MODELING, RENDERING, AND SIMULATING KNITS

by

Kui Wu

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Department of Computing

The University of Utah

August 2019

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Kui Wu**
has been approved by the following supervisory committee members:

Cem Yuksel ,	Chair(s)	**Apr 2 2019** Date Approved
Charles Hansen ,	Member	**Apr 2 2019** Date Approved
Ladislav Kavan ,	Member	**Apr 2 2019** Date Approved
Daniele Pazzono ,	Member	**Apr 2 2019** Date Approved
Doug James ,	Member	**Apr 2 2019** Date Approved

by **Ross Whitaker** , Chair/Dean of
the Department/College/School of **Computing**
and by **David B. Kieda** , Dean of The Graduate School.

ABSTRACT

Knitting is one of the most popular techniques to manufacture clothing. In this dissertation we present a framework for modeling, fabricating, and simulating knit structures. We first introduce a fully automatic pipeline to convert arbitrary 3D shapes into a stitch mesh, which provides a mesh-based representation of the yarn-level knit structure. We also extend the concept of stitch mesh modeling with a list of novel shaping techniques to allow modeling more complex structures. We then introduce a scheduling algorithm to convert the structure into step-by-step hand knitting instructions for knitters. We further embed low-level machine knitting operations to each face and allow users to edit the stitch mesh in a way that preserves the machine knittability. Moreover, we present a real-time rendering method to display knitwear containing more than a hundred million individual fiber curves at real-time frame rates with shadows and ambient occlusion on the GPU. We also present a GPU parallelization method to address the computational challenges of using the Material Point Method to simulate knits.

For my wife, Zhen Li.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Knitting is one of the most popular techniques to manufacture clothing such as t-shirts, socks, and various forms of winter garments. As a popular choice in the textile industry, knitting offers extra stretchability by its special yarn-level construction. Thus, knitting is especially useful for form-fitting and comfortable clothing. Knit structures are constructed by continuously looping yarn through existing loops and shaping is added as the structure is created by using shaping techniques like stitch increases, decreases, and short rows. This creates a stronger structure than other fabrication methods, which can include seams, welds, etc., that can thereby introduce points of weakness and stress. Seamless fabrics produce minimal waste, which is important when the material is expensive. The 3D shapes of knit structures are defined by the types of stitches used for constructing them (i.e., the knitting pattern). However, the yarn-level structure also leads to various challenges in digitalizing the knitting design pipeline, including modeling, rendering, and simulation, for computer graphics purpose and fabrications.

1.1 Motivation and Contributions

Knitting is actually a construction technique that starts with a long thread (referred to as *yarn* below) and produces a surface using interlocked stitches. The final 3D shapes are produced by using basic variations on how stitches are formed. That particular construction mechanism brings up both geometric and topological requirements for knit modeling. First of all, the size of each stitch is controlled by both needle size and yarn weight (i.e., thickness), but all stitches in the same garment usually have approximately the same size. Second, on a knit surface, consecutive stitches are placed side-by-side along the *course* knitting direction to form rows, and consecutive rows are connected along the *wale* knitting direction. These two primary directions are roughly perpendicular to each other over the entire surface at most places.

More importantly, the knitting procedure can be decomposed into a small set of low-level knitting instructions performed in a particular order. Industrial knitting machines with arrays of needles and multiple yarn feeders can rapidly produce knit products, but also introduce additional constraints on the knitting pattern that they can fabricate, as compared to hand knitting, which is more flexible. Knowing the set of knitting instructions does not guarantee that the designed knit model can be fabricated by a given knitting machine, as the knitting machine may not be able to perform these instructions in the required order without breaking the yarn. The knitting instructions must be converted to low-level machine instructions, scheduling the fabrication of each stitch and making sure that the fabrication process does not exceed the limitations of the knitting machine used (such as the number of needles available) and the physical constraints of the material (such as extreme stretching that could break the yarn). Thus, controlling those machines for fabricating a desired 3D form is a notoriously difficult problem.

Knits structure is made by intertwining yarn pieces with each other to form a stable structure. Each yarn consists of multiple plies, and each ply has tens to hundreds of individual fibers. Composite materials can be formed by holding different ply or fiber materials together and provide high tensile strength or extra elasticity. This complex geometric structure leads to various challenges on rendering knit models.

Regarding simulating the animation of knits, yarn-level simulation can produce realistic deformations of knit structures. Yarn-yarn contacts keep the knit structure stable but are also very expensive to detect and handle while simulating knits.

Designing knitting pattern for a given 3D surface has also been an open problem. Knitting patterns are currently designed using a high level of expertise and numerous iterations of trial and error to figure out how one could knit a particular 3D shape. That is why most knitting patterns used today are merely derivations of a limited number of well-known and well-understood shapes. In computer graphics, stitch meshes [164] provide a powerful interface for modeling knit garments. However, they still require users to manually design the topology of the given (typically low-resolution) input mesh. This requires the user to know exactly how to knit the desired shape and prepare an input mesh accordingly. Therefore, it is extremely difficult and time consuming to design knitted models for complex and uncommon shapes, each of which would require

numerous design iterations by a knitting expert. We introduce the first fully automatic pipeline to convert arbitrary 3D shapes into knit models. Our pipeline is based on a global parametrization remeshing pipeline to produce an isotropic quad-dominant mesh aligned with a 2-RoSy field. The knitting directions over the surface are determined using a set of custom topological operations and a two-step global optimization that minimizes the number of irregularities. The resulting mesh is converted into a valid stitch mesh that represents the knit model. The yarn curves are generated from the stitch mesh, and the final yarn geometry is computed using a yarn-level relaxation process. Thus, we produce topologically valid models that can be used with a yarn-level simulation.

Existing computer-aided design interfaces for knittable structures have been severely limited. Current tools for the fashion industry include only a few templates for common items. Details can be customized, but experimenting with shape requires extensive expertise, large amounts of time, and extra materials to iterate on pattern designs. Other tools developed for designing knit structures use basic primitives like tubes or flat patches. Stitch meshes [164] do not produce actually knittable structures. Therefore, they cannot be used for fabrication purposes. On the other hand, stitch meshes offer a convenient interface for designing 3D knit structures by hiding the complexity of the yarn-level model and providing a 3D workspace that is powerful enough to represent arbitrary shapes. We introduce *knittable stitch meshes* for modeling complex 3D knit structures that can be fabricated via hand knitting, by extending the concept of stitch mesh modeling. Knittable stitch meshes ensure that the final model can be knitted. Moreover, they include novel representations for handling important shaping techniques that allow modeling more complex knit structures than prior methods. In particular, we introduce *shift-paths* that connect the yarn for neighboring rows, general solutions for properly connecting pieces of knit fabric with mismatched knitting directions without introducing seams, and a new structure for representing *short rows*, a shaping technique for knitting that is crucial for creating various 3D forms, within the stitch mesh modeling framework. Furthermore, we present a scheduling algorithm for providing step-by-step hand knitting instructions to a knitter, so that anyone who knows how to knit can reproduce the complex models that can be designed using our approach.

Computer-controlled knitting machines are powerful tools for computer-aided fabrication, and they are widely used in the garment and accessory industries. When properly programmed, they can turn yarns into soft 3D surfaces in a wide range of shapes, textures, and colors. Knitting machines create these objects by using a small vocabulary of operations that manipulate loops on their *needle beds*, two long rows of loop storage locations. Once programmed, objects can be manufactured quickly and with minimal wasted yarn. Yet, knitting machine programming is notoriously challenging because shape, structure, texture, and color effects must all be created concurrently using a small set of low-level operations. In addition, these operations must be *scheduled* to limited needle bed locations on a knitting machine, and they are conventionally selected with limited visual and structural feedback. Thus, designing patterns for industrial knitting machines requires extensive training. We present the first general visual programming interface for creating 3D objects with complex surface finishes on industrial knitting machines. At the core of our interface is a new, augmented, version of the stitch mesh data structure. The augmented stitch mesh stores low-level knitting operations per-face and encodes the dependencies between faces using directed edge labels. Our system can generate knittable augmented stitch meshes from 3D models, allows users to edit these meshes in a way that preserves their knittability, and can schedule the execution order and location of each face for production on a knitting machine. Our system is general, in that its knittability-preserving editing operations are sufficient to transform between any two machine-knittable stitch patterns with the same orientation on the same surface.

Rendering fiber-level knit models not only has enormous high memory requirements but also has extreme computational cost. 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 realistic cloth rendering. 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. We present a real-time fiber-level cloth rendering method for current GPUs. Our method procedu-

rally 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 self-shadow computation method that allows lighting with self-shadows using relatively low-resolution shadow maps. We also provide a simple distance-based ambient occlusion approximation as well as an ambient illumination precomputation approach, both of which account for fiber-level self-occlusion of yarn. Finally, we discuss how to use a physical-based shading model with our fiber-level cloth rendering method and how to handle cloth animations with temporal coherency.

Yarn-level simulation can produce realistic deformations of knit structures. Yarn-yarn contacts keep the knit structure stable but also are very expensive to detect and handle while simulating the knits. Traditional methods either handle the collision explicitly [76, 77, 90] or assume persistent contact between yarn pieces throughout the simulation [27, 29]. Recently, by assuming yarns as codimensional objects, a hybrid Lagrangian/Eulerian material point method (MPM) method [72] is used to simulate knits. Yarn-yarn contacts and collisions are simply handled through the constitutive modeling in the grid only. While considerably faster than yarn-level cloth simulations, MPM has been computationally expensive and could not be efficiently implemented on high-performance modern multiprocessors, like GPUs, with parallelization. We introduce methods for addressing the computational challenges of MPM and extend the capabilities of general simulation systems based on MPM, particularly concentrating on GPU optimization. In addition to our open-source high-performance framework, we also conduct performance analyses and benchmark experiments to compare against alternative design choices, which may superficially appear to be reasonable, but can suffer from suboptimal performance in practice.

In this dissertation, my key contributions are listed as follows:

- *The first fully automatic pipeline to convert arbitrary 3D shapes into knit models.* After a global parametrization remeshing to produce an isotropic quad-dominant mesh, the knitting directions over the surface are determined using a set of custom topological operations and a two-step global optimization that minimizes the number of

irregularities.

- *A framework for modeling complex 3D knit structures that can be fabricated via hand knitting.* To extend the concept of stitch mesh modeling, novel shaping representations are introduced to allow modeling more complex knit structures than prior methods. A scheduling algorithm is proposed to for provide step-by-step hand knitting instructions to a knitter for knitting the complex 3D models.

- *The first general visual programming interface for designing complex knit objects,which can be fabricated on industrial knitting machines.* The augmented version of the stitch mesh data structure stores low-level knitting operations per-face and encodes the dependencies between faces using directed edge labels. Our system allows users to edit stitch meshes, while preserving their knittability, and can schedule the execution order and location of each face for production on the industrial knitting machine.

- *The first real-time fiber-level cloth rendering method for current GPUs.* Fiber-level geometric details are generated procedurally on-the-fly. A 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.

- *A GPU parallelization method to address the computational challenges of the Material Point Method.* In addition to the open-source high-performance framework, the performance analyses and benchmark experiments are conducted to compare against alternative design choices, which may superficially appear to be reasonable, but can suffer from suboptimal performance in practice. It demonstrate that more than an order of magnitude performance improvement can be achieved with our GPU solvers. Practical high-resolution knits examples with up to ten million particles run in only a few seconds per frame.

1.2 Dissertation Statement

This dissertation explores approaches to address various challenges to digitalize the knitting design pipeline, including modeling, rendering, and simulation, for computer graphics purpose and fabrications. Specifically, this work introduces an automatic way to convert arbitrary shapes into knit structures. This work introduces interactive modeling frameworks for design of 3D knit shapes for both hand knitting and industrial knitting machine. This work also introduces a real-time rendering technique to display knitwear with more than a hundred million individual fiber curves and a parallelization method to simulate knitwear with up to ten million particles in only a few seconds per frame for current GPUs.

1.3 Dissertation Organization

The remaining chapters are organized as follows. In Chapter 2, we briefly overview related previous works. In Chapter 3, we present an automatic approach to convert arbitrary 3D shape into knit structure. In Chapter 4, we present a framework to convert stitch mesh structure into knittable structure as well as a scheduling method to knit it by hand. In Chapter 5, we present an interactive designing framework for users to edit the stitch mesh while preserving the machine knittability. Each stitch mesh face is embedded with customized machine codes to provide extra flexibility for users to create patterns, cables, and colorworks and fabricate the result using industrial knitting machines. In Chapter 6, we present a real-time rendering approach to render the knit structure with fiber-levels on the GPU. In Chapter 7, we present a GPU-accelerated material point method to simulate the knit structure efficiently. Finally, we conclude in the last chapter.

CHAPTER 2

BACKGROUND

In this chapter, we provide a brief overview of knits modeling methods as well as previous works of yarn-level rendering and simulation.

2.1 Modeling

We briefly overview of modeling fabrics and knit structures as well as the stitch mesh structure [164] that we use in our pipeline for representing the final yarn-level model in Chapter 3. We also provide an overview of prior works on quad-dominant remeshing.

2.1.1 Cloth Modeling

Much of the work in computer graphics involving cloth has been aimed towards simulating woven fabrics using sheet-based representations [6, 16, 19, 49, 52, 147]. The modeling approaches for sheet-based cloth mainly concentrate on fitting garment models on virtual characters [9, 21, 34, 55, 120, 141, 143, 148]. 3D modeling approaches have also been used in virtual garment prototyping [97, 146]. To produce more complex cloth models, Mori and Igarashi [104] proposed a method for designing 3D plush toys by sketching 2D patterns. More recently, cloth capturing methods using multiview systems [15], single images [31, 172], 3D scans [22], and motion sequences [114] have been shown to successfully produce virtual garment models.

Akleman et al. [3] introduced a method for converting arbitrary quad-meshes into plain-woven structures using graph rotation systems. While this approach is similar in spirit to our method, the knit structures we produce have entirely different constructions and requirements than plain-woven structures. 3D printing is often paired with computational fabrication techniques. Methods to design and fabricate various structures, such as flexible rod meshes [113], ornamental curve networks [165], and wireframe meshes [160] have been explored in prior work.

2.1.2 Knit Modeling

Knit structures are constructed by pulling yarn loops through other yarn loops to form stitches. Shaping techniques, such as *increases* that pull multiple yarn loops through one yarn loop or *decreases* that pull a yarn loop through multiple yarn loops, allow forming complex 3D shapes without introducing seams. Due to the properties of this construction, knit fabrics have low resistance to stretching even if the yarn itself is not stretchable.

Modeling the yarn-level structure needed for knits, however, is a more complex procedure. Commercial knitting design software provides templates of standard designs with limited scope for editing [126, 131]. More complex or nonstandard patterns must be hand-designed at the stitch level, though there exist guidebooks of advanced techniques that can assist with this process [144]. Others focus only on flat panels of texture and color, only altering the appearance but not the shape of a pattern [127].

2.1.2.1 Machine Knitting

Meisner et al. [103] proposed one of the earliest graphics based approaches for visualization and design of machine knitted structures. Recently, researchers have expanded this scope by offering general tube and sheet primitives in the context of a knitting compiler [101]. Since their approach still requires a designer to place and configure these primitives by hand, Narayanan et al. [106] recently proposed an automatic approach knit a wide variety of 3D surfaces on machine. However, their method focuses on matching the topology and geometry of the input model and does not offer any options to edit these patterns for texture or colorwork. Popescu et al. described a similar system that works on topologically-disc-shaped patches [115], which can later be connected manually. In addition to making feasible knitting patterns, researchers are also beginning to examine how to create efficient patterns [92], though this work is limited to flat knitting patterns.

The textile community has also attempted to model knit structures using spline curves [24, 25, 35, 48, 119] or by simplifying a pattern using representative cells holding swatches of knit structure [85–87]. Generating yarn geometry can be similar to texture synthesis [59, 88], and some methods have attempted to use these algorithms for adding fine-level repeating geometry to high-level shapes [20, 89, 173].

2.1.2.2 Hand Knitting

Relative to machine knitting, hand knitting is very flexible. Human knitters are dextrous and able to form complex stitches using loops from anywhere in the existing fabricated item. Thus, designing for human knitters is a substantially different problem than designing for machine knitting, and while approaches for the latter can be used for the former, the reverse is not true.

It is known that a 2D surface of any topology can be hand knit [8]. Igarashi et al. [67, 68] presented a design assistant that semiautomatically creates a knitting pattern from a 3D model by covering the surface with a winding strip and finding areas where increases or decreases are needed. Yuksel et al. [164] introduced stitch meshes, a data structure for modeling knit structures for visualization and simulation. Recently, Wu et al. extended stitch meshes for hand knitting by introducing a list of hand-knittable mismatch faces [156] and introducing an automatic pipeline to convert arbitrary shape into labeled quad-dominant mesh as input [155].

2.1.3 Stitch Meshes

The stitch mesh structure [164] is an abstraction of the yarn-level geometry that provides a powerful interface for modeling knit structures. Each stitch-mesh face corresponds to a stitch of the knit structure, shown in Figure 2.1a. These faces are placed side-by-side along the *course* knitting direction, forming *rows* (Figure 2.1b). Consecutive rows are connected along the *wale* knitting direction (Figure 2.1c). Each stitch-mesh face has two *wale edges* that are aligned with the wale knitting direction. Most stitch mesh faces are quads with two wale edges and two *course edges* that are aligned with the course knitting direction and separate the wale edges. The yarn used for knitting the stitch represented by a face enters the face from one of the wale edges, forms the stitch, and then exists the face from the other wale edge (green yarn curves in Figure 2.1a). The top part of a yarn loop formed by the stitch in the previous row enters and exits the face from the *bottom course edge* (green yarn curves in Figure 2.1a). Similarly, the loop forming the stitch of a face exits and enters the face from the *top course edge*, connecting it to the next row.

Stitch mesh faces can have more than four edges. Faces with multiple top course edges are called *increases*, as they increase the number of stitches on the next row. Similarly,

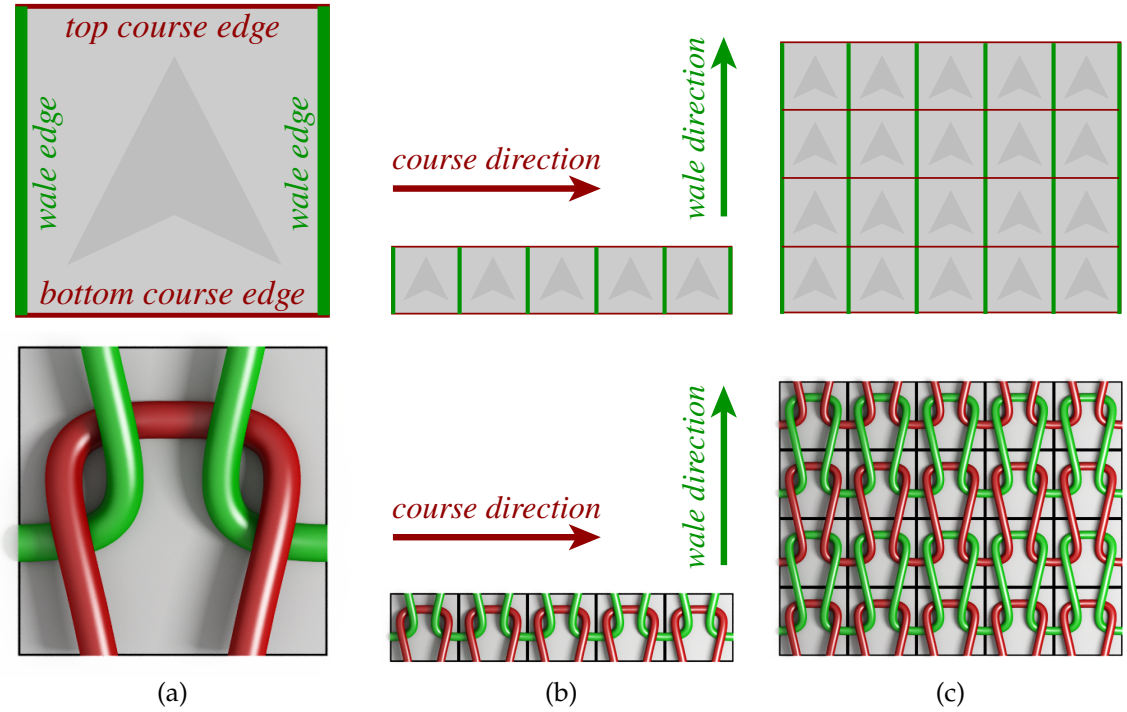


Fig. 2.1: Stitch mesh representation: (a) a typical stitch mesh face and the corresponding yarn-level model, (b) stitch mesh faces on a row, and (c) multiple rows of stitch mesh faces representing interlocked stitches on consecutive rows.

faces with multiple bottom course edges are called *decreases*. The stitch mesh structure also permits faces with no bottom course edges or no top course edges, but such faces must be placed with caution, since they do not form stable stitches and placing them side-by-side may cause the yarn-level model to unravel. That is why we entirely avoid such faces in our framework. Yet, this limitation has no practical consequence, since the yarn-level models including such faces can be represented differently. For example, a triangular face next to a quad face can also be represented by a face with five edges.

2.1.4 Structured Meshing

The generation of quadrilateral or quadrilateral-dominant meshes has received a lot of attention in the last two decades. We restrict our review to the most recent works in global and local parametrization, and we refer an interested reader to [11] for a complete survey.

Global parametrization methods [4, 54, 79, 98] flatten the surface after cutting it into a topological disk, generate a regular lattice on the plane, and then lift it back to the original surface, producing a structured mesh. To control edge alignment, it is possible

to solve an optimization that strive to align the parametrization gradients to a guiding field [12, 40, 75, 108]. Designing the guiding field is a difficult problem on its own [30, 61, 74, 83, 84, 89, 109, 110, 118], and we refer an interested reader to the recent state-of-the-art report of Vaxman et al. [145]. These methods fix the singularities of the quadrilateral mesh during the orientation field design, and they thus inevitably introduce distortion in the parametrization (since the orientation field is not integrable [36]), which results in quads of varying size. While this is not problematic for most remeshing applications, it is not acceptable for stitch meshes, since the size of stitch has to be uniform.

Local parametrization methods [45, 71, 117, 128] provide a radically different approach, where a perfectly isometric parametrization is computed locally for every vertex/triangle of a surface. Local inconsistencies between neighboring parameterizations, which are unavoidable since exact isometry is enforced, lead to the introduction of nonquad elements or T-junctions, producing hybrid meshes composed of a majority of isotropic quadrilateral elements. These meshes are ideal for our purposes, since they contain minimal distortions, and they produce approximately uniform face sizes.

Our remeshing algorithm is heavily based on the Robust Instant Meshing (RIM) quad-dominant meshing pipeline of Gao et al. [45]. In RIM, the orientation field is encoded as a unit vector attached to every vertex, which is unique up to an integer rotation. The position field encodes a local isometric parametrization whose gradient is aligned with the orientation field, i.e., it encodes a regular grid in the tangent space. It is called position field, since the only available degree of freedom is the origin of the grid (up to an integer translation), which is represented as a 3D point. The position field can be visualized as a new set of 3D coordinates for the vertices of the input triangle mesh, that are mapping each vertex to the position of the closest vertex of the output quadrilateral mesh. RIM extracts the final quad mesh by collapsing the edges of the input mesh, using the position field to identify which edges should be preserved as final edges of the quad-dominant mesh, which edges are diagonals, and which edges should be collapsed.

2.2 Cloth Rendering

Fabric appearance has been an active research area in computer graphics. A specialized Bidirectional Reflectance Distribution Function (BRDF) combined with texture mapping is

used to produce realistic results for sheet-based cloth rendering. However, a fiber-level cloth model for a full-size garment would require terabytes of memory for storing all fiber curves. To avoid using a considerable amount of memory for storage, researchers typically convert the cloth model with fiber-level geometric details into volumetric data that approximates it. Yet, this volumetric data can also require extensive amount of storage, the size of which depends on the level of details represented in the volume data. Recently, researchers started combining the fiber-level geometric complexity with the optical complexity of light interaction for cloth rendering.

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 [5, 122, 151]. For woven fabrics procedural patterns [2, 78] were used for approximating fabric appearance with limited yarn-level detail, and the far-field appearance was improved using mip-maps [163]. Fitting measured data to a detailed procedural model was used for capturing the anisotropic specular reflections for woven fabrics [69]. Recently, Schröder et al. [124] 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 [2, 78, 163], they can only handle woven cloth and cannot reproduce fiber-level details.

Rendering yarn-level cloth 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 makes it more favorable to use volumetric approximations that convert the entire cloth model into volume data [53, 161]. The volume data are 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. [94] 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. [70] proposed a framework for volumetric modeling and rendering of materials with anisotropic microstructure. Micro CT imaging was used for repeated fabric patterns for volumetric fabric modeling [169] and explicitly modeling the interaction of light with microgeometry [80]. For reducing the extensive storage requirements of volumetric fabric

rendering, Zhao et al. [171] used the SGGX microflake distribution [60] 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.

Recently, Zhao et al. [170] proposed a procedural fiber-level representation of the yarn structure, the parameters of which are fitted from the real data by CT scan. Luan at el. [95] introduced a offline rendering method with this procedural representation. The ray-fiber intersection are performed only when the ray enters the yarn volume. No regular fibers are stored since they can be generated on-the-fly using the procedural model. Their method can generate exactly the same results as using fully explicit fiber geometries.

2.3 Yarn-Level Simulation

Yarn-level simulation can produce realistic deformations of knit structures. Yarn-yarn contacts keep the knit structure stable, but they are very expensive to detect and handle.

2.3.1 Explicit Yarn-Yarn Contact Methods.

Kaldor et al. [76] first introduced a yarn-level simulation method for knit models, which can produce impressive animations with complex dynamic motion. The yarns are explicitly modeled as inextensible but flexible spline curves, and yarn-yarn contacts are handled explicitly as well. Later, Kaldor et al. [77] introduced an adaptive contact linearization force model, which is used to avoid most of the penalty-based contact force computation at each timestep. Recently, Leaf et al. [90] introduced an efficient GPU-based method for designing periodic yarn-level cloth patterns based on the same force model.

2.3.2 Reduced Contact Methods.

Sueda et at. [134] introduced a method that combines Lagrangian and Eulerian approaches to handle the constraints independently from the initial discretization of the yarn. Cirio et al. [27] extended this method to simulate woven structure with a reduced-order model for yarn-yarn contacts handling. They later extended this method to knit cloth simulation [28, 29]. Their methods handle contacts between interlaced/interlocked yarns implicitly by discretizing yarn crossings as sliding contacts and assuming persistent contact between yarn pieces throughout the simulation.

2.3.3 Material Point Method (MPM).

The Material Point Method was introduced by Sulsky et al. [135] as the generalization of the hybrid Fluid Implicit Particle (FLIP) method [14, 18, 174] to solid mechanics. It has been recognized as a promising discretization choice for animating various solid materials including snow [132], foam [116, 162], sand [33, 82], cloth [56, 72], fracture [154], cutting [64], and solid fluid mixture [42, 133, 136].

For GPU-based simulation methods, many researchers divided the simulation domain in order to parallelize computation on the GPU [26, 63, 93]. Wang [150] performed a GPU optimization on sewing pattern adjustment system for cloth. Others explored solutions to the computationally heavy task of self-collision detection, using GPUs [50, 51] with improved spatiotemporal coherence and spatial hashing [137–139, 152, 153]. In particular, the spatial hashing table [153] has been proposed as both an acceleration structure and a substitution to the hierarchy of uniform grids for collision query in order to lower the memory consumption. Furthermore, the histogram sort (alternative to radix sort) has been proven to be capable to significantly reduce the overhead of sorting if the number of bins is adequately smaller than the total count of elements [152]. These practices provided great foundations for our MPM pipeline as they fit well with the sparse grid structure.

From the Eulerian view, the simulation domain is represented by a discretized grid. Museth [105] developed OpenVDB, which is a tree with a high branching factor that yields a large uniform grid at leaf nodes. This adaptive data structure has been shown to be very efficient and broadly used. Inspired by that, Hoetzlein et al. [62] proposed GVDB Voxels, a GPU sparse grid structure. The voxel data are represented in dense n^3 bricks allocated from a pool of subvolumes in a *voxel atlas*. The atlas is implemented as a 3D hardware texture to enable trilinear interpolation and GPU texture cache. Recently, Wu et al. [157] extended it with dynamic topology update and GPU optimized matrix-free conjugate gradient solver for fluid simulations with tens of millions of particles. Although the use of texture to store volumetric data can benefit performance for general purpose usage, such as hardware trilinear interpolation and fast data accessing, it prevents GVDB from using scattering rather than gathering because atomic operations on textures are not allowed to be used under current GPU hardware.

SPGrid [125] provided an alternative sparse data structure; it has been adopted in large-

scale fluid simulations [1, 93, 125] and in the MPM context [42, 43, 64]. SPGrid improves the data locality by mapping from a sparse 2D/3D array to a linear memory span by following a modified Morton coding map. Furthermore, it exploits hardware functions to accelerate the translation between geometric indices and the 64-bit memory offsets. The neighborhood accesses can be achieved in $\mathcal{O}(1)$, rather than $\mathcal{O}(\log n)$ for traditional tree-based sparse storage schemes (n is the number of the leaf nodes).

Recently, Jiang et al. [72] used the Material Point Method to represent yarn curves as volumetric elastoplastic continua. The particles are used to represent discrete samples of the continuous yarn curve. Thus, yarn-yarn contacts and collisions can be handled through the constitutive modeling in the grid only.

CHAPTER 3

STITCH MESHING

Knitted garments are common in our daily lives, going from socks and T-shirts to winter clothing and accessories, and are thus ubiquitous in movies and games. There are two good reasons for favoring knitting over its alternatives: knitted fabrics easily stretch and the shaping techniques used in knitting allow producing complex 3D surfaces without any seams.

However, designing knitting pattern for a given 3D surface is still an open problem. Knitting patterns are currently designed using a high level of expertise and numerous iterations of trial and error to figure out how one could knit a particular 3D shape. That is why most knitting patterns used today are merely derivations of a limited number of well-known and well-understood shapes. In computer graphics, stitch meshes [164] provide a powerful interface for modeling knit garments. However, they still require users to manually design the topology of the given (typically low-resolution) input mesh. This requires the user to know exactly how to knit the desired shape and prepare an input mesh accordingly. Therefore, it is extremely difficult and time consuming to design knitted models for complex and uncommon shapes, like the ones shown in Figure 3.1, each of which would require numerous design iterations by a knitting expert.

In this chapter, we introduce the first automatic pipeline to deal with this challenging problem: our method takes a 3D surface as input and the desired stitch size, and automatically produces a topologically-valid yarn-level knit model. The resulting model can be either directly used in computer graphics applications with yarn-level simulation or realized for 3D printing. The challenge we are tackling is the design of a dense network of closed, intertwined yarn curves that are not self-intersecting and hold together thanks to the interlocked curves formed by knitted stitches. The problem is inherently global, i.e., a change in one stitch can affect not only its neighborhood, but the entire shape.

Fig. 3.1: Example yarn-level models generated from input 3D surfaces using our fully automatic pipeline.

We use stitch meshes [164] as an intermediate step in our pipeline. We begin by converting the input 3D shape into an isotropic quad-dominant mesh with approximately uniform face sizes using a two-fold rotational symmetry (2-RoSy) orientation field. This process provides a good starting point that minimizes the distortions of the final knit structure. Then, we automatically determine the knitting directions over the entire model surface using a two-step global optimization process along with custom topological operations.

Finally, we subdivide the resulting mesh to generate a valid stitch mesh. After the stitch mesh is ready, we can use it to create the final yarn curves. Since the yarn-curves are topologically valid, we can use them with yarn-level cloth simulations. Also, fabricating them using 3D printing produces interlocked curves that form flexible surfaces.

We show the effectiveness of our approach by automatically producing yarn-level knit models for complex 3D shapes (Figure 3.1) and its robustness by generating stitch meshes from a large number of complex 3D models. We also present yarn-level models automatically generated using our pipeline and fabricated via 3D printing.

3.1 Overview

Our pipeline begins with an input model. Unlike the stitch mesh modeling framework, however, we do not rely on the topology of this input model. Instead we begin with remeshing the model to produce a quad-dominant mesh. Then, we perform a series of optimizations and topological operations to generate the knitting directions over the mesh. Finally, we perform a subdivision operation that produces a valid stitch mesh, and the final

yarn-level model can be easily created from this stitch mesh. Figure 3.2 demonstrates the individual steps of our pipeline that are listed below:

- **Remeshing:** Starting with a given input model, we generate an isotropic quad-dominant mesh that only contains triangles and quads (Section 3.2).

- **Labeling:** We formulate a Mixed-Integer Programming (MIP) problem and perform custom topological operations to label each edge as a wale or course edge (Section 3.3).

- **Knitting Direction Assignment:** We determine the knitting (wale) direction based on the edge labels by solving another optimization problem (Section 3.4).

- **Stitch Mesh Generation:** The stitch mesh is formed via a subdivision operation that considers the knitting directions and edge labels (Section 3.5).

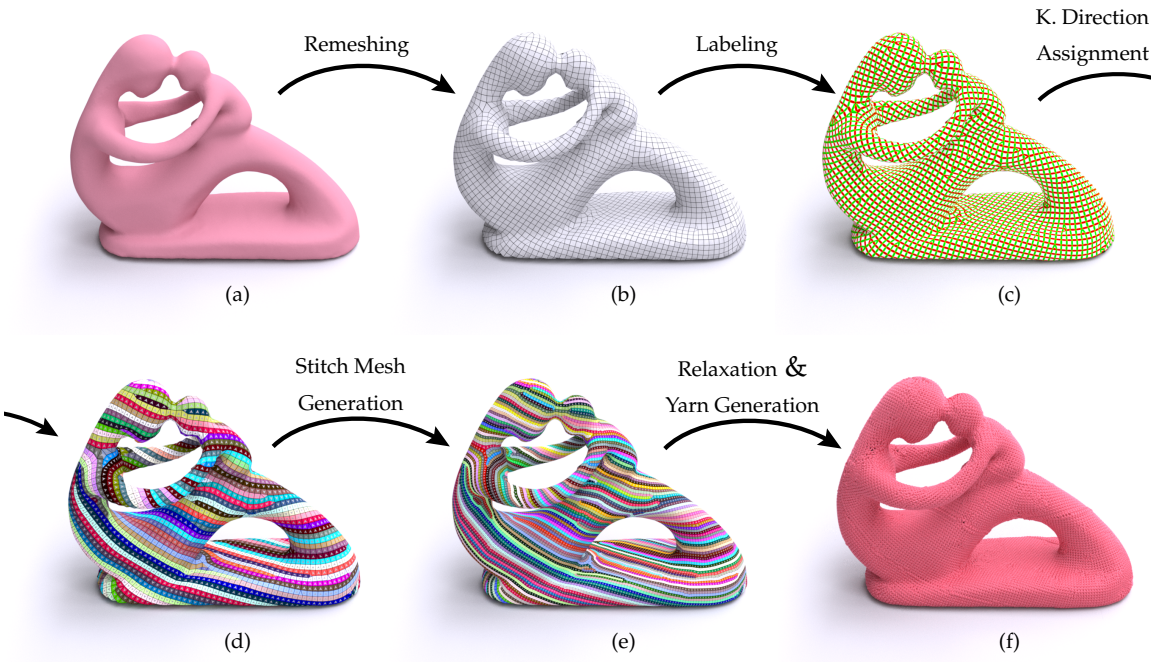


Fig. 3.2: The overview of our pipeline: (a) an arbitrary input 3D model is converted into, (b) an isotropic quad-dominant mesh with only quads and triangles via remeshing. Then, (c) the edges of the mesh are labeled, and (d) knitting directions over the surface are determined (arrows showing the wale knitting direction on each face). Finally, (e) a stitch mesh is generated, and (f) the final yarn-level model is produced from the stitch mesh via relaxation and yarn generation operations.

- **Relaxation and Yarn Generation:** We perform mesh-based relaxation and then generate the yarn-curves from the stitch mesh (Section 3.6). The final yarn-level model is produced via yarn-level relaxation.

This process allows us to produce a yarn-level knit model starting with an arbitrary 3D shape. No user interaction is required at any step. We describe each one of these steps in detail in the following sections.

3.2 Remeshing

The requirements for producing valid stitch meshes are different from traditional FEM applications. Stitch meshes are quad-dominant meshes that must satisfy two requirements:

1. **Topology Requirement**: It must be possible to separate the faces into groups of rows, such that the knitting directions are aligned along edges (Figure 2.1c),

2. **Geometry Requirement**: All faces must have approximately the same size.

The first requirement ensures the existence of a valid set of knitting patterns for the faces, while the second models the physical constraint that the stitch size is constant, and thus the stitch mesh faces need to have a homogeneous size.

Stitch meshes contain two primary directions (course and wale) that are roughly perpendicular to each other over the entire mesh. Therefore, a 2-RoSy field, which produces a 2-colorable mesh [91], provides a suitable topological construction for generating stitch meshes, where one of the primary directions is later aligned with the field. A more typical 4-RoSy field, on the other hand, leads to directional misalignments that require additional topological operations to resolve them. However, 2-RoSy fields are rarely used in other applications, since their singularities induce large geometric distortions: a low-order 2-RoSy field singularity can be approximated by 2 quads, which necessarily have flat angles, and introduce large distortions in the neighboring regions. Fortunately, this is not a problem for stitch meshes, since stitches near singularities naturally deform, making modern field-guided, quad-dominant pipelines ideal for our purpose.

We extend the RIM [45] quad-dominant meshing pipeline to produce meshes that satisfy (in a soft sense) the requirements of stitch meshes. Our method for orientation and

position field generation is identical to RIM, with the exception of using a 2-RoSy symmetry instead of a 4-RoSy symmetry, which is a trivial modification. RIM automatically adds T-junctions and triangles to ensure a uniform mesh element size, which satisfies our geometry requirement. The mesh extraction part is modified to restrict the nonquad elements to be either pentagons (using T-junctions) or triangles. We implemented this as a postprocessing step, which is applied after the extraction procedure of RIM, using:

1. For each triangle, we pick the edge whose opposite angle is closer to 90 degrees (excluding the edges corresponding to sharp features, identified by a negative dot product between the normal of the two incident faces) and mark it as a diagonal, encouraging the extraction algorithm to merge it with the neighboring elements, if possible.

2. We split each polygon with more than five sides by adding the edge that is most aligned with the orientation field. The splitting is done recursively until all subpolygons have less than five sides.

These operations are interleaved with the extraction algorithm in RIM, until no changes to the final mesh are made in one iteration. To complete the pipeline, each pentagon (T-junction) is split into three triangles, connecting the T-junction with the two vertices on the opposite side. As a result, at the end of our remeshing step we get a quad-dominant mesh that contains a relatively small number of triangles.

3.3 Labeling

Labeling helps us determine the knitting directions over the mesh surface. Similar to stitch mesh modeling, our goal is to label each edge as a wale edge or a course edge.

Our labeling process begins with representing each edge as two *half-edges*, each belonging to one of the two faces sharing the edge. Obviously, border edges that are used by a single face would only have a single half-edge. We label each half-edge, following certain rules that will allow us to define valid knitting directions over the surface (Section 3.3.1). This process involves solving an optimization problem that would minimize the number of edges with conflicting half-edge labels. Thus, we find a valid half-edge labeling that maximizes the number of edges with consistent half-edge labels. Then, we assign the edge

labels by resolving the half-edge labeling conflicts using simple topological modifications (Section 3.3.2). Finally, we perform post-processing operations to ensure that we have desirable final edge labels and mesh topology (Section 3.3.3).

3.3.1 Labeling Half-Edges

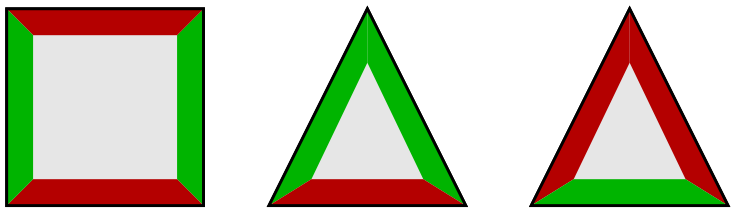
Our half-edge labeling must follow certain rules, so that the resulting labels define valid knitting directions over the surface. Thus, we can only permit a limited number of configurations for labeling.

Each quad face must have two wale edges and two course edges. Also, wale edges must be separated by course edges. Therefore, the only acceptable combination of labeling for quad faces is the one shown in Figure 3.3a.

Our final stitch meshes do not contain triangles, but we do have triangles at this intermediate step. When labeling triangles, we cannot permit all edges of a triangle to be labeled as course edges, because this would prevent building stitches within the triangle, effectively turning the triangle into a hole. Similarly, we cannot permit labeling all edges of a triangle as wale edges either, since this would also prevent building stitches within the triangle. Therefore, the only two labeling alternatives we can permit for triangles include either one wale edge or one course edge, as shown in Figure 3.3b-c.

