the same periodic shape. To the optimizer any such state is equivalent, and the actual result may depend arbitrarily on numeric solver parameters. This nullspace does not affect the homogenized energies in [Sperl et al. 2020] by definition. However, interpolating between two shifted parameterizations may produce completely different shapes, as shown in Figure 5. In our experience, this nullspace creates distracting interpolation artifacts even for nearly-identical deformations. The problem remains as we sample the deformations more densely, manifesting as sharp discontinuities in deformation space.

We eliminate this parametric yarn sliding by adding a constraint to the optimization, effectively removing the nullspace. Specifically, we fix one vertex per periodic yarn to remain on the boundary of the pattern:

\[ \mathbf{\xi} \cdot (\nabla \varphi \mathbf{N}) = 0, \]

where \( \mathbf{N} \) is the undeformed normal to the respective pattern boundary, either \( \mathbf{N} = (1, 0)^T \) or \( \mathbf{N} = (0, 1)^T \). This sparse set of vertex constraints efficiently and effectively removes the aforementioned nullspace.
interpolation artifacts. This strategy allows us to find a physically realistic yarn shape for a given large-scale deformation described by I and II.

3.2 Sampling

Now that we can produce yarn geometry for a given deformation, we want to precompute a series of representative samples and interpolate them at runtime. I and II are each symmetric $2 \times 2$ tensors, so parameterizing deformation by I and II directly yields a 6-dimensional function which is expensive to precompute, store, and read at runtime. We aim to reduce this dimensionality by parameterizing the deformation with as few variables as possible.

Following Sperl et al. [2020], we reparameterize I as a 3-dimensional function using the in-plane strains

$$s_x = \sqrt{I_{11} - 1}, \quad s_y = \frac{I_{22}}{\sqrt{I_{11}I_{22}}}, \quad s_y = \sqrt{I_{22} - 1}. \quad (4)$$

II can be parameterized similarly by introducing additional curvature variables and increasing the dimensionality of the dataset. However, we show in Section 4 that bending deformation can be reasonably approximated in terms of stretching variables alone. This strategy lets us sample the entire large-scale deformation space with only three variables $s_x, s_y$, and $s_y$, significantly reducing memory and computation overhead. We sample these deformations on a regular 3D grid, and we discuss the performance, data size, and quality for different numbers of samples in Section 5.

3.3 Material-Space Displacements

At runtime, our interpolated yarn geometry will be mapped onto a deformed triangle mesh, where it will naturally inherit the deformation of its triangle (the mapping from material-space to world-space in Figure 4). Thus, we want to store all of the deformation except the large-scale deformation in our precomputation as material-space displacements (the "pullback" column in Figure 3).

To do this, we need to find the modified material space coordinates $\hat{X}$ which give us our desired world-space deformations $x$ when deformed by only the large-scale deformation $X(\hat{X})$: in other words, find $\hat{X}$ s.t. $x = X(\hat{X})$. We perform this solve using Newton’s method and provide more details in Appendix B.

### Algorithm 1 Real-time Yarn Animation

**Input:** mesh animation, yarn pattern, displacement data $\Delta Q$
**Output:** deformed yarn geometry $q$

1. **procedure** `Animate Yarns`
2. $Q \leftarrow$ yarn pattern tiled over mesh
3. **Each Frame:**
4. compute I, II per face
5. interpolate I, II to mesh vertices
6. **for each yarn vertex do** \(\triangleright\) on the GPU
7. interpolate I, II from mesh
8. compute linearized bending $\iiota(X_3)$ \(\triangleright\) (7)
9. clamp compression \(\triangleright\) Section 4.3
10. compute strains $s_x, s_y, s_y$ \(\triangleright\) (4)
11. look up displacements $\Delta Q$ \(\triangleright\) Section 3.3
12. displace: $\hat{Q} \leftarrow Q + \Delta Q$ \(\triangleright\) (8)
13. map: $q \leftarrow (x(\hat{X})^T, \hat{\theta})^T$ \(\triangleright\) Section 4.5
14. **end for**
15. tessellate and render $q$ \(\triangleright\) Supplementary S2
16. **end procedure**

Concatenating $\hat{Q} = (X^T, \hat{\theta})^T$, we subtract the initial material state to get displacements

$$\Delta Q = \hat{Q} - Q. \quad (5)$$

Note that at the rest pose, $I = I_d$, $II = 0$, and $\Delta Q = 0$.

To summarize, we can now build a database of material-space yarn displacements $\Delta Q$ for each vertex $i$ of a pattern and for a range of in-plane deformations $j$: $\Delta Q_i(s_{xj}, s_{aj}, s_{yj})$. This database can be interpreted as a grid of example deformations, or similarly as a 3D displacement texture per yarn vertex. Interpolating between these samples takes a planar deformation $s_{xj}, s_{aj}, s_{yj}$ and maps it to a yarn-level displacement map. Note that this will recover the exact deformed yarn pattern for the strains sampled in the database and approximate patterns for intermediate strains.

4 REAL-TIME DISPLACEMENT

Now that we have a database of yarn pattern displacements for a range of deformations, we apply them to a yarn pattern tiled over an animating triangle mesh. In the following discussion, we denote $Q$ as the undeformed (material space) coordinates, $\hat{Q}$ as the coordinates deformed by local displacements, and $q$ as the final world-space coordinates, as illustrated in Figure 4. Algorithm 1 outlines our procedure.

As a precomputation, we first create the initial undeformed yarn mesh corresponding to the undeformed triangle mesh (inset). We generate a 2D background grid in the mesh’s UV-coordinates with cells the size of the periodic pattern, and we copy the yarn geometry into every cell that overlaps an undeformed triangle. We then remove
yarn vertices that do not lie within a triangle and delete yarn fragments shorter than a user-specified length for aesthetic purposes as described in the supplementary (Section S1). Finally, we precompute the material-space barycentric coordinates for each yarn vertex.

4.1 Mesh Strains
Each animation frame, we compute discrete fundamental forms for each triangle as in [Sperl et al. 2020]:

\[
I = F^\top F, \quad II = F^\top \Lambda F, \quad (6)
\]

where \(F\) is the triangle deformation gradient and \(\Lambda\) is the triangle-averaged shape operator from [Grinspun et al. 2006]. We then distribute them to triangle vertices using modified Shepard weights from Phong interpolation [James 2020], and finally interpolate them to each yarn vertex.

4.2 Linearized Bending
As alluded to in Section 3.2, we can approximate the effect of bending behavior by adding stretching and compression depending on the surface curvature. We show in Appendix C.1 that we can enhance the first fundamental form with one that varies along the surface normal:

\[
I(X_3) \approx I - 2X_3II, \quad (7)
\]

We illustrate this idea in the inset figure to the right, which approximates the extruded volume around a curved blue midsurface (top) with a linearized volume that is stretched above and compressed below the midsurface (bottom). We compare this idea to other bending models in Section 5.

4.3 Compression Clamping
Like most elastic materials, cloth buckles out of plane when compressed. The chaotic nature of this buckling can make our yarn optimization reach multiple visually distinct configurations that do not vary smoothly over deformation space (Figure 6, top). Sperl et al. [2020] dealt with this issue regularizing the fit of their output energies. We similarly regularize compression by clamping the eigenvalues of \(I\) to a lower bound before looking up the yarn displacement, which reduces buckling in a user-tunable way.

Using the method of Deledalle et al. [2017], we set a minimum value \(\lambda_{\text{min}}\) for the eigenvalues of \(I(Z)\). (Unless stated otherwise, all of our experiments use \(\lambda_{\text{min}}=0.8\), where \(\lambda<1\) is compression.) Because \(\Delta Q\to 0\) as \(\lambda\to I\), our technique for clamping compression will only reduce local deformations, but it will still deform due to the triangle embedding. Figure 6 (bottom) shows how this clamping reduces buckling while preserving the large-scale deformation.

4.4 Local Displacement
After clamping \(I\), we convert it to strains \(s_x, s_y, s_z\) (Equation (4)), tri-linearly interpolate the yarn displacement \(\Delta Q(s_x, s_y, s_z)\), and compute the deformed material space yarn coordinates

\[
\hat{Q} = Q + \Delta Q. \quad (8)
\]

We clamp strains outside of the sampled range to their nearest neighbor in the \(\Delta Q(s_x, s_y, s_z)\) dataset. Similar to compression clamping, this constant extrapolation will limit the local deformations while still inheriting large-scale deformations from the mesh embedding (See Figure 7).