Following these labeling rules for quads and triangles shown in Figure 3.3 as hard constraints, we label each half-edge in a way that would minimize the number of edges with inconsistent half-edge labels. We achieve this by representing the half-edge labeling problem as a mixed integer programming problem.

Let $\ell_0^{e_i}$ and $\ell_1^{e_i}$ represent the labels of the two half-edges for the edge e_i with index i.



(a) (b) (c)

Fig. 3.3: Valid half-edge configurations for quad and triangle faces. Course half-edges are colored as red, and wale half-edges are colored as green.

We assign them integer values 0 or 1 to indicate labels wale or course, respectively. These labels can also be accessed using face indices, such that $\ell_0^{f_j}$, $\ell_1^{f_j}$, $\ell_2^{f_j}$, and $\ell_3^{f_j}$ are the four half-edge labels of a quad face f_j with index j. Thus, if e_i is the first edge of f_j and f_j is the first face of e_i, we can write $\ell_0^{e_i} = \ell_0^{f_j}$. Using this notation, our optimization problem can be written as

$$
\begin{aligned}
&\textbf{minimize} && \sum_{i=0}^{n-1} (\ell_0^{e_i} - \ell_1^{e_i})^2 \\
&\textbf{subject to} && \\
&\text{for each quad face } f_j, && \ell_0^{f_j} = \ell_2^{f_j}, \quad \ell_1^{f_j} = \ell_3^{f_j}, \quad \ell_0^{f_j} \neq \ell_1^{f_j} \\
& && \ell_k^{fj} \in \{0,1\}, \quad k = 0,1,2,3 \\
&\text{and for each triangle face } f_j, && 1 \leq \ell_0^{f_j} + \ell_1^{f_j} + \ell_2^{f_j} \leq 2 \\
& && \ell_k^{f_j} \in \{0,1\}, \quad k = 0,1,2
\end{aligned}
$$

where n is the number of non-border edges. Note that since $\ell_0^{f_j}$ and $\ell_1^{f_j}$ can only be 0 or 1, $\ell_0^{f_j} \neq \ell_1^{f_j}$ is modeled as $\ell_0^{f_j} + \ell_1^{f_j} = 1$. The constraints ensure that quad and triangle faces use one of the valid half-edge configurations. We solve this optimization problem using branch-and-bound that returns a solution with the minimum number of edges that contain conflicting half-edge labels.

In most cases, the resulting labeling would contain edges with inconsistent half-edge labels, such that $\ell_0^{e_i} \neq \ell_1^{e_i}$ for some edges e_i. In fact, around certain types of singularities, we are guaranteed to have inconsistent half-edge labels. In particular, vertices with odd valance that are surrounded by quad faces, such as the example in Figure 3.4, would have at least one edge with inconsistent half-edge labels. Therefore, before we solve the optimization problem, we triangulate the faces surrounding singularities containing vertices with odd valence. This provides additional flexibility in assigning half-edge labels around such singularities and makes it possible to label the half-edges around them consistently. At the end of the labeling process, we can recover some of these triangulated quads via our post-processing operations (Section 3.3.3).

3.3.2 Labeling Edges

After we label the half-edges, we can label all edges with consistent half-edge labels. Edges with inconsistent half-edge label, however, require topological modifications to the

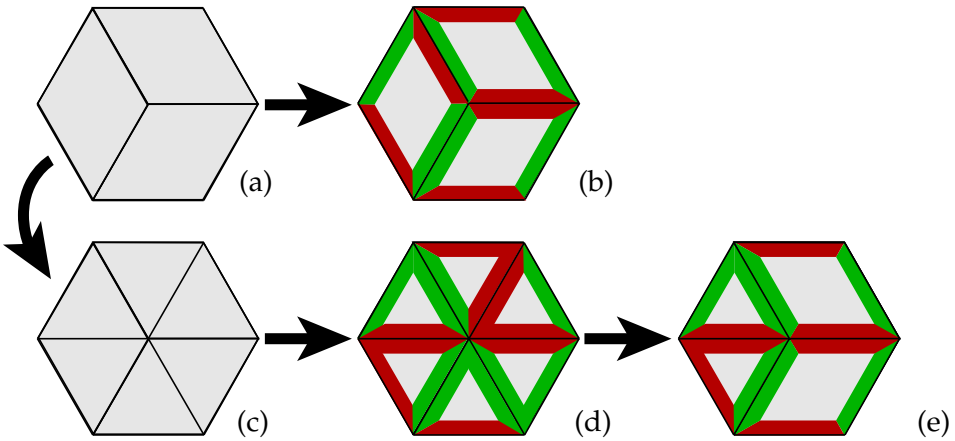


Fig. 3.4: Triangulation near singularities: (a) singularities containing vertices with odd valance lead to (b) inconsistently labeled half-edges; therefore, (c) we first triangulate the quads near such singularities to provide more flexibility during half-edge labeling, and then (d) such triangles can be merged at the end of the labeling process.

mesh. There are three alternatives that are handled differently: an edge with inconsistent half-edge labels might be between two quads, a quad and a triangle, or two triangles.

If an edge with inconsistent half-edge labels is between two quads, we label the edge as a course edge, then split the quad with the wale half-edge label into two triangles. The alternative of labeling the edge as a wale edge and splitting the other quad is also an acceptable solution, but this would split the row on one side of the edge (since neighboring quad faces sharing wale edges form rows), so we prefer the other alternative. A quad can be split into two triangles in two different ways along either one of its diagonals. They produce similar results, so we randomly pick one diagonal. Once we split a quad into two triangles, any possible half-edge labeling configuration can be represented by combinations of the two triangle labeling configurations we permit, as shown in Figure 3.5. Therefore, while assigning the half-edge labels for these two new triangles, we make sure that they do not contain other edges with inconsistent half-edge labels. Thus, we simply use the half-edge labels on the other sides of their edges. An example of this operation is shown in Figure 3.6, where one of the quad faces sharing an edge with inconsistent half-edge labels is split into two triangles, and the half-edge labels of the new triangles are assigned such that the triangles do not contain edges with inconsistent labels.

If an edge with inconsistent half-edge labels is between a quad and a triangles, we split the quad face. Again, we can use either one of the diagonals for splitting the quad

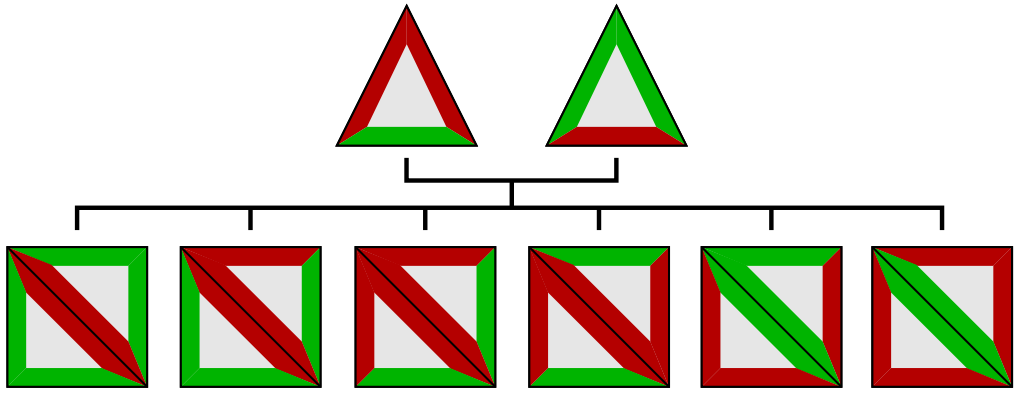


Fig. 3.5: Triangulation of quad faces: (top) the two valid configurations for labeling half-edges of triangles can be used for representing (bottom) all possible configurations for labeling half-edges of quads.

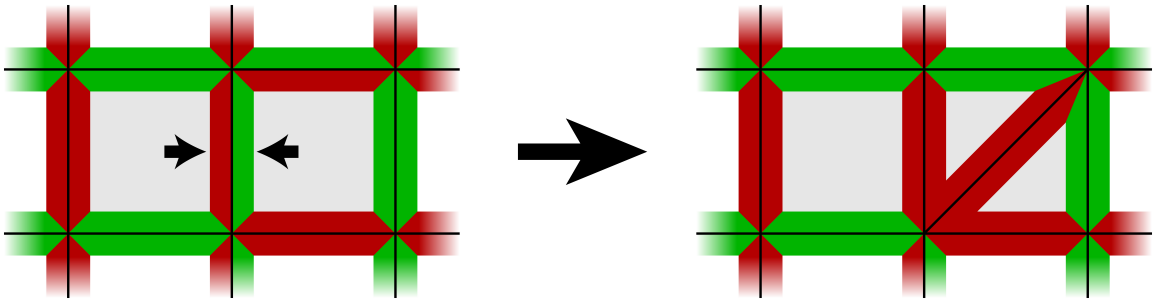


Fig. 3.6: Splitting quad faces: (left) if an edge with inconsistent half-edge labels is between two quad faces, (right) the face with the wale half-edge label is split into two triangles.

face. Similarly, we make sure that the two new triangles do not contain other edges with inconsistent half-edge labels.

If an edge with inconsistent half-edge labels is between two triangles, we rotate the edge, as shown in Figure 3.7, and we label the rotated edge as a course edge. Note that labeling the rotated edge as a course edge is guaranteed to form two valid triangle configurations on either side of the edge. This is because the shared edge between two triangles can have inconsistent half-edge labels only when the other half-edges of one triangle are labeled as course and other half-edges of the other triangle are labeled as wale. Otherwise, the optimization process for assigning the half-edge labels would have resolved the inconsistency in half-edge labeling.

Note that none of these topological operations lead to new inconsistencies in half-edge

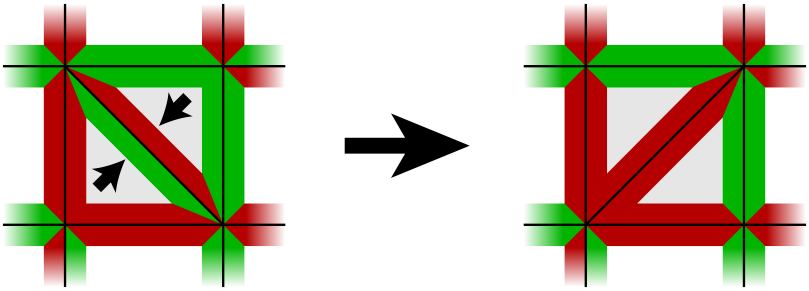


Fig. 3.7: Rotating edges between triangle pairs: (left) if an edge with inconsistent half-edge labels is between two triangles, (right) the edge is rotated.

labeling. Therefore, all edges can be labeled in a single pass without the need for multiple iterations.

3.3.3 Postprocessing

Our postprocessing operations involve pairs of neighboring triangles. If the edge labels of a pair of neighboring triangles are such that merging them into a quad by removing the common edge between them would lead to a quad with an acceptable labeling configuration (as in Figure 3.3a), we merge the two triangles into a quad. An example of this operation is shown in Figure 3.8. This operation reduces the number of triangles and unnecessary complexity in the final knit structure. Such pairs of triangles commonly appear around singularities containing a vertex with an odd valence, since we triangulate the quad faces around such singularities before labeling the half-edges. Thus, this operation can recover some of the quad faces around those singularities. Yet, such pairs of triangle can appear on other parts of the model as well. In particular, the triangulation process used for labeling edges can also produce such triangle pairs, which are converted to quads in this step.

In some cases flipping the label of a course edge between two triangles can allow merging these triangles with other neighboring triangles, as shown in Figure 3.9. Therefore, we scan course edges between pairs of triangles with at least one of them connected to another triangle and check if flipping the edge label would allow merging the nearby triangle into quads. If so, we flip the edge label and merge the triangles.

Finally, we consider pairs of neighboring triangles sharing an edge labeled as a wale edge. If both triangles of such an edge have other edges labeled as wale edges and that

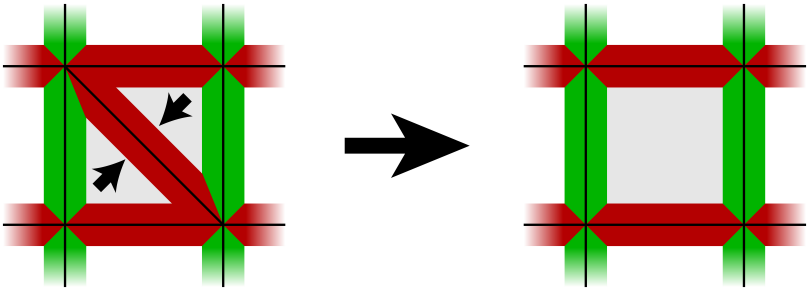


Fig. 3.8: Merging triangles: (left) if removing an edge between two triangles would lead to a quad with valid labeling, (right) we merge the two triangles.

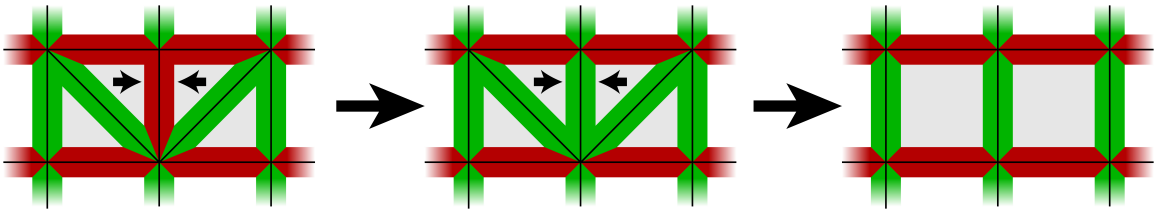


Fig. 3.9: Merging triangles after flipping the label of a course edge: (left) two pairs of triangles separated by a course edge are labeled such that (middle) flipping the label of the course edge allows (right) merging the triangles into quad faces.

merging them would not lead to a quad face with a valid configuration, we flip the label of the shaded edge to a course edge. The reason for this operation becomes more clear after discussing the subdivision operation that generates the final stitch mesh (Section 3.5). This is because if the common edge label for this particular pair of triangles is kept as a wale edge, the resulting stitch mesh would contain triangular stitch-mesh faces that cannot always be safely eliminated, which may result in unstable stitches that would unravel during yarn-level simulation.

3.4 Knitting Direction Assignment

After labeling the edges, we must determine the knitting directions over the model surface. On each face the course and wale knitting directions are aligned with the course and wale edges, respectively. We can arbitrarily pick either one of the two possible course directions (i.e., left-to-right or right-to-left), since a stitch can be formed using either direction. The choice for the wale directions, however, is not arbitrary, since it determines which course edges of a face are the bottom course edges and which ones are the top course edges.

We would like the wale direction to be uniform over the entire model. This means that if an edge is treated as a bottom course edge for one face, the other face sharing the edge should treat it as a top course edge. This aligns the wale knitting directions for the two faces. However, we cannot enforce this as a hard constraint, because some shapes would require having mismatched wale directions in certain places, depending on how the knit structure form the surface. Therefore, we perform another optimization that provides a solution with the minimum number of course edges that are along mismatched wale directions.

Note that in our labeling each quad face is assigned exactly two wale edges, and each triangle can have one or two wale edges. Therefore, a group of edges connected with wale edges form a string of faces that we call a *row*. Each row can either form a closed loop, or it can begin and end with two triangle faces, each with a single wale edge. Each face belongs to a single row, and neighboring rows are separated by course edges.

One hard constraint for this optimization is that the wale directions of two neighboring faces sharing a wale edge must be aligned. Otherwise, the resulting wale directions would not form a valid stitch mesh. This means that the wale direction along each row must be consistent. Therefore, instead of formulating the optimization problem for determining the wale directions per face, we can reduce the dimensionality of the problem by formulating it per row of faces. We achieve this by building a metagraph of the mesh, such that each row of the mesh corresponds to a node of the metagraph. An example metagraph generated from a mesh is shown in Figure 3.10. Two nodes of the metagraph are connected to each other via undirected weighted edges, if the rows that correspond to these nodes have common course edges. The number of common course edges determine the weight of the edge. Each node of the metagraph contains two halves: one half corresponds to the group of course edges on one side of the row and the other half corresponds to the group of course edges on the other side. Thus, the edges between nodes connect one half of a node to one half of another node.

We formulate a similar mixed integer programming problem on the metagraph. The two halves of each metagraph node are labeled as either *top* or *bottom*, indicating that the course edges corresponding to those halves are either top course edges or bottom course edges. Let $L_0^{M_r}$ and $L_1^{M_r}$ represent the labels of the two halves of a metagraph node M_r

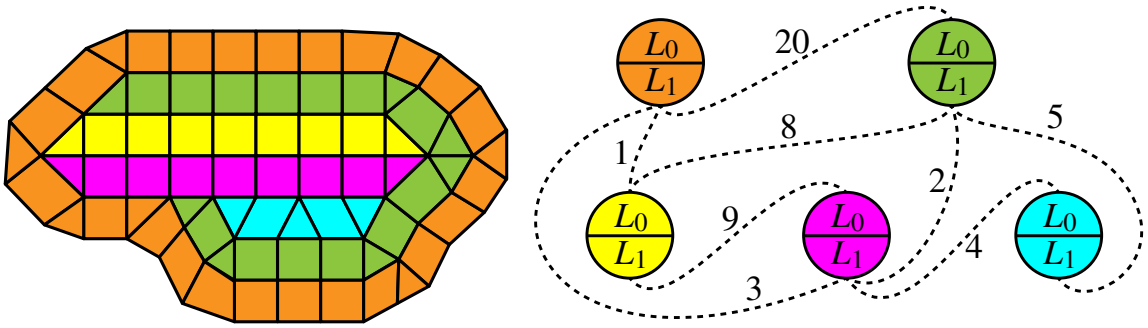


Fig. 3.10: An example metagraph: (left) mesh with separate rows colored differently, and (right) its metagraph.

with index r. We assign them integer values 0 and 1 to indicate top or bottom labels, respectively. The same indices can also be accessed using the edges of the metagraph, such that $L_0^{E_s}$ and $L_1^{E_s}$ are the labels of the two metagraph node halves that are connected by the metagraph edge E_s with index s. Using this notation, we can write the optimization problem that minimizes the number of course edges with mismatched wale directions as

$$\begin{aligned} \textbf{minimize} \quad & \sum_{s=0}^{N-1} W_s\left(1-(L_0^{E_s}-L_1^{E_s})^2\right) \\ \textbf{subject to} \quad & \text{For metagraph node } M_r, L_0^{M_r}+L_1^{M_r}=1 \\ & L_k^{M_r}\in\{0,1\}, \quad k=0,1, \end{aligned}$$

where N is the number of metagraph edges, and W_s is the weight of the edge E_s (i.e., the number of course edges between the two rows). The constraint $L_0^{M_r}+L_1^{M_r}=1$ ensures that the two halves of the node M_r are assigned different labels. We solve this problem using branch-and-bound. Since the metagraph contains a relatively small number of nodes (as compared to the number of faces), this optimization can be solved efficiently.

3.5 Stitch Mesh Generation

The resulting mesh after assigning the knitting directions can be directly used as a stitch mesh. However, it contains triangle faces, which are undesirable. In particular, each triangle face with a single wale edge, marking the beginning or ending of a row, would lead to a knot in the yarn-level model. To avoid this, we perform a subdivision operation, similar to Catmull-Clark subdivision [164], which converts each quad face into four quads and each triangle face into three quads.

There are three cases to consider for labeling the new edges generated by the subdivision operation, as shown in Figure 3.11. Quad faces form four regular quads. Triangle faces, however, form two regular quads and one special quad with a different labeling configuration, where wale edges are not separated by course edges. We handle these quad faces with different labeling configuration differently.

Triangles with two course edges form a special quad with one bottom course edge and one top course edge. Such quads mark the beginnings and endings of stitch mesh rows. Therefore, they are handled differently than other stitch mesh faces when generating the yarn curves, as explained in Section 3.6.

Triangles with two wale edges, however, form a special quad with either two bottom course edges or two top course edges. Such quads do not correspond to a valid stitch; therefore, we eliminate them. We begin with triangulating these quads by splitting them with a diagonal wale edge that forms two triangles, each with a single course edge. Finally, we merge these two triangles with the quad faces on either side, forming pentagons that represent either increase or decrease type stitches. Note that our postprocessing after labeling edges (Section 3.3.3) ensures that there is always a quad face next to these triangles, since we do not permit having two triangles with two wale edges side-by-side, sharing a wale edge.

Figure 3.12 shows an example row that is subdivided into a stitch mesh. Special quad faces appear on either ends of the row as well as the top center of the row, which are handled differently. Note that after the subdivision operation, all rows of the resulting stitch mesh form closed loops with no end points.

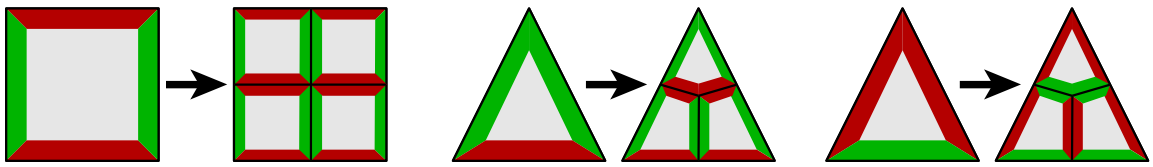


Fig. 3.11: Subdivision rules for (a) quad faces, (b) triangle faces with two wale edges, and (c) triangle faces with two course edges.

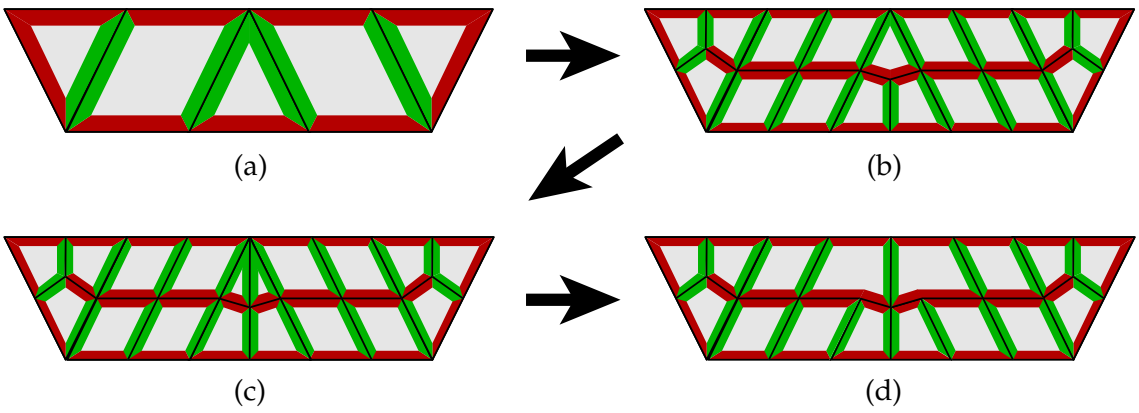


Fig. 3.12: Stitch mesh generation: (a) the faces on each row are (b) subdivided into quad faces; (c) the face at the center with two bottom course edges is triangulated; and finally (d) the triangles are merged with the neighboring quad faces.

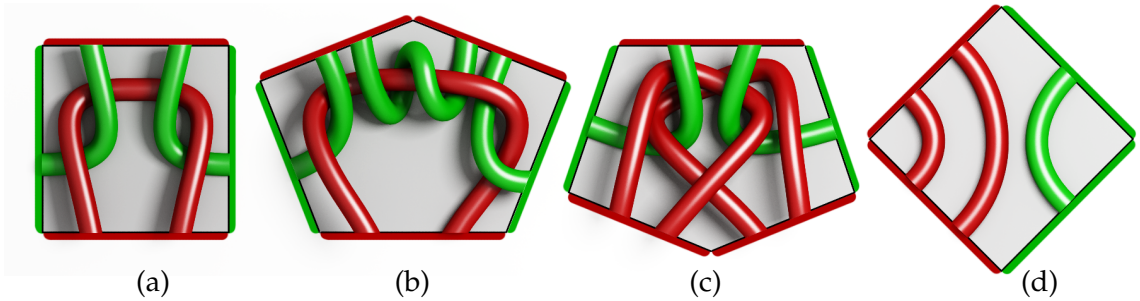
3.6 Relaxation and Yarn Generation

Before we generate the yarn curves from the stitch mesh, we perform mesh-based relaxation [164] . While the initial remeshing step provides a good starting point that results in faces with approximately the same size over the entire model, due to the topological operations we perform during labeling and stitch mesh generation, an optional mesh-based relaxation step can provide some minor improvement in unifying the edge lengths and minimizing the deformation of quad faces. Figure 3.13 demonstrates zoom in view of "fertility" model before and after mesh-based relaxation.

The stitch mesh models we generate have four different types of faces: (1) regular quad faces, (2) pentagon faces representing increases, (3) pentagon faces representing decreases, and (4) special quad faces marking row ends. For regular quad faces, we generate yarn curves of a knit stitch (k), shown in Figure 3.14a. Pentagon faces representing increases use a knit followed by a purl that are pulled through the same loop (kp), as shown in Figure 3.14b. Pentagon faces representing decreases use a knit stitch that is pulled through two loops ($d_{12}k$), as shown in Figure 3.14c. Finally, special quads that mark the end of the rows simply connect the yarn of the course edges together and the wale edges together, as in Figure 3.14d. Note that the stitch mesh representation allows replacing these stitches with other stitch types, if desired. However, this would require manual stitch mesh editing, so we simply use the corresponding stitch type in Figure 3.14 for all faces.

(a) (b)

Fig. 3.13: Mesh-based relaxation: (a) stitch mesh before mesh-based relaxation, and (b) stitch mesh after mesh-based relaxation.



(a) (b) (c) (d)

Fig. 3.14: Stitch types: (a) regular quads, (b) increases, (c) decreases, and (d) special quads.

3.7 Results

Figure 3.1 shows complex 3D models that are automatically converted to yarn-level knit structures using our pipeline. Notice that the resulting yarn-level models have uniform stitch sizes over the model surfaces. Our pipeline supports high genus surfaces, as demonstrated in Figure 3.2f. We can also handle models with high-curvature areas, such as the example in Figure 3.15.

The surface details preserved in the final yarn-level model depends on the resolution of the generated stitch mesh. Figure 3.16 shows the "bunny" model with five different resolutions. While all five results are valid stitch meshes with uniform stitch sizes, only the one with highest resolution captures the small-scale details of the input surface. Notice that representing small-scale surface details also introduces additional singularities that are needed for shaping the knitted model. When the resolution of stitch mesh is too low, the remeshed result will lose features such as the ear in the 1.3K knit bunny.

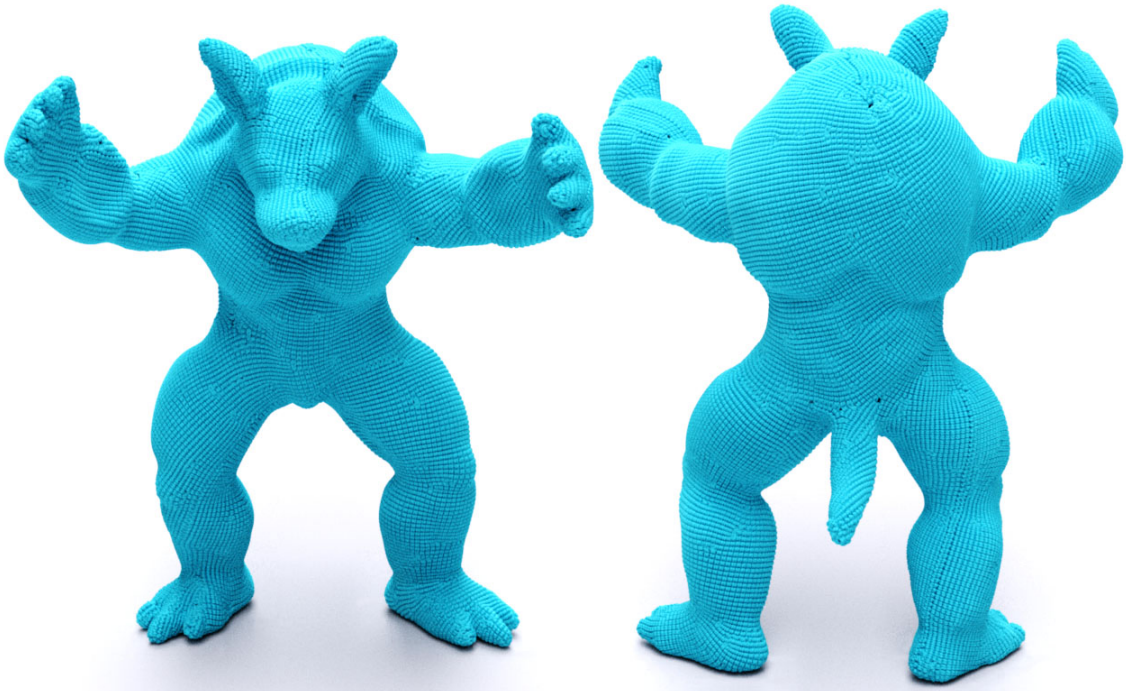


Fig. 3.15: Yarn-level knit structure for the Armadillo model with high-curvature areas around the ears, the tail, the fingers, and the toes.

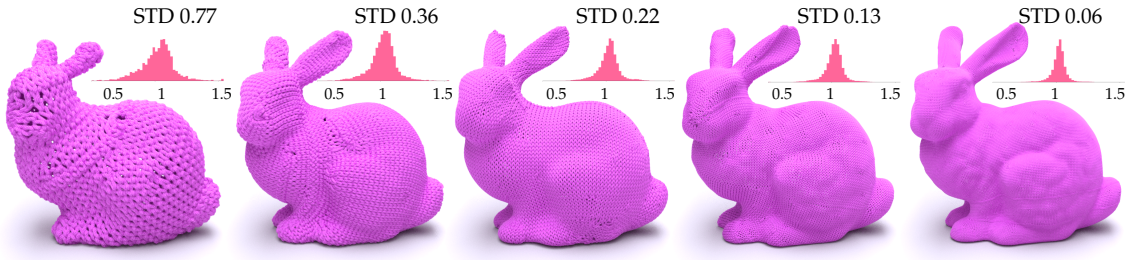


Fig. 3.16: Yarn-level knit structures generated from the "bunny" model with three different resolutions: 1.3K, 4K, 7K, 16K, and 48K stitches.

Even CAD models with sharp features can be processed by out pipeline (Figure 3.17): the sharp features of the input model are partially smoothed due to the relatively low resolution of the yarn-level model, but the overall shape is preserved. Notice that when using an intrinsic orientation field (Figure 3.17a), the knitting directions are not aligned with the surface details [66, 71]. Using an extrinsic orientation field instead (Figure 3.17b) makes the knitting directions of the final model follow the surface details better, but it also introduces additional singularities to align the orientation field with the model features. Therefore, the yarn-level models in Figures. 3.1, 3.2, 3.15, and 3.16 are generated using an intrinsic orientation field.

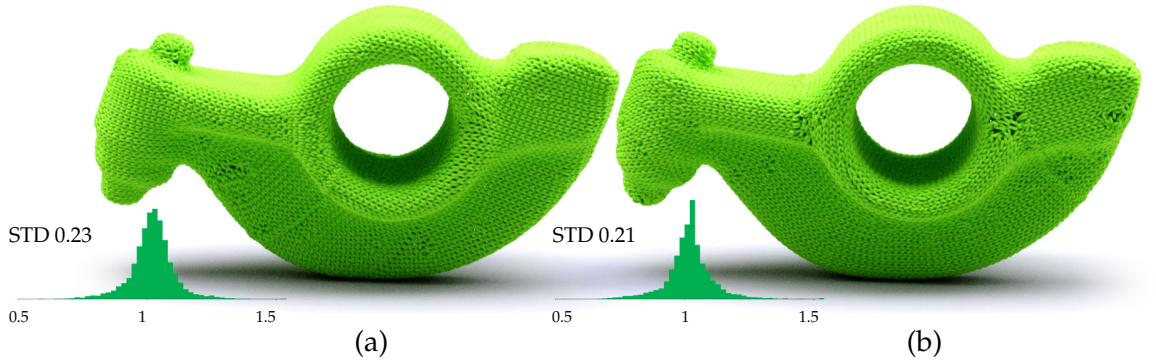


Fig. 3.17: Yarn-level "rocker arm" model generated using (a) an intrinsic orientation field and (b) an extrinsic orientation field.

Our pipeline can also be used with custom orientation fields, to provide additional control over the final knitting directions. Figure 3.18 shows the stitch meshes and the final yarn-level model generated from the same input shape using both the intrinsic orientation field and a custom orientation field, generated with a small set of user-defined strokes. The stroke compete with the field smoothness, leading to a small increase in the number of irregularities in the knitting pattern.

3.7.1 Performance

Notice that most of the steps in our pipeline can be computed within several seconds to a few minutes, depending on the size and complexity of the input model and the resolution of the output model. However, after we generate the yarn curves, the yarn-level relaxation step that produces the final yarn curve shapes can take hours. Aside from the relaxation operations, the most expensive component of our pipeline is the optimization we use for labeling the half-edges. Figure 3.19 shows half-edge labeling time per-face for different quad-dominant mesh resolutions, indicating that computation time per-face increases with mesh resolution, and it depends on the topological complexity of the mesh.

The two-step optimization for labeling and direction assignment (Sections 3.3 and 3.4) is the key to the efficiency of our algorithm. We experimented with an integrated optimization tackling jointly both problems, and we observed that the larger solution space of the combined optimization dramatically increases the computational requirements. On the 16K "bunny" model (Figure 3.16), the combined optimization finished the 16GB of available memory after 40 minutes of computation and started thrashing. In comparison,

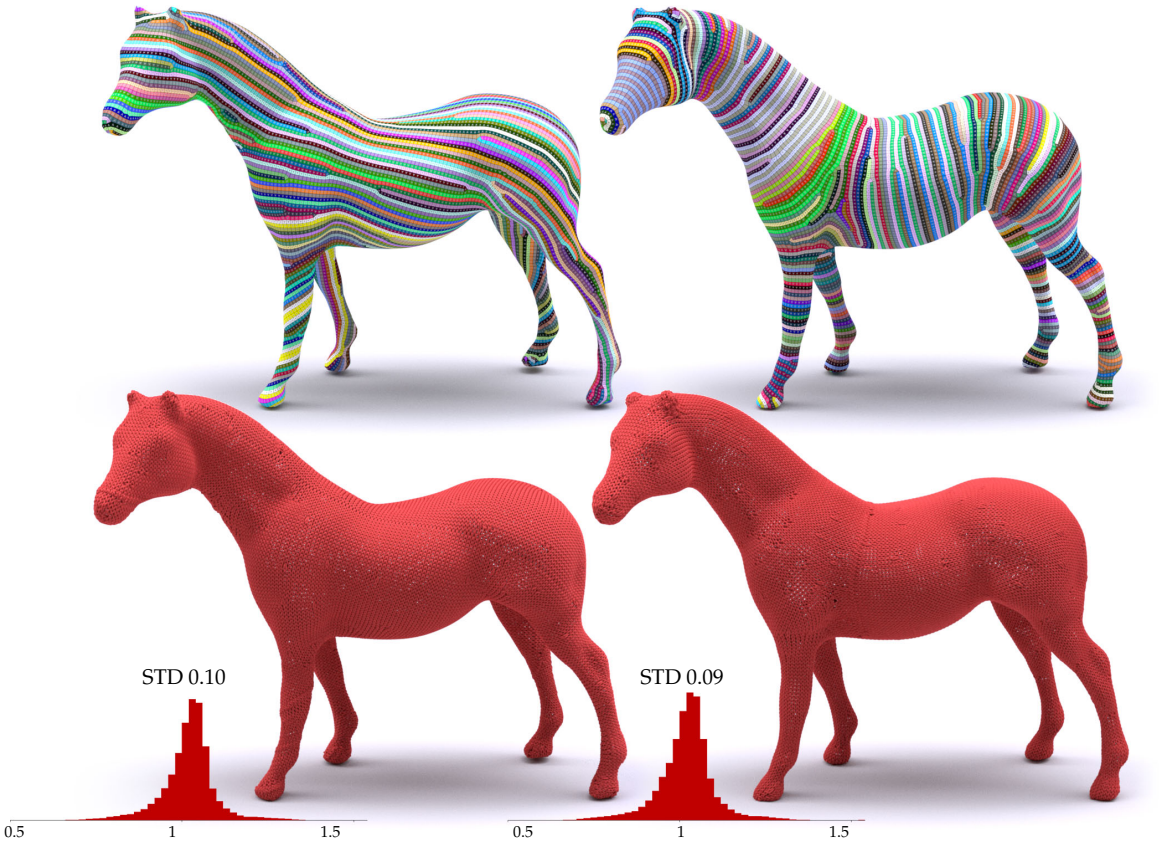


Fig. 3.18: Stitch meshes and yarn-level knit models generated using (a) the default orientation field, and (b) user-defined orientation field with orientation constraints interactively drawn on the model surface.

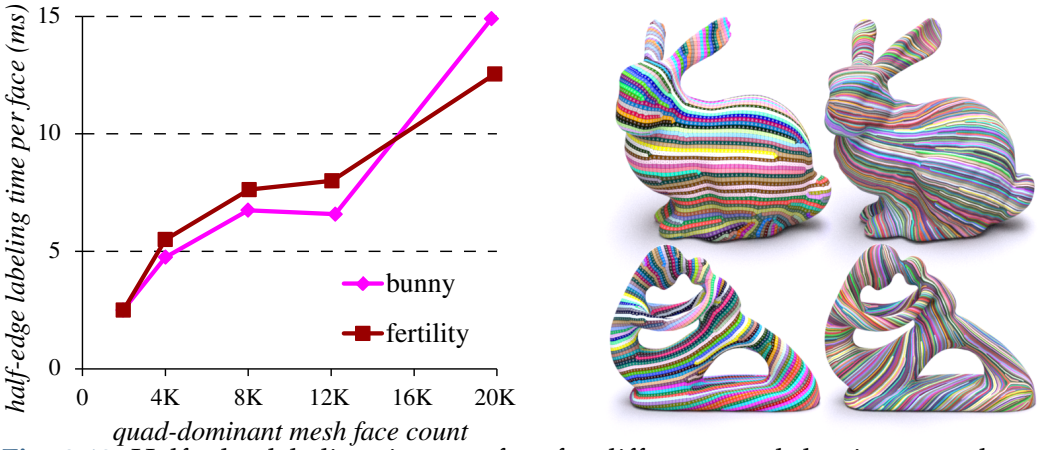


Fig. 3.19: Half-edge labeling time per face for different quad-dominant mesh resolutions. The final stitch meshes for the lowest and highest resolution examples in the graph are shown on the right.

our two-step solution takes 24 seconds for half-edge labeling and less than a second for knitting direction assignment. While it is possible that the combined optimization could produce results with fewer mismatched knitting directions, our two-step optimization provides superior computational performance and lower memory usage.

The performance results of our pipeline for generating various yarn-level models that are presented in this section are shown in Table 3.1.

3.7.2 Robustness

We demonstrate the robustness of our pipeline by automatically processing a collection of 104 models (Figure 3.20). The set includes models with high genus, sharp features, and thin parts, and our algorithm generated a valid stitch mesh model with roughly uniform face sizes for all of them models. We also demonstrate the robustness by generating low resolution knit model from the complicated shape (Figure 3.21).

We measure edges length for stitch meshes after mesh-based relaxation and show histograms in Figures 3.16, 3.17, and 3.18. The standard deviation (STD) mainly depends on the mesh size after remeshing rather than intrinsic/extrinsic field and whether custom orientation filed is used. Variation of edge length is mostly introduced due to singularities and stitch size variations around singularities and also appear in real-world knitting.

Table 3.1: The computation performance measurements for the steps of our pipeline.

	Remesh	Labeling	K. Direction Assignment	Stitch Mesh Gen.	Mesh-based Relaxation	Yarn Gen.
Rocker Arm	2 s	8 s	99 ms	593 ms	12 s	18 ms
Rocker Arm	2 s	4 s	127 ms	583 ms	9 s	22 ms
Chinese Lion	4 s	19 s	198 ms	1049 ms	18 s	39 ms
Kitten	4 s	16 s	124 ms	1083 ms	16 s	37 ms
Dragon	4 s	26 s	370 ms	1234 ms	53 s	35 ms
Horse	6 s	17 s	159 ms	1297 ms	25 s	55 ms
Horse	6 s	18 s	306 ms	1311 ms	45 s	52 ms
Elephant	13 s	26 s	237 ms	1421 ms	28 s	51 ms
Fertility	8 s	32 s	192 ms	1495 ms	46 s	54 ms
Armadillo	13 s	58 s	567 ms	1963 ms	88 s	77 ms
Bunny (1.3K)	4 s	2 s	45 ms	119 ms	6 s	4 ms
Bunny (4K)	4 s	2 s	66 ms	315 ms	8 s	12 ms
Bunny (7K)	4 s	2 s	131 ms	550 ms	10 s	19 ms
Bunny (16K)	5 s	16 s	147 ms	1101 ms	12 s	45 ms
Bunny (48K)	37 s	84 s	399 ms	3526 ms	130 s	159 ms

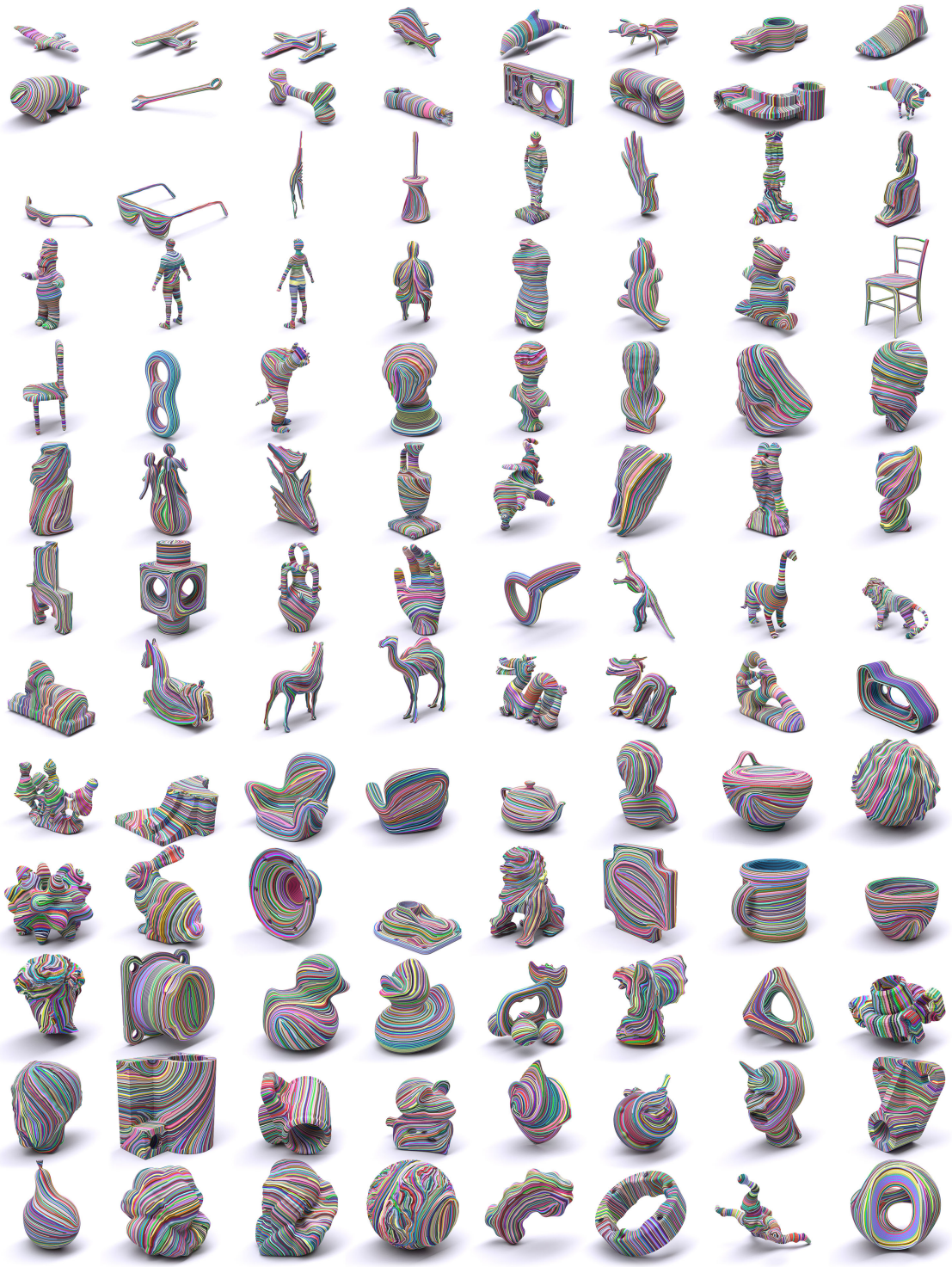


Fig. 3.20: Stitch meshes generated by our fully automatic pipeline using an extrinsic orientation field.

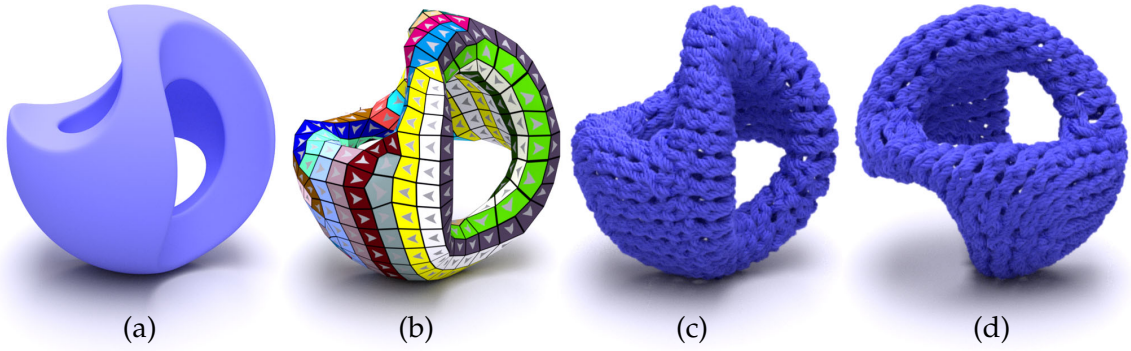


Fig. 3.21: Our method used for designing a custom fitted glove: (a) the input shape, (b) the stitch mesh, (c) and (d) front view and side view of yarn-level knit model.

3.7.3 Remeshing and Labeling

We compare different methods for generating quad-dominant meshes in Figure 3.22, measuring the quality using the number of inconsistencies produced after labeling half-edges, which correspond to irregularities in the stitch-mesh. Directly using RIM [45] or our modified method with a 4-RoSy field, we get a large number of inconsistent edge labels. Switching to a 2-RoSy field greatly improves the quality, but still struggles due to the topology of the mesh near some singularities. Triangulating the neighborhood of singularities before labeling the half-edges, enlarges the solution space and allows our optimization to substantially reduce the number inconsistent edge labels. Note that Gurobi library [57] is used for solving MIP problems for labeling and direction alignment with 8 threads.

3.7.4 Simulation

We produce valid stitch meshes and, therefore, yarn-level models with topologically correct knitted structures. All our models can be directly used for yarn-level simulation. Figure 3.23 shows example frames from an animation of a bunny model deforming under gravity computed using a yarn-level simulation. Notice that all stitches remain intact during the simulation.

3.7.5 Fabrication via 3D printing

We show a 3D printed yarn model: the sweater is made of black nylon, and it has been printed in nylon using Fused Deposition Modeling and a water-soluble supporting material (Polyvinyl Alcohol). A clip documenting the fabrication procedure using the

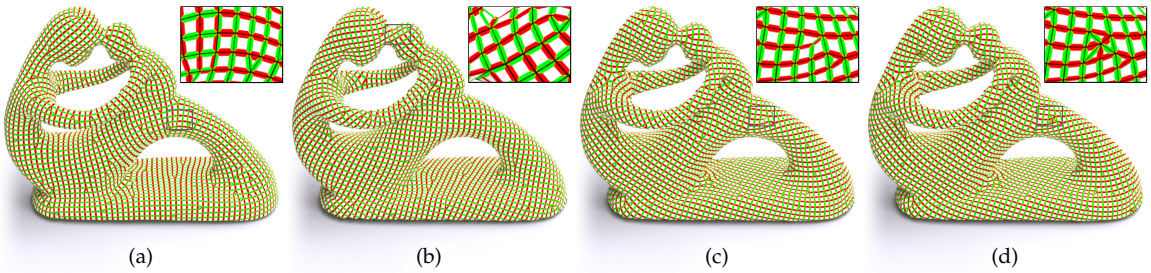


Fig. 3.22: Comparison of different methods for orientation field generation: the number of inconsistent edge labels they produce after half-edge labeling, showing (a) 115 inconsistencies using [45], (b) 130 inconsistencies using our method with a 4-RoSy field, (c) 52 inconsistencies using our method with a 2-RoSy field ab, and (d) 22 inconsistencies using our method with a 2-RoSy and triangulated singularities. Note that different methods create inconsistencies on the different parts of the model surface as highlighted, but the 2-RoSy field leads to fewer inconsistencies, especially when combined with triangulated singularities.

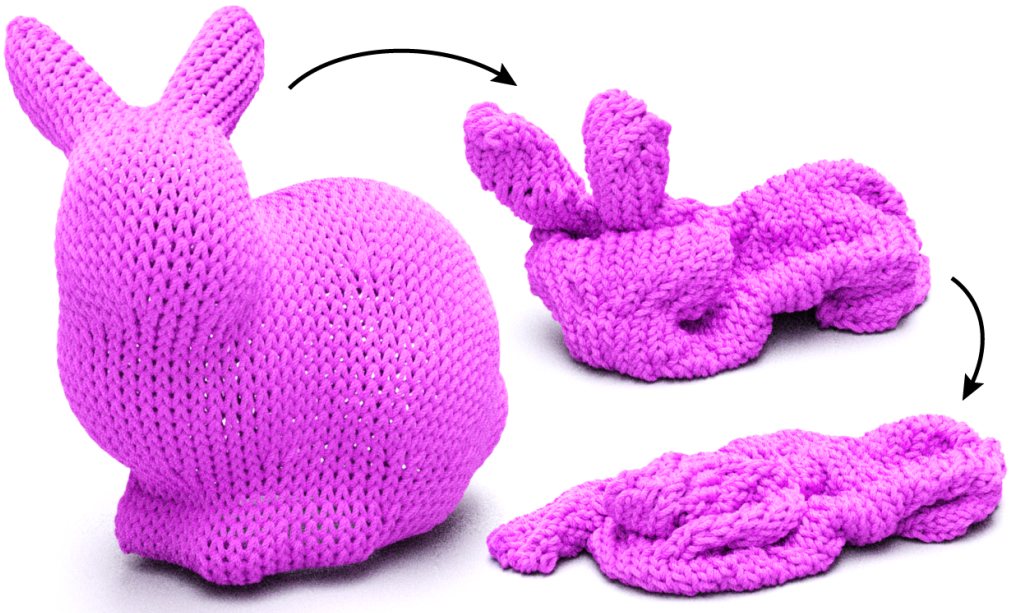


Fig. 3.23: Example frames from a yarn-level simulation of a bunny model deforming under gravity.

"Ultimaker 3" [142] printer is attached in the additional material. Since nylon is a stiff material, the sweater is only mildly flexible, and it does not collapse under its weight as shown in Figure 3.24. This fabrication method is affordable, enabling the production of decoration and lightweight physical realizations of 3D shapes using 3D printers.

We also prototyped a design pipeline for tailored gloves, combining this fabrication method with a 3D scanning pipeline (David 3D Scanner [32]), shown in Figure 3.25. First,

Fig. 3.24: An octopus wearing a knitted sweater: (left) simulated model and (right) fabricated via 3D printing.

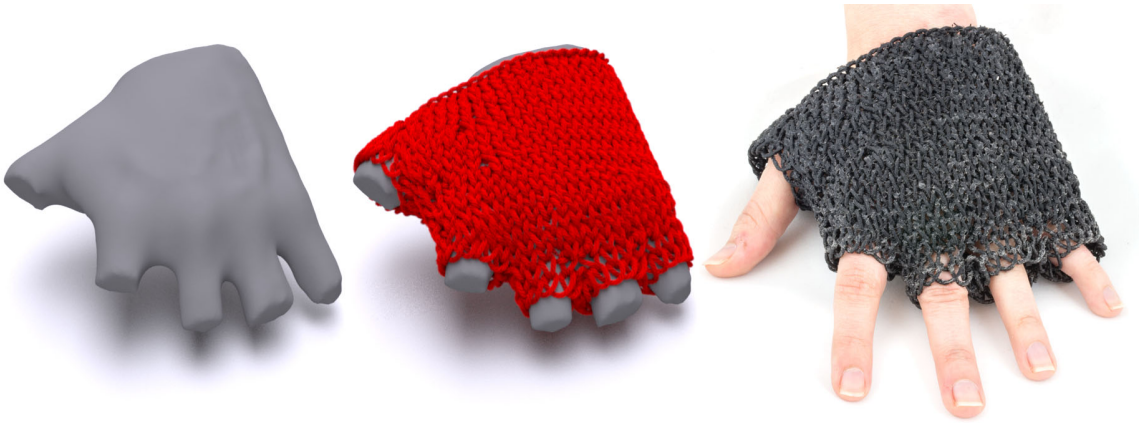


Fig. 3.25: Our method used for designing a custom fitted glove: the hand model acquired using a structured light 3D scanner, the simulated model, and 3D printed glove.

the user's hand is scanned; then, a desired part is manually selected and enlarged; finally, the model is automatically transformed into a yarn model by our method and printed, leading to wearable nylon glove. The comfort and flexibility of the final model depends on the material used for printing.

Finally, we printed a knitted Bunny using selective laser sintering (Figure 3.26). The fabricated model is flexible and robust, as shown in the supplementary video.

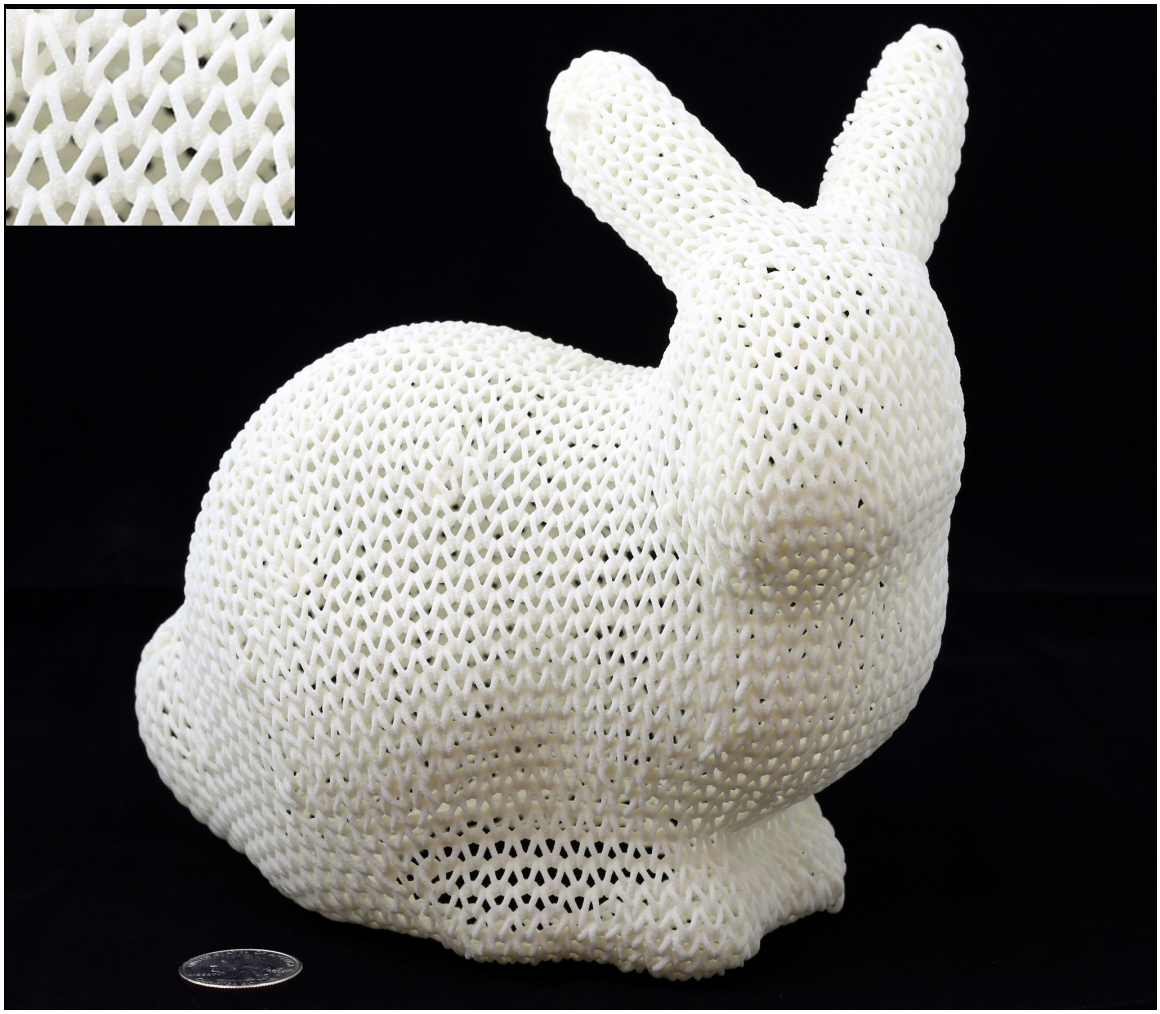


Fig. 3.26: A knitted "bunny" model generated with our pipeline and printed using selective laser sintering.

3.8 Conclusion

We have introduced a fully automatic method for converting arbitrary 3D shapes into knit structures, starting with quad-dominant mesh generation, followed by a two-step optimization process and topological operations that generate a valid stitch mesh. We have demonstrated the effectiveness of our approach with complex knit models generated using our pipeline and the robustness of our method by processing a large number of different 3D models. The yarn-level models we produce are guaranteed to have valid knit topologies, and they are ready to be used with yarn-level simulations. To our knowledge, this is the first fully automatic method that can produce yarn-level knit model for arbitrary 3D shapes.

One important limitation of our approach is that fine-scale details of the input surface may not be properly represented in the final knit model, unless a high-enough resolution stitch mesh is generated. Since we rely on stitch meshes, we share the limitations of the stitch mesh representation. In particular, we cannot produce multilayer knit structures that are used for colored knitting patterns. In addition, our algorithm can be extended to handle nonorientable surfaces, such as a Mobius strip. However, our current implementation uses a half-edge data structure that cannot represent nonorientable surfaces.

Our current remeshing method can generate isotropic quad-dominant meshes by assuming each quad-shaped stitch face would ideally be square. However, in reality the ratio between the width and height of stitches can vary depending on the yarn type, the needle size, and the details of the knitting operations. It would be an interesting future direction to investigate variations of our pipeline that generate rectangular stitch mesh faces with a user-specified aspect ratio.

Though our method allows custom orientation fields to be used to provide additional control over the final knitting direction, automatically generating an optimal orientation field for minimizing singularities or to better preserve the shape is an interesting direction for future work.

CHAPTER 4

KNITTABLE STITCH MESH

In computer graphics, yarn-level simulations of cloth is known to produce compelling animations [28, 29, 72, 76, 77]. This is particularly important for knitted cloth, as it often exhibits complex yarn-level deformations. The challenging problem of designing yarn-level knitted cloth models is solved by the stitch mesh modeling framework [164]. Stitch meshes provide a full 3D modeling interface for knits with a collection of high-level and low-level modeling operations that allow designing arbitrary knitting patterns, including complex constructs like knitted cables. In fact, the stitch mesh modeling framework, with its ability to model arbitrary 3D forms, is superior to the modeling interfaces used in the textile industry.

Yet, stitch meshes do not produce knittable models (except for flat swatches). This is partially because each row of stitches on a stitch mesh form a closed yarn piece with no end points and knitting direction inconsistencies of neighboring faces are ignored, resulting in models that can be used for yarn-level simulation, but cannot be produced via knitting operations. Furthermore, the original stitch mesh representation does not support short-rows–a crucial 3D shaping technique used heavily with most knit cloth. This severely limits the 3D forms that can be reliably modeled (and then simulated) using stitch meshes, and precludes using the stitch mesh modeling framework for fabrication purposes.

In this chapter, we introduce *knittable stitch meshes* that can guarantee that the designed final model is actually knittable. The technical contributions in this chapter include the following:

- *Shift-Paths:* a novel concept, which is extremely simple to implement, for converting nonknittable tubular sections of a stitch mesh into knittable helix form,
- *Knittable Mismatched Directions:* four different (including two novel) types of mismatched knitting directions that are needed for designing complex 3D forms, and

the corresponding automated modifications to the stitch mesh structure for ensuring that they are knittable,

- *Short-Rows:* a novel structure required for representing short-rows within the stitch mesh modeling framework, and

- An algorithm for generating step-by-step knitting instructions from a given knittable stitch mesh.

Thus, we extend the concept of stitch mesh modeling for designing more complex 3D models and for producing knittable structures that can be fabricated via knitting. As long as the input mesh can be completely labeled (using the labeling process of the stitch mesh modeling framework [164]) and thus can be converted to a stitch mesh, we guarantee that it can be converted to a knittable stitch mesh using our approach. To verify that our results are knittable, we also developed a graphics interface for aiding a knitter by providing step-by-step instructions. We present topologically complex models knitted by following the knitting instructions using this interface. An example model generated using our modeling framework is shown in Figure 4.1.

Our main goal in this chapter has been solving the problems of the stitch mesh modeling framework for representing realistic knit structures and actually knittable models. We do not, however, consider the physical limitations of knitting machines. Therefore, while

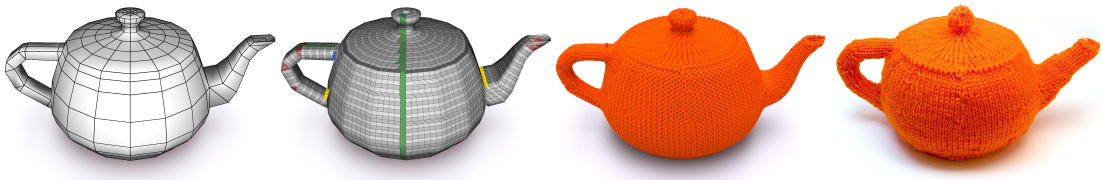


Fig. 4.1: Stages of our knittable garment modeling system: (a) We begin our interactive modeling process with an input polygonal mesh that specifies the global shape of the model. (b) Using this polygonal mesh, we produce a high-resolution stitch mesh including shift-paths (shown as green faces) that form knittable spiral structures and splitting (yellow) and joining (blue) mismatched faces that connect them without seams. Afterwards, we can either (c) generate the yarn curves from the stitch mesh and use a physically-based relaxation process to produce the final yarn-level shape for rendering, or (d) knit the model using the knitting instructions generated from our knittable stitch mesh.

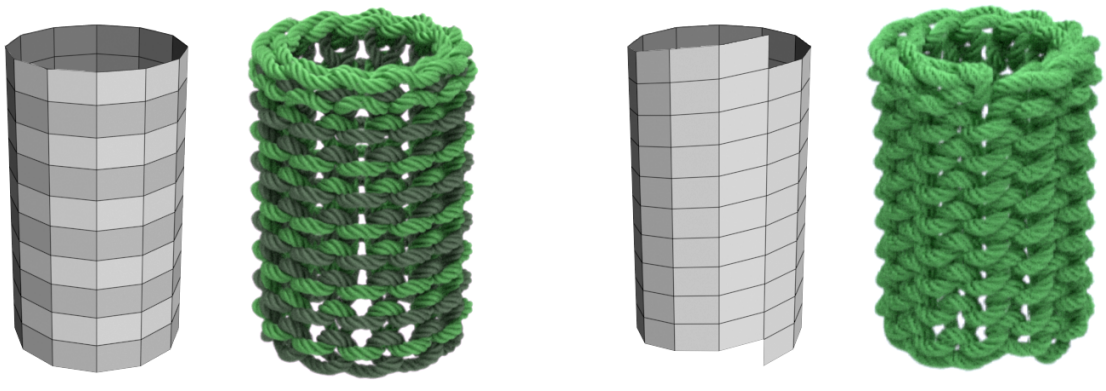
we guarantee that the resulting models are knittable, we do not test whether the required sequence of knitting instructions can be performed by a particular knitting machine.

4.1 Knittable Stitch Meshes

Knittable stitch meshes extend the concept of stitch meshes for designing actually knittable structures. This is achieved by removing the nonrealistic assumptions of original stitch meshes and introducing fundamental shaping concepts. In particular, our knittable stitch meshes incorporate novel concepts to the stitch mesh modeling framework in three groups: *shift-paths* (Section 4.1.1), *knittable mismatched directions* (Section 4.1.2), and *short-rows* (Section 4.1.3). These concepts are necessary for modeling general 3D knittable structures (Section 4.2) and sufficient for making sure that the final outcome is indeed knittable (Section 4.3).

4.1.1 Shift-Paths

An important nonrealistic assumption of the original stitch meshes is that the yarn piece for each row is handled separately. Therefore, a row on a tubular part of a model corresponds to a closed yarn curve with no end points (Figure 4.2a). Instead, knit structures are formed by following a helix progression, as shown in Figure 4.2b. However, this helix formation effectively converts multiple rows into a single (helix-shaped) row, and introduces unnecessary complexity to all stages of the modeling framework from labeling



(a) (b)

Fig. 4.2: An example stitch mesh with (a) separate rows that lead to separate yarn pieces per row with no end points and (b) helix progression that connects the rows and can be knit by a single yarn piece with end points at the first and the last stitches of the helix.

to stitch mesh generation and editing. In fact, this complexity is even avoided in real-world knitting instructions by defining each row separately. Therefore, an ideal solution must preserve the separate-row representation, but form the helix structure only when needed.

We introduce the concept of *shift-paths* for slightly modifying the stitch mesh structure to automatically construct a helix formation at the very end of the modeling process. A shift-path merely marks a set of connected wale edges at consecutive rows (Figure 4.3a). The structure of the stitch mesh remains unaltered until the end of the modeling process, so all prior stitch mesh modeling methods can be used without modification. Each wale edge along the shift-path indicates where the row begins and ends, and where the last stitch should be connected to the first stitch of the next row.

It is easy to automatically pick a set of wale edges as a shift-path where needed, but since it impacts the shape of the final model, we leave the choice of where to place the shift-paths to the user. Once a shift-path is specified by the user, the stitch mesh can be

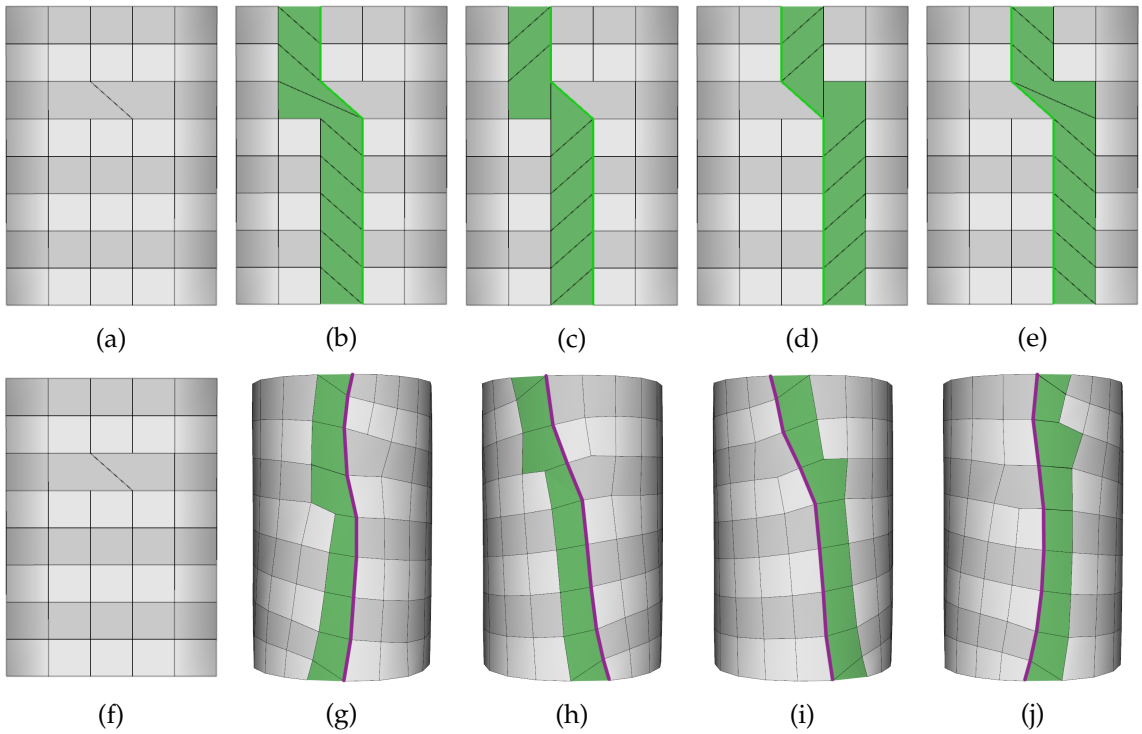


Fig. 4.3: Shift-path and the shift operation on a tubular stitch mesh: (a) full shift-path, (b-e) four different shift operation options that can be selected by the user, shift left-up, shift left-down, shift right-up, and shift right-down, (f) an alternative shift-path that partially covers the tube, and (g-j) stitch meshes for the shift-path in (b-e) after mesh-based relaxation.

automatically converted to the helix progression via a *shift operation* that alters the course edges on one side of the shift-path by sliding them along the shift-path (similar to prior helix creation methods [10]). These course edges can "shift" along the shift-path either up or down, effectively determining the knitting (course) direction. Therefore, a shift-path can modify the stitch mesh in four different options shown in Figure 4.3. All four options produce valid knittable structures and the choice merely determines the knitting order. In our implementation, one of the options is automatically selected by default, but we allow the user to pick a different option.

Selecting a single wale edge is enough to identify an entire shift path.[1] If the chosen shift-path begins in the middle of an increase stitch and/or end in the middle of a decrease stitch (as shown in Figure 4.3f), additional shift-paths are needed for handling the rows that are not covered by the chosen shift-path.

Shift-paths can be specified automatically by repeatedly picking an arbitrary wale edge (marking an entire shift-path) until all rows are covered. However, since the shift-path choice determines the knitting order and the knitting pattern along a shift-path is altered (due to the shift operation), we leave this choice to the user.

With typical 2D knitting instructions, the beginning and ending of each row is implicitly defined. Shift-paths are indeed a simple modification to stitch meshes, but they are essential for marking the row ends within a 3D modeling framework.

Note that shift-paths are needed for tubular parts of a model. If a series of rows have open ends, knitting can be done by simply moving to the next row and *turning* after reaching one end of a row (Figure 4.4). Shift-paths can be used for such rows as well, but they are not needed to make them knittable. Thus, the original stitch meshes provide knittable structures only when all rows of a model are open and there are no tubular pieces.

4.1.2 Knittable Mismatched Directions

Typically, the wale and course directions on neighboring faces of a stitch mesh are aligned with each other. However, restricting that the knitting directions of *all* faces would match their neighbors would significantly limit the 3D shapes that can be modeled. Since

[1] If the stitch mesh contains triangular faces, multiple wale edge selections might be required for specifying the desired shift-path.

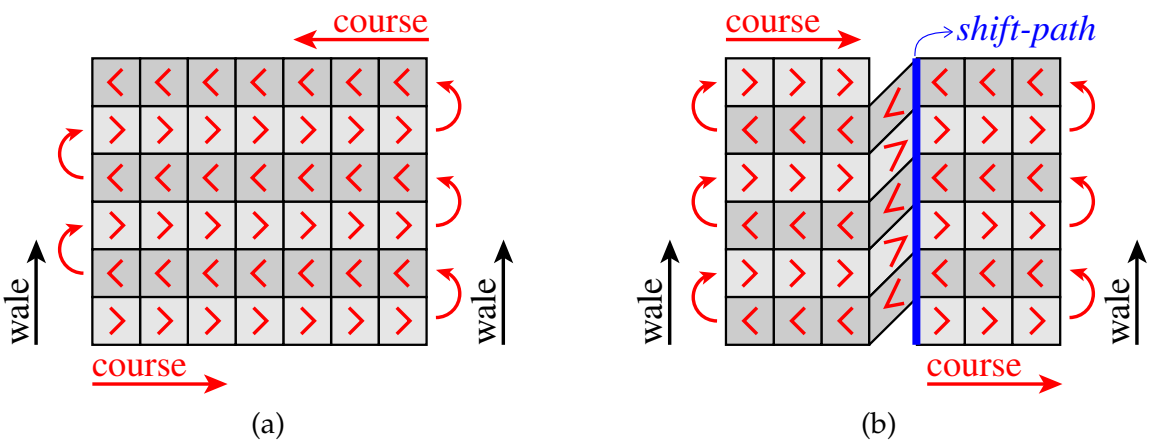


Fig. 4.4: Stitch mesh rows with open ends: (a) without a shift-path and (b) with a shift-path. The < or > symbol on a face indicates the course direction for the face.

what makes stitch mesh modeling powerful is its ability to represent complex 3D shapes, it is important to provide support for having neighboring faces with mismatched knitting directions.

We support four types of mismatched directions shown in Figure 4.5 that cover all possible cases that include two stitch mesh faces with a common edge. The first two types are *joining* and *splitting* mismatches (Figure 4.5a-b), where two faces on consecutive rows sharing a course edge disagree on the wale direction. We also introduce two new types of perpendicular mismatched directions (Figure 4.5c-d). These take place when two neighboring faces disagree on the type of the shared edge, which is treated as a wale edge for one and a course edge for the other.

4.1.2.1 Joining Mismatched Directions

As shown in Figure 4.5a, joining mismatched directions happen when the wale directions of neighboring stitch mesh faces point towards the mismatched edge. Joining mismatched directions are needed for handling certain types of singularities, such as the example.

A common knitting technique for handling joining mismatched directions is called *three needle bind-off*. The knitter uses two needles to hold the two knitted pieces and a third needle to knit through two stitches (one from each needle) at a time, combined with a bind-off stitch. To represent three needle bind-off within the stitch mesh modeling framework, we extrude the mismatched edges along the inverse normal direction and

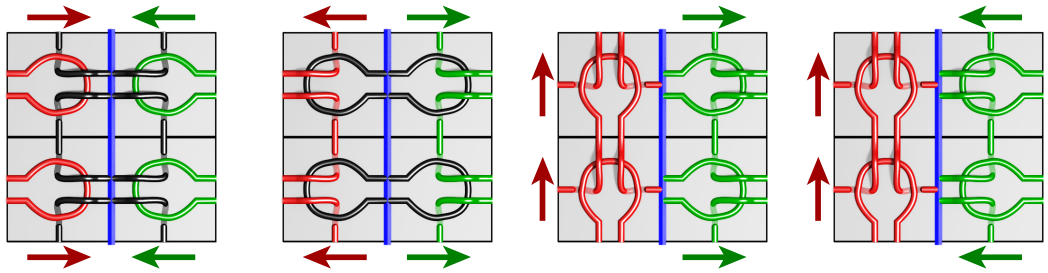


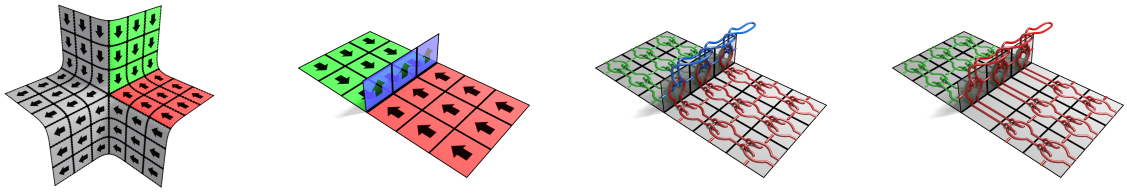
Fig. 4.5: Four types of mismatched directions, covering all possible cases that involve two stitch mesh faces with a shared edge: (a) joining, (b) splitting, (c) expanding perpendicular, and (d) contracting perpendicular. Arrows indicate the wale direction.

form faces that are (locally) perpendicular to knit surface (Figure 4.6b). The new faces act like decrease type stitches, each joining two loops connected to the faces on either side of the extruded edge. A bind-off stitch is used to terminate the knitting progression after joining (Figure 4.6c). This extrusion is performed automatically and it takes place at the very end of the stitch mesh modeling process, right before generating the yarn curves or knitting instructions.

Three needle bind-off can use a separate piece of yarn for the extruded row to join two knitted pieces on either side (Figure 4.6c). Alternatively, it is possible to use the yarn of one of the two knitted pieces instead. In this case, the loop on the second row is extended, as shown in Figure 4.6d. This alternative version not only avoids the extra piece of yarn but also avoids introducing an extra row of stitches used for joining the two pieces. In practice, both of these alternatives are acceptable solutions with slightly different final results, so we support both of them, leaving the choice to the user.

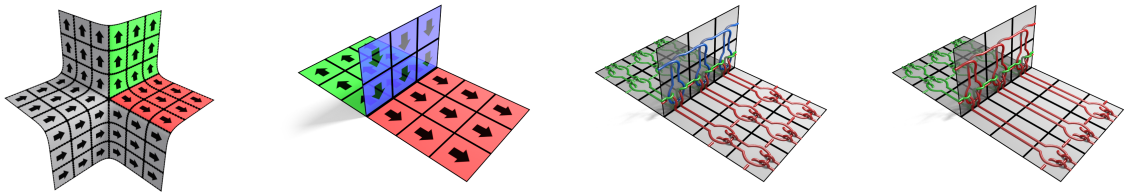
4.1.2.2 Splitting Mismatched Directions

Similar to joining mismatched directions, with splitting mismatched directions (Figure 4.5b) the wale directions of neighboring stitch mesh faces point away from the shared mismatched edges. Splitting mismatched directions appear in similar cases as joining mismatched directions, but with inverted wale directions (Figure 4.7a). While the actual knitting operations needed for handling splitting mismatched directions are different (i.e., three needle bind-off cannot be used), we can handle them similarly in the context of stitch mesh modeling. As in joining mismatched directions, we extrude the mismatched edges



(a) (b) (c) (d)

Fig. 4.6: Joining mismatch directions: (a) mismatched edges on one side of a valance 6 vertex between green and red faces, (b) the extruded row of faces (shown in blue) with arrows showing the wale direction, (c) yarn curves using an extra piece of yarn (shown in blue) along the extruded row, and (d) yarn curves using the yarn piece on one side of the mismatched edges.



(a) (b) (c) (d)

Fig. 4.7: Splitting mismatch directions: (a) mismatched edges on one side of a valance 6 vertex between green and red faces, (b) the extruded rows of faces (shown in blue) with arrows showing the wale direction, (c) yarn curves using an extra piece of yarn (shown in blue) along the extruded rows, and (d) yarn curves using the yarn piece on one side of the mismatched edges. The arrows on the stitch mesh faces indicate the wale knitting direction.

along the inverse normal direction to introduce additional faces. However, this time we extrude two rows, as in Figure 4.7b. Then, we can use an extra piece of yarn for producing cast-on stitches (Figure 4.7c). The stitches on both sides of the mismatched edges are moved to the extruded row, and they are pulled through the same loops formed by the cast-on stitches. This operation effectively connects the two rows on both sides of the mismatched edges. Alternatively, one of the existing yarn pieces can be used to form the cast-on stitches (Figure 4.7d). This effectively replaces the stitches on one side with the cast-on stitches along the extruded rows and moves the stitches on the neighboring row to the first extruded row.

4.1.2.3 Expanding Perpendicular Mismatched Directions

It is also possible to connect two neighboring stitches that disagree on the type of the common edge between them. As shown in Figure 4.5c, expanding perpendicular

mismatched directions happen when the wale direction on one side of the mismatched edge is pointing away from the edge. Figure 4.8 shows two possible ways of handling such cases. The first one (Figure 4.8a) is done by pulling perpendicular loops though previously knitted loops, and the second one (Figure 4.8b) corresponds to increase stitches that are placed on a separate needle. Nonetheless, these two stitch types are merely two example ways of handling expanding perpendicular mismatched directions and one could imagine other stitch types that could serve the same purpose.

Note that two stitches on consecutive rows on one side of the mismatched edges are paired with slightly different yarn-level geometry. Yet, this does not require the number of stitches along the mismatched edge to be even. When there is an odd number of stitches, one end of the yarn is simply tied, forming a knot.

4.1.2.4 Contracting Perpendicular Mismatched Directions

The last possibility, shown in Figure 4.5d, appears when the wale direction on one side of a mismatched edge is towards the edge. In this case, we simply handle the stitch on the other side as a decrease type of stitch with a minor modification that effectively treats the mismatched wale edge as a course edge (Figure 4.8c). In spite of the complexity of the yarn geometry, this type of mismatched directions are relatively easy to handle during knitting by simply moving stitches from one needle to another and knitting them together to form a decrease stitch.