4.5 World-Space Mapping
To map the yarn vertices to world space \(x\), we use:

\[
x(\hat{X}) = \varphi(\hat{X}_1, \hat{X}_2) + \hat{X}_3n(\hat{X}_1, \hat{X}_2), \quad (9)
\]

which corresponds to extruding the world-space mesh surface \(\varphi\) along its normal \(n\). To avoid piecewise linear embedding artifacts, we employ Phong deformation [James 2020] and interpolated vertex normals to generate a smoother surface \(\varphi\) and shell-volume (Figure 8).

To map yarn twists, we simply copy the updated twist values \(\theta=0\), and we co-transform the edge normals \(d'\) using an approximate mapping of the Jacobian of (9), as detailed in the supplementary material Section S2.1.

The computation of yarn vertex deformation and world-space mapping are trivially parallel, so we implement them as GPU compute shaders. The interpolation of \(\Delta Q\) results in a single 3D texture interpolation per yarn vertex.
4.6 Real-Time Rendering

Computer graphics researchers have developed a number of algorithms for rendering yarns and fibers [Montazeri et al. 2020, 2019; Wu and Yuksel 2017]. For the examples in this paper, we tessellate the deformed yarns as cylindrical meshes in a geometry shader. We approximate ply- and fiber-level detail with procedurally twistable normal maps and ambient occlusion maps, and we approximate volume conservation by locally rescaling yarn radii when they are stretched. We provide the full rendering details in the supplementary document Section S2.

5 RESULTS

We will now discuss examples generated by our mechanics-aware yarn animation technique. Figure 9 shows how our algorithm compares to a more typical embedded approach to animating yarn-level geometry detail. Our method naturally reacts to the deformation of the underlying cloth by causing loops to rearrange and tighten up as the tension is increased. As a natural consequence, our approach also reproduces the tendency of knitted and mesh fabrics to become more transparent when stretched. Figure 10 shows how we can easily apply different yarn-patterns to any cloth mesh, producing visually distinct geometry which depends on both the deformations of the mesh and the precomputed yarn mechanics. Our method can add yarn-level details onto any deforming triangle mesh: examples in this paper use deforming cloth meshes from ArcSim [Narain et al. 2013, 2012], position-based dynamics [Müller et al. 2007], and Blender [2020].

We enhance an offline cloth simulation solver which reproduces the large-scale effects of knitted garments [Sperl et al. 2020]. Figure 11 compares this approximation to ground-truth yarn-level simulations, yielding visually plausible recreations of the local pattern deformation. We note that the accuracy of our method highly depends on the accuracy of the underlying triangle deformation, so we see higher discrepancy where the FEM cloth simulation deviates from yarn-level simulation.

We also used our approach to add yarn-level details to a position-based dynamics cloth solver [Bender et al. 2015], to approximate yarn-level cloth simulation in real-time. Figure 12 shows how a user can perform an interactive dressing operation by pulling a knitted sock onto a foot. Note that our method still produces plausible yarn-level deformations, despite the fact that the default position-based elastic material model is quite far from a model derived from yarn physics.

Figure 13 shows we can render our yarn geometry offline to produce higher quality path-traced scenes with “fuzz” from a procedural particle system. Figure 14 shows how our GPU-based yarn
displacement algorithm allows for the animation of garments with millions of individual yarn vertices at interactive rates. Please refer to our supplementary video for additional highly detailed results. For reproducibility, we release the source code and data used to generate our results (https://git.ist.ac.at/gsperl/MADYPG).

Performance. We ran our method on a desktop computer with an Intel Core i7-7820X processor and an NVIDIA GeForce GTX 1080 Ti graphics card and gathered performance statistics. Table 1 breaks down the computational cost of our algorithm for each of the various examples in this paper. Almost all examples in this paper perform both geometry generation and rendering in real-time (well over 60 fps). To stress-test our method, we created the knitted sweater in Figure 14 (right), which has more than 40 million yarn vertices and generates yarn geometry at over 14 fps. The total time including rendering (depending on the complexity of the renderer and the amount of geometry per pixel due to camera zoom) is around 4 fps for this example. Please see our submission video for more detailed demonstrations of our method.