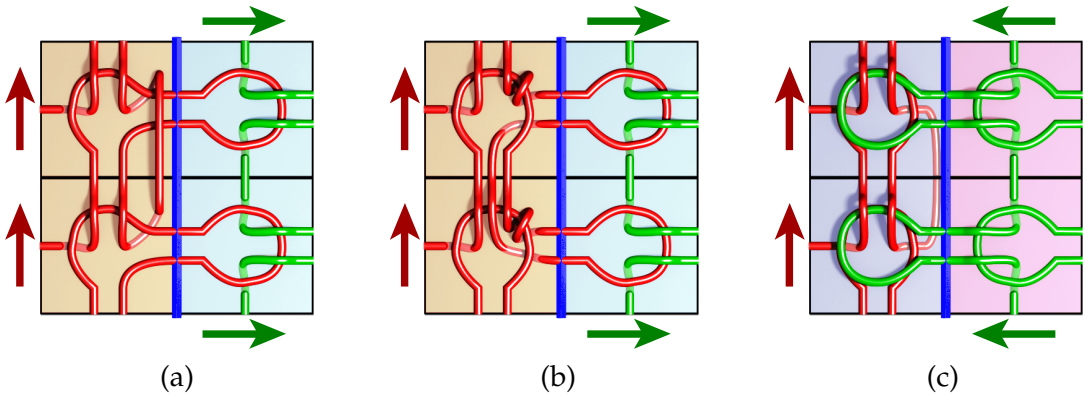
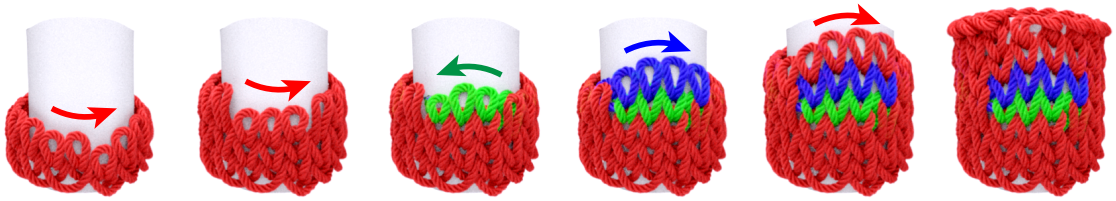


Fig. 4.8: Three types of perpendicular mismatches: (a) expanding type 1, (b) expanding type 2, and (c) contracting. Expanding and contracting perpendicular mismatched directions and the corresponding yarn curves generated from the stitch mesh. The mismatched edges are shown as blue lines. Arrows indicate the wale direction.

4.1.3 Short-Rows

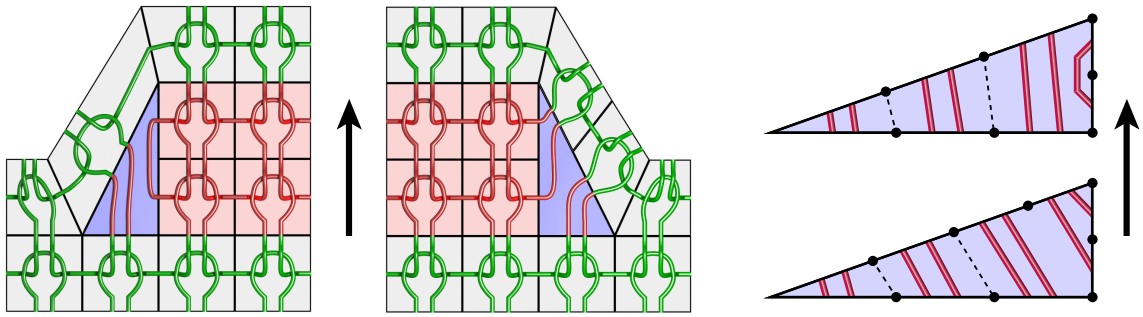
Short-rows are essential for shaping knit structures. Indeed, short-rows are so commonly used in knitting that even some simpler design tools for knitting support them [101], but they are not supported by the original stitch meshes. A regular knit row is formed by knitting through all of the stitches on a needle from start to end (Figure 4.9a). A short-row, however, uses only a subset of the stitches on a needle. This is accomplished by stopping at a certain stitch before reaching the end of the row (Figure 4.9b) and starting to knit in the opposite (course) direction (Figure 4.9c). The stitches knit in the opposite (course) direction form a short-row. After a desired number of stitches are knit, another turn, changes the knitting (course) direction back to its original. After the same number of stitches are knit (Figure 4.9d), two short-rows are completed. At this point, knitting can continue towards the end of the row (Figure 4.9e-f) or additional pairs of short-rows can be added. Thus, short-rows typically appear in pairs. Short-rows cause the knit surface to bend by adding partial rows. A typical example of short-rows would be the heels of knit socks that force the straight tubular shape of a sock pattern to bend and make room for the heel.

For introducing short-rows to the stitch mesh modeling framework, we introduce a new type of stitch mesh face that we call a *short-row face*. Unlike typical stitch mesh faces that have two wale edges separated by one or more course edges on either side, a short-row face places the two wale edges together on a straight line, forming a triangular shape with four (or more) edges, as shown in Figure 4.10a. Thus, a short-row face makes room for a



(a) (b) (c) (d) (e) (f)

Fig. 4.9: Knitting short-rows: (a) regular rows knit through all stitches on the previous row, (b) short-rows begin with stopping knitting before reaching the end of a row, then (c) reversing the course direction and knitting stitches in the opposite direction, afterwards (d) knitting direction is reversed again to finish a pair of short-rows, and (e-f) knitting continues in the original direction. Arrows indicate the course direction used for knitting the stitches below them.



(a) (b) (c)

Fig. 4.10: Short-Rows: marked as red faces (a) beginning with a short-row face marked as the blue face and (b) ending with another short-row face. A short-row face can have four or five edges with different yarn-level connections. We also support (c) short-row faces with more than five edges that simply connect the yarn pieces of corresponding edges. Arrows indicate the wale knitting direction.

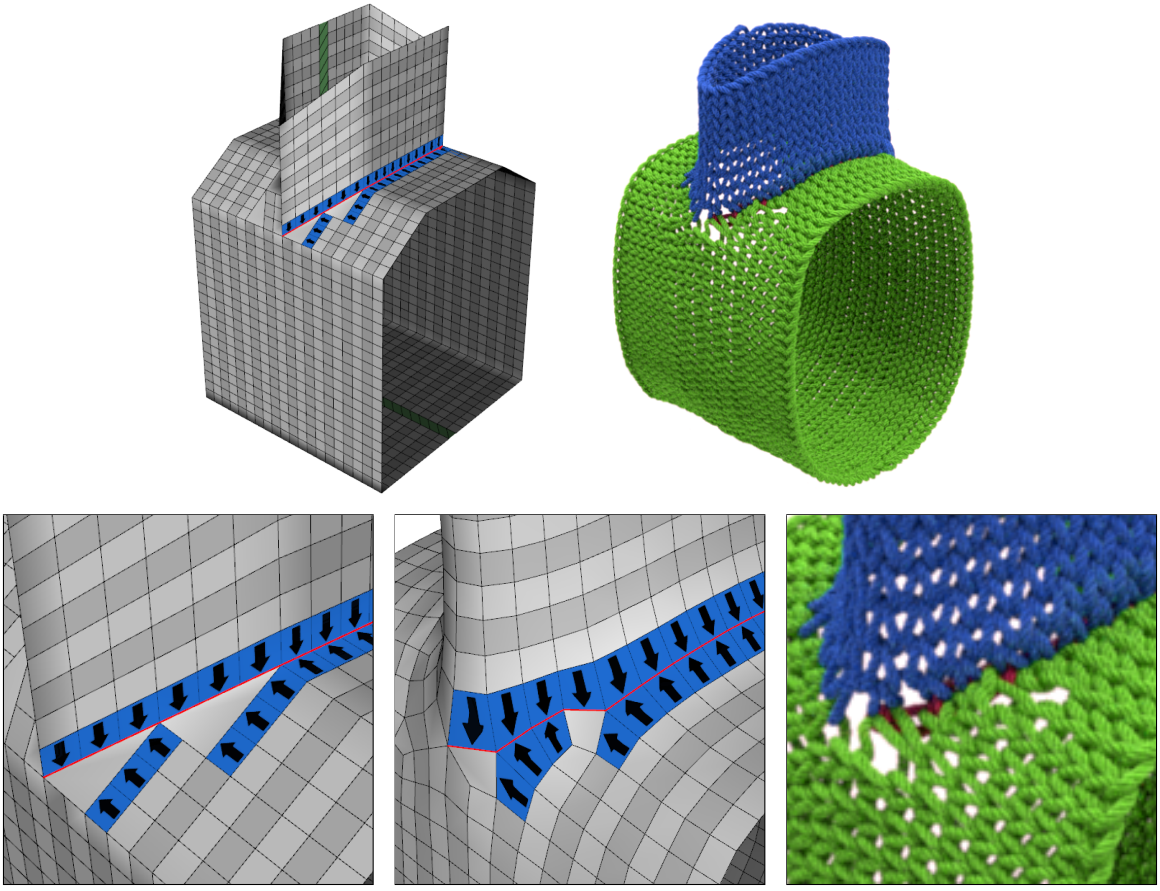
pair of short rows on one side. Therefore, it is often paired with another short-row face marking the other ends of these short-rows (Figure 4.10b).

We introduce two different types of short-row faces that correspond to two different techniques for knitting short-rows. The first type contains four edges and the corresponding yarn-level model has the form in Figure 4.10a. The other type includes five edges and forms yarn-level geometries in Figure 4.10b. This second type forms more stitches near the ends of the short-rows, thereby avoiding large holes near short-row ends.

The fact that a short-row face spans two rows on one side causes the neighboring stitches to deform significantly. Some amount of deformation is expected near short-row ends, and they are minimized after mesh-based relaxation. However, prior to mesh-based relaxation, during stitch mesh editing, the extreme deformation around short-row faces provides a poor representation of the knit model. Therefore, we allow short-row faces to have more than five edges, where the corresponding yarn pieces for the extra edges are directly connected, as shown in Figure 4.10c. During mesh-based relaxation, we connect the corresponding vertices of the short-row faces with zero-length springs that effectively collapse these extra segments, as can be seen in Figure 4.11.

4.2 Modeling Framework

In our knittable stitch meshes, we follow the original stitch mesh modeling framework [164]. Yet, some modifications are required to accommodate the changes needed



Stitch Mesh Relaxed Stitch Mesh Yarn-level Model

Fig. 4.11: An example containing short-row faces with more than five edges near joining mismatched directions, before and after mesh-based relaxation, and the final yarn-level model. Arrows on the stitch mesh faces indicate the wale knitting direction.

for handling knittable stitch meshes.

These modifications begin with the labeling process, which is the first step in stitch mesh modeling that allows the user to specify the knitting directions on the input mesh faces. Labeling is typically done by starting from one open end of the model. Simply clicking on an edge along an open end automatically selects the neighboring face without a label and labels an entire row of faces. However, the input for knittable stitch meshes can contain triangles that would eventually turn into ends of short-rows, so simply selecting an edge does not always provide enough information for identifying a short-row. To accommodate this, we allow the user to mark triangular faces of the input mesh as short-row ends. Yet, depending on the order of labeling, short-row ends can be detected automatically, when all other faces surrounding a triangle are already labeled.

Labeling perpendicular mismatched directions also requires some minor changes to the labeling process. It is possible to automatically detect perpendicular mismatched directions when one side of the mismatched edge can be labeled without considering the mismatched directions, which is typically the case. When this is not possible, mismatched edges can be marked by the user explicitly or the labeling process can be done one face at a time, instead of automatically labeling an entire row of faces.

Knittable stitch meshes support the same low-level and high-level stitch mesh editing operations of the original stitch meshes, so no modification is needed for these operations. However, editing edges along a shift-path may invalidate the shift-path, which must be monitored during stitch mesh editing.

The mesh-based relaxation step also requires some minor modifications. First of all, the stretch and shear forces are computed considering the modified mesh after the shift operation. This can be done on-the-fly without actually modifying the topology of the mesh, so that the user can change the location of the shift-path after mesh-based relaxation (followed by another mesh-based relaxation to get the final stitch mesh shape).

Second, mismatched edges along perpendicular mismatched directions use the average rest length of wale edges and course edges, since they are treated as course edges for one of their faces and wale edges for the other.

Third, for handling short-row faces that contain more than five edges, we introduce zero-length springs, so that pairs of these short-row faces collapse after mesh-based relaxation, as shown in Figure 4.11.

Finally, joining and splitting mismatched directions, which are handled by extruding the mismatched edges perpendicularly, must consider the impact of the extruded rows. Since these edges are extruded right before generating yarn curves, these new faces do not exist during mesh-based relaxation. Thus, we modify the rest lengths of neighboring wale edges. If a joining mismatched direction would use an extra piece of yarn, the rest lengths of the wale edges that are connected to the mismatched course edges are extended (by a factor of 1.5 in our implementation) to make room for the extra row of stitches (Figure 4.12a). If the yarn on one side would be used for the extruded faces (Figure 4.12b), the rest lengths of the wale edges on that side are reduced (by a factor of 0.5 in our implementation). In case of splitting mismatched directions with an extra piece of yarn

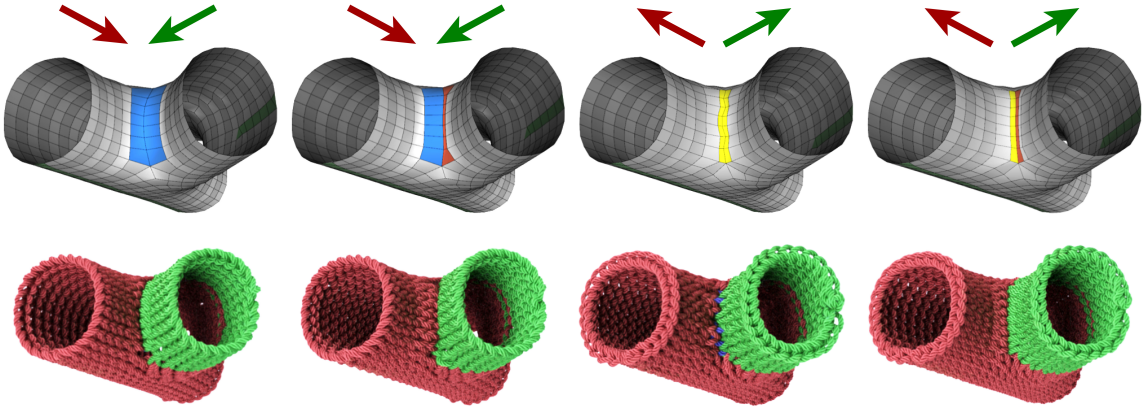


Fig. 4.12: Joining and splitting mismatched directions after mesh-based relaxation and after yarn-level relaxation: (a) joining mismatched directions using an extra (blue) yarn piece, (b) joining mismatched directions using the green yarn piece, (c) splitting mismatched directions using an extra (blue) yarn piece, and (d) splitting mismatched directions using the green yarn piece. Arrows indicate the wale knitting direction.

(Figure 4.12c), the rest lengths of the wale edges on both sides of the mismatched edges are reduced (by a factor of 0.5 in our implementation). If one of the yarn pieces are used for the cast-on stitches along the extruded row (Figure 4.12d), the rest lengths of the wale edges on the second row on that side are also reduced (by a factor of 0.5 in our implementation). Note that the purpose of the mesh-based relaxation is to provide a rough approximation of the relaxed shape, and the final model is produced by yarn-level relaxation after the yarn curves are generated. Therefore, the scaling factors we use for the mesh-based relaxation do not have to be precise.

4.3 Generating Knitting Instructions

Knittable stitch meshes provide a novel mechanism for representing and modeling complex 3D knit structures. However, even though a knittable stitch mesh includes all information needed for precisely describing a knit structure, determining the sequence of knitting instructions needed for fabricating the represented knit structure can be highly complicated. Therefore, we describe a scheduling algorithm that converts the information in a knittable stitch mesh into step-by-step knitting instructions. This process also crucial for verifying that the models we produce are indeed knittable.

4.3.1 Dependency and Knittability

Knitting begins with *casting on*, a term that refers to placing the first stitches onto a needle. Only then can a second needle be used to pull the next stitch through an existing one, an operation that transfers the original loop from the first needle onto a new loop that now exists on the second needle. Therefore, there is a strict order dependency in knitting. A knitting project is completed and removed from the needles by *binding off*. The *bind-off* stitches transfer loops on the needle to a row neighboring stitch, closing all the open loops except for one. A yarn end is pulled through this last stitch (as well as the first stitch) to prevent unraveling.

In a knittable stitch mesh, the knitting order dependency is simple: each stitch depends on a neighboring stitch on the same row in the inverse course direction, and it depends on all neighboring stitches on the previous row (in the inverse wale direction). Extruded stitches along joining mismatched directions depend on stitches on both sides of the mismatched edges. Stitches on one side of expanding perpendicular mismatched directions that treat the mismatched edges as course edges depend on the stitches on the other side. Similarly, stitches on one side of contracting perpendicular mismatched directions that treat the mismatched edges as wale edges depend on the stitch on the other side.

Note that the knittable stitch mesh structure is general enough to represent unknittable models as well. Yet, given an arbitrary knittable stitch mesh, it is easy to verify that the represented model is knittable. Consider the dual graph of the knittable stitch mesh, which replaces the stitch mesh faces with graph nodes and connects the neighboring nodes with directed edges that follow one of the two knitting directions (course or wale). If this dual graph has a cycle, the model is not knittable, since a cycle would mean circular dependency. If the dual graph has no cycle, we can conclude that the model is knittable.

Nonetheless, our knittable stitch mesh modeling framework produces knittable models, so this verification is not needed in practice. This is because the initial labeling step follows the dependency order. There is only one exception that can lead to a unknittable model, caused by the flexibility we provide on handling mismatched directions. Depending on the shift paths, a model may beunknittable if it meets *both* of the two following conditions:

1. If there are two or more *separate* groups of joining or splitting mismatched directions

on the same row.

2. If extra pieces of yarn are not used for handling the mismatched directions, and one yarn is used for handling one group of mismatched directions while the *other* yarn is used for handling another group.

A simple example of this special case is shown in Figure 4.13, and it is unknittable, because the specific way that the mismatched directions are handled in this case can cause circular dependency. Yet, it is easy to avoid this special case. If there are multiple separate groups of joining or splitting mismatched directions on the same row, we can only permit using extra yarn pieces or the yarn on one side of the mismatched edges for handling *all* mismatched directions on the same row. Except for this special case, which is related to the extra flexibility we provide to the user for selecting which yarn piece should be used for handling joining/splitting mismatched directions, input meshes that can be labeled *always* produce knittable models.

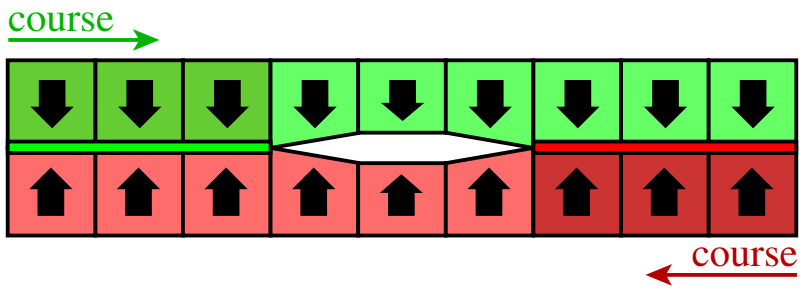


Fig. 4.13: An example of the special unknittable case, which includes two groups of joining mismatched directions. The dark faces depend on the stitches on the opposing row. One piece of yarn is used for knitting the top row, and another yarn is used for knitting the bottom row. The mismatched edges are colored according to the yarn pieces that are to be used for knitting the stitches along the mismatched directions. The first group of joining mismatched directions (on the left side) use the yarn from the top row, and the second group (on the right side) uses the yarn from the bottom part. This case leads to deadlock due to circular dependency, since the red stitches must be knit before knitting the stitches for the first group of mismatched directions, and the green stitches must be knit before knitting the stitches for the second group. Slightly modifying this model by either using the same yarn for both groups of mismatched directions or using additional pieces of yarn would avoid the deadlock case.

4.3.2 Identifying Separate Yarn Pieces

Before we start generating knitting instructions, we must identify how many yarn pieces are needed for knitting the model and which yarn piece should be used for knitting which stitch. Obviously, the stitches on the same row are knit using the same yarn piece. Stitches on consecutive rows can share the same yarn piece as well, depending on the placement of the shift-paths and the row types. A knittable stitch mesh can have three types of rows:

- Closed-rows that form complete loops with no ends,
- Open-rows that begin and end on either a border (marked as a wale edge) or a perpendicular mismatched direction boundary, and
- Short-rows that are placed between other rows.

Note that one end of a pair of short-rows can be on a border or a perpendicular mismatched direction boundary.

Closed-rows form tubular pieces and neighboring closed-rows are connected to each other via shift-paths. Therefore, closed-rows along the same shift-path can be knit using the same yarn piece. Consecutive open-rows that share wale edge borders can be knit using the same yarn piece without needing a shift-path. Short-rows that end on a border can be handled similarly. Short-rows between closed-rows either connected to other rows via shift-paths passing through them or they are knit using separate yarn pieces.

We can find the number of yarn pieces needed by counting the number of separate shift-paths, separate groups of open-rows, short-rows that are not connected to other rows, and joining/splitting mismatched directions that use extra yarn pieces. Thus, stitch mesh faces within the same group of rows that share a yarn piece are assigned the same yarn index.

Then, we determine the first stitch for each yarn piece. For a group of closed-rows, the first stitch corresponds to the stitch mesh face on one side of the first edge along its shift-path (the side is determined by the shift direction). The first stitch of a group of open-rows that are not connected by a shift-path would be a stitch on either end of the first row (the rows are ordered based on the wale direction). Pairs of short-rows that are not connected to other rows can be knit starting from either end of the first row. Separate yarn

pieces for joining or splitting mismatched directions can also be knit starting from either end.

4.3.3 Step-by-Step Knitting Instructions

Algorithm 1 shows how a knittable stitch mesh can be used for generating step-by-step knitting instructions. We begin with placing the first stitch of each yarn piece in a queue. Then, we pick a stitch that is ready to be knit from the queue. A stitch is considered *ready* after all other stitches that it depends on are knitted. Initially, only cast-on stitches can be considered ready, since they are not connected to a stitch on a previous row. Knitting can begin with any cast-on stitch in the queue.

We use the stitch type of the stitch mesh face to produce the knitting instructions. The knitting instructions for a single stitch mesh face can be as simple as a single *knit* or *purl* instruction or it might consist of a series of instructions (as with increases and decreases) and it might include complex instructions like combining previously knit stitches from other needles.

ALGORITHM 1: Generate Knitting Instructions

```
Q ← list of first stitches for each yarn piece
while Q is not empty do
    s ← the next ready stitch from Q
    while true do
        Generate knitting instructions for s.
        Mark s as knitted.
        s_c ← the next stitch after s along the course direction
        if s_c exists then
            s ← s_c
        else
            s_w ← the stitch after s on the next row in wale direction
            if s_w exists and yarn of s_w = yarn of s then
                s ← s_c
            else
                Terminate the yarn piece
                break
        if s is not ready then
            Enqueue s into Q.
            break
```

We mark a stitch as *knitted* after it is knit and move to the next stitch along the course direction. If there is no next stitch in the course direction, we check the next stitch in the wale direction. If there is no next stitch in the wale direction or if the next stitch in the wale direction belongs to a different yarn piece, it means that we have completed knitting all stitches of the current yarn piece. After we move to a next stitch, we check if it is ready to be knit. If it is ready, we repeat the same process; otherwise, we place it into the queue.

Note that knitting can begin and continue with any stitch in the queue that is ready. Therefore, for complex models, the implementation of the queue determines which parts of the model are knit first. In our implementation, we used a stack with First-In-Last-Out order. When a stitch s is enqueued, we check its first dependent stitch t, and we move the stitch in the queue that uses the same yarn piece as t to the top of the queue. This effectively prioritizes knitting one part of a model before starting to knit the other parts, attempting to reduce the number of needles needed during knitting.

4.4 Implementation and Results

We have tested our knittable stitch mesh modeling framework in two fronts: (1) generating complex yarn-level models containing short-rows and various forms of mismatched directions, and (2) fabricating models via assisted hand knitting.

4.4.1 Graphics Interface for Hand Knitting

We have developed a graphics interface that displays the step-by-step knitting instructions to a knitter, shown in Figure 4.14. At every step our interface instructs the user to perform a series of knitting instructions. For simplification, short sets of knitting instructions that are consecutively repeated are grouped together. In these cases, the interface presents how many times the given knitting instructions should be repeated. We have found this approach favorable to providing a single knitting instruction at a time. A 3D viewport shows the previously knitted part of the model and the part that corresponds to the current set of knitting instructions. We use the real-time fiber-level cloth rendering method of Wu and Yuksel [158, 159] to display the yarn-level model.

One difficulty with hand knitting is that it is easy to lose count. A simple solution that is used in hand knitting is placing markers between stitches on needles. Therefore, our

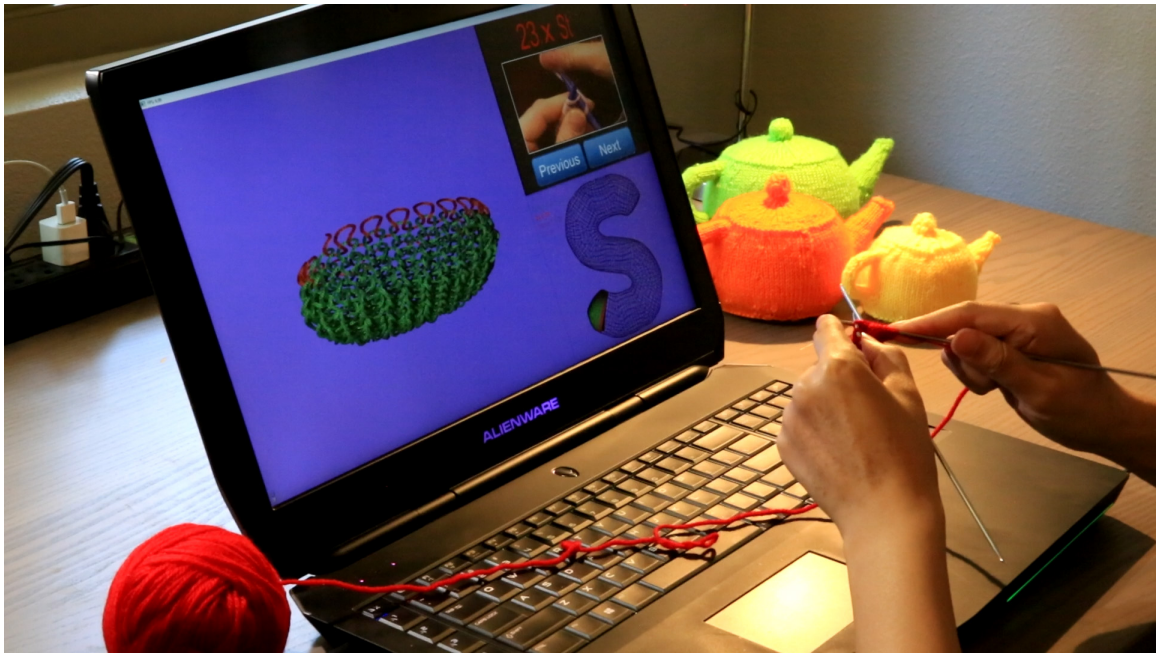


Fig. 4.14: Our knitting interface: On the top-right corner the knitting instruction code is displayed along with how many times the instruction should be repeated. Below the instruction code a short video-clip show how to perform the instruction. At the bottom-right side the entire model is displayed, along with the previously knitted part shaded in green and the stitches that correspond to the current instructions shaded in red. The main view provides a yarn-level rendering of the part of the model that is previously knit and the part that is currently being knit.

knitting interface includes markers as well. Our system instructs a user to place a marker when a shift path is reached, right before the first stitch of a new row. The user is also instructed to place markers before and after a group of mismatched directions and the beginning and ending of short-rows. Our system also keeps track of these markers and provides knitting instructions using them. For example, instead of instructing the user to perform a set of instructions a certain number of times, we can instruct the user to perform the instructions until a marker is reached. We have found that this simple feature makes it much easier to follow the knitting instructions in practice. We also permit users to place markers after any groups of knitting instructions. After a marker is placed by the user, upcoming knitting instructions can use that marker.

4.4.2 Knitted Models

An example teapot model prepared using our system is shown in Figure 4.1. The input mesh model (Figure 4.1a) is converted to a knittable stitch mesh (Figure 4.1b) with three

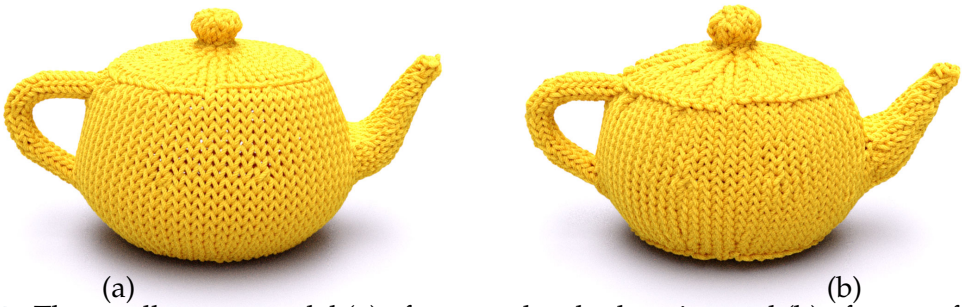
shift paths: one group of joining, two groups of splitting, and six groups of perpendicular mismatched directions, and multiple short-rows. The simulated model (Figure 4.1c) is produced via yarn-level relaxation after generating the yarn curves from the knittable stitch mesh. The knitted model (Figure 4.1d) is fabricated via hand knitting following the step-by-step knitting instructions provided by our knitting interface.

Figure 4.15 shows three different teapot models generated from the same input mesh with different levels of tessellations used for generating the knittable stitch meshes. Notice that higher levels of tessellations lead to models with more stitches that can represent more details, but take longer to knit. Note that knittable stitch meshes can be designed using any modeling operations of stitch meshes (Figure 4.16). The simulated models (Figure 4.17a) are constrained to take the shape of the stitch mesh model. We have noticed that removing the shaping constraints and using one frame of yarn-level simulation (Figure 4.17b) produces shapes with similar features to the hand knitted models that are stuffed with cotton.

We test our approach for handling different mismatched directions using simple models in Figures 4.18 and 4.19. The model in Figure 4.18 is knitted using two long yarn pieces

Fig. 4.15: Knitted teapots with different numbers of stitches using different knittable stitch meshes generated from the same input mesh in Figure 4.1. They are all knitted using six separate yarn pieces, and they contain 6.3K, 4.4K, and 2.6K stitches from left to right.

Fig. 4.16: Teapot models with different stitch mesh patterns.



(a) (b)

Fig. 4.17: The small teapot model (a) after yarn-level relaxation and (b) after one frame of simulation with gravity.

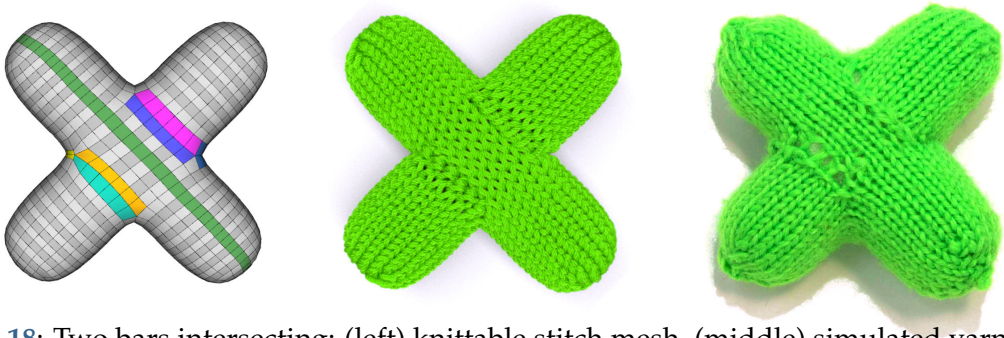


Fig. 4.18: Two bars intersecting: (left) knittable stitch mesh, (middle) simulated yarn-level model, and (right) final knitted model.

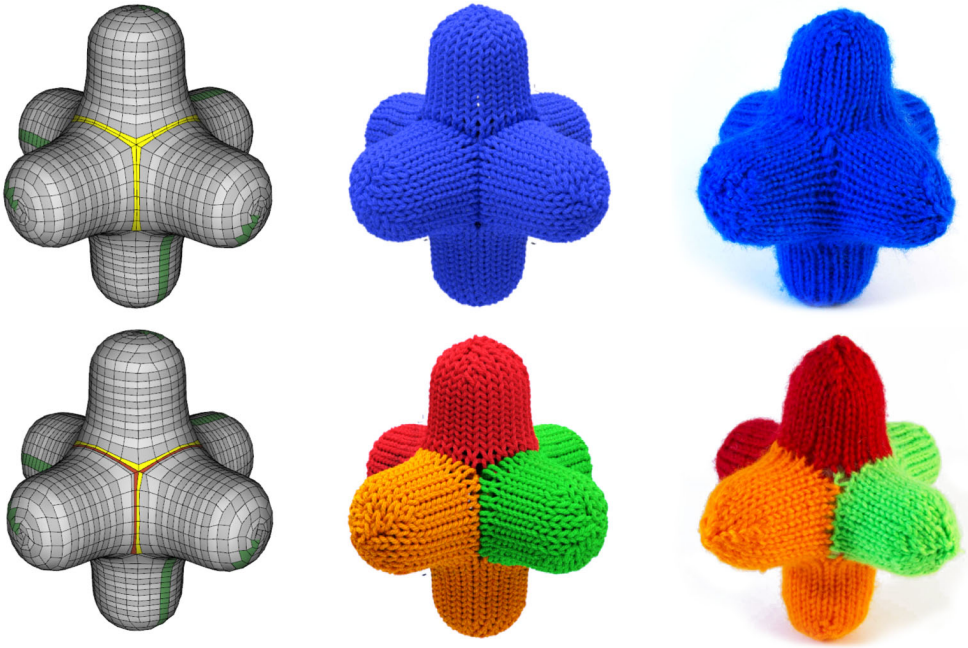


Fig. 4.19: Three bars intersecting: (left) knittable stitch meshes, (middle) simulated yarn-level models, and (right) final knitted models.

along with two shorter yarn pieces used for handling the joining and splitting mismatched directions. Expanding and contracting perpendicular directions appear along the flat face of the model. The model contains 1.5K stitches, and it is knitted using four yarn pieces, two of which are used for handling joining and splitting mismatched directions. The model in Figure 4.19 shows two different ways of handling joining and splitting mismatched directions. It shows joining and splitting mismatched directions handled (top row) using four extra yarns pieces, two for cast-on stitches along splitting mismatched directions, and two for bind of stitches along joining mismatched directions, in addition to the three yarn pieces needed for knitting the rest of the model, and (bottom row) using one of the yarn pieces near the mismatched edges, requiring only three yarn pieces for knitting the entire model. Both models contain 2.1K stitches.

For testing more challenging cases of topological connections and short-rows, we have modeled and knitted some example letters shown in Figure 4.20. Most letters demonstrate how short-rows (red faces) can be used for producing curved shapes; "R" and "A" include sharp corners that can be generated using short-rows, and "R," "P," and "H" show the importance of handling perpendicular mismatched directions (yellow-green and blue-purple face pairs) for knitting complex shapes. While these letters are relatively small knitting projects, as compared to a larger model like a full-size sweater, they are excellent examples of complex shapes and topologies. In that respect, they are more complex cases than typical garment models.

4.4.3 Performance

By far the slowest part of our pipeline is hand knitting. It can take hours or days to knit a model, depending on the experience of the knitter and the number of stitches needed to complete the model. The second slowest component is yarn-level relaxation that is used for generating the simulated models, which can also take hours, depending on the complexity of the model (see the performance results of Yuksel et al. [164] for yarn-level relaxation times of different models). The rest of the operations we use can be handled interactively, except for the mesh-based relaxation that take several seconds to about a minute.

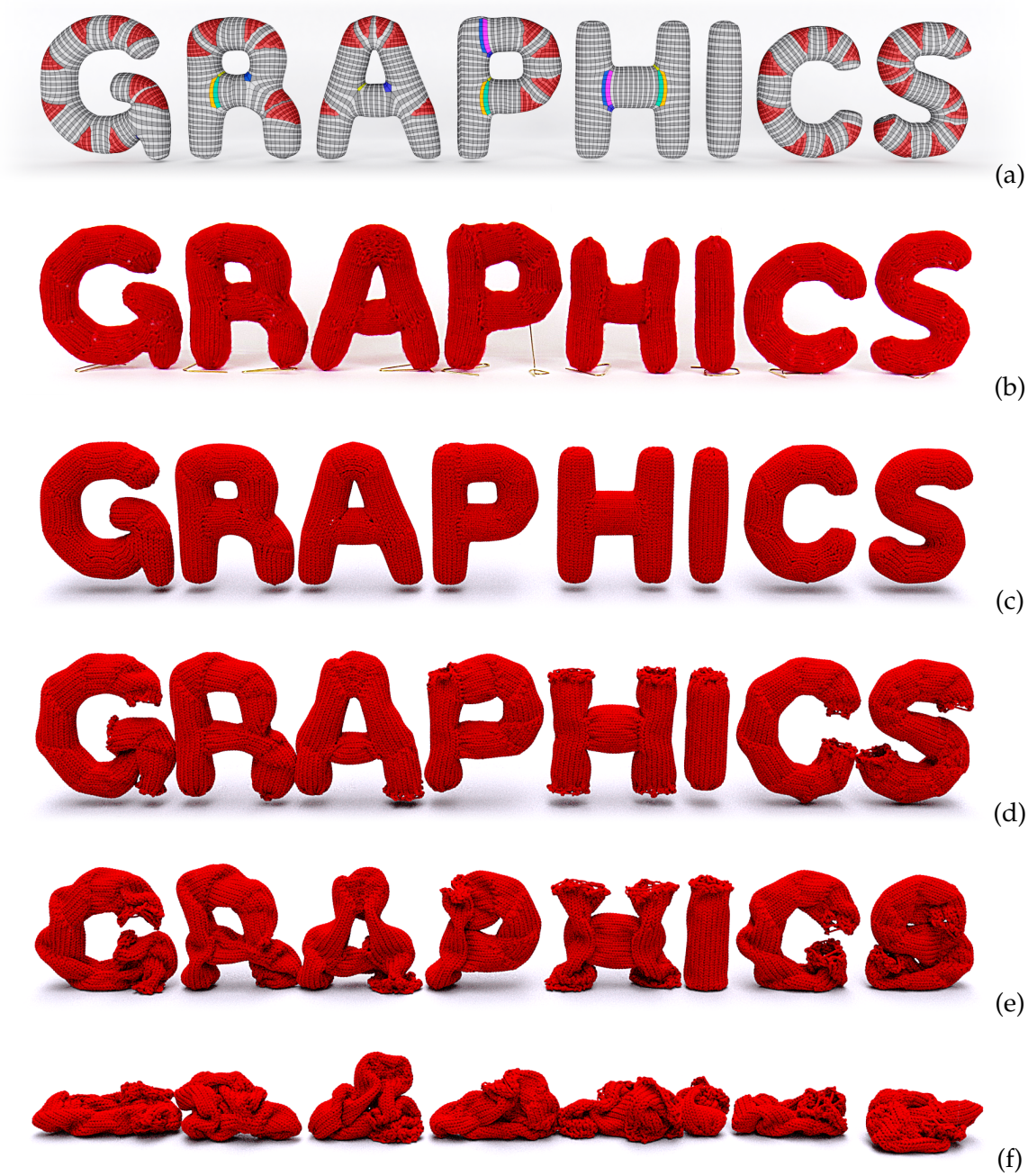


Fig. 4.20: Example letters: (a) knittable stitch mesh models, (b) knitted models, (c) simulated models, and (d-f) yarn-level simulation results with gravity after removing the cast-on and bind-off stitches. Each letter model contains between 2.5K and 2.9K stitches, except for "I," which has only 1K stitches. "G" is knitted using 3 yarn pieces; "R," "A," and "P" need 4 yarn pieces; "H" contains 6 separate yarn pieces for handling the joining and splitting mismatched directions, and "I," "C," and "I" are knitted using a single piece of yarn.

4.5 Conclusion

We have presented knittable stitch meshes that extend the powerful concept of stitch meshes into models that can be fabricated via knitting. By introducing shift paths and properly handling mismatched knitting directions, we can convert any stitch mesh into a knittable structure. We have also introduced novel representations for handling shaping techniques that allow designing knit structures with unprecedented complexity. Finally, we have presented an algorithm that generates step-by-step instructions from a given knittable stitch mesh. We have shown a variety of example models with complex topologies to demonstrate the effectiveness of our approach.

Knittable stitch meshes provide a powerful structure for representing and designing knittable models. While they are general and able to incorporate a wide variety of knitting patterns and complex shapes, they cannot represent *everything* that can be knit. For example, multilayer patterns that are used for knitting multicolor designs cannot be represented using our formulation. With hand knitting, it is possible to pull a loop through any previously knitted loop, not just the loops on a previous row, but such operations cannot be represented with stitch meshes. Consequently, we cannot represent models that are knit as separate pieces and then sewn together. The knittable stitch mesh representation is limited to the stitch types that can be abstracted using stitch mesh faces.

Our modeling framework relies on the topology of the input mesh. This allows precisely designing the desired knit structure, but requires the user to have some understanding of the knitting process. Thus, our framework cannot convert any polygonal mesh into a knittable stitch mesh. In particular, modeling the input meshes for unusual models, such as the teapot model in Figure 4.1, may require many iterations to figure out how the model can be knit. However, once the user determines how to knit a model and prepares the input mesh, the knittable stitch mesh modeling framework allows for quickly designing the final model. Afterwards, the knitting instructions generated from the final model can be used by any knitter to fabricate the model. Therefore, combining our framework with an automated stitch mesh generation process for a given arbitrary 3D model would be an important direction for future research [155].