Dataset Size. The number of samples in our precomputed yarn database only modestly affects the visual quality of the animation. Figure 15 shows a deformed rib pattern with a varying number of database samples and a visualization of the associated interpolation error. We also provide animations with several sampling densities in our supplementary material. In practice, we found that a small number of samples was sufficient to capture strong effects like loops tightening under tension, and we noticed surprisingly few obvious interpolation errors (like interpolated threads leading to self-collisions).

On the other hand, a large number of database samples led to unwanted noise in the animations, like sudden ‘pops’ from one configuration to another, instead of gradual ones. We believe our observations are consistent with those of Sperl et al. [2020], who noted that a dense sampling rate (especially in the compressive regime where fabric buckles chaotically) required explicit filtering to remove high-frequency noise. In our case, simply removing samples from the database acts as an effective low-pass filter, and our compression clamping technique strongly reduces popping.

The complexity of our yarn database has only a small affect on runtime. Table 2 relates the number of precomputed yarn-level simulations in our database to the memory and runtime of our method. Notably, a 238× increase in database size caused a proportional increase in memory but only a 2.8× increase in runtime. Unless stated otherwise, the results shown in this paper use datasets of 9^3 samples sampled uniformly over the ranges sx, sy ∈ [−0.2, 1.0] and sa ∈ [−0.7, 0.7].

Bending Models. Finally, we compare the effect of our linearized bending approximation in Figure 16. We compare our linearized model described in Section 4.2 to a bending model that explicitly captures combined stretching and bending. (See Appendix C for details on these comparison models, and see our supplementary videos for an animated version of this and similar tests.) First of all, we found that the importance of bending depends on the knit pattern: “thick” patterns like the rib (Figure 10 red, center) exhibit more differences than nearly planar ones like the stockinette (Figure 10 blue, right). The further away the geometry is from the surface φ, the more it is locally stretched or compressed by bending. Second, even for a thick pattern, the differences between the two models are minor and localized to regions of strong curvature. Most importantly, the linearized model is much more efficient — when comparing CPU-based implementation, our linearized model ran roughly 8× faster.

An extra benefit of the linearized bending model is that it transforms bending-induced buckling into compression, which can be filtered with our compression clamping algorithm (Section 4.3). Otherwise, it would be non-trivial to filtering both compression- and bending-related buckling in a compatible and coherent manner.

Fig. 13. Two examples of our yarn geometry that were rendered offline using path tracing and hair particles.

Fig. 14. Our method scales favorably with increasing yarn density. We deform yarn geometry for a sweater with 0.8 million vertices at well over 100 FPS (left), and a sweater with 42.7 million vertices at over 14 FPS (right).

Fig. 15. Comparison of different data sampling densities. The top and bottom rows show results generated with 5 × 5 × 5 and 31 × 31 × 31 samples respectively. In the top images, we color-code the L2 error in vertex positions between the coarse and fine model, compared to the yarn radius R; red color corresponds to a larger difference. The coarse model produces plausible geometry even in the presence of larger errors.

Table 2. Data sampling density and performance.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>9^3 samples</td>
<td>14 FPS</td>
</tr>
<tr>
<td>31^3 samples</td>
<td>11 FPS</td>
</tr>
<tr>
<td>256^3 samples</td>
<td>7 FPS</td>
</tr>
</tbody>
</table>

ACM Trans. Graph., Vol. 40, No. 4, Article 168. Publication date: August 2021.
Table 1. Performance breakdown of results in this paper. From left to right, we list: the respective figure, the number of animated yarn vertices and average per-frame times of mesh strain computation (Section 4.1), data look-up and local yarn displacements (Sections 4.2 to 4.4), and embedded world-space mapping (Section 4.5). The mesh stage is implemented on the CPU, whereas yarn displacement and mapping are implemented on the GPU. In comparison, the CPU-implemented position-based dynamics step for the sock example averaged to 19.24 ms.

<table>
<thead>
<tr>
<th>animation</th>
<th># vertices</th>
<th>strains_{CPU}</th>
<th>displacement_{GPU}</th>
<th>mapping_{GPU}</th>
</tr>
</thead>
<tbody>
<tr>
<td>sleeve</td>
<td>Fig. 9</td>
<td>94k</td>
<td>1.02 ms</td>
<td>0.13 ms</td>
</tr>
<tr>
<td>patterns (averaged)</td>
<td>Fig. 10</td>
<td>82k</td>
<td>0.82 ms</td>
<td>0.09 ms</td>
</tr>
<tr>
<td>honeycomb stretch</td>
<td>Fig. 11 top</td>
<td>74k</td>
<td>0.61 ms</td>
<td>0.08 ms</td>
</tr>
<tr>
<td>rib stretch</td>
<td>Fig. 11 center</td>
<td>154k</td>
<td>0.77 ms</td>
<td>0.16 ms</td>
</tr>
<tr>
<td>stockinette stretch</td>
<td>Fig. 11 bottom</td>
<td>119k</td>
<td>0.59 ms</td>
<td>0.12 ms</td>
</tr>
<tr>
<td>stockinette sweater</td>
<td>Fig. 14 left</td>
<td>862k</td>
<td>2.59 ms</td>
<td>0.81 ms</td>
</tr>
<tr>
<td>fine stockinette sweater</td>
<td>Fig. 14 right</td>
<td>42.7M</td>
<td>2.59 ms</td>
<td>35.34 ms</td>
</tr>
<tr>
<td>rib twist</td>
<td>Fig. 13 left</td>
<td>433k</td>
<td>0.38 ms</td>
<td>0.51 ms</td>
</tr>
<tr>
<td>table cloth</td>
<td>Fig. 13 right</td>
<td>130k</td>
<td>1.50 ms</td>
<td>0.16 ms</td>
</tr>
<tr>
<td>sock</td>
<td>Fig. 12</td>
<td>139k</td>
<td>7.5 ms</td>
<td>0.14 ms</td>
</tr>
</tbody>
</table>

Table 2. Comparison of rib pattern displacement data for different dataset sizes. "generation" refers to the time needed to precompute the data, and "displacement" refers to the per-frame time of computing $\hat{Q} = Q + \Delta \hat{Q}(s_x, s_a, s_z)$ on a representative sweater animation with 1.8 million yarn vertices.

<table>
<thead>
<tr>
<th># samples</th>
<th>memory</th>
<th>generation</th>
<th>displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5^3 = 125$</td>
<td>0.7 MB</td>
<td>2.2 min</td>
<td>1.58 ms</td>
</tr>
<tr>
<td>$9^3 = 729$</td>
<td>4.2 MB</td>
<td>12.7 min</td>
<td>1.82 ms</td>
</tr>
<tr>
<td>$15^3 = 3375$</td>
<td>19.2 MB</td>
<td>59.8 min</td>
<td>2.30 ms</td>
</tr>
<tr>
<td>$31^3 = 29791$</td>
<td>169.7 MB</td>
<td>533.6 min</td>
<td>4.35 ms</td>
</tr>
</tbody>
</table>

Fig. 16. We compare our linearized bending model against a model incorporating bending strains explicitly on an animated knit sweater. We measure the difference between the models as the $L^2$ norm of the difference in yarn vertex positions, and we color code them relative to the yarn radius $R$; red color corresponds to a larger difference. The cut-outs show how knit loops appear to tighten more with the linearized model (top) than the explicit bending model (bottom).

Fig. 17. Approximating a seam by folding one piece of cloth over another on the mesh-level, rather than the thread level.

6 DISCUSSION

Because our method is based on geometric interpolation of yarn vertices rather than an exact simulation of yarn-level geometry, it cannot exactly reproduce all yarn-level effects. Our method fundamentally assumes that the deformations sampled in the database are representative of the deformations in the online simulation. Because our per-vertex mesh strains only communicate deformations on the scale of a single triangle, the method will not be able to accurately react to fine-scale collision events like pulling on an individual thread. Similarly, our method relies on the mesh simulation to handle collisions; it will not exactly resolve object collisions on the level of the individual thread, and it will not exactly resolve collisions for very thick fabrics if the mesh is modeled as infinitely thin.