The stitch mesh modeling framework allows defining knitting instructions on a given 3D input model shape, but it does not guarantee that the final rest-shape of the knit model

would align with the shape of the input model. This is because the final shape of a model depends on the knitting instructions, and the knittable stitch mesh modeling framework provides enough flexibility to design knitting instructions that would contradict with the shape of the input model. In fact, it is a challenging problem to determine the set of knitting instructions that would produce a desired shape. Yet, a simple analysis of the excessive stretching or compression of stitch mesh faces after the mesh-based relaxation step often provides a good indication of how well the knitting instructions agree with the input model shape. Therefore, majority of the modeling iterations can be done at the stitch mesh modeling phase, without having to fabricate each iteration via knitting.

The same knitting instructions performed by different knitters often produce somewhat different results. This is because the final shape of a knit structure not only depends on the properties of the yarn and the needles used, but also the forces applied during the knitting process. Therefore, accurately predicting the final shape of a model that would be produced by a particular knitter is a challenging problem that our approach does not address.

CHAPTER 5

VISUAL KNITTING MACHINE PROGRAMMING

Computer-controlled knitting machines are powerful tools for computer-aided fabrication, and they are widely used in the garment and accessory industries. When properly programmed, they can turn yarns into soft 3D surfaces in a wide range of shapes, textures, and colors. Knitting machines create these objects by using a small vocabulary of operations, which manipulate loops on their *needle beds*, two long rows of loop storage locations. Once programmed, objects can be manufactured quickly and with minimal wasted yarn.

Knitting machine programming is notoriously challenging because shape, structure, texture, and color effects must all be created concurrently using a small set of low-level operations. These operations must be *scheduled* to limited needle bed locations on a knitting machine, and they are conventionally selected with limited visual and structural feedback.

We demonstrate the first general visual programming interface for 3D machine knitting, which tackles all these challenges. The core of our system is an *augmented* stitch mesh data structure, where each face contains both a visual representation and a specific set of low-level knitting operations. We couple this representation with an automatic mesh generator and a set of knittability-preserving editing operations to provide a system that can navigate the space of *all* tube-based 3D knitting programs. Together, these innovations result in the first general visual editor for knitting programs as shown in Figure 5.1.

Our system stands in contrast to previous systems, which either focused exclusively on machine-knittable shaping [101, 106], while neglecting color and texture customization, or on representing knit structures with no guarantees on machine knittability [156, 164].

We demonstrate the power and flexibility of our pipeline by using it to create and knit objects featuring a wide range of patterns and textures. Therefore, the main technical contributions presented in this chapter are the following:

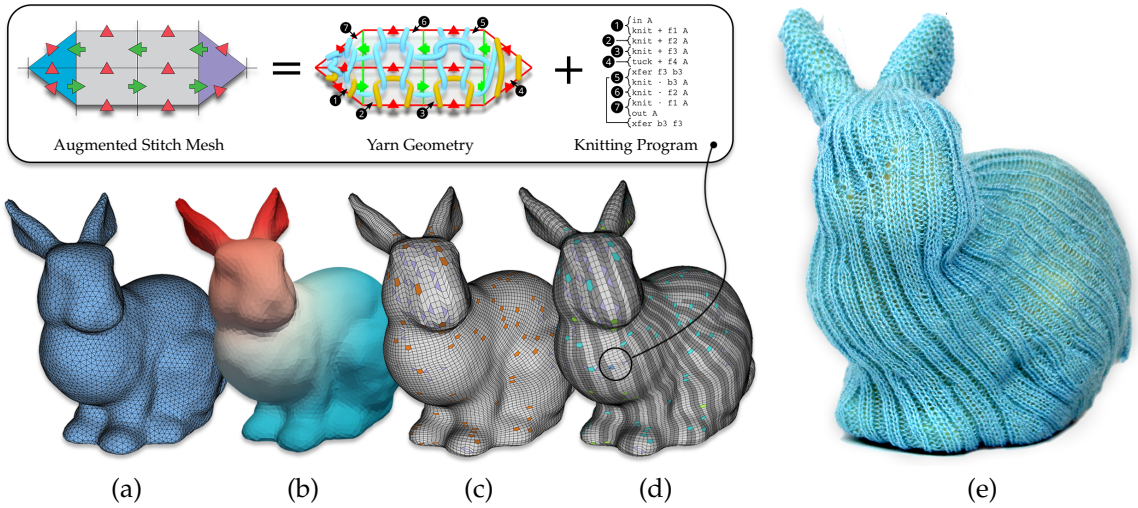


Fig. 5.1: Stages of our visual knit programming system: (a) our system begins with an input mesh; (b) generates a knitting time function; (c) remeshes the surface to create an augmented stitch mesh; (d) allows the user to interactively edit and add patterns, textures, and colorwork; and (e) generates instructions for fabrication on an industrial knitting machine. At the core of our interface is the augmented stitch mesh, which associates yarn geometry, dependency information, and a knitting program with each face.

- an *augmented* stitch mesh data structure, where each face is associated with a local machine knitting program;
- a scheduling system that assigns needle bed locations and times to stitch mesh faces in order to automatically fabricate 3D knitting patterns from augmented stitch meshes;
- and an interactive visual design system to edit knitting programs directly in 3D, while preserving the machine knittability.

We provide a brief overview of related background on machine knitting (Section 5.1), then describe the details of our augmented stitch mesh structure (Section 5.2). Next, we describe how our system allows users to generate, edit, and machine-knit augmented stitch meshes. Particularly, we demonstrate how to create a machine-knittable augmented stitch mesh from a 3D object (Section 5.3), how to edit meshes in a general and machine-knittability-preserving way (Section 5.4), and how the final mesh faces can be ordered (Section 5.5.1) and their loops can be assigned needle locations (Section 5.5.2) for machine fabrication. Finally, we present our results (Section 5.6) and discuss the limitations of our work and future directions (Section 5.7), before we conclude (Section 5.8).

5.1 Background

Industrial knitting machines are programmable systems that manipulate loops into knit structures using hundreds of *needles* arranged in two parallel *beds*. Needles receive yarn from *carriers* and can be actuated to perform one of three basic instructions:

1. `tuck D N CS` : add a loop to needle `N` using yarn carrier set `CS` in the specified direction `D`.

2. `knit D N CS` : add a loop through all the existing loops on needle `N` using yarn carriers `CS` in the specified direction `D`. All previously held loops at the location are dropped.

3. `xfer N1 N2` : move all loops on needle `N1` to target needle `N2`. Needles `N1` and `N2` must be aligned and on opposite beds.

These operations, along with others to perform machine configuration (bed alignment, stitch sizing) and yarn handling, comprise the *knitout* low-level knitting language, which we use for output in our system. Details of knitout can be found in its specification [100], while more information about knitting machines in general can be found in [129]. Even though there are only three basic instructions, they can be combined to form a dazzling array of textures, colors, and shapes [144]. Indeed, knitting machines are used to fabricate items as diverse as shoes, car upholstery, and glass-fiber reinforcement for composites.

While most arbitrary sequences of needle operations can be executed by a knitting machine, few will produce results beyond a yarn tangle. This is because operations depend on previous operations to place loops and yarn carriers in specific places in order to run smoothly. Specifically, for a knitting program to form a desired set of stitches, it must respect the wale and course edge dependencies of those stitches.

We formalize this notion of validity with two properties:

Property 5.1.1 (Order). *All stitches must be constructed in the course direction order specified by the pattern. All loops that a stitch depends upon must be constructed before it and must be available on a needle at the time of constructing the stitch.*

Property 5.1.2 (Adjacency). *All course direction adjacent stitches must be constructed on adjacent needles.*

In other words, a given sequence of stitches is machine knittable if and only if its associated knitting program can be ordered and scheduled to machine needles such that the loop(s) and yarn(s) each stitch depends on are held on adjacent needles at the time of its construction. In the next section, we introduce a data structure for knitting representation built with this important notion in mind.

5.2 Augmented Stitch Meshes

The data structure at the core of our system is the *augmented* stitch mesh (Figure 5.1), a 3D mesh in which each face is labelled with both yarn topology and machine instructions, and each edge is labelled with yarn or loop dependency information.

The stitch mesh data structure, introduced by Yuksel et al. [164], represents complex yarn topologies as compositions of a few basic face types, which can be connected as long as their edge labels match–essentially, as a set of generalized Wang tiles [149]. By selecting the appropriate vocabulary of face types, stitch meshes can be used to represent knit, knotted, and woven structures. We augment this representation in two ways in order to support the editing of machine knitting programs: first, we add directed edge labels to track dependency information; second, we associate with every face type a knitting machine program that can construct the yarn-level topology on that face.

- Directed edges. The edge labels in our augmented stitch mesh capture dependencies between faces (Figure 5.2, left) – *loop in* edges indicate that a loop is needed, while *loop out* edges indicate that a loop is produced; *yarn in* and *yarn out* give similar information about yarns. Any *in* edge may only connect to an *out* edge of the same type, and visa-versa. Notice that these directed edge labels induce a directed graph on the faces, which is useful when checking for dependency cycles.

- Face programs. Each face in our augmented stitch mesh data structure represents a fragment of a knitting program that operates on the yarns and loops provided by its *in* edges in order to produce the yarns and loops indicated by its *out* edges (Figure 5.2, right). That is, the edge labels provide a type signature – input and output loop and yarn counts – for a knitting program fragment. The basic face types provided by our system, along with pseudocode for their knitting program fragments, are shown in

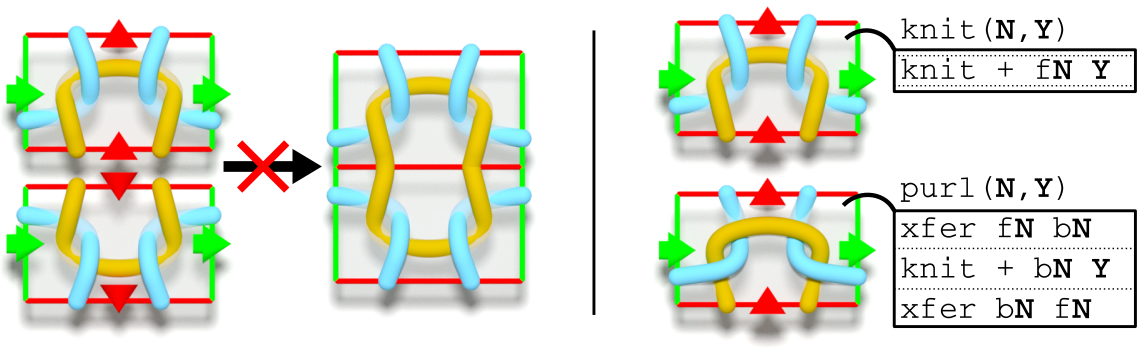


Fig. 5.2: Augmented stitch mesh faces have, *left*, directed edges to prevent locally unknittable assembly; and, *right*, associated knitting programs.

Figure 5.3. Our system makes it easy to extend this list, since the editing operations it supplies depend on face edge labels, not on the referenced program.

- Knittability. For an augmented stitch mesh to be machine-knittable, both the ordering property (Property 5.1.1) and the adjacency property (Property 5.1.2) must hold. The ordering property is easy to check – it holds locally by construction, and can be checked globally with a topological sort (Section 5.5.1). The adjacency property is somewhat more subtle, in that it depends on the existence of a valid schedule among the (exponentially-large) set of possible needle allocations. Fortunately, previous work demonstrated that the existence of a valid schedule is a property of the mesh topology along with an easy-to-check feasibility condition at splits and merges within the mesh [106]. The former can be checked once on mesh import and the latter can be preserved through UI design.

Therefore, we focus our efforts on maintaining the ordering property.

5.3 Stitch Mesh Generation

Our pipeline begins by creating a machine-knittable augmented stitch mesh from a given oriented manifold 3D surface, as illustrated in Figure 5.4. As a first step, the mesh is segmented into tubular regions using either a user-specified time function (as per [106]) as an input to the system or the Fiedler vector of the mesh Laplacian [166]. The user may also edit boundaries for better alignment (as in [67]). Then, boundaries are discretized based on stitch width such that segments align one-to-one. Once the starting and ending boundary

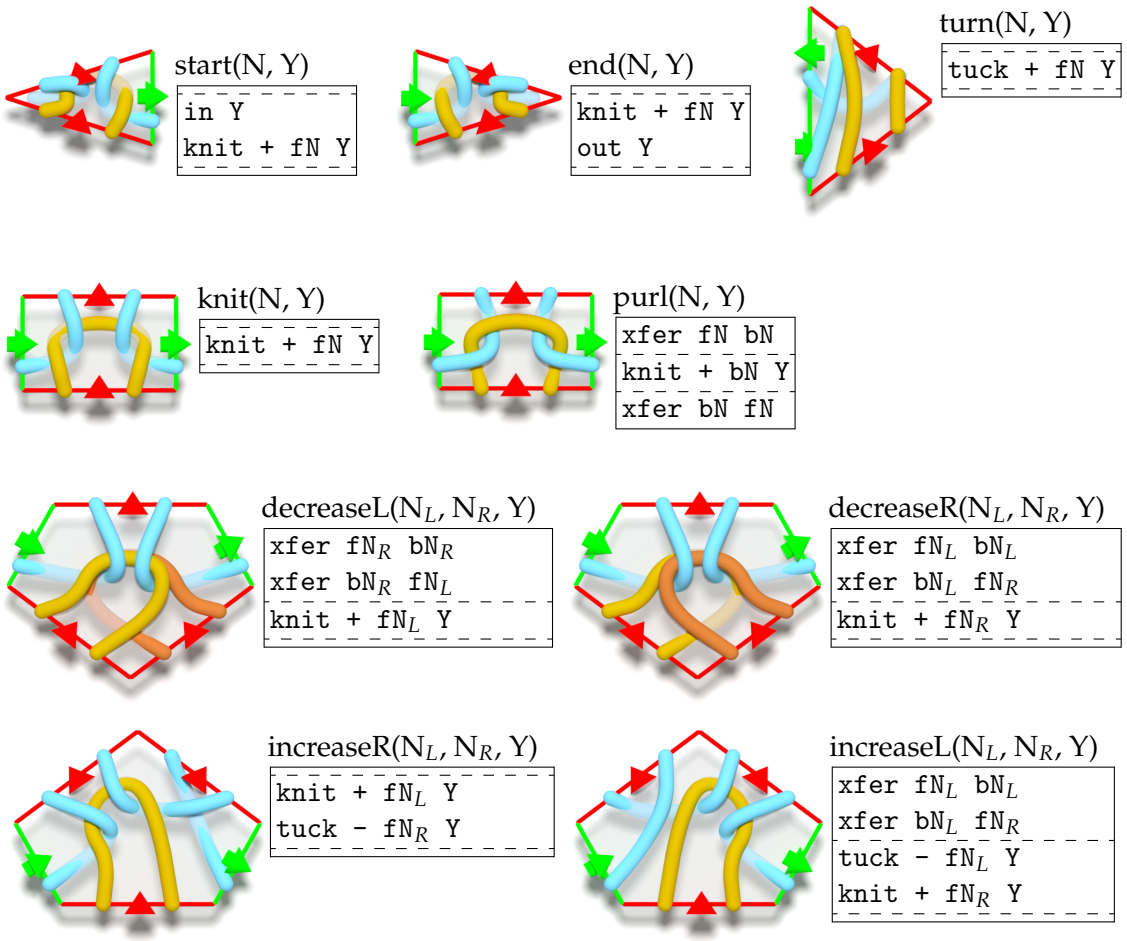


Fig. 5.3: Basic face types and their associated knitting code fragments. Faces with opposite yarn direction proceed similarly. Dashed lines indicate divisions between construction passes. For increases, the input loop is always stored on needle $\texttt{fN}_L$. For decreases, the output loop can arrive at either needle $\texttt{fN}_L$ or $\texttt{fN}_R$. This is pseudocode; the (JavaScript) code used in our system are available in the supplementary material.

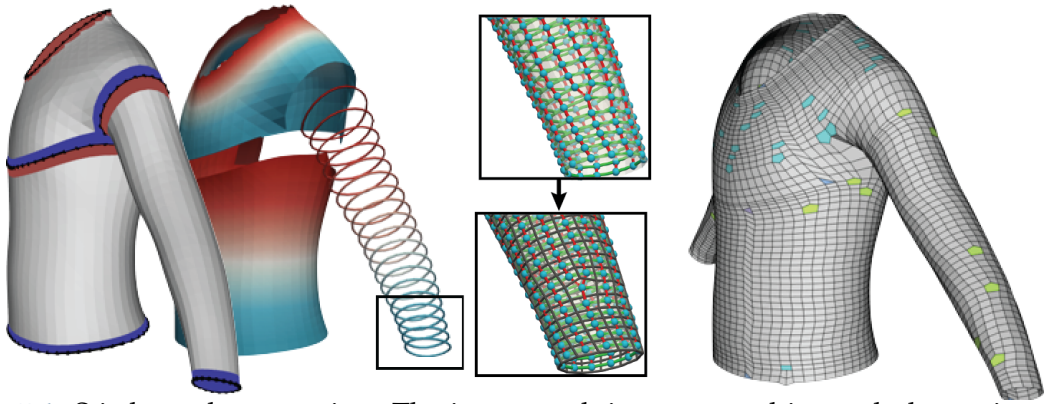


Fig. 5.4: Stitch mesh generation: The input mesh is segmented into tubular regions and remeshed to identify stitch connectivity. Its dual is extracted and refined to generate an initial stitch mesh.

counts are computed, each segment is quad meshed based on the stitch dimensions [37].

To maintain boundary lengths while limiting the change in stitch counts between rows for reliable fabrication, stitch counts are optimized by relaxing integer constraints on the counts and rounding the results:

$$s_f = \underset{x}{\operatorname{argmin}} \sum_{i}^{n} (x_i - s_i)^2 \tag{5.1}$$

$$\text{subject to: } \frac{2}{3} x_{i-1} \leq x_i \leq \frac{3}{2} x_{i-1} \tag{5.2}$$

$$x_1 = s_1, x_n = s_n \tag{5.3}$$

where s_f is the vector of final stitch counts, and s_i is the vector of initial counts.

The stitch mesh is extracted from the dual graph of this structure and faces with higher length distortion are refined or merged. This refinement can generate *short-rows* – a chain of faces that form a partial loop. The mesh is sub-divided along its rows to ensure an even number of short-rows followed by local edits to convert it into a single helical sequence of faces [156]. This ensures that to begin with, each tubular region can be constructed from a single yarn and yarn-wise label consistency is maintained. Loopwise label consistency is maintained by following the time function for remeshing. This initial mesh is machine knittable as long as the input model topology is feasible as proposed by [106] and requires at most one yarn per tubular region.

From this starting mesh, the user can edit the mesh to refine topology or change stitch types as described next.

5.4 Stitch Mesh Editing

Our system provides a suite of editing operations which allow users to navigate the space of knittable augmented stitch meshes (Figure 5.5). These operations all involve replacing some portion of an augmented stitch mesh while maintaining compatible edge labels. Edits of this sort can still introduce global dependency cycles (i.e., violate Property 5.1.1), and we will discuss how our system prevents this at the end of this section. During editing, vertex positions are updated using projective dynamics to ensure faces retain the approximately correct size [13]. From the open source ShapeOp library, edge-strain constraints are used to ensure faces retain the approximately correct size and bending and plane constraints are used to maintain the 3D shape.

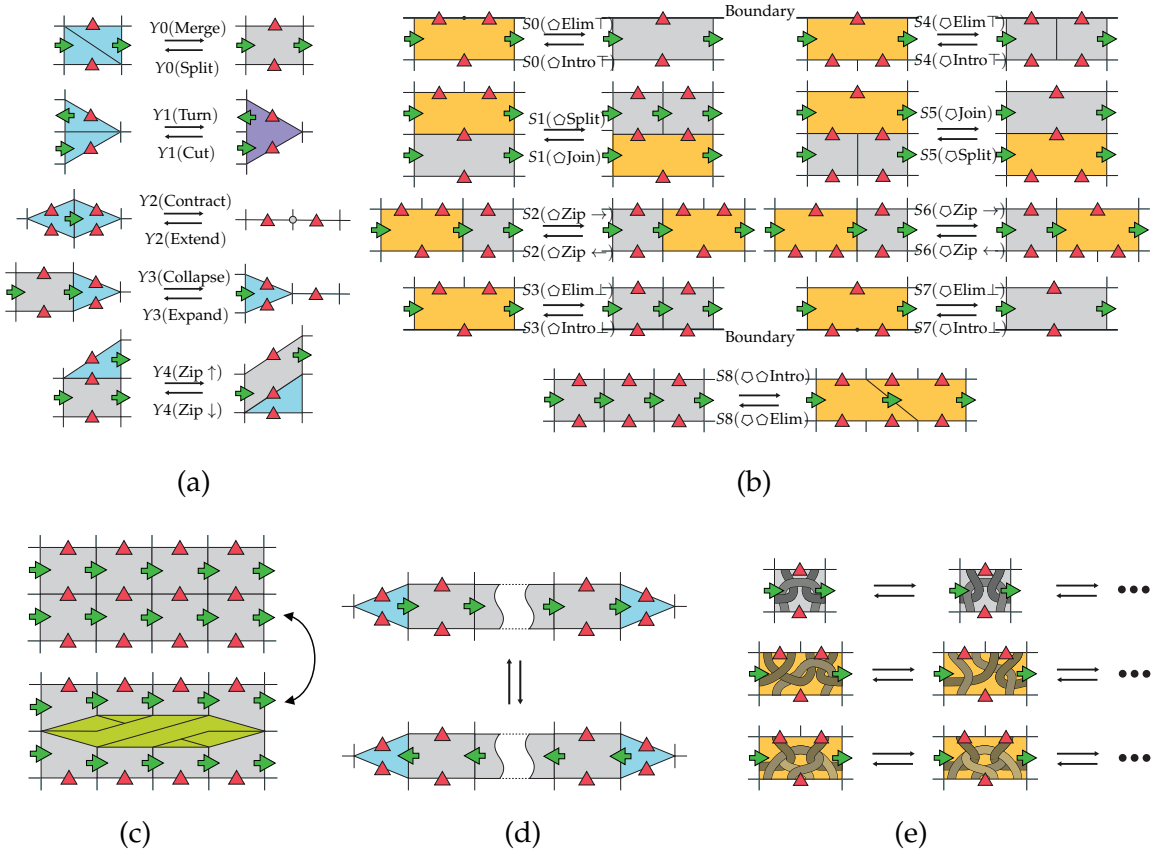


Fig. 5.5: Editing Operations: (a) yarn operations, (b) shape operations, (c) cable operation, (d) yarn reversal, and (e) type operations. Blue indicates yarn-end faces, grey indicates regular faces, orange indicates pentagons, purple indicates short-row faces, and green indicates cable faces. Green and red arrows indicate yarn direction and loop directions, respectively.

5.4.1 Editing Operations

Our system supports two classes of editing operations: mesh editing and data editing. Mesh editing operations change the mesh structure (face counts or connections) and include *yarn operations* that deal with yarn start/end faces, *shape operations* that modify pentagons (for increasing or decreasing loops), and *cable operations* that add/remove cable faces (for reordering loops after construction). Editing operations do not change the mesh structure, and include *type operations* that change face type and *yarn reversal* that reverses yarn direction along a row. Although the stitch mesh structure might be modified by mesh editing, the genus of the input surface is not modified by any of the editing operations.

- Yarn Operations. We call operations that involve single-yarn-edge triangles yarn

operations (Figure 5.5a). Merging two yarn-start/end triangle faces over a *loop* edge will either form a regular quad, $Y0(\text{Merge})$, or short-row face to turn the yarn based on the types of remaining four edges, $Y1(\text{Turn})$. On the other hand, one row can be broken into two rows by their reverse operations. Removing or adding pair of yarn-start/end triangles can be used to add or remove a row, $Y2(\text{Contract/Extend})$. Yarn-start/end triangles are allowed to move along the wale direction as well as along the course direction ($Y3$ and $Y4$).

- Shape Operations. Shape operations allow the user to move pentagons along the wale direction and course direction, as illustrated in Figure 5.5b ($S1$, $S2$, $S5$, and $S6$). Note that operations $S0$, $S3$, $S4$, and $S7$ can remove/add a vertex without causing problems because they do so at the boundary of the mesh.

- Cable Operations. Transposing loops after knitting them create interesting patterns called *cables*. Our system supports insertion and removal (Figure 5.5c) of cable faces of any length between two rows of regular stitches. These faces do not have yarn edges, so they cannot construct new loops, only rearrange them.

- Type Operations. These simple editing operations change face types, for example, swapping a "knit" face for a "purl" face.

  Our system will use the corresponding face program to generate machine code during instruction generation.

- Yarn Reversal. In addition to these face modifications, our system also includes an operation for reversing yarn direction by changing the face types and internal edge labels of an entire row (Figure 5.5d). While not strictly necessary, this operation is much more convenient than removing and reinserting a yarn stitch-by-stitch to change its direction.

These edits work together to enable natural dragging-based edits, where faces are moved across the mesh, locally altering topology. For example, Figure 5.6, users can "zipper" in and out partial rows of yarn by moving yarn-start/end faces, and do the same with columns of loops by moving increase/decrease faces. This gives users access

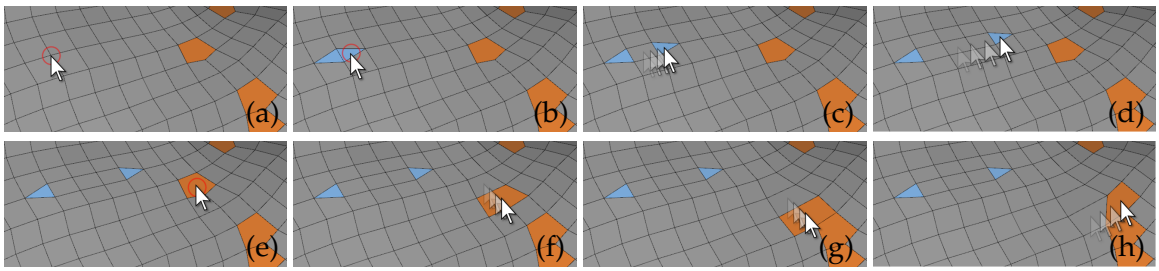


Fig. 5.6: Demo of applying $Y2$ in (a) and $Y3$ in (b - d) to create a short-row as a Yarn "zipper" and aligning the pentagon using the Shape "zipper" by $S5$ in (e - g) and $S6$ in (h).

to shaping techniques in a very intuitive way. Our dragging-based tools are similar to singularity editing [112], but also preserve the machine knittability.

5.4.2 Preserving Machine Knittability

Though our local edits will never introduce locally conflicting edge labels, they are not always legal to apply because they can potentially introduce a dependency cycle, as shown in Figure 5.7. When editing, our interface checks the legality of each operation by attempting a topological sort on the dependency graph induced by the edge labels; if a directed cycle is found between a face and itself, then the ordering property has been violated and the edit is not permitted. This ensures that the ordering property (Property 5.1.1) is preserved. Also, as discussed earlier, none of our edit operations modify the topology of the input model, and therefore the adjacency property (Property 5.1.2) is never violated.

5.4.3 Generality

We refer to an editing operation that passes the global ordering check, and thus can be executed, as *valid*. Importantly, there is always a sequence of valid editing operations that can be used to transform one machine-knittable augmented stitch mesh into another (of the same input topology).

Indeed, we can prove a restricted version of this statement, though we first need a few lemmata:

Lemma 5.4.1. *Shrinking a row using operations $Y2(Contract)$ and $Y3(Collapse)$ is always valid.*

Proof. Removing nodes and edges from a directed graph will not create a cycle. □

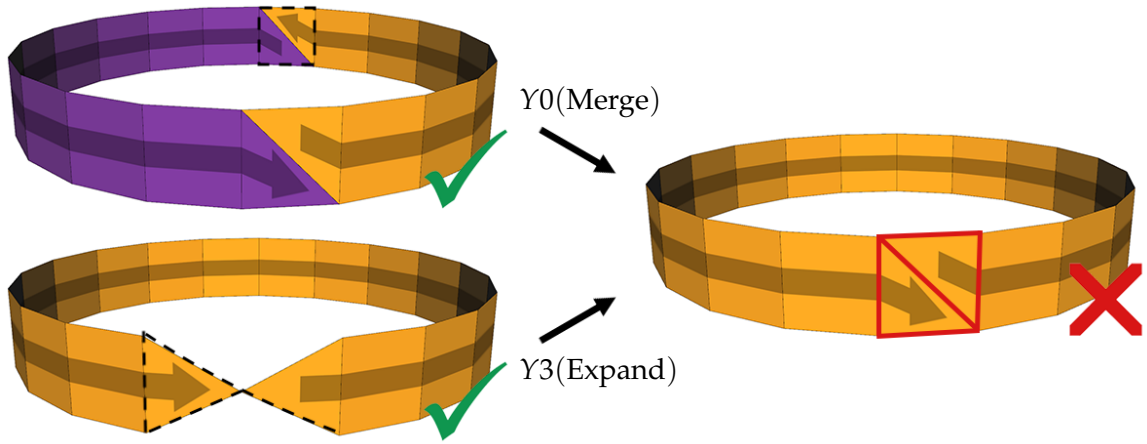


Fig. 5.7: Examples of edits (shown with dashed edges) that our interface would prevent because the resulting mesh contains a cyclic dependency between faces. Arrows show yarnwise dependencies, and loopwise dependencies (not shown) point from bottom to top. Yarn-end and yarn-start faces (highlighted in red) form an unknittable structure by introducing a cyclic dependency.

Lemma 5.4.2. *If the edit f is valid on machine-knittable mesh A, its inverse operation f^{-1} is valid on $f(A)$.*

Proof. $f^{-1}(f(A)) = A$ is machine-knittable by hypothesis, so it passes the ordering check and f^{-1} is valid on $f(A)$. □

Lemma 5.4.3. *Splitting faces by $S1(\pentagon Split)$, $S3(\pentagon Elim\bot)$, $S4(\pentagon Elim\top)$, and $S5(\pentagon Split)$ are always valid.*

Proof. Duplicating an edge in a directed graph will never create a cycle. □

Lemma 5.4.4. *Any pentagon face can be removed from the stitch mesh while maintaining machine-knittability.*

Proof. As shown in Figure 5.8, by repeatedly applying operation $S5(\pentagon Split)$, the decreasing face can be moved to the top boundary and the stitch mesh remains valid (Lemma 5.4.2). If the pentagon is trapped by a yarn-end face, the yarn-end can be moved (Lemma 5.4.1), the pentagon can be moved to the boundary. Then, operation $S4(\pentagon Elim\top)$ can be used to remove a decrease pentagon. Similarly, repeatedly applying operation $S1(\pentagon Split)$ and $S3(\pentagon Elim\bot)$ can be used for eliminating increase pentagons. □

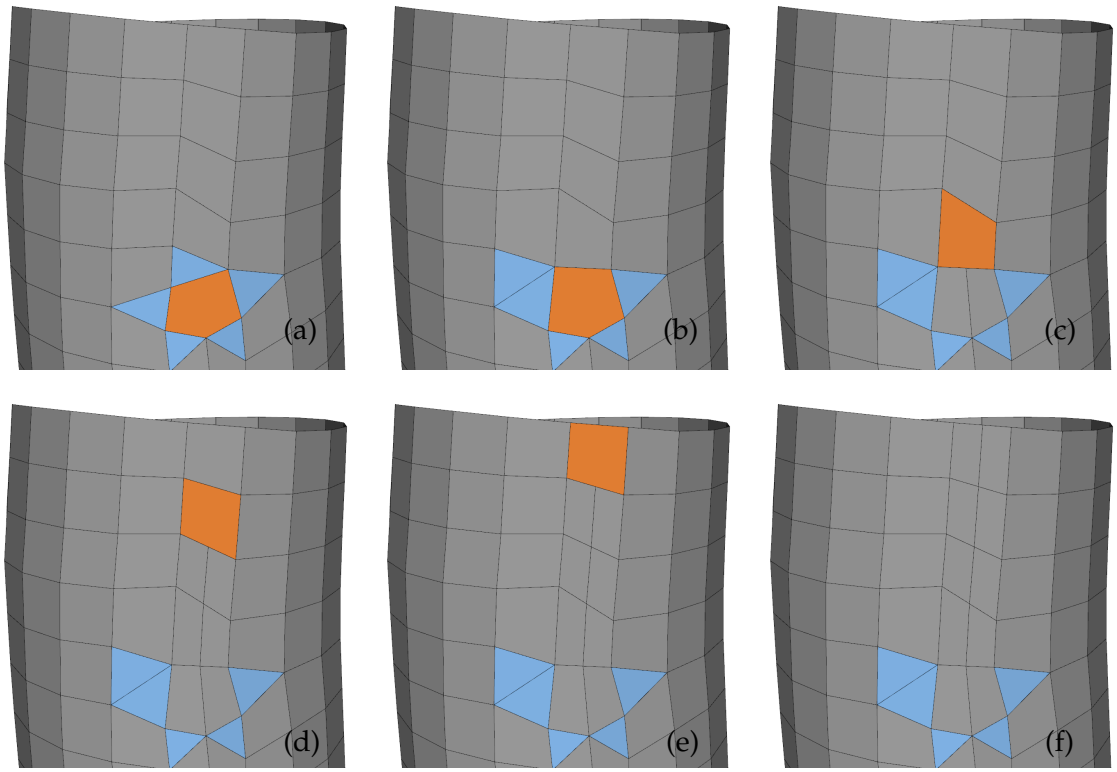


Fig. 5.8: Removing a pentagon while preserving knittability: (a - b), move the yarn-end face above the pentagon; (c - e), move the pentagon to the boundary; (f), remove the pentagon.

Theorem 1. *The editing operations supplied by our interface are sufficient to connect the space of all machine-knittable augmented stitch mesh tubes.*

Proof. First, we show that any valid stitch mesh tube can be turned into a trivial pattern. Any pentagon face can be edited out by Lemma 5.4.4 without breaking validity. Operation $Y1(\text{Cut})$ can be used to remove any yarn-turn triangles. Finally, all yarn-start triangles can be moved closer to their yarn-end by repeatedly applying operation $Y2(\text{Contract})$ and $Y3(\text{Collapse})$ (Lemma 5.4.1). The result is a stitch mesh consisting only of a ring of edges and no faces.

Finally, these edges can be collapsed to a ring with just two edges by applying $Y2(\text{Extend})$ and $Y3(\text{Expand})$ to fill the ring with a single of quads, followed by $S4(\pentagon\text{Intro}\bot)$ and $S0(\pentagon\text{Elim}\top)$ to reduce the number of quads to one, followed by $Y2(\text{Contract})$ to remove the yarn.

Let $f_1, f_2, \cdots, f_n$ be the sequence of n operations to turn a stitch mesh F into the trivial

pattern. Let $g_1, g_2, \cdots, g_m$ be the sequence of m operations to turn a stitch mesh G into the trivial pattern. By Lemma 5.4.3, $g_1^{-1} \circ g_2^{-1} \circ \cdots \circ g_m^{-1}$ is valid on the trivial pattern, so $g_1^{-1} \circ \cdots \circ g_m^{-1} \circ f_n \circ \cdots f_1$ is valid on F and results in G. □

We believe that a similar, more general, proof can be conducted for nontube-like meshes by using a variant of the same construction, but some care must be taken to keep the "trivial configuration" compatible with the underlying topology (since no edits allow, e.g., the creation of a figure-8 of empty edges).

5.5 Instruction Generation

In order to convert an augmented stitch mesh into a list of machine instructions for knitting, our system must determine the order in which the faces should be created and assign machine locations to all loops produced and consumed by faces.

5.5.1 Face Ordering

Given that our interface preserves the ordering property (Property 5.1.1), our system will always be able to find *some* dependency-obeying order of the faces. Our system selects an ordering using a topological sortwith a modified queue that always returns the next face along the currently-being-traced yarn or – failing that – the least-recently-used yarn among possible ready faces. New yarn-start faces are only knit if no other ready faces exist.

We chose this particular ordering heuristic because it avoids switching yarns and favors finishing in-action yarns before starting new yarns, while still knitting every in-action yarn reasonably frequently. Switching yarns can slow down the knitting machine, having too many active yarns can lead to time wasted by the machine moving carriers out of the way, and loops held for a long time on the bed can occasionally be worn out and broken.

Results of our tracing algorithm on several tricky cases are shown in Figure 5.9. Note that, in our system, users are also allowed to manually add dependencies between faces in order to, e.g., steer the heuristic away from nonoptimal orderings (Figure 5.9b), or to prevent two logical yarns they intend to assign to the same physical yarn from being active at the same time. Once the faces have been ordered, our interface allows the user to map each logical yarn to a physical yarn ID that is passed on along with the face sequences to the scheduling system.

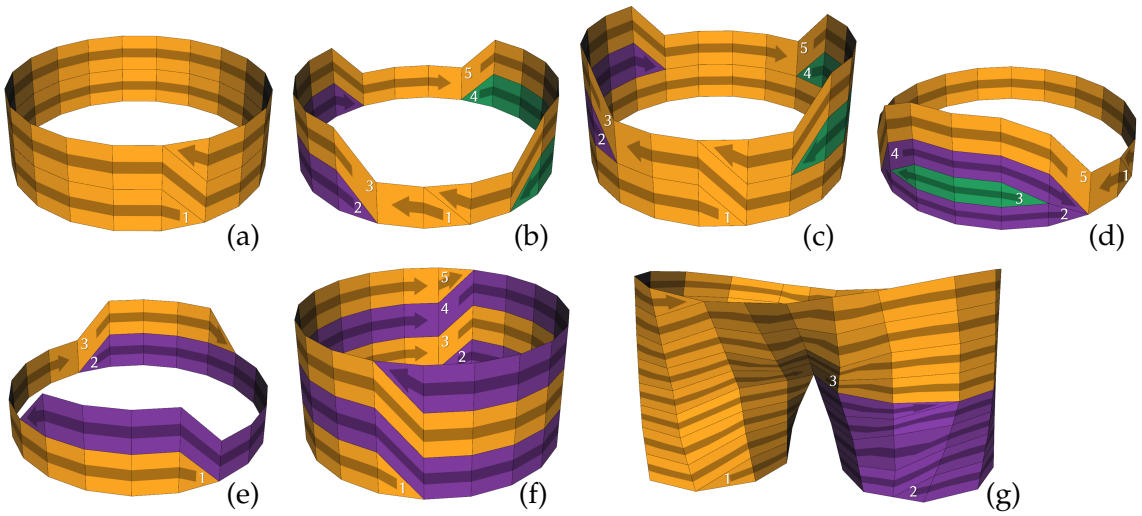


Fig. 5.9: Examples of face ordering: (a) a simple spiral; (b) a case where the heuristic's arbitrary choice of starting face leads to a nonoptimal ordering (if green and purple had been started first, fewer yarn changes would be needed); (c) short-rows embedded in a spiral require yarn carrier switches; (d) nested short rows; (e) yarns that depend on each-other; (f) two helicies that depend on each other, requiring multiple yarn carrier switches; and (g), two tubes merging.

5.5.2 Scheduling

Once the knitting order has been determined by tracing, our system must *schedule* (assign) storage locations for loops. Our scheduler is based on the open-source implementation released by Narayanan, et al. [106], which we have modified to support general faces instead of a fixed menu of stitch types.

Our scheduler turns each face into one more or *fragments* – low-level items that, together, produce and consume the same number of loops as a face – and breaks the sequence of fragments into logical *passes*. These fragments serve as a placeholder for the faces, and allow our system to decouple scheduling for storage locations from the operations performed on those locations. In addition to the knitting program, the configuration data associated with each face type indicate if the face requires its own pass and if it has any custom pass-level programs associated with it. Passes are constructed based on the following conditions:

- All fragments in a pass have the same direction (clockwise or counter-clockwise).
- A pass does not have more than a limited (4) number of "increase" or "decrease" shaping operations that changes its width.

- All fragments in a pass must have the same *basic type* – faces with different basic types force a pass break.

A pass a is said to depend on a pass b if pass a uses the loops produced by pass b (i.e., it reads from storage locations last written to by pass b). Each pass may be dependent on zero or more passes. If a pass depends on exactly one previous pass, it is referred to as a *regular* pass and the rest are *critical*.

Once passes are constructed, scheduling proceeds through the pipeline of knitting scheduler introduced by Narayanan et al. [106]: An upward planar embedding is identified by enumerating all embeddings of the critical passes. Based on the shape of these critical passes, intermediate passes are filled in by assigning a shape that minimizes transfer operations. Finally, needles are assigned to all loops using the computed shapes, and instructions can be generated.

During instruction generation, where the scheduler [106] only ran "knit" operations, our scheduler needs to respect the programs stored with each knitting face. To execute a pass, our scheduler emits instructions as follows: First, any prepass programs associated with the basic stitch type of the fragments are invoked with the storage locations of the previous and current passes. Next, transfer operations (planned by the method of [101]) are executed to align locations. Finally, each face included in the pass is generated by calling the program associated with the face on the storage locations associated with that face's fragments.

Prepass programs can be used for custom transfer planning and for reordering existing loops in a user defined manner and are invoked by passing the locations computed by the transfer planner for *all* the stitches active on the bed and for the stitches participating in the pass with the function signature:

```
function pre_pass(from, to, pass_from, pass_to)
```

The face program associated with the stitch mesh face is invoked by passing the yarn direction determined during tracing, the needle allocations determined by the scheduler, and the ID(s) of the current yarn(s). Instead of invoking a single function, the face programs are divided into a preamble, main execution and a postamble that share the same function signature:

```
function face_pre(dirs, bns, carrier)
```

```
function face_main(dirs, bns, carrier)
function face_post(dirs, bns, carrier)
```

These three functions allow for coarse instruction re-ordering, which we will discuss further after introducing an non-trivial face program:

A purl (i.e., a back knit) constructs the stitch on the *opposite* bed. This simple alteration generates a backward facing loop, which appears structurally different from a forward facing loop. A combination of front and back stitches can give rise to a wide variety of patterns (Figure 5.10, columns 7-9). The face program for a purl first transfers the loop held in the storage location to the holding position on the other bed, knits on this new storage location and transfers the loop back. No pass level programs are necessary.

```
function purl_pre(dirs, bns, carrier){
        xfer(bns[0], opposite(bns[0]));
}
function purl_main(dirs, bns, carrier){
        knit(dirs[0], bns[0], carrier);
}
function purl_post(dirs, bns, carrier) {
        xfer(opp(bns[0]), bns[0]);
}
```

On a single system machine like the one used to fabricate our examples, knit and transfer instructions must be performed in separate carriage movements; this means that a row of N purl stitches – each of which requires an `xfer`, `knit`, `xfer` instruction chain – takes $3N$ carriage movements to fabricate if all the instructions are in the "main" function, but only 3 carriage movements if they are distributed across "pre," "main," and "post" functions.

5.6 Results

We have used our pipeline to create a wide range of objects, which we discuss in this section. All results were knit on a Shima Seiki SWG091N2 industrial knitting machine with 15 needles per inch; the SWG*N2 series are basic v-bed machines typically used for hat, glove, and accessory production. All examples were knit from a 2-ply acrylic yarn (Tamm Petit). With this yarn and machine, a knit stitch is 2.88mm wide and 1.75mm tall; so we used this to set the face sizes in our interface. Knitout code for all our results is included in the supplementary material.

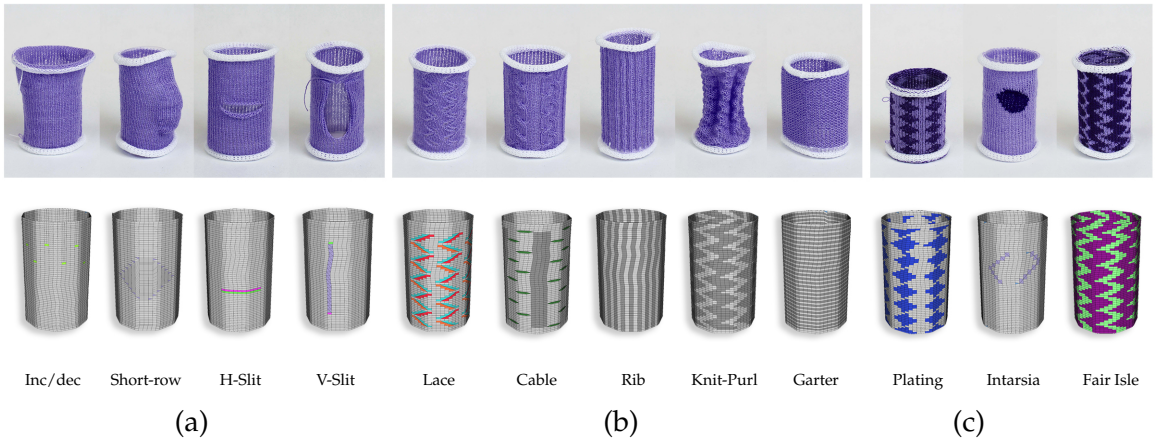


Fig. 5.10: A sampler of knitting techniques available in our system: (a), increase/decrease shaping, short-row shaping, a horizontal slit using bind-off and cast-on, and a vertical slit using C-knitting; (b), lace with ordered increases and decreases, cables, rib (alternating columns of knits and purls), a complex knit-purl pattern, and garter (alternating rows of knits and purls); and (c), colorwork with the plating, intarsia, and Fair Isle methods.

5.6.1 Basic Techniques

Thanks to the generality of the augmented stitch mesh structure, our system can be used to program a wide variety of common knitting techniques. A sampler of the techniques our system supports is presented in Figure 5.10. Of course, further techniques can be added by defining more face types.

5.6.2 Colorwork

Our system supports three common styles of knitting with multiple colors. *Intarsia* colorwork involves knitting different sections of the object with yarns of different colors. This is easy to achieve with our basic stitch types, but is relatively inflexible. In *Fair Isle* colorwork, yarns of different colors run along the inside of the pattern, and knit stitches are made out of whichever color the designer wants to be shown. We implemented this in our system using faces that pass multiple yarns along their yarn edges but knit with only one of them (Figure 5.11, left). Figure 5.12 shows the addition of ribs and Fair Isle colorwork in a basic sock pattern. Manually editing the face types of the initial mesh took around 60 minutes for this pattern. Finally, in *plating* colorwork, two yarns are used in every stitch, with one running slightly in front of the other in order to make it appear on the front of the stitch. This has the advantage of making a reversible pattern, with the downside of some loss of contrast. This, again, was simple to implement in our system with special

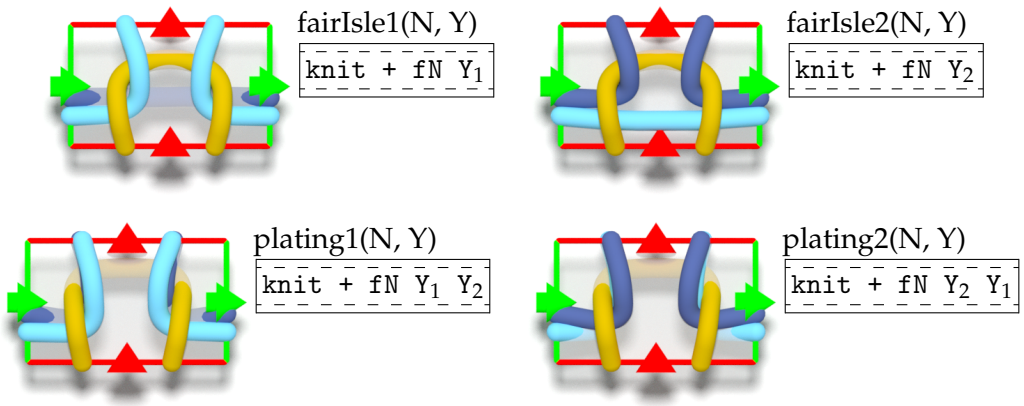


Fig. 5.11: Additional face types created for Fair Isle and plating colorwork. We use Y_1 and Y_2 to denote the first and second elements of the yarn array, Y, respectively.

Fig. 5.12: A sock pattern can be easily edited by adding color and ribs.

plating face types (Figure 5.11, right). The two-sided decorative item showing a skyline in Figure 5.13 was created using plating and a texture map to guide face selection.

5.6.3 Adding Detail

Our system can be used to add fine-scale detail to automatically-generated patterns. Functional and decorative knit and purl textures as well as cable patterns can be easily created by switching face types. We created a hand warmer, Figure 5.14, from a tube by introducing a vertical slit using modified (tuck-less) short-row turn faces – a technique known as *C knitting* – and decorating it with ribs and cable faces. Starting from the tube,

Fig. 5.13: A knit skyline decorating a cylinder by picking plating face types from the image (top right)

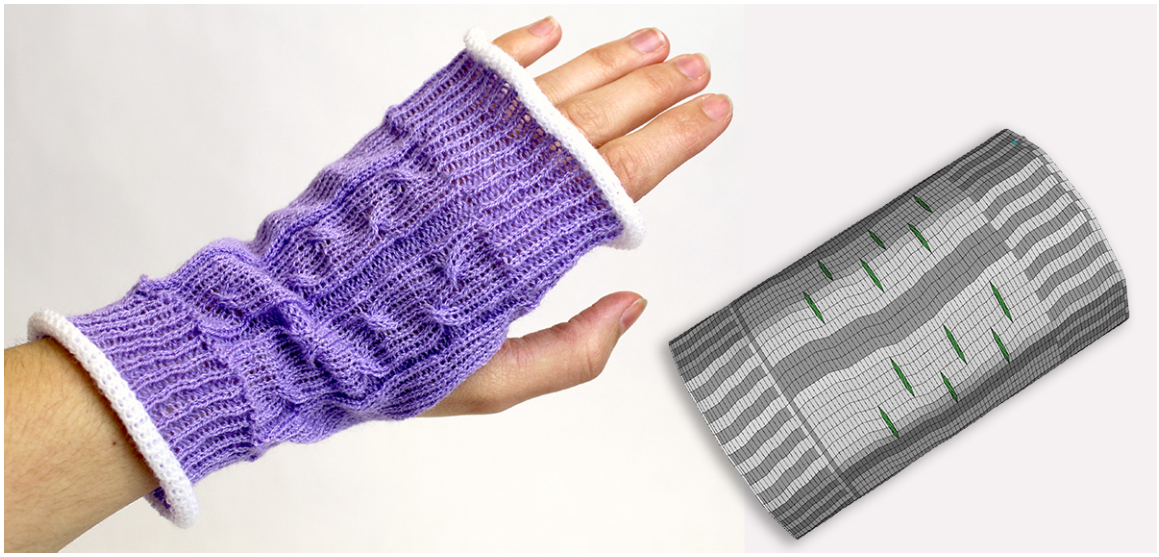


Fig. 5.14: A hand-warmer designed and edited in our system.

these edits took less than 15 minutes to finish. The finished product has a significantly more interesting surface than could be produced by previous work on high-level design for knitting machines (e.g., Figure 14 in [101]).

Beginning from the same input mesh of a sweater, variations can be quickly created by editing the pattern using our system (Figure 5.15). By introducing paired increases and decreases with specific stacking orders, lace can be created. By stamping a simple

Fig. 5.15: Multiple variations – including lace (center) and colorwork (intarsia stripes left, Fair Isle right) – on the same base pattern can be easily introduced.

pattern over the body of the sweater, *Fair Isle* style colorwork and details can be added. By directly modifying yarn path using the edit operations *Intarsia* style colorwork can be created. Starting from the initial mesh, each pattern required around 20 minutes of manual editing for generating the variations.

We also used our system to add surface texture to a classic computer graphics model (Figures 5.1, 5.16a), producing a much more interesting surface appearance than was possible in previous systems. Editing the texture involved editing face types throughout the pattern and took around 60-90 minutes. Our system incorporates basic block-level pattern specification tools for tiling patterns. In general, introducing more midlevel patterning tools would further reduce editing time.

Switching yarn types can be used to not only introduce color but also to vary yarn properties. In Figure 5.16b, our system was used to add conductive yarn "rails" to carry current to LEDs in a cap.

5.6.4 Shaping Adjustment

Professionally-designed knitting programs often feature carefully aligned increases and decreases, while current automatic generation approaches place them somewhat sporadically – this produces higher shape fidelity, but can appear messy. Our system's editing tools makes it easy to reposition shaping features, as can be seen in the sweater example in Figure 5.17.

(a) (b)

Fig. 5.16: Our knitted results: (a) the Stanford bunny with ribs on the body and garter pattern on the ears and textures created by knits and purls generates a distinct look in comparison to the image from the supplementary material in [106] shown on the right, and (b) conductive yarn (cyan faces) can be integrated into the body of the object for incorporating electronics.

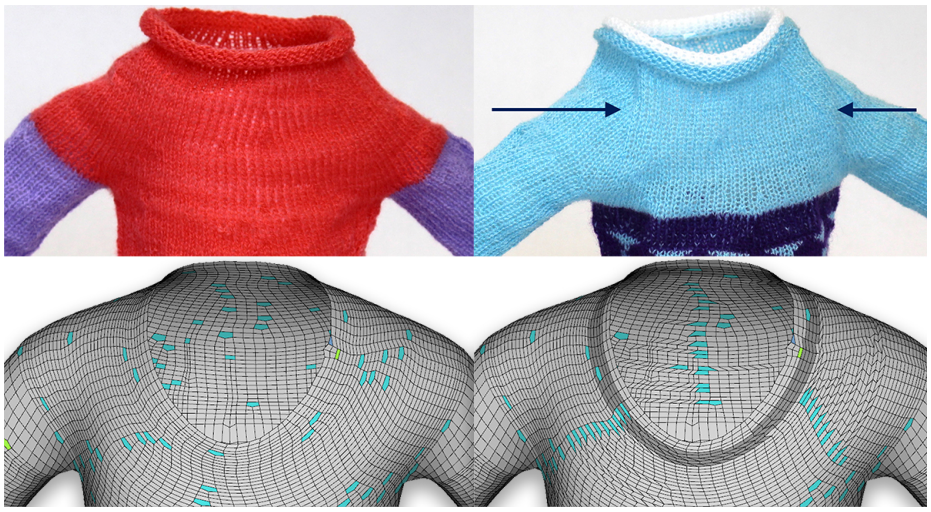


Fig. 5.17: By aligning shaping edits using our system (right), a more traditional appearance can be created.

5.6.5 Improving Robustness

Narayanan et al. [106] reported that in their fully automatic stitch generation system, patterns could occasionally fail to properly knit owing to suboptimal schedules or extreme shaping. In our system, such meshes can be edited to avoid these situations. For example, the sweater pattern for the bear had a large number of concentrated decreases that caused the resulting pattern to fail. Our editing tool allows those decreases to be staggered across rows to produce a similar but robust pattern (Figure 5.18). These edits were performed in under 15 minutes

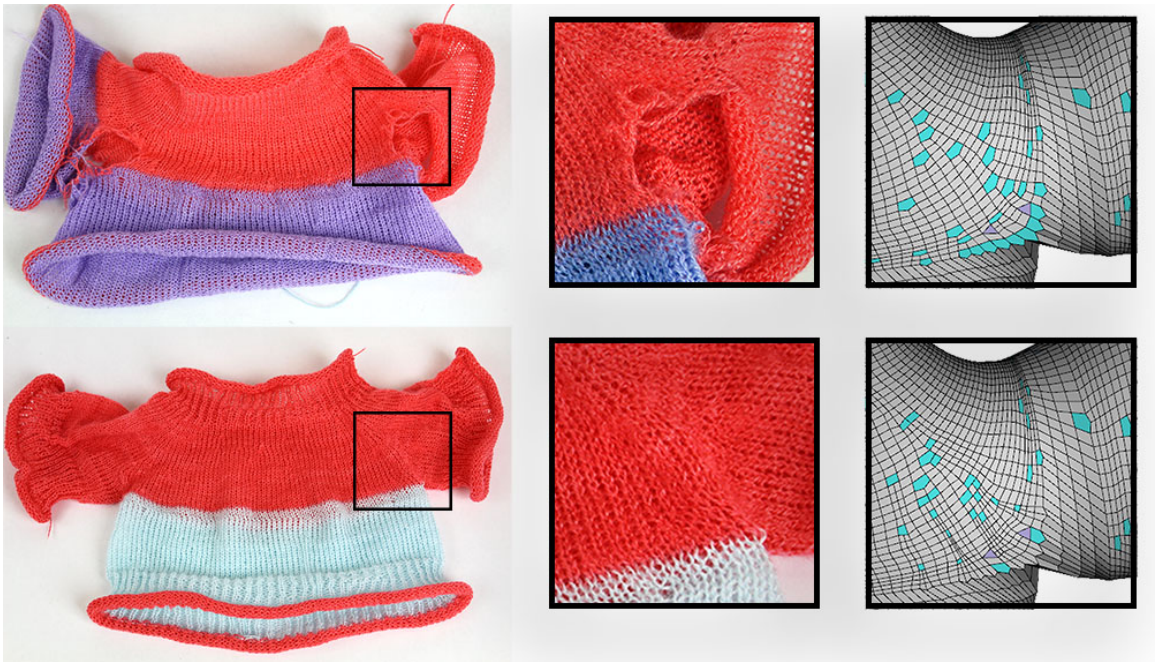


Fig. 5.18: Clustered decreases in one row (top) were distributed to improve the robustness of the sweater pattern (bottom). Notice the holes under the sleeves in the top example.

5.6.6 Complex Results

By using a combination of our automatic remeshing tool, editing tools, and scheduling system, novel three-dimensionally shaped seamless knit objects can be designed and fabricated in an intuitive way (Figure 5.19).

5.7 Limitations and Future Directions

In this chapter, we have introduced a flexible and useful visual editor for machine knitting programs. However, there remain areas for improvement in our system.

In our system, remeshing runs before faces types are selected, so it must assume all faces are the same, generic, size. In practice, not only can face programs modify the machine's stitch size settings, but faces such as cables or combinations of adjacent knits and purls can introduce desirable but pronounced local deformation. This means that editing can change the shape of the object in ways that the initial remeshing did not account for. In the future, this could be addressed by including some coarse local stitch sizing information when doing the initial remeshing, or by adding real time simulation into the design pipeline to provide clear feedback on deformations.

By design, none of our editing operations can introduce a surface topology change.

Fig. 5.19: Toys with accessories designed in our system. The bear (left) is wearing a custom beanie with slits for ears and a matching sweater. The cat (right) is wearing a cap with knit and purl patterns and a sweater with cable work and ribbed sleeves.

Introducing this class of edits with knittability guarantees would require quickly verifying that the mesh retains an upward planar embedding, which is hard [46]; however, incremental reverification approaches may make this tractible for interactive editing.

An implementation limitation that is worth pointing out involves *balancing* faces participating in splits and merges so that a valid layout exists (Property 6 in [106]). Although easy to detect, our system does not implicitly maintain balance and relies on the user to ensure that shaping edits on the splitting/merging cycles are symmetric. Similarly, our current implementation allows mapping logical yarns to two (or more) physical yarns to support plating and Fair Isle colorwork; however, it does not allow faces that merge or split logical yarns. Supporting such faces should not require any major algorithmic changes.

Our system remains agnostic to machine specifications such as choice of yarn carrier and yarn carrier position. However, when multiple yarn carriers are active, their exact position and relative motion can potentially introduce yarn tangling. A possible extension to our system's instruction generation phase would detect these machine configurations and trigger special-case logic to avoid them, e.g., by adding extra dependencies between stitches or providing scheduling hints.

Our scheduler can only perform resource allocation for *tubular* surfaces (and flat surfaces that can be viewed as a part of a tube); in part, this is because it depends on having a specific arrangement of free needles to allow it to rearrange held loops. Therefore, it cannot handle techniques that require specific global layouts of stitches (e.g., fiber inlay), or that occupy needles in a way that prevent flexible racking adjustments (e.g., complex full-gauge flat-knitting patterns). We believe that, in the future, additional constraints can be introduced into the scheduling process to handle these cases, though more user intervention may be required to produce valid schedules.

Finally, our system makes use of an automatic transfer planning algorithm that can perform substantially worse than hand-designed solutions for the same loop movements [92]. This is acceptable for prototyping, but may be a problem in manufacturing. Better instruction generation through improved planning and scheduling algorithms – essentially, the work of building better compilers for machine knitting – remains an interesting open problem.

5.8 Conclusion

In this chapter, we demonstrated a new visual programming tool for computer-controlled knitting machines. The core of our approach is an *augmented* stitch mesh data structure which associates each face in a mesh with machine knitting instructions and maintains dependency information between the faces. Our system allows users to automatically create a machine-knittable augmented stitch mesh from a 3D model, edit this augmented stitch mesh while preserving machine-knittability, and transform the mesh into a knitting program for machine fabrication.

Knit programmers are still doing the same thing they were 80 years ago. Though the media has changed – from cam cylinders to card chains to paper tape to floppy disks to flash drives and ftp servers – programmers still explicitly tell the machine what to do with each needle on each carriage pass. In contrast, our system allows a user to manipulate output structures, describing exactly what the machine should make. This shift – from the *how* of fabrication to the *what* of the finished product – is empowering and delightful.

This is the knitting design tool we have always wanted, and we are exceptionally pleased to have finally created it.

CHAPTER 6

REAL-TIME KNITS RENDERING

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 [27–29, 76, 77]. 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 chapter, 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 two methods for approximating ambient occlusion for high-quality lighting with limited resources.

We compare our simplified fiber-level models with reference models generated by a recent procedural yarn model [170] 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 6.1). Since our method does not rely on any precomputation of the yarn-level control points, it is suitable for yarn-level cloth animation. For rendering animated yarn-level cloth models, we only need to update the yarn-level shadow map and the

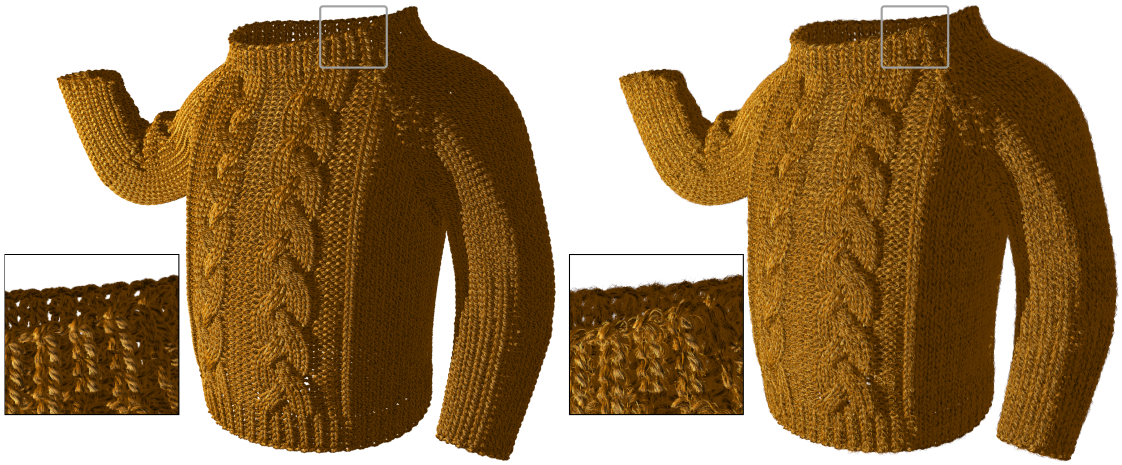


Fig. 6.1: Examples of rendering fiber-level cloth at real-time frame rates: A sweater model that consists of 356K yarn curve control points and over 20M fiber curves, rendered using different yarn types with different fiber-level geometry. Notice the difference in appearance.

rest of our precomputations can be used without the need for any update due to 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.

This chapter 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. We extend our prior work on this topic [159] that only uses a simple shading model by describing how a physically-based shading model can be incorporated with our fiber-level cloth rendering approach. Moreover, this extension provides a new fiber-level ambient occlusion computation and compares it to the simple approximation used in our prior work [159]. In addition, we describe how animated yarn-level cloth models can be rendered with temporal coherency. Nonetheless, our underlying rendering approach is targeted for current GPUs, and it is designed for rasterization only, so it does not provide a trivial mechanism for integrating it with ray tracing. Computing multiple scattering of light or global illumination is beyond the scope of this section. Therefore, the methods we describe in this section cover only a portion of a full cloth appearance modeling process with fiber-level details.

6.1 Procedural Yarn Model

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 6.2, formed by twisting a few substrands 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.

Recently, Zhao et al. [170] described a procedural representation of fiber geometry forming the yarn structure, and they 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 section 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

$$\mathbf{c}_i(\theta) = \left[\, R\cos(\theta_i + \theta), R\sin(\theta_i + \theta), \alpha\,\theta/2\pi \,\right]^T , \tag{6.1}$$

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

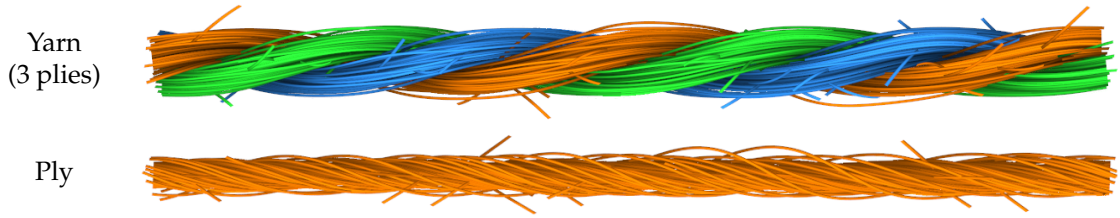


Fig. 6.2: Yarn structure: Yarn typically consists of multiple plies, each of which is made up of tens to hundreds of micron-diameter fibers, depending on the yarn type.

$$R(\theta) = \frac{R_i}{2}\left(R_{\max}+R_{\min}+(R_{\max}-R_{\min})\cos(\theta_i+s\theta)\right), \tag{6.2}$$

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 6.2 with a larger value $R_{\max}^{\text{loop}}$. Finally, hair fibers are fibers that have open endpoints sticking outside of the their plies. They significantly contribute to the fuzzy appearance of yarn.

6.2 Fiber-Level Geometry

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 minimize the data stored and sent to the GPU and fiber segments actually drawn on the screen.

6.2.1 Fiber Generation

We use a procedural fiber generation method based on the model of Zhao et al. [170]. 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 6.3. 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

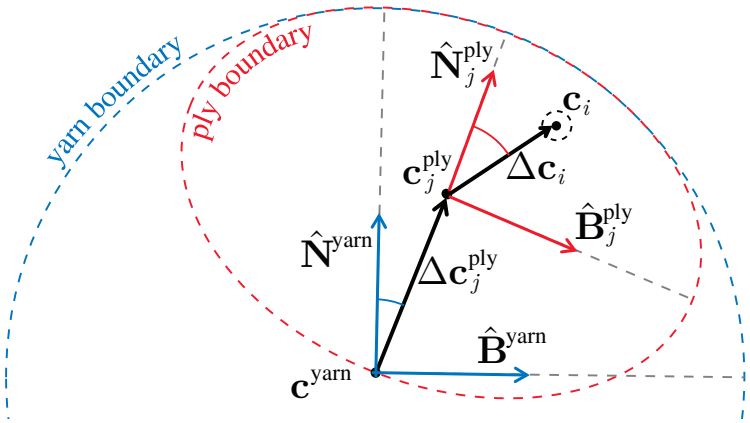


Fig. 6.3: Placing fibers around the yarn: The computation of fiber positions takes place on the cross-section plane perpendicular to the yarn curve.

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

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

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

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 6.2) along the z-axis of the ply with a period of α (see Equation 6.1). We consider each full period

of R separately and assign a fiber type. Figure 6.4 shows an example that includes four periods and their fiber types. The migration fibers simply follow Equation 6.2 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. [170]. 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.

6.2.2 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

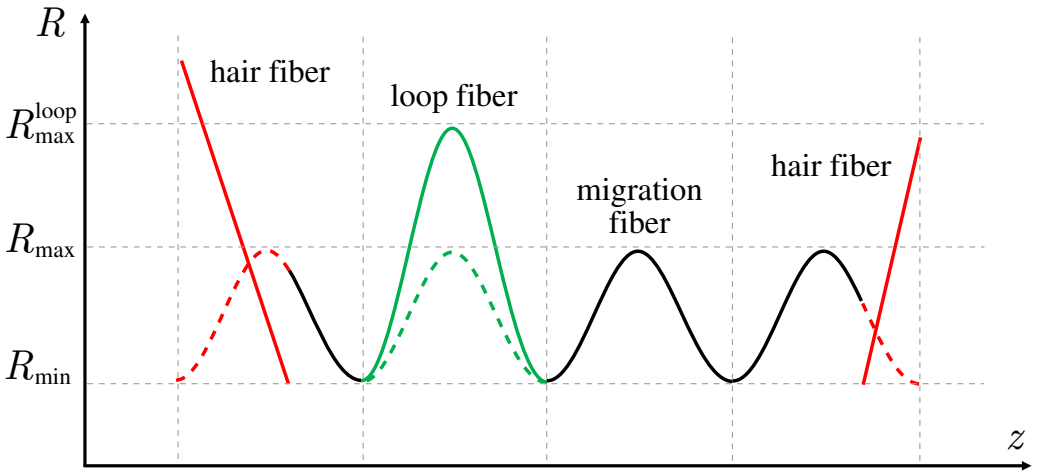


Fig. 6.4: Fiber types: Black curves are migration fibers with R values changing between R_{min} and R_{max}. The green curve is a loop fiber, and the red curves are hair fibers.

minimize the 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 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 6.5, and it is updated only when the parameters of the procedural model are modified. In order to reproduce the appearance of all fibers represented by the core fiber, we store a height-map and 2D surface normals. We also need an alpha channel that indicates the visible holes in the core fiber and marks its outline. However, instead of storing a separate alpha channel, in our implementation we use the height-map to determine these holes, where the height values are zero. We also store 3D surface direction of the fibers in the core fiber, computed by taking the u-direction of the texture as the ply direction.

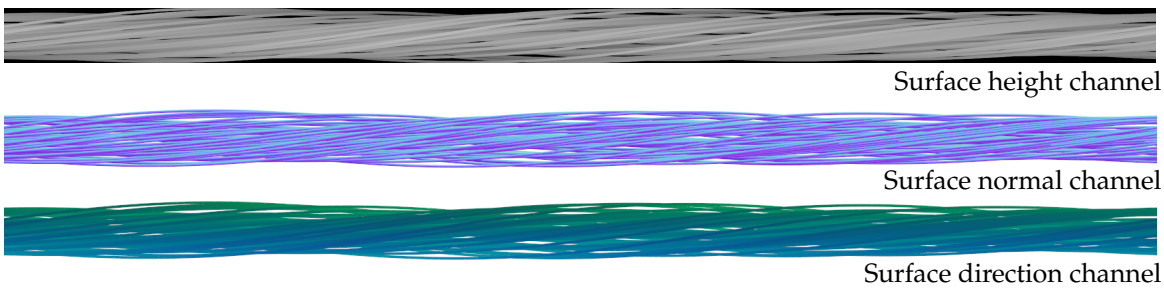


Fig. 6.5: Different channels of an example core fiber texture, packed into two textures in our implementation.

This surface direction component is used for computing the specular component of the core fiber BRDF (bidirectional reflection distribution function) during shading. In our implementation we store the surface direction as a secondary texture (Figure 6.5) and transform the texture-space directions to camera space during shading. 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

$$u = \frac{1}{2\pi}\left(\frac{\theta\,(\alpha^{\text{ply}})^2\,\alpha}{\alpha^{\text{ply}} - \alpha} + \frac{\psi_{\text{view}}}{2}\right), \tag{6.3}$$

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

6.2.3 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

$$\lambda = \begin{cases} 1, & \text{if } \omega^{\text{LoD}} \leq \omega^{\text{fiber}} \\ 0, & \text{if } \omega^{\text{ply}} \leq \omega^{\text{LoD}} \\ \frac{\omega^{\text{fiber}}}{\omega^{\text{LoD}}} \left(\frac{\omega^{\text{ply}} - \omega^{\text{LoD}}}{\omega^{\text{ply}} - \omega^{\text{fiber}}} \right), & \text{otherwise,} \end{cases} \tag{6.4}$$

where $n_{\text{max}}^{\text{fiber}}$ 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.

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.

6.3 Illumination and Shading

Fabric appearance is a combination of both the fiber-level fabric geometry and the shading model that defines the light interaction with the fibers. In this section, we discuss efficient methods for handling self-shadows of the fabric geometry and ambient occlusion of a piece of yarn. We also discuss shading models for the fiber-level geometry, including physically-based shading.

6.3.1 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 fibers 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. [170]). 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 6.6. 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 6.6 top). 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 (Figure 6.6 bottom).

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

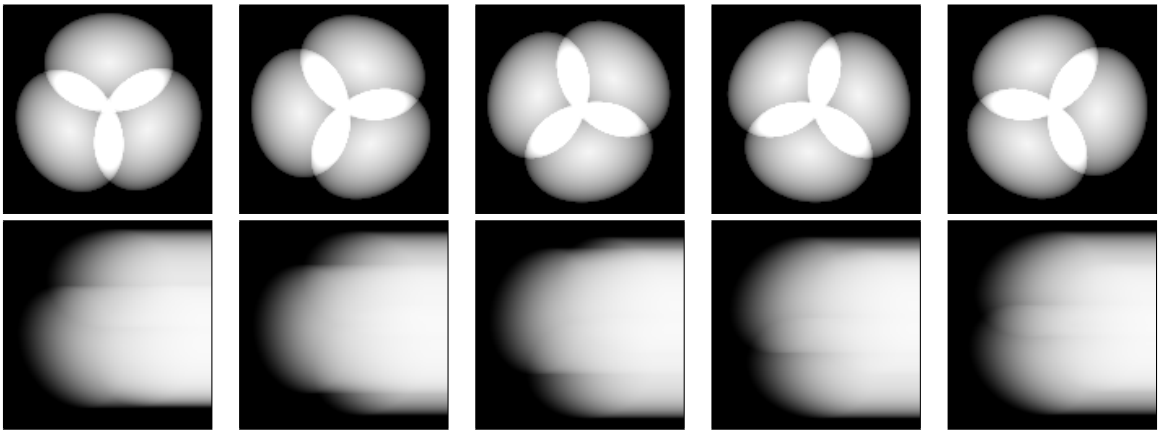


Fig. 6.6: Precomputed self-shadows: (Top row) density slices for different orientations of yarn, and (bottom row) their corresponding self-shadow densities with light coming from the left sides of the images.

light direction. This total density value can be used with an exponential decay function to estimate the light reaching any point within the yarn volume during shading.

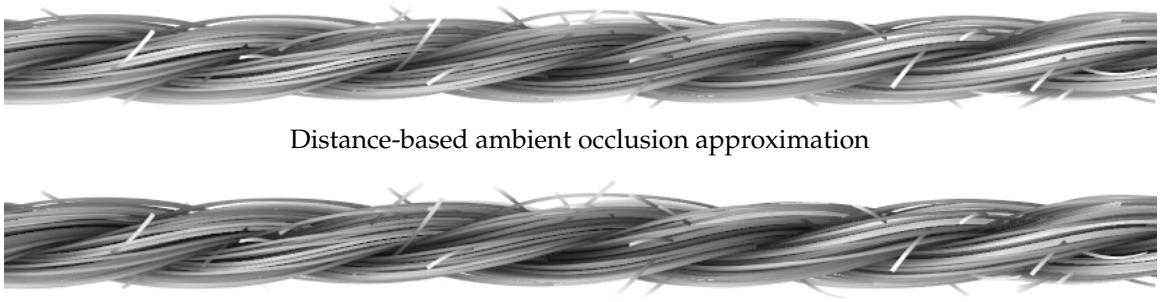
While computing the shadow map for handling the shadows between yarn pieces, we simply render each yarn curve as a thick tube, completely disregarding the fiber-level structure of yarn. Nonetheless, since the shadow map is not used for computing the self-shadows within the yarn, this provides an acceptable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

6.3.2 Ambient Occlusion

Fibers that form a piece of yarn occlude each other in all directions. Therefore it is important to consider the ambient occlusion within the yarn. We describe two approaches for efficiently handling the ambient occlusion within the yarn. Yet, it is important to note that neither one of these approaches consider the ambient occlusion caused by different yarn pieces that are in close proximity or other scene geometry close to the rendered fabric. Such relatively large-scale ambient occlusion factors can be computed using existing techniques [7, 102, 140]. The approaches described here can be used in combination with existing techniques for efficiently incorporating the small-scale ambient occlusion that takes place within the yarn due to fibers occluding each other.

The first approach we consider is an extremely simple ambient occlusion factor that merely uses the distance of a fiber from the center of the yarn curve. One could also consider the distance of a fiber from the ply center for approximating ambient occlusion; however, that would not account for the occlusion of separate plies that form the yarn structure. By using the distance to the yarn center we approximate the ambient occlusion as a circularly symmetric function around the yarn center, which, according to our experiments, provides a reasonable approximation of ambient occlusion within yarn.

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 6.7 top. 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.



Distance-based ambient occlusion approximation

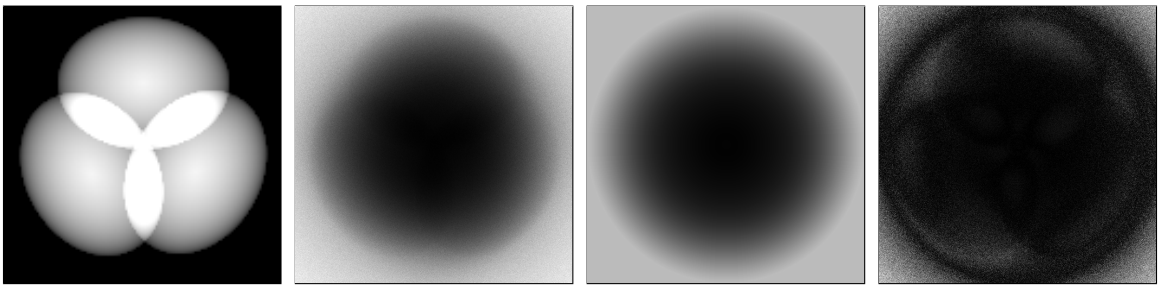
Precomputed ambient illumination

Fig. 6.7: An example straight yarn model rendered with ambient occlusion computed using two different methods.

Our alternative approach is precomputing ambient occlusion into a 2D texture that represents the ambient occlusion value anywhere within a cross-section of the yarn. For computing this ambient occlusion texture we again directly sample the fiber density function that is used for placing the fibers of each ply (as in fiber-level self-shadow computation), instead of using a set of example fibers. This provides an efficient method for precomputing the ambient occlusion texture and the resulting texture does not correspond to a particular set of fibers generated using the density function and their orientations along the yarn. Using a density function, we also inherently ignore the ambient occlusion contributions of hair fibers. Nonetheless, this is a reasonable approximation, since hair fibers are typically sparse, as compared to other fibers.

For simplifying the precomputation of the ambient occlusion texture further, we assume that the yarn has no twist and that the plies are perfectly parallel to the yarn direction. This also is a reasonable assumption, especially when the ambient occlusion radius is much smaller than α. Yet, this simplification ignores the (typically) subtle impact of the yarn twist on ambient occlusion. It also allows computing ambient occlusion entirely in 2D.

The ambient occlusion texture represents the amount of ambient light within a cross-section of the yarn for an arbitrary orientation of the yarn twist. The same texture is used for any orientation of the yarn twists by rotating the texture coordinate accordingly. We precompute this texture using Monte Carlo sampling. We begin with generating a temporary 2D texture for the fiber density function (Figure 6.8a). For each texel of the



(a) (b) (c) (d)

Fig. 6.8: Ambient Occlusion Texture: (a) The density texture used for ambient occlusion texture computation, (b) the final precomputed ambient occlusion texture, (c) the simple distance-based ambient occlusion converted to a texture, and (d) four times the difference between the precomputed ambient occlusion texture and the simple distance-based ambient occlusion.

ambient occlusion texture (Figure 6.8b), we pick a number of random 2D directions and trace rays starting from the texel location through the 2D density texture. The accumulated density along a ray indicates the visibility in the ray direction. For faster convergence we use stratified sampling of all possible 2D directions. Note that this is equivalent to computing the visibility in 3D with infinite radius when the yarn is straight and has no twist.

Figure 6.8 also compares the precomputed ambient occlusion texture to our simple distance-based ambient occlusion. For this comparison, we compute the radial average of the precomputed ambient occlusion texture and fit a cubic polynomial that we use as the distance-based ambient occlusion function. While the differences on the texture space is minor (Figure 6.8d), some slight differences can be observed in the rendered images, as seen in Figure 6.7. Nonetheless, these comparisons show that our simple distance-based ambient occlusion approximation can provide a good approximation of ambient occlusion. Therefore, the results in this section use the distance-based ambient occlusion method, unless specified otherwise.

6.3.3 Shading

We have tested our method using two different shading methods. The first one is a simple shading model with a diffuse and a specular component. The second shading model uses a physically-based BSDF (bidirectional scattering distribution function) model.

The diffuse component of the simple shading model is based on the surface normal of a

fiber, obtained from the fiber texture. For the specular component f_{specular} we use a far-field approximation of surface reflectance for tubular shapes, similar to the hair shading model of Sadeghi et al. [123], computed as

$$f_{\text{specular}} = k_{\text{specular}} \cos(\phi/2) \exp(-\theta_h^2/2\beta^2), \tag{6.5}$$

where k_{specular} 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.