Our database currently also ignores time-dependent effects like hysteresis and damping, so repeating a mesh deformation will yield exactly the same fine-scale yarn arrangements. Our method interpolates the precomputed behaviors of a periodic yarn pattern, so we cannot yet simulate clothes consisting of completely aperiodic or disorganized threads, and it will be significantly more expensive to simulate ornate patterns with a large number of yarn vertices. We also do not yet handle non-periodic connections between different patches of cloth, so our method cannot yet sew together seams on the individual thread level. However, we can approximate larger seams by folding a piece of thin fabric over itself (Figure 17).

On the other hand, our geometric approximation affords us a number of advantages and leads to a few novel challenges. The data-driven approach completely avoids the expense of online collision...
handling, and its exploitation of periodic structures avoids redundant computations resulting from structurally similar yarn patterns. The method cuts the complexity of brute-force yarn level cloth by several orders of magnitude, and as far as we know, our method is the only way to simulate millions of yarn vertices at interactive rates. The interpolative nature of the approach also guarantees unconditional numerical stability, so the method cannot blow up even in unrealistic video game environments. The speed and detail of our approach for animating knitted garments also creates new research challenges: zooming out from a pattern made of millions of threads can cause aliasing patterns when many of them occupy a single pixel, potentially requiring novel geometry anti-aliasing techniques in the future.

Conclusion. We have presented a method for deforming yarn patterns in a mechanics-aware manner. It reproduces characteristic yarn-level cloth behaviors like knitted loops that tighten when the fabric is stretched. We introduced practical heuristics such as linearizing bending and limiting buckling, which make the method significantly more efficient and tunable by artists. The method is lightweight and GPU-parallelizable, so it is capable of animating millions of yarn vertices at real-time rates. In the future, we are interested in new research challenges introduced by the massive scale of these yarn simulations. The method could further benefit from level-of detail approaches simplifying yarn geometry where it is not visible. Our technique might also be useful for research into deformation-dependent microfacet rendering or anti-aliasing techniques, for smoothly replacing extremely dense yarn geometry with an analytic or data-driven shading model.

ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers and the members of the Visual Computing Group at IST Austria for their valuable feedback. We also thank Seddi Labs for providing the garment model with fold-over seams.

This research was supported by the Scientific Service Units (SSU) of IST Austria through resources provided by Scientific Computing. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 638176. Rahul Narain is supported by a Pankaj Gupta Young Faculty Fellowship and a gift from Adobe Inc.

REFERENCES


A Optimization mid-surface

The fundamental forms I and II define the mid-surface ϕ used in the optimization of Section 3. We construct ϕ like Sperl et al. [2020] by solving for ∇ϕ ≈ RS in the least-squares sense (with ∇ϕ = RS exactly for single curvature). To do so, we compute a rotation R describing curvature and an in-plane deformation matrix S from the two input fundamental forms I and II. The 3 × 2 matrix S is computed from the principal square root of I as

\[
S = \sqrt{T} \cdot 0.
\]

To compute the rotation, we first compute normal derivatives

\[
\nabla n = \begin{bmatrix} n_{1,1} & n_{1,2} \\ n_{2,1} & n_{2,2} \end{bmatrix} = -\left(\sqrt{T}\right)^T II.
\]

Then, with

\[
a = \frac{1}{2} \sum_{a=1}^{3} n_{a,1} X_{a}, \quad b = \frac{1}{2} \sum_{a=1}^{3} n_{a,2} X_{a}, \quad r = \sqrt{a^2 + b^2},
\]

we compute \(R(X_1, X_2)\) as

\[
R = \begin{bmatrix} 1 - \frac{1}{2} a^2 \sin(r/2)^2 & -\frac{1}{2} ab \sin(r/2)^2 \sin(r/2) \\ \frac{1}{2} ab \sin(r/2)^2 & 1 - b^2 \sin(r/2)^2 \cos(r) \end{bmatrix}.
\]

Finally, we solve for \(ϕ\) with the Poisson equation \(\nabla^2 ϕ = \nabla \cdot RS\) and natural boundary conditions discretized on a grid. For further details, we refer to the paper and supplementary document of [Sperl et al. 2020]. The sliding constraint (3) is thus \(\hat{u} \cdot (R N) = 0\).