For demonstrating that our method can also be used with a physical-based fiber shading model, we follow the BSDF model of Khungurn et al. [80]. This model includes two specular illumination components: R and TT for handling reflection and transmission respectively. The R component is isotropic in azimuthal directions and the TT component includes a single forward scattering lobe. Following model of Khungurn et al. [80], we omit the TRT component, which would account for the internally reflected light from fibers, since this component was determined to be unimportant for textile fibers [80]. The BSDF is written as a product of azimuthal (N) and longitudinal (M) components

$$\begin{aligned} f_{\text{s}}(\omega_{\text{i}}, \omega_{\text{o}}) = {} & M_R(\theta_{\text{i}}, \theta_{\text{o}})\, N_R(\phi_{\text{i}}, \phi_{\text{o}}) \\ & + M_{TT}(\theta_{\text{i}}, \theta_{\text{o}})\, N_{TT}(\phi_{\text{i}}, \phi_{\text{o}}), \end{aligned} \tag{6.6}$$

where the direction $\omega_{\text{i,o}}$ is represented in spherical coordinates by a longitudinal angle $\theta_{\text{i,o}}$ and an azimuthal angle $\phi_{\text{i,o}}$ for the incoming and outgoing direction, respectively (note that in this section we use θ_i with a different meaning than the same symbol used in other sections).

Since we do not consider indirect illumination or multiple scattering, but we only incorporate ambient occlusion for ambient illumination L_{amb}, for a scene with n light sources, we can write the scattered radiance from a fiber as

$$L_{\text{o}}(\omega_{\text{o}}) = \sum_{i}^{n} L_{\text{i}}\, f_{\text{s}}(\omega_{\text{i}}, \omega_{\text{o}}) + L_{\text{amb}} \int_{\Omega} f_{\text{s}}(\omega_{\text{i}}, \omega_{\text{o}})\, d\omega_{\text{i}}\,, \tag{6.7}$$

where L_{i} is incoming radiance from the light (including shadows), and Ω represents all directions over a sphere. Note that we treat ambient illumination L_{amb} as constant illumination coming from all directions, so it is placed outside of the integral and the integral

evaluates to a constant term. The integral for the ambient illumination term can be written as the sum of R and TT components, both of which are split into products of azimuthal and longitudinal components, resulting

$$\int_{\Omega} f_s(\omega_i, \omega_o)\, d\omega_i = \int_{-\pi/2}^{\pi/2} M_R(\theta_i, \theta_o) \cos^2\theta_i\, d\theta_i \int_0^{2\pi} N_R(\phi_i, \phi_o)\, d\phi_i + \int_{-\pi/2}^{\pi/2} M_{TT}(\theta_i, \theta_o) \cos^2\theta_i\, d\theta_i \int_0^{2\pi} N_{TT}(\phi_i, \phi_o)\, d\phi_i. \tag{6.8}$$

Following Khungurn et al. [80], N_R is defined as a constant term and N_{TT} is taken the von Mises distribution f, and the integrals of the azimuthal terms evaluate to constant values, such that

$$\int_0^{2\pi} N_R(\phi_i, \phi_o)\, d\phi_i = \int_0^{2\pi} \frac{1}{2\pi}\, d\phi_i = 1 \tag{6.9}$$

$$\int_0^{2\pi} N_{TT}(\phi_i, \phi_o)\, d\phi_i = \frac{\gamma_{TT}^{-2}}{I_0(\gamma_{TT}^{-2})} I_0(1), \tag{6.10}$$

where γ_{TT} is the azimuthal width of the TT component, and $I_0(x)$ is the modified Bessel function of order 0 [80].

On the other hand, the longitudinal functions M_R and M_{TT} are more complicated, making it difficult to find closed-form solutions to their integrals in Equation 6.8. Therefore, we compute those integrals numerically and store the results in 1D tables for a number of θ_o values within the range $[-\pi/2, \pi/2]$. Since the resulting values change slowly with changing θ_o values, a low-resolution 1D texture can be used for these values during rendering.

In our implementation, we also add the diffuse component of the simple shading model to this BSDF.

6.4 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 thickness alone makes it indifferent to the fiber type (i.e., core fiber or regular fiber). 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. Thus, 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.

In addition to the 1D texture that stores fiber coordinates R_i and θ_i mentioned above, the only geometry data we send to the GPU are the 3D control points of the yarn-level curves and a 1D *length coordinate* for each one of these control points, which determines the position of the control point along the total length of the yarn. Thus, each cubic yarn curve piece not only knows its length but also knows the total length of the yarn preceding it, so that both yarn and ply twists can be computed consistently for neighboring cubic curves without discontinuity.

Our method is suitable of rendering yarn-level cloth animations at real-time frame rates. At each frame, we only update yarn-level control points and the shadow map, which

only has a minor cost, since the shadow map is rendered using only the yarn curves (without generating fiber-level details). It is important that the length coordinate is not updated for each frame and it is kept constant throughout the animation. Otherwise, minor changes in the length of each cubic yarn curve piece lead to noticeable erratic changes in yarn and ply twists, perceived as temporal incoherence.

6.5 Results

We show the components of our fiber model in Figure 6.9. Notice that yarn with only core fibers looks too regular, as compared to the reference [170]. 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 6.10. Note that our model cannot produce an exact

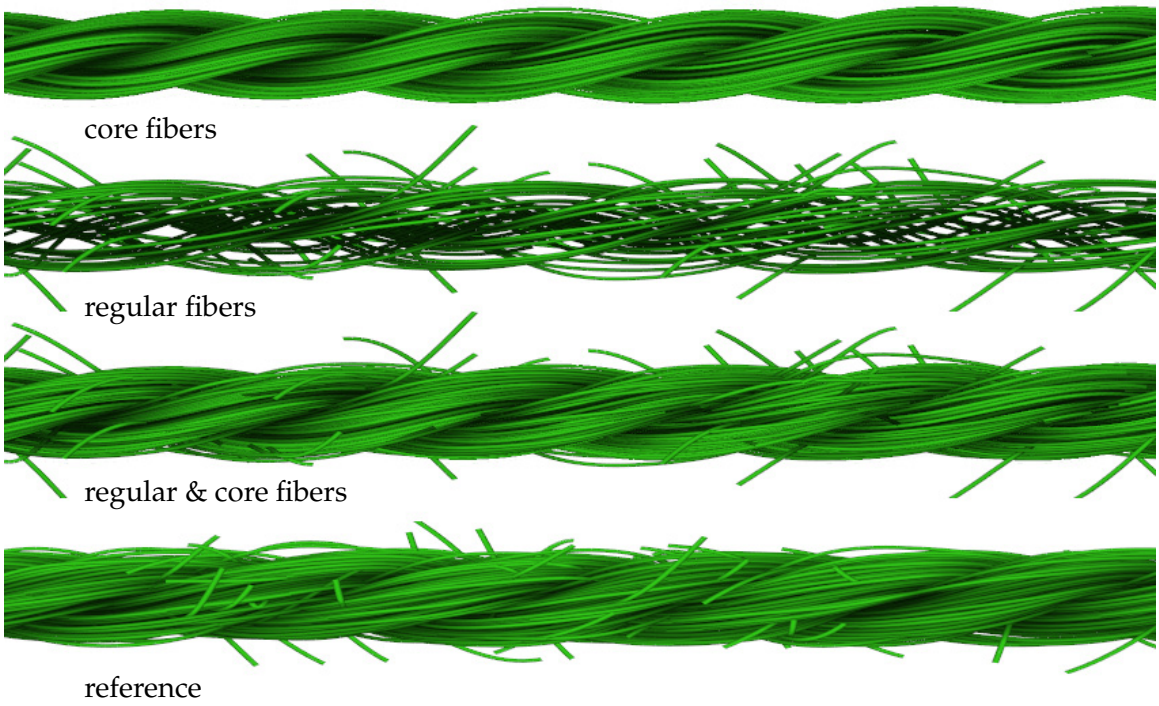


Fig. 6.9: An example yarn model with our core fibers and regular fibers, and the comparison of our full model to a reference model generated using the method of Zhao et al. [170].

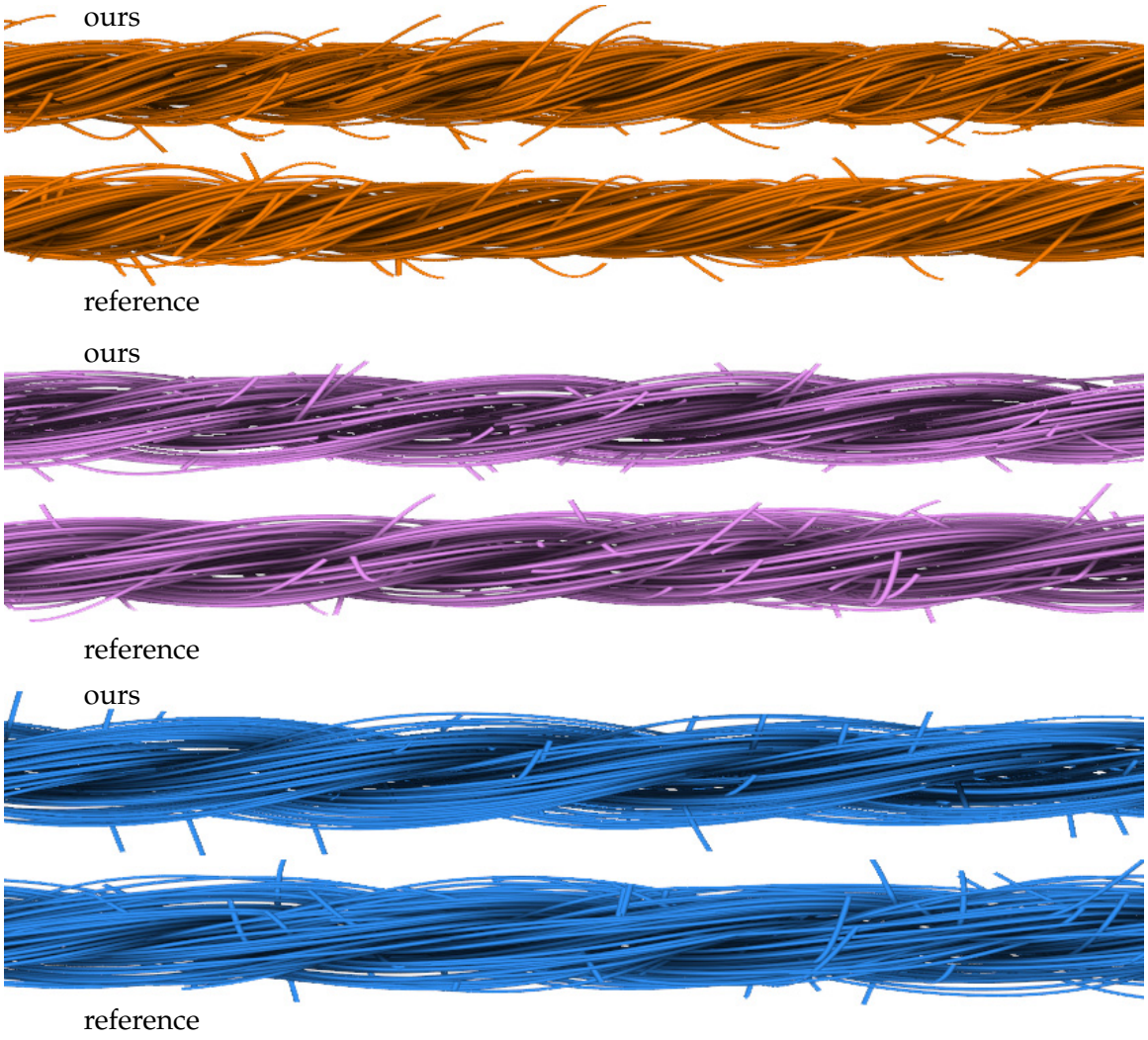


Fig. 6.10: Comparison of our fiber generation method to reference models generated using the method of Zhao et al. [170].

match for the reference mainly because we handle hair fibers differently.

Figure 6.11 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 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 6.1.

We display the contributions of different shadow components on an example glove

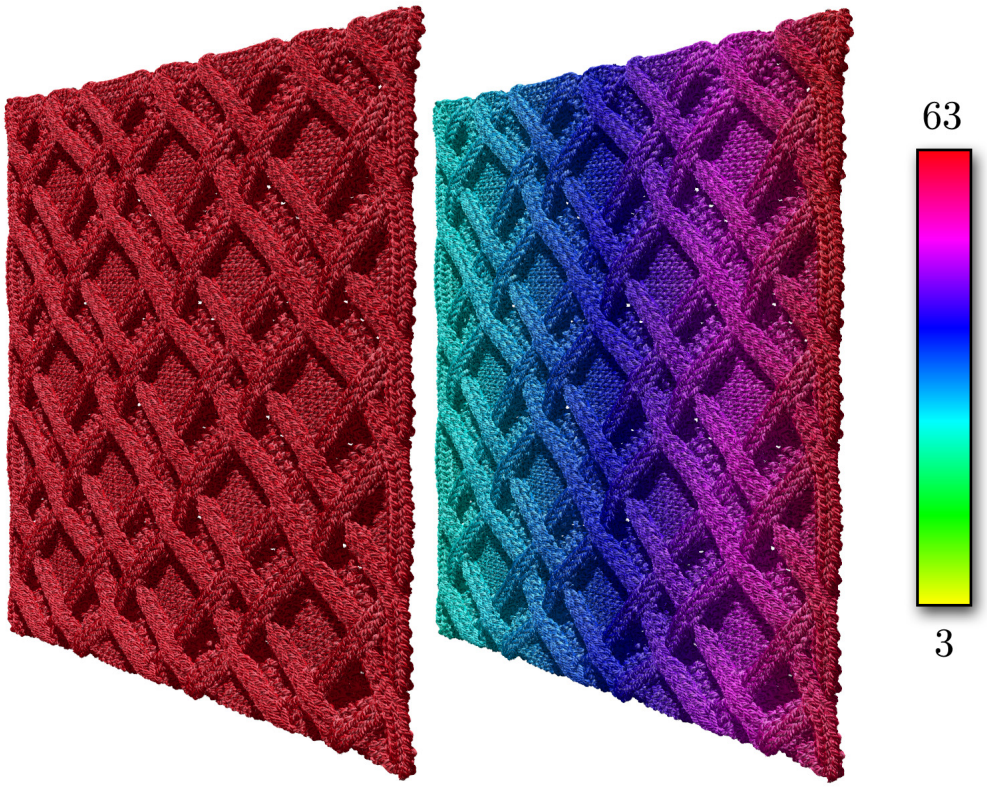


Fig. 6.11: Level-of-detail: (Left) disabling LoD generates 63 fibers everywhere, (right) enabling LoD generates varying numbers of fibers between 3 and 63 based on the fiber thickness in screen space, displayed with color coding.

model in Figure 6.12. In the case of rendering without any shadows (Figure 6.12a), 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 6.12b) accentuate the knitted structure formed by the yarn curves, but they lack fiber-level details, which are introduced by the self-shadows (Figure 6.12c). The ambient occlusion component (Figure 6.12d) 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 6.1 and 6.13 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

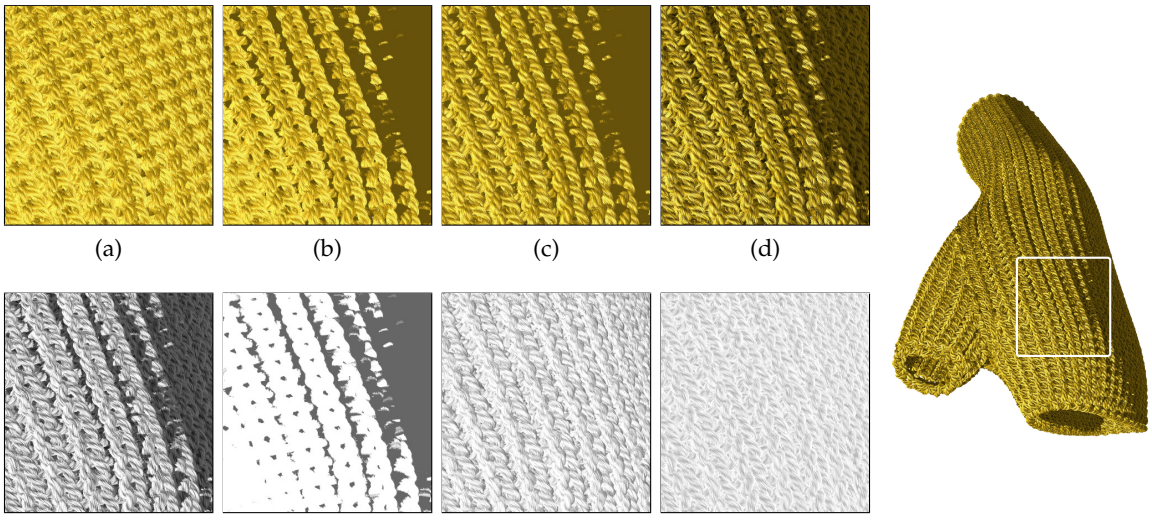


Fig. 6.12: Shadow components: (a) No Shadow, (b) + Shadow Map, (c) + Self-shadows, (d) + Ambient Occlusion, (e) Full Shadow, (f) Shadow Map, (g) Self-shadows, and (h) Ambient Occlusion.

Fig. 6.13: Different yarn geometry: A sweater model with 681K yarn curve control points and 1.5G fiber segments using different procedural yarn parameters, rendered using the same yarn control points, the same simple shading parameters, and the same lighting.

shading parameters, the underlying fiber geometry still impacts the overall appearance.

The components of the physically-based shading model are shown in Figure 6.14 along with the simple shading model for two different materials using the same fiber-level cloth geometry. Notice that the specular components of the physically-based shading model produces natural looking highlights, but the rest of the cloth appears dark without the diffuse component. The ambient illumination with our ambient occlusion factor reveals the material color. The diffuse component also adds the color of the material and it is

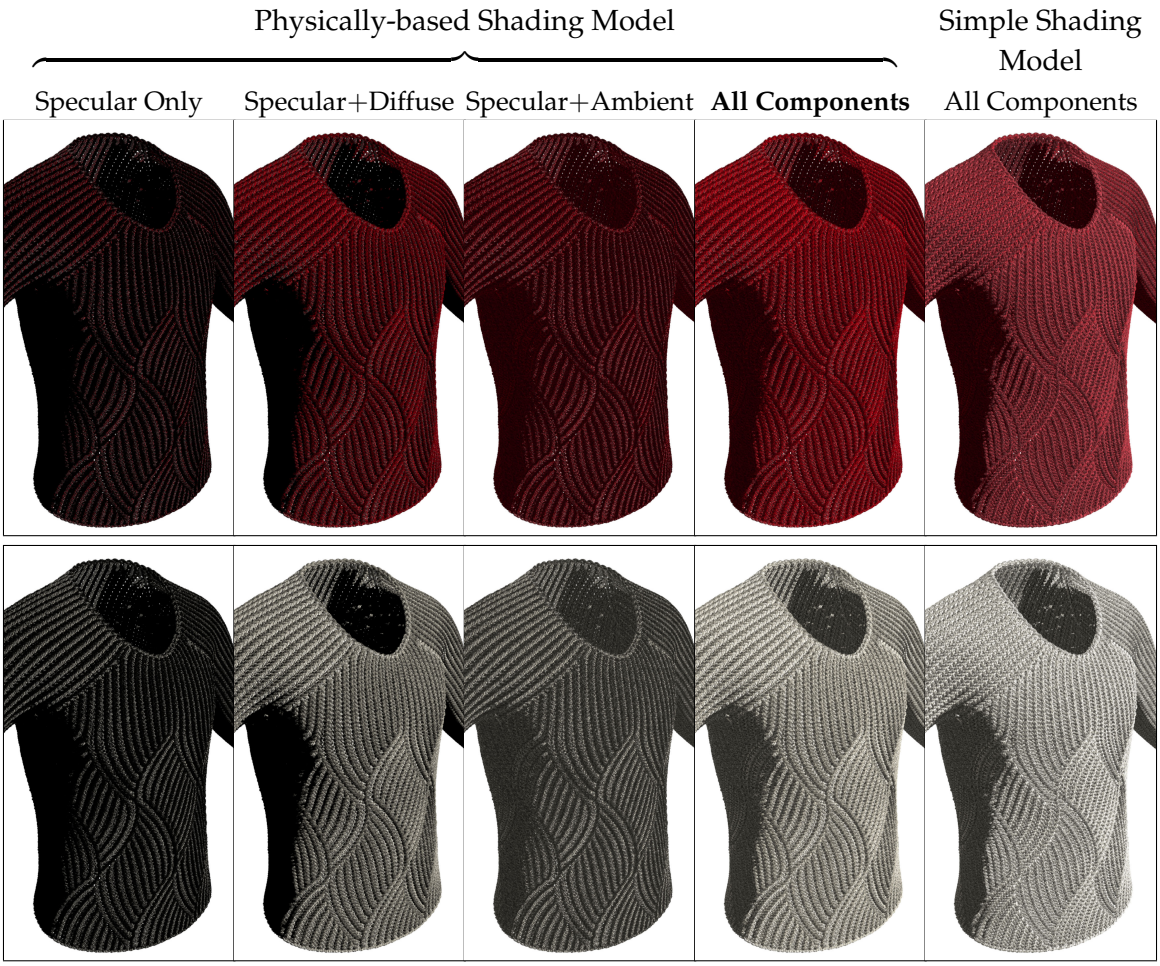


Fig. 6.14: Components of the physically-based shading model: Silk (top row) and cotton (bottom row) material parameters on the same fiber-level sweater model, showing the components of the physically-based shading model along with the simple shading model using the same diffuse component, but different specular and ambient components.

particularly important for exposing the small-scale tubular shape of the fibers in close-up views as shown in Figure 6.15. The final result is generated using all components of the physically-based shading model, which produces natural cloth appearance. In comparison to the simple shading model using the same diffuse color but different specular and ambient formulations, the physically-based shading model produces (arguably) more realistic results due to improved specular highlights.

Example dress models with different yarn-level structures are shown in Figure 6.16, rendered using both physically-based shading model and simple shading model. 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

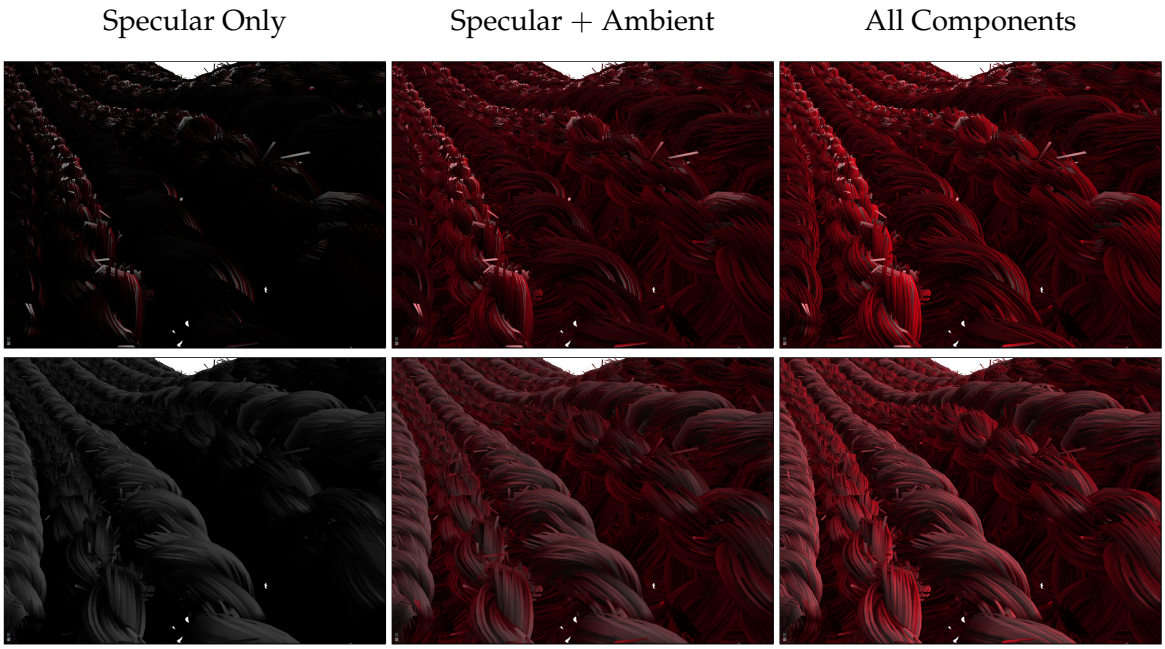


Fig. 6.15: Close-up view: showing the fiber-level details with physically-based shading model and the simple shading model. The top row is using physically-based shading model and the bottom row is using simple shading model.

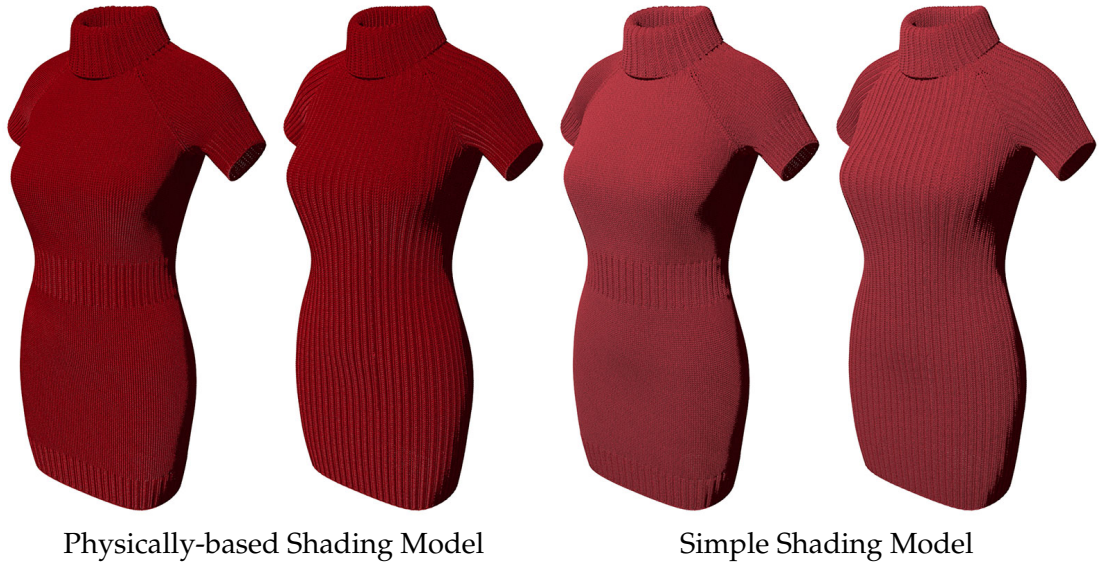


Fig. 6.16: Two dress models with about 2M yarn curve control points and over 100M fiber curves, rendered using the same yarn type and lighting conditions with both physically-based shading and simple shading models. The only differences between the two models are the control points of the yarn curves. The two shading models share the same diffuse component, but they use different specular and ambient components.

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.

We provide the performance graphs for all models in the section in Figure 6.17. 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 6.1 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. As expected, the performance results show that the physically-based shading model is more expensive. Consequently, the worst case performances for the simple shading model and the physically-based shading model do not necessarily appear at the same camera distances, since the performance depends on both the amount of geometry processed and the pixel fragments shaded. As the model moves away from the camera, the performance difference between the two shading models become less significant.

We provide the performance results in Table 6.1 with different models as shown in Figure 6.18 using both the simple shading model and the physically-based shading model. Note that when the camera is far enough that models are fully visible, we generate fewer

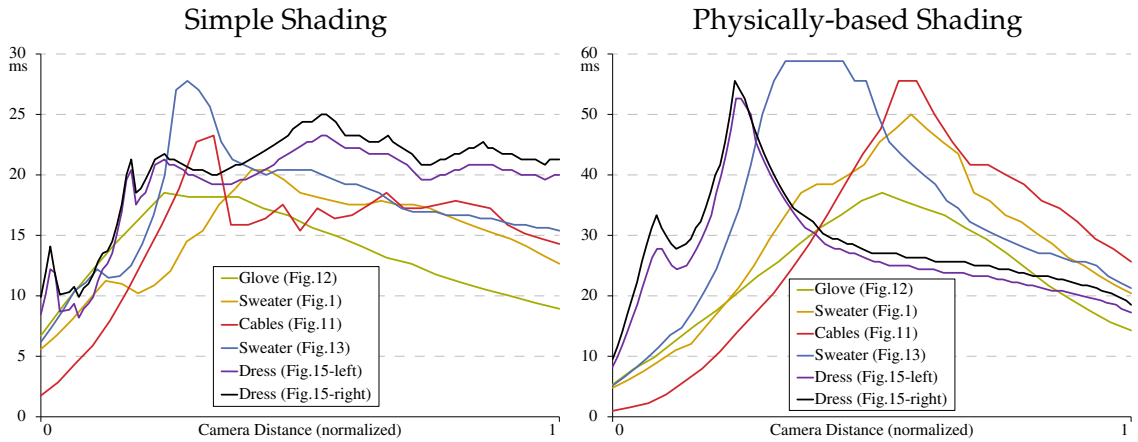


Fig. 6.17: Performance Graphs: The dependence of the frames per second performance values on camera distance for all examples in the section using both the simple shading model and the physically-based shading model. The camera distances are normalized such that the models are fully visible and cover the screen at their maximum distances (full view in Figure 6.18), shown at the camera distance value of 1.

Table 6.1: Performance Results for Different Camera Distances. All performance results are obtained on an NVIDIA GeForce GTX 1080 GPU, rendering to an OpenGL viewport of size 1280 × 960

			Simple Shading			Physically-based Shading		
Model		# of Control Points	Close View	Worst Case	Full View	Close View	Worst Case	Full View
Glove	(Figure6.12)	148K	9 ms	19 ms	9 ms	13 ms	37 ms	14 ms
Sweater	(Figure6.1)	356K	11 ms	23 ms	13 ms	18 ms	50 ms	20 ms
Cables	(Figure6.11)	362K	2 ms	23 ms	14 ms	2 ms	56 ms	26 ms
Sweater	(Figure6.13)	681K	11 ms	28 ms	15 ms	21 ms	59 ms	21 ms
Dress	(Figure6.16-left)	1.89M	9 ms	23 ms	20 ms	24 ms	53 ms	17 ms
Dress	(Figure6.16-right)	1.99M	11 ms	25 ms	21 ms	28 ms	56 ms	18 ms

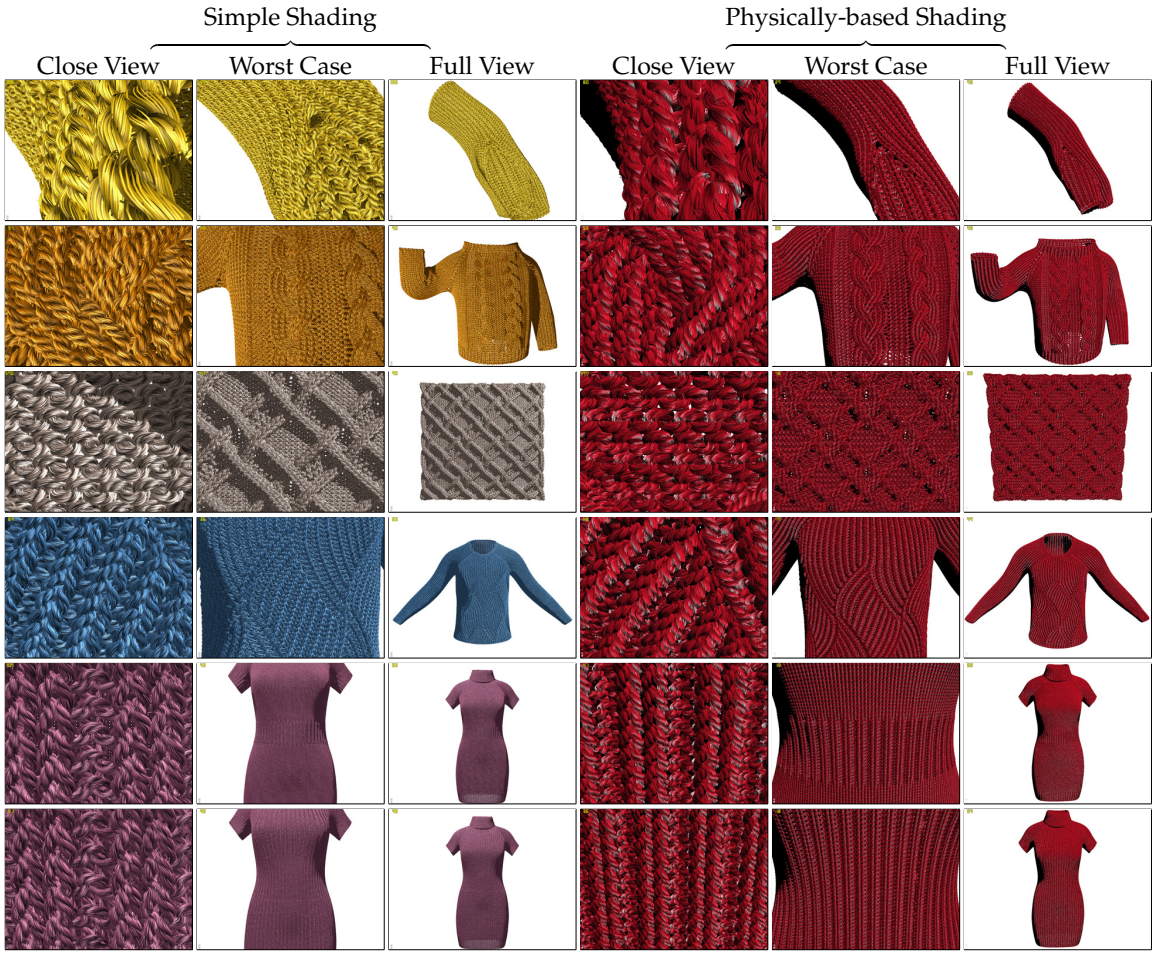


Fig. 6.18: Frames selected for performance results shown in Table 6.1. All images are rendered with 1280 × 960 resolution using the simple shading model and the physically-based shading model.

fibers using our level-of-detail strategy and achieve high performance (the "Full View" column of Table 6.1 and Figure 6.18). Close-ups achieve high performance as well, since we avoid rendering the yarn curves that are outside of the view frustum (the "Close View" column of Table 6.1 and Figure 6.18). 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 "Worst Case" columns of Table 6.1 and Figure 6.18 show the worst performance we encountered by manually adjusting the camera distance.

6.6 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. We have also provided a simple ambient occlusion approximation and an ambient occlusion precomputation approach using the the 2D cross-section of the yarn. In addition, we have demonstrated how our technique can be used with a physical-based fiber shading model. 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.

CHAPTER 7

YARN-LEVEL SIMULATION

The Material Point Method (MPM) is a hybrid Lagrangian/Eulerian computational scheme that has been shown to simulate a large variety of traditionally challenging materials with visually rich animations in computer graphics. Recent examples of MPM-based methods developed for such materials include simulations of snow [132], granular solids [33, 82], multiphase mixtures [42, 133, 136], cloth [72], foam [162], and many others. MPM has been shown to be particularly effective for simulations involving a large number of particles with complex interactions. However, the size and the complexity of these simulations lead to substantial demands on computational resources, thereby limiting the practical use cases of MPM in computer graphics applications.

Using the parallel computation power of today's GPUs is an attractive direction for addressing computational requirements of simulations with MPM. However, the algorithmic composition of an MPM simulation pipeline can pose challenges in fully leveraging compute resources in a GPU implementation. Indeed, MPM simulations include multiple stages with different computational profiles, and the choice of data structures and algorithms used for handling some stages can have cascading effects on the performance of the remaining computation. Thus, discovering how to achieve a performant GPU implementation of MPM involves a software-level design-space exploration for determining the favorable combinations of data structures and algorithms for handling each stage.

In this chapter, we introduce methods for addressing the computational challenges of MPM and extending the capabilities of general simulation systems based on MPM, particularly concentrating on a high-performance GPU implementation. We present a collection of alternative approaches for all components of the MPM simulation on the GPU and provide test results that identify the favorable design choices. We also show that design choices that may superficially appear to be reasonable can suffer from suboptimal

performance in practice. Furthermore, we introduce novel methods for thermodynamics and simulation of granular materials with MPM. More specifically, this chapter includes the following contributions:

1. A novel, efficient, and memory-friendly GPU algorithm for accelerated MPM simulation on a GPU-tailored sparse storage variation of CPU SPGrid [125].

2. A performance analysis of crucial MPM pipeline components, with several alternative strategies for particle-grid transfers.

Our experiments show that more than an order of magnitude performance improvement can be achieved using the favorable choices for data structures and algorithms, as compared to optimized solutions using different computational models. We demonstrate that complex MPM simulations with up to 10 million particles can be simulated within a minute per frame using the methods we describe. We demonstrated million of yarn particles can be simulated efficiently in a few second per frame on modern GPU as shown in Figure 7.1.

7.1 Related Work

In this section, we provide a briefly overview of material point method and previous methods for particle-to-Grid transfer.

7.1.1 MPM Overview

MPM, as a hybrid spatial discretization method, benefits from the advantages of both Lagrangian and Eulerian views. MPM uses Lagrangian particles to carry material states

Fig. 7.1: The yarn-level simulation with 1.2 million particles at an average 5 seconds per 48Hz on Nvidia Titan Xp.

including mass m_p, position $\mathbf{x}_p$, velocity $\mathbf{v}_p$, volume V_p, deformation gradient $\mathbf{F}_p$, etc. The grid acts as an Eulerian scratchpad for computing stress divergence and performing numerical integration. Grid nodes represent the actual degrees of freedom, which store mass m_i, position $\mathbf{x}_i$ and velocity $\mathbf{v}_i$ on each node i.

A typical first-order MPM time integration scheme for incremental dynamics from t^0 to t^1 (with $\Delta t = t^1 - t^0$) contains the following essential steps:

1. Particles-to-grid (P2G) transfer of masses and velocities:
   $\{m_i^0, \mathbf{v}_i^0\} \leftarrow \{m_p, \mathbf{v}_p^0\}$;

2. Grid velocity update using either explicit or implicit integration: $\mathbf{v}_i^1 \leftarrow \mathbf{v}_i^0$;

3. Grid-to-particles (G2P) transfer of velocities and strain increments: $\{\mathbf{v}_p^1, \mathbf{F}_p^1\} \leftarrow \mathbf{v}_i^1$;

4. Particle-wise stress evaluation and plasticity treatment that modifies $\mathbf{F}_p^1$.

Assuming the usage of a matrix-free Krylov solver for the linearized system (due to its superior efficiency in implicit MPM, where dynamically changing sparsity pattern of the stiffness matrix could cause significant overhead in explicitly storing the corresponding linear system), both explicit and implicit Euler time integrations of MPM break the entire computation procedures in (1)-(3) into particle-grid transfer operations of physical quantities and their differentials. As such, the key to high-performance MPM is the optimization of particle-grid transfer operators.

Unsurprisingly, the algorithmic choice of the transfer scheme plays a considerable part in affecting the computing performance. We design our benchmarks based on three possible choices of the transfer scheme: FLIP [174], APIC [73] and the recently proposed MLS-MPM [64]. Note that MLS-MPM provides an additional algorithmic speed-up by avoiding kernel weight gradient computations in step (2) (discussed in more detail in §7.3 and by Hu et al. [64]).

7.1.2 Particle-to-Grid Transfer

As in many other Lagrangian-Eulerian hybrid methods, the particles carry material states and transfer them to the grid as mentioned in §7.1.1 step (1). On the GPU, the two most common approaches for performing parallel state transfer are *gathering* and *scattering*.

The former one gathers states from all nearby particles to one particular grid node; while the latter one distributes all states of one particular particle to all influenced grid nodes.

The domain decomposition technique is commonly used in the gathering strategy [65, 111, 130, 168]. Using this strategy the simulation domain is divided into several small subdomains such that each node only needs to check all the particles within the neighboring subdomains instead of the whole domain. Binning is used for avoiding conflicts in atomic operations and accessing each particle only once [81], but the precomputation can be very expensive and having a different number of particles per cell leads to thread divergence. Ghost particles are introduced to minimize the number of communications and barriers within subdomains [111, 121]. Chiang et al. [23] maintain and update particle lists for each node during the simulation. Obviously, it requires a huge amount of GPU memory as well as the processing time for updating particle lists. In order to reduce the workload, Dong et al. [39] extend the influencing range of the grid nodes and further adapt their method to multiple GPUs [38]. Wu et al. [157] precompute a subset of the particles for each grid cell. However, their method still needs to unnecessarily examine a large amount of particles within each voxel. Furthermore, for storing the subsets, it not only requires expensive data movements but also consumes a large amount of memory. To summarize, all the fundamental issues of gathering are due to the need for accessing associated particles of a node. There is no satisfying solution reading the data efficiently as well as saving the extra memory for particle lists, etc. The other inherent problem is that the workload of each GPU thread is closely related to the length of the particle list of each grid node, and thus thread divergence generally exists and slows down P2G.