B Pullback into material space

Here, we detail the Newton iteration to transform optimized yarn geometry from Section 3 back into material space. The goal is to find \(X\) s.t. \(x = \overline{X}(X)\), which we do using Newton’s method on \(f(\overline{X}) = x - \overline{X}(X)\), for which we need the gradient \(∇f\).

The mapping \(\overline{X}\) is defined as

\[
\overline{X}(X) = ϕ(\overline{X}_1, \overline{X}_2) + X_3 n(\overline{X}_1, \overline{X}_2),
\]

where \(ϕ\) is the deformed mid-surface and \(n\) its normal. Its gradient is

\[
∇\overline{X}(X) = \begin{bmatrix} ∇ϕ(\overline{X}_1, \overline{X}_2) + X_3 ∇n(\overline{X}_1, \overline{X}_2), & n(\overline{X}_1, \overline{X}_2) \end{bmatrix}.
\]

By definition (see [Sperl et al. 2020]) we have

\[
∇ϕ = RS,
\]

and we compute \(∇n\) with (11). The Newton iterations are

\[
\overline{X}_{i+1} = \overline{X}_i - ∇f(\overline{X}_i)^{-1} f(\overline{X}_i)
\]

starting from the rest configuration \(\overline{X}_0 = X\).

In our experiments, the iterations converge to a fraction of the yarn radius within three iterations and the cost is negligible compared to the elasticostatic optimization. For pure in-plane deformations, \(Π = 0\), and thus \(R = Id\) and \(∇f = -\sqrt{T} \begin{bmatrix} 0 \\ 0 \end{bmatrix}\), simplifying the pullback into the constant expression

\[
\hat{X} = X + \begin{bmatrix} √T^{-1} & 0 \\ 0 & 1 \end{bmatrix} \hat{u}.
\]
the $X_1$ axis and the eigenvector corresponding to $\lambda_1$, we distribute the bending contributions to get

$$
\Delta Q(s, \mathbf{II}) = c^2 \Delta Q_{X_1}(s, \lambda_1) + (1 - c^2) \Delta Q_{X_1}(s, \lambda_2) + c^2 \Delta Q_{X_2}(s, \lambda_2) + (1 - c^2) \Delta Q_{X_2}(s, \lambda_1)
$$

(19)

Since both datasets include the displacements caused by pure in-plane deformations $\lambda_1 = \lambda_2 = 0$, we have to subtract these displacements once to avoid double counting. This model exactly reproduces the data samples for bending aligned with either $X_1$ or $X_2$. However, it requires five 4D-texture look-ups and is thus much more expensive than the linearized model requiring only a single 3D-texture look-up. For the speed comparison mentioned in Section 5, we used two sets of $9^4$ samples for the 4D model and $9^3$ samples for the linearized model. The linearized model was still 7× faster even when increasing its sample density to $15^3$.

C.1 Linearized Bending

For a thin shell represented by a midsurface $\varphi$ with normal $\mathbf{n}$, its full domain is

$$
\mathbf{x} = \varphi + h \mathbf{n},
$$

(20)

with the normal coordinate $h \in [-H/2, H/2]$ for shell thickness $H$. Its right Cauchy-Green deformation tensor is

$$
\nabla \mathbf{x}^\top \nabla \mathbf{x} = \nabla \varphi^\top \nabla \varphi + h (\nabla \varphi^\top \nabla \mathbf{n} + \nabla \mathbf{n}^\top \nabla \varphi) + O(h^2),
$$

(21)

which can be interpreted as a first fundamental form $I(h)$ depending quadratically on $h$. The quadratic term is commonly neglected (e.g. [Kiendl et al. 2015]). Additionally, we see that $\nabla \varphi^\top \nabla \varphi = I$ and similarly $\nabla \varphi^\top \nabla \mathbf{n} = \nabla \mathbf{n}^\top \nabla \varphi = -\mathbf{II}$ are the fundamental forms of $\varphi$. As a result, we get the linearized expression

$$
I(h) \approx I - 2h \mathbf{II}.
$$

(22)

Then, with precomputed data $\Delta Q_s(s)$ for in-plane deformations $s$, the linearized bending model is

$$
\Delta Q(I, \mathbf{II}) = \Delta Q_s(s(I - 2h \mathbf{II})).
$$

(23)