On the contrary, the scattering method is free from all these critical issues, but its performance suffers from write conflicts. Once the write conflicts are resolved, the scatter-based transfer scheme should be superior, because it has more balanced workload and it also avoids the overhead of maintaining particle lists. Both Harada et al. [58] and Zhang et al. [167] introduced parallel scattering methods to distribute particle attributes to neighboring particles in SPH. In their methods, particle attributes are written into 2D texture buffer simultaneously using graphic rasterization pipeline to solve write-conflicts. However, their scattering methods cannot be easily employed in a 3D Eulerian simulation framework because the current hardware rasterization technique only works for

2D texture, not to mention the added complexity to build a mapping between a sparse grid and 2D texture during rasterization. Fang et al. [41] and Hu et al. [64] recently proposed an asynchronous MPM simulation method for acceleration on the CPU. Their implementation adopted the parallel scatter-based scheme and thus also suffered from the data race during the particle-to-grid transfer. To avoid the expensive per-cell locking and guarantee correctness as well, they partitioned all the blocks into several sets, in each of which no two blocks share any overlapping grid node. But this method only prevented the write conflicts between blocks, since blocks were used as the scheduling units on the CPU, and it cannot resolve the conflicts when multiple particles are writing to the same grid node simultaneously, which is the case on the GPU. We propose a novel scatter-based GPU implementation of transfer kernels, which alleviates the heavy use of atomic operations by utilizing modern graphics hardware features, and thus avoids the majority of the overhead.

7.2 Optimized GPU Scheme for MPM

In this section, we describe our optimized GPU scheme for MPM using a GPU-tailored sparse paged grid data structure. The overall algorithm is summarized in Algorithm 2.

7.2.1 GSPGrid Tailored to MPM

We introduce *GSPGrid*, the GPU adaptation of the SPGrid [125] data structure that facilitates the sparse storage required for efficient simulations. Though not limited to such

ALGORITHM 2: Sparse MPM simulation on the GPU

$P \leftarrow$ initial points	
$P \leftarrow$ sort(P)	Section 7.2.3
for *each timestep* **do**	
$dt \leftarrow$ compute_dt(P)	
$V \leftarrow$ GSPGrid structure(P)	Section 7.2.1
$M \leftarrow$ particle_grid_mapping(P, V)	Section 7.2.1
$V \leftarrow$ particles_to_grid(P, M)	Section 7.2.2
$V \leftarrow$ apply_external_force(V)	
$V \leftarrow$ grid_solve(V, dt)	
$P \leftarrow$ grid_to_particles(V, M)	
$P \leftarrow$ update_positions(P, dt)	
$P \leftarrow$ sort(P)	Section 7.2.3

a choice, a $4 \times 4 \times 4$ spatial dissection of the computational domain into GSPGrid blocks is assumed in our implementation.

We take a similar strategy to [43], and briefly summarize it here. We use the quadratic B-spline functions as the weighting kernel, so each particle is associated with $3 \times 3 \times 3$ grid nodes (3×3 in 2D as shown in the Figure 7.2). We assign each particle to the cell whose "min" corner collocates with the "smallest" node in the particle's local $3 \times 3 \times 3$ grid. All particles whose corresponding cells are within the same GSPGrid block are attached to that block. Geometrically, the particles of a particular GSPGrid block all reside in the same dual block, i.e., the $\Delta x/2$-shifted block (the red dashed block in Figure 7.2). All computations for transferring particles' properties to/from the grid will be conducted locally in the same GSPGrid block.

We assign each particle to one CUDA thread. The particles are sorted such that the computations of particles sharing the same dual cell are always conducted by consecutive threads. In scenarios involving high particle densities, the number of particles within a

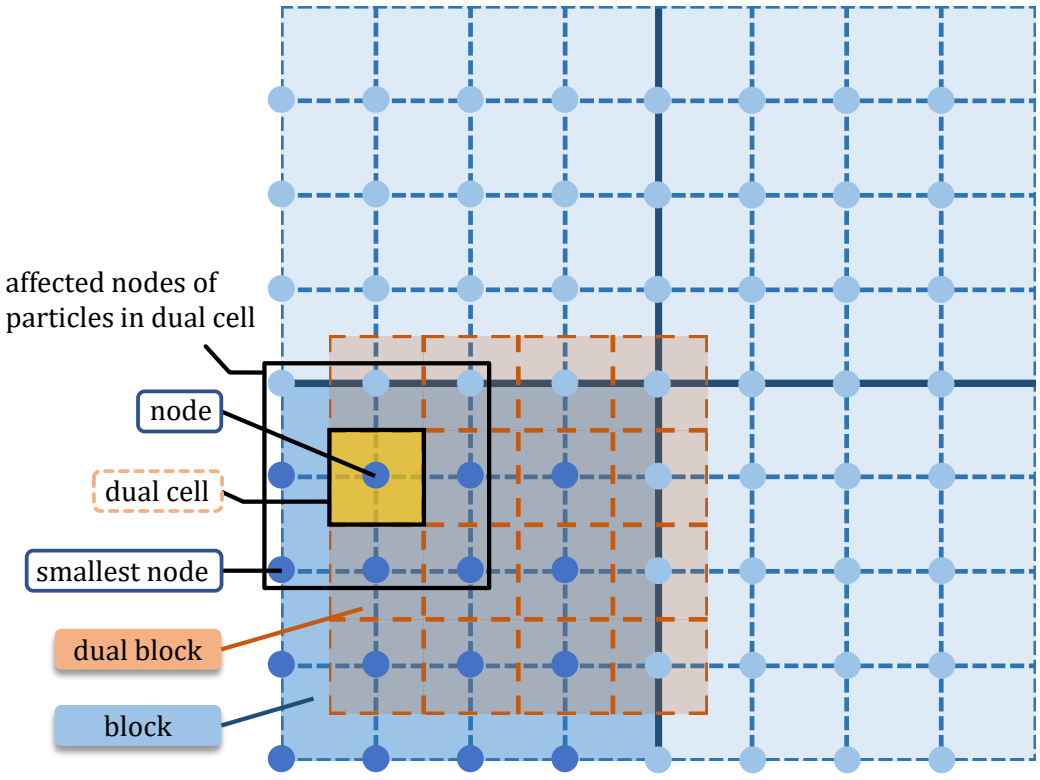


Fig. 7.2: Mapping from particles to blocks: All particles in the yellow dual cell interact with the same set of 27 nodes (9 in 2D). These particles also distribute states to the grid nodes of the top neighboring block.

single GSPGrid block can easily go beyond the maximum number of threads allowed in a CUDA block under the current graphic architecture. To solve this issue, we assign each GSPGrid block to one or several CUDA blocks and generate the corresponding *virtual-to-physical* page mapping. In this way, we can treat each CUDA block separately without considering their geometric connections.

Notice that, for some particles, e.g., the ones in the yellow cell in Figure 7.2, reading and writing will involve nodes from neighboring blocks. To deal with such particles, the shared memory of each CUDA block temporarily allocates enough space for all 7 neighboring blocks (3 in 2D). One CUDA block usually handles hundreds of particles in parallel, which makes it affordable to allocate enough shared memory for all neighboring blocks. For example, in grid-to-particle transfer, we can first fetch all required data into the shared memory from the global grid, and then update particles' properties.

As discussed above, it is possible for a particular block to access data from its neighboring blocks. In SPGrid, the offsets of the neighboring blocks can be easily computed for addressing them. However, without the support of virtual memory space in GPU, we also need to store the address information of neighboring blocks. Fortunately, the spatial hashing algorithm is able to construct the topology of neighboring blocks in $\mathcal{O}(1)$ complexity on the GPU.

7.2.2 Parallel Particle-to-Grid Scattering

For particle-to-grid transfers, geometrically neighboring particles can write into the same nodes, making write hazards a critical problem which has been examined in many Eulerian-Lagrangian hybrid methods as discussed in Section 7.1.2. There are generally two schemes for resolving this problem, scattering and gathering. Scattering methods simply use atomic operations to avoid conflicts. On the other hand, for gathering, usually a list of particles are created and maintained during the simulation; thus, each node can track down all the particles within its affecting range.

Almost all previous papers opt for a gathering approach, since it is widely believed that highly frequent atomic operations in scattering can significantly undermine the performance especially in GPU-based parallel applications. However, we propose a method to handle most of the write conflicts from scattering without atomic operations, inspired

by the concept of parallel reduction sum [96] and show that the performance is superior to the gathering ones. As we divide the whole grid domain as well as their corresponding particles into blocks and cells, there are two levels of write hazards in P2G, within-block hazards and crossing-block hazards. As observed in practice, the within-block conflicts are the absolute majority. The within-block ones can be further divided into within-warp and crossing-warp conflicts and our solution particularly tackles the within-warp ones.

As shown in Figure 7.3, within a warp, a few groups of particles tend to add their attributes to the corresponding grid nodes. For particles of a particular group (e.g., particles 1-4), simultaneous write operations into the same location are conventionally done through atomic operations. We exploit the warp-level CUDA intrinsic functions, i.e. *ballot* and *shfl*, to resolve this problem. First, a representative of each group is chosen (i.e. the left most one whose boundary mark is set on). Then attributes from all the other particles within the same group are added to the one from the representative by iteratively shuffling from right to left. The stride of shuffle doubles after each iteration, and the total number of iterations should be enough to cover the longest region in the warp by using Algorithm 3. Finally, the representative particle is responsible for writing the sum

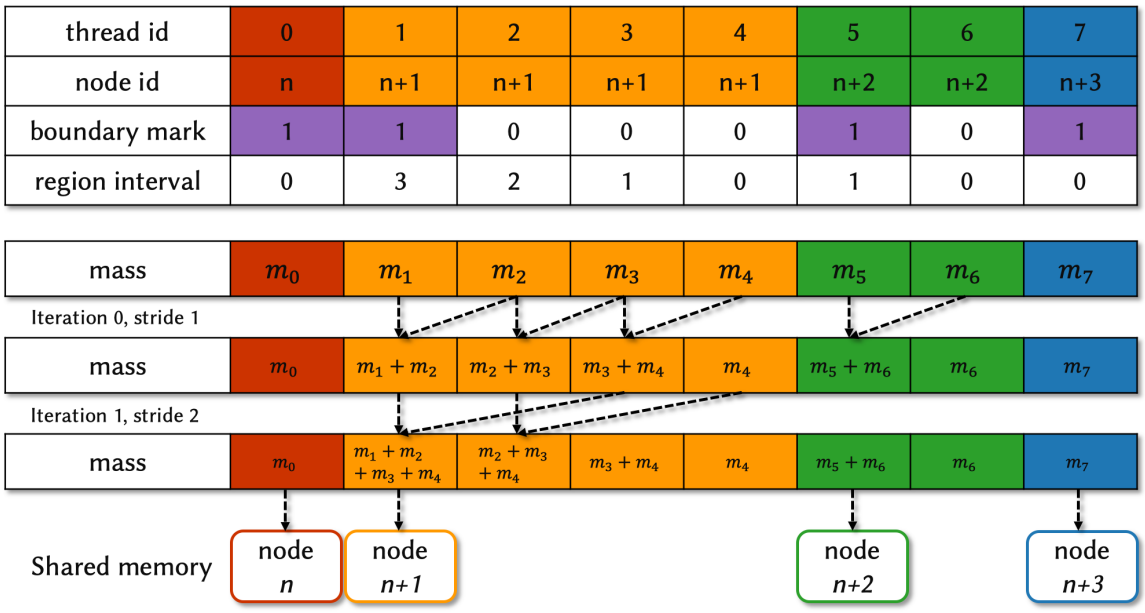


Fig. 7.3: Optimized particles-to-grid transfer. By using CUDA warp intrinsics to accelerate attribute summations, results are first written to the shared memory and then transfered to global memory in bulk.

to the target grid node, as illustrated in Algorithm 4. In this way, all warp-scale write conflicts are eliminated. To further reduce the global write conflicts across different warps, the shared memory acts as a buffer for temporarily holding the summation results from different warps. Note that this crossing-warp conflicts only infrequently happen such that the negative impact caused by atomic-adds to the shared memory is almost negligible.

7.2.3 Cell-Based Particle Sorting

Since the positions of the particles are updated in each time-step, the GSPGrid data structure should be refreshed accordingly. To build the new mapping from the GSPGrid blocks to the continuous GPU memory, we need first to re-sort the particles to help identify all the blocks occupied or touched by the particles.

ALGORITHM 3: ComputeBoundaryAndInterval(**int** *laneid*, **int**∗ *cellids*)

```
cellid ← cellids[laneid]
cellid_prev ← cellids[laneid − 1]
if laneid = 0 ∨ cellid ≠ cellid_prev then
    boundary ← true
mark ← brev(ballot(boundary))
interval ← countFollowingZeroes(mark, laneid)
stride ← 1
iter ← interval
while stride < 32 do
    tmp ← shfldown(iter, stride)
    iter ← max(tmp, iter)
    stride ← stride << 1                    // move on to a higher level
iter ← shfl(iter, 0)                        // broadcast the max iteration
```

ALGORITHM 4: GatherAndWrite(**T**∗ *buffer*, **T** *attrib*, **int** *iter*)

```
stride ← 1
val ← attrib
while stride <= iter do                     // hierarchical summation
    tmp ← shfldown(val, stride)
    if stride <= interval then              // only sum within the group
        val ← val + tmp
    stride ← stride << 1                    // move on to a higher level
if boundary then                            // only write boundary node
    ∗buffer ← ∗buffer + val                 // apply AtomicAdd
```

SPGrid translates the cell index of each particle to a 64-bit integer offset, which is used as the keyword for sorting. Radix sort is generally considered the fastest sorting algorithm on GPU. In our simulations, the count of offsets in use is almost negligible compared to what a 64-bit integer can represent. We therefore use spatial hashing to transform these sparse offsets of GSPGrid blocks into consecutive numbers. Whenever a new block is touched, the index assigned to the block increments by one (starting from 0). The transformed block indices of all the particles are thus consecutive, and the maximum number is usually several orders of magnitude smaller than the particle number. Furthermore, all particles within the same block can be partitioned by $4 \times 4 \times 4$ cells. This particular layout is required by the follow-up computations including precalculating the smallest index of each particle and reducing write conflicts during P2G transfer (Section 7.2.2).

With CPU SPGrid, radix sort facilitates proximity (in memory) of geometrically neighboring blocks to improve prefetching efficiency. In GPU, this is not a concern anymore; thus we use histogram sort instead of the conventional radix sort. The new keyword, which is a combination of the transformed block index and dense cell index within the block, works as the reference to the bin. The observed performance speedup approaches an order of magnitude, and in collaboration with *delayed ordering technique* (Section 7.2.4), reordering the particles is no longer a bottleneck.

7.2.4 Particle Reordering

Given the sorted indices from (Section 7.2.3), one natural thing to do next is to reorder all particles' properties accordingly since coalesced memory accesses are always preferable in CUDA kernels. However, for a high-resolution simulation, the reordering itself can be the bottleneck, since each particle carries various properties such as positions, velocities, and deformation gradient, etc., and reading and writing those data in a noncoalesced manner can be time consuming. We propose a practical way to completely dispense with the reordering. We observe that any functions which take in scattered data and write back the updated results in order can actually sort the data as a byproduct. In essence, a reordering function simply writes unordered inputs back in a different order. If we apply the reordering function to the particle property optionally rather than sorting all the properties of a particle, the cost / latency due to the scattered memory reads can be

largely mitigated and the overhead of reordering is avoided.

The position of a particle is the only property we decide to reorder since they are essential for almost all kernels that build mappings between particles and grid blocks / cells, and for all transfer kernels that needs to compute weights and weight gradients from the positions. We treat all the other properties accordingly based on how they are used in the related computations. We list a few of them as the most typical cases. Note that most CUDA kernels follow the positions' order, except for certain material-based computations.

- Mass. Since each particle's mass remains constant, their sequence will never be changed; for retrieving the correct mass, we only need to map from the current particles' order to the original order.

- Velocity. Velocities are updated every time-step in the G2P kernel. The new velocities are forced to follow the order of the threads / positions by simply writing them back in a coalesced manner. However, when the velocities are being used as the inputs to some kernels in the next time-step, they are not matching with new positions' order since we choose only to reorder positions at the end of each time-step. As a result, a mapping from the current order to the previous order is required.

- Stress. When computing the stresses, we choose not to change the order of either the inputs $\mathbf{F}$ or the outputs $\mathbf{P}$. One reason is that the GPU SVD function can be optimized (Section 7.3.3) so fast that it may be inefficient to amortize the scattered writes.

7.3 Benchmarks and Performance Evaluation

To evaluate our GPU MPM algorithm, we create several benchmarks, starting with the uniform particle distribution (Section 7.3.1). Firstly, we compare the performance of the transfer kernels between our method and one SIMD-optimized multicore CPU implementation as well as one gathering-based GPU implementation with GVDB as the sparse background grid. Secondly, with the total number of particles being fixed, we vary the particle densities (particles per cell, PPC) to examine how our pipeline can be affected.

We further evaluate the performance with nonuniform particle distributions, e.g., Gaussian distributions (Section 7.3.2). In this experiment, we fix both the total number of

particles and the total number of cells being occupied by the particles; while the PPC of each single cell can vary following particular Gaussian distributions.

In addition to the explicit pipeline, we also measure the performance of the key kernels merely used in the implicit time integration [132]. Since the results of the singular value decomposition (SVD) of the deformation gradient are required in the computation of the stress derivative in every single iteration, one can always precompute the SVD and store the results in advance. However, our experiment (Section 7.3.3) reveals some interesting findings.

Notice that, as mentioned before, the 3D GSPGrid block with 16 channels is of resolution $4 \times 4 \times 4$ and eight particles per cell are usually required for stability considerations in MPM applications. Hence, we choose to allow each CUDA block to process at most 512 particles, due to the limitation of the current hardware architecture.

Unless otherwise stated, all of our GPU tests are performed on Quadro P6000 and all CPU tests are performed on an 18-core Intel Xeon Gold 6140 CPU.

7.3.1 Uniform Distribution Benchmarks

We first introduce uniform distribution benchmarks to compare with state-of-the-art implementations on CPU and GPU and then fix the total number of particles to compare methods with different particle densities.

7.3.1.1 Comparisons with State-of-the-Art Implementations

We create a benchmark with $\Delta x = \frac{1}{128}$. The particles are uniformly sampled on a grid from $(\frac{1}{8}, \frac{1}{8}, \frac{1}{8})$ to $(\frac{7}{8}, \frac{7}{8}, \frac{7}{8})$ with spacing $\frac{1}{256}$. The total number of the particles is just over 7 million particles. For this test, the CPU benchmark was performed on an 18-core Intel Xeon Gold 6140 CPU and all GPU measurements were performed on a NVIDIA TITAN Xp. The results are in Figure 7.4.

- Comparison with CPU implementation. We compare our scatter-based GPU implementation of particle-grid transfers to SIMD optimized CPU implementations of FLIP [43] and MLS [64]. In our tests, our P2G and G2P kernels achieved more than $16\times$ speedups in comparison to the CPU implementation of FLIP [43], and about $8\times$ and $13\times$ speedups as compared to the CPU implementation of MLS [64],

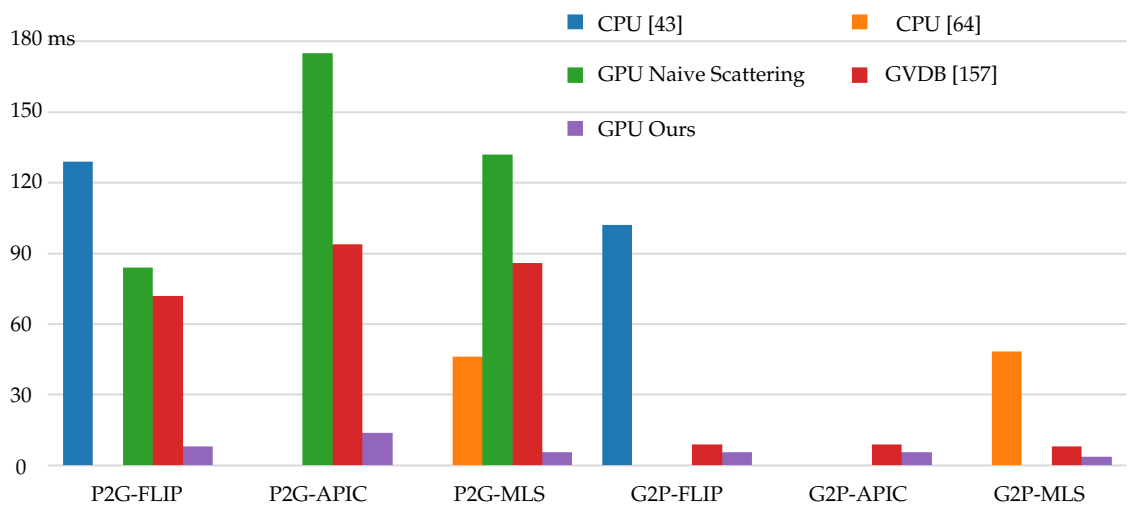


Fig. 7.4: Transfer benchmark. Comparison of our GPU scattering to a SIMD CPU implementations of FLIP [43] and MLS [64] transfer schemes, a naive GPU scattering implementation using atomic operations, and a GPU gathering implementation using GVDB [157] on Nvidia TITAN Xp.

respectively.

- Comparison with GPU implementation with atomic operations. We perform comparisons against an atomic-only scattering implementation, which is an order of magnitude slower than our proposed method as shown in Figure 7.4. Of course, the fact that we did not find this route to be attractively efficient in current platforms, is no indication that the performance of atomics will not be faster in future generations. However, since MPM for simulating solids needs quadratic kernels to acquire continuous force fields, the amount of conflicts encountered is usually much more than expected.

- Comparison with GVDB-based implementation. We compare with one of the most recent GPU gathering implementation with sparse grid structure [157]. Their idea is basically to precompute a subset of particles for each grid cell. All particles influencing the grid nodes inside the cell will be included in that list. When performing the particle-to-grid transfer, each node needs to check all particles in the list to determine whether they are close enough. Therefore, each node has to check much more unnecessary particles than needed (only 20% utilization for MPM FLIP with

subcell size 4^3). In order to do a comprehensive comparison, we use three different transfer schemes, including FLIP, APIC, and MLS. Since their gathering method needs to create lists for all particle attributes, such as velocity, position, stress, and deformation gradients, it takes half of the computation time to load and store data. More importantly, it consumes tremendous amounts of GPU memory to store the neighboring particle indices.

For our P2G kernels, the computing workload and the memory access workload are well balanced. Therefore, the timing is bounded by both memory and computations. FLIP only needs to compute nodal mass, traditional translational momentum and forces; while APIC also needs to load one additional matrix for the affine velocity modes. In contrast, MLS completely avoids the computation of weight gradients, and the two matrix-vector multiplications of APIC (i.e., computing the force and the affine modes) can be merged into one [64].

For our G2P kernels, memory is utilized more heavily than computing units. Compared to APIC and MLS, FLIP also needs to load the nodal velocity increments for updating the particles' velocities. However, APIC and MLS have to refresh one extra matrix for recording the affine velocity modes; while MLS can merge the updates of $\mathbf{F}$ and that extra matrix into one to reduce the total cost [64]. GVDB uses a 3D texture to store volume data to utilize the hardware trilinear texture interpolation functions; however, MPM cannot benefit from this because of the higher order B-spline weighting functions. Furthermore, we exploit the shared memory to preload grid data for all particles within the same block.

7.3.1.2 Particle Density Benchmark

In this subsection, while fixing the total number of particles, we run the benchmarks for cases with different particle densities (particles per cell, PPC). And we start to include all other critical kernels in addition to the transfer kernels; all tests are conducted using the MLS transfer scheme. As shown in Figure 7.5, it is reasonable to observe that when the particle density increases, the transfer kernels take less time to finish, since the higher PPC renders a smaller sparse grid structure. For the other kernels, which are mostly particle-oriented, i.e., the underlying grid structure does not really interfere with them, the impacts of the varying PPC seem to be negligible.

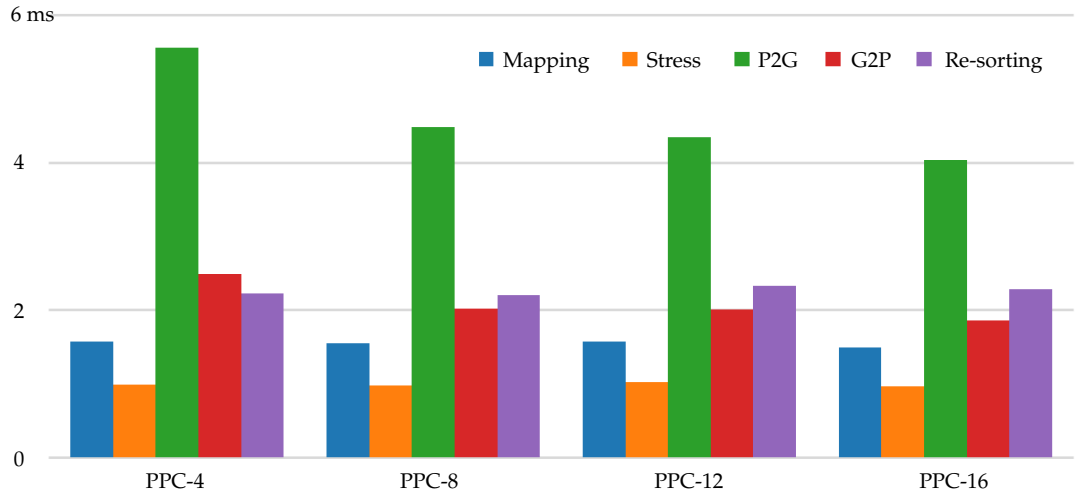


Fig. 7.5: Particle density benchmark. The total number of particles is approximately fixed as 3.5M. The stress kernel includes the SVD computation.

7.3.2 Gaussian Particle Distribution Benchmarks

To further examine the impacts of non-uniform particle distributions, we also run some benchmarks in which the particle-per-cell varies based on a Gaussian distribution. All tests are with MLS. We use the same box domain from $(\frac{1}{8}, \frac{1}{8}, \frac{1}{8})$ to $(\frac{7}{8}, \frac{7}{8}, \frac{7}{8})$. For the two Gaussian distributions, the minimum particle-per-cell are 4 while the maximums are 16 and 32; while the corresponding uniform cases are with particle-per-cell being 10 and 18 respectively. As shown in Figure 7.6, the performances are almost identical, proving that our scheme is not affected by the particle distribution when the background sparse grid remains the same.

7.3.3 Implicit Iteration and SVD

We adopt the matrix-free Krylov solver for the implicit step in which the multiplication of the system matrix and a vector can be expressed by concatenating a G2P transfer and a P2G transfer (ref.[132] for more details). Notice those two transfer kernels are not the same as the ones used in the explicit MPM solver. We name them as P2G-Implicit and G2P-Implicit kernels as in Figure 7.7. Both FLIP and APIC (non-MLS) have to compute weight gradients while MLS approximates weight gradients with weights. From the right half of Figure 7.7, the G2P-MLS is slightly slower than G2P-non-MLS because G2P-MLS needs to do additional 9 multiplications due to the extra term $(\mathbf{x}_p - \mathbf{x}_i)$ at all 27 nodes.

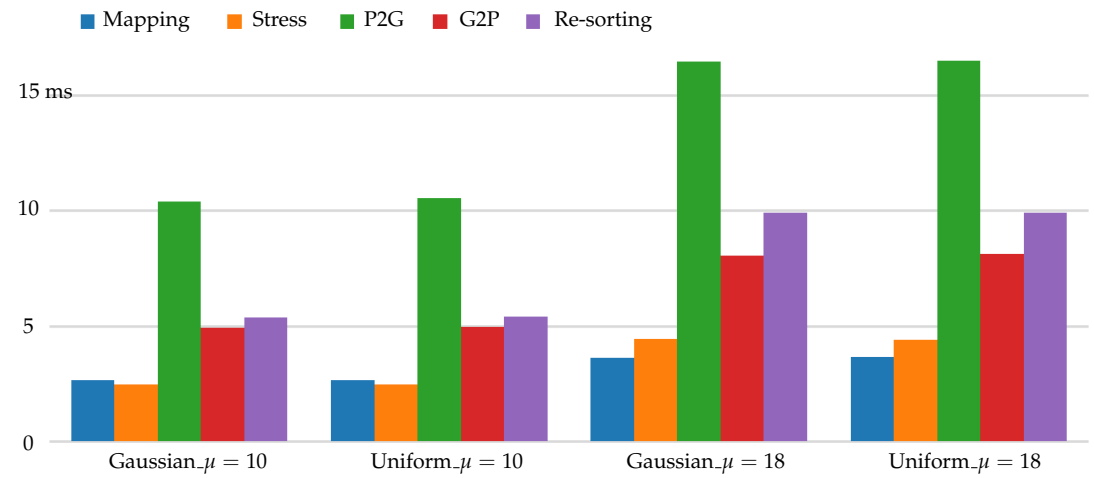


Fig. 7.6: Gaussian benchmark. We compare the performance of each critical kernel when the particle-per-cell distributions following Gaussian and uniform distributions. The stress kernel also includes the SVD computation.

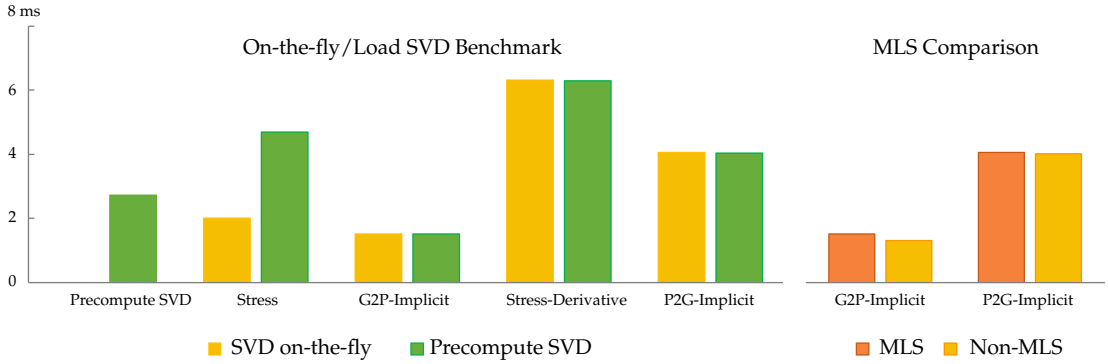


Fig. 7.7: On-the-fly/load SVD benchmark and MLS comparison. Left, when we precompute SVD and store the results, the stress and stress-derivative kernels load the SVD results directly from the memory; otherwise they recompute SVD on-the-fly; right, we compare the performance for one implicit iteration of MLS and non-MLS implicit integrations.

We also consider the possibility of precomputing and storing the results of SVD at the beginning of each time step. Whenever the stress kernel or stress-derivative kernel needs, we simply load the SVD results from memory. Notice that, for stress kernel, it is faster to simply recompute SVD repeatedly; while for stress-derivative kernel, there is only negligible difference. The main reason for this discrepant behavior in the two kernels is that the computing workload in Stress kernel is already lighter than the memory workload, loading more data in can significantly impede the performance. On the other hand, Stress-derivative has enough computing workload to mitigate the memory cost for loading SVD

results.

In the same CPU used in Section 7.3.1.1, a AVX512 SVD implementation of [99] takes about 2.3 ns per particle (SVD) and implicit symmetric QR SVD [47] takes 17.0 ns; while our GPU implementation takes about 0.37 ns. Our GPU implementation is six times faster than the state of art CPU implementation.

7.3.4 Reorder Benchmark

We compare the performances of 7M particles example with particle reordering and without particle reordering (i.e., delayed reordering) using the explicit integration. As shown in Figure 7.8, computing stress, P2G, and G2P are barely affected by reordering, while our method only reorders the particle positions rather than all attributes such as velocity and deformation gradient.

7.4 Results

In addition to the benchmarks, we also demonstrate the efficiency of our GPU implementation and the efficacy of our new heat discretization and the semidefinite sand model with several simulation demos. We list the performance and the parameters used in these simulations in Table 7.1 and 7.2. Note that all particles are sampled using Poisson Disk [17] for uniform coverage. For the Krylov solver, which are typically used for implicit MPM, each iteration consists of a call to the P2G kernel and a call to the G2P kernel. Detailed

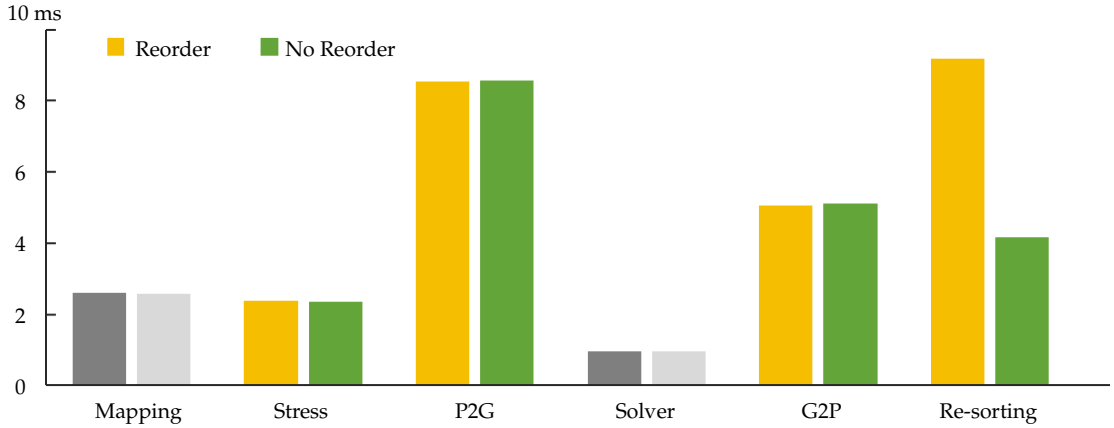


Fig. 7.8: Reorder benchmark. Our delayed ordering technique can reduce sorting time dramatically while all other kernels are barely impacted. Only colored components are impacted by reordering.

Table 7.1: Resolution and time step for each example

	Particles #	Domain	Δt
Dragon Cup	9.0M	512^3	1×10^{-4}
Granulation	6.7M	512^3	2.5×10^{-4}
Gelatin	6.9M	512^3	1×10^{-3}
Melting	4.2M	256^3	8×10^{-3}

Table 7.2: Average simulation time per frame. Timings are in seconds and frame rate is 48.

	Mapping	Stress	P2G	Solver*	G2P	Sorting	Others	Total (s)
Dragon Cup	0.64	0.57	2.30	13.94	1.00	1.35	1.15	**20.95**
Granulation	0.26	1.34	1.88	33.47	0.74	0.41	0.32	**38.42**
Gelatin	0.08	0.05	0.26	5.50	0.10	0.26	0.02	**6.27**
Melting	0.02	0.01	0.07	10.32	0.02	0.03	0.01	**10.48**

timings for the solver are shown in Table 7.3. Others section includes vector addition, inner-product, and other solver operations, the time of which depend on the number of activated voxels. Beside P2G, computing gradient and SVD also take a large amount of computation time in the solver, however, for the "Granulation" example in the Table 7.3, since sand particles are very sparse, more voxels are activated. As a results, P2G and others become more expensive. G2P becomes more expensive too, because the more voxels data need to be fetched for interpolation. Thus, the percentage of computing gradient and SVD is reduced, because it only depends on the particle number. Using either an atomics-only scattering or optimized gathering, like GVDB does, P2G occupies more than 90% of the solver time and is the bottleneck of the entire simulation. In contrast, our proposed P2G-MLS method is 23× faster than atomic-only scattering and 15× faster than GVDB gathering as shown in Figure 7.4. The P2G time is reduced down to around 40% for the solver.

In Figure 7.9, eighteen elastic dragons are stacked together in a glass to generate interesting dynamics. This simulation contains 9.0 million particles on a 512^3 grid with an average 21.8 seconds per $48Hz$ frame.

Time performance is not the only concern of our implementation. The memory consumption should also be dealt with appropriately, especially in high-resolution animations. In our simulator, the memory budget is partitioned into three categories.

- Particle. This part of the memory is used to store all particles' attributes. Its size

Table 7.3: Average percentages for all components in the solver. The MLS transfer scheme is used for the implicit solver.

	Compute			
	G2P	Gradient & SVD	P2G	Others
Dragon Cup	12.8%	36.8%	43.3%	7.1%
Granulation	13.4%	12.9%	45.3%	28.4%
Gelatin	12.4%	34.2%	42.9%	10.6%
Melting	11.3%	37.0%	39.5%	12.2%

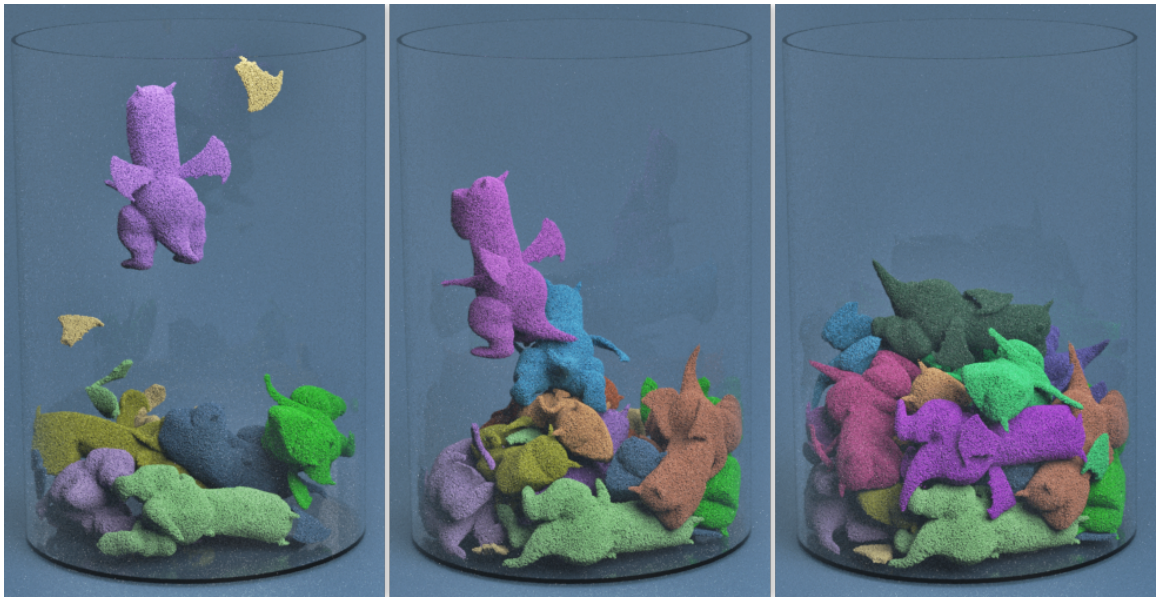


Fig. 7.9: How to stack your dragon. Stacking elastic dragons in a glass. This simulation contains 9.0 million particles on a 512^3 grid with an average 21.8 seconds per $48Hz$ frame.

grows linearly with the particle count and the number of attributes in each particle.

- Grid. This part of the memory is used for the sparse structure, where the actual memory usage has a linear correlation with the number of occupied GSPGrid blocks. The worst case degenerates to a uniform grid. In our tests, we mostly set the total amount of memory allocated to be 60% to that of the uniform grid.

- Auxiliary. Logically speaking, this part consists of two blocks of memory. Since we use spatial hashing for particle sorting and block topology construction, we need a hash table to perform the task. Its capacity shares a linear correlation with the number of cells in use (64× that of sparse GSPGrid blocks). To reduce hash collision

conflicts as well as saving memory, the coefficient is set to 64 (not the same meaning as before) as a trade-off. Note that the key value of the hash table is a 64-bit integer. The other trunk of memory works as the storage for other intermediate computations, including the ordering of particle positions, velocities, deformation gradient $\mathbf{F}$, etc. Its size is linear with the number of particles.

In the actual implementation, the total amount of auxiliary memory set in advance for these intermediate computations is decided by the maximum memory in use at the same time. The hash table is only used for particle ordering and page topology construction which are the two beginning tasks of each time step. In the rest steps, this hash table is no longer needed.

As a result, up to 2.4GB is spent on the cube example with 7M particles, 900K cells, and 17K GSPGrid blocks, of which the majority is particle-related, while the memory budget for the dragon collision example with 95K particles, 6K cells, and 500 blocks is 70MB.

7.5 Extra Results

We can also simulate a Gelatin Jello bouncing off another larger one in Figure 7.10. Lava is poured to a cool elastic dragon in Figure 7.11. Our heat solver is capable of accurately capturing the process of heat transport and phase change. As the temperature of certain parts of the dragon increases, the dragon liquefies. We demonstrate in Figure 7.12 that a knited sweater with 0.5 million particles can be simulated at an average 2 seconds per $48Hz$ frame on Nvidia Titan Xp. We can also even simulate the knited sweater in the ply level as shown in Figure 7.1 with 1.2 million particles at an average 5 seconds per $48Hz$ on Nvidia Titan Xp. Finally, we demonstrate in Figure 7.13 that our new granular material model manages to produce visually pleasing dynamics with a semi-implicit solver.

7.6 Limitations and Future Work

Our work focuses at porting the transfer kernels of the MPM pipeline to the GPU with the least amount of performance compromises. In addition to our data representation via (G)SPGrid, our analysis identified the overhead associated with a scatter-approach as a significant hindrance, due to the fine-grained atomic operations necessary to avoid hazards. Rather than avoiding the scatter paradigm, we introduced an approach that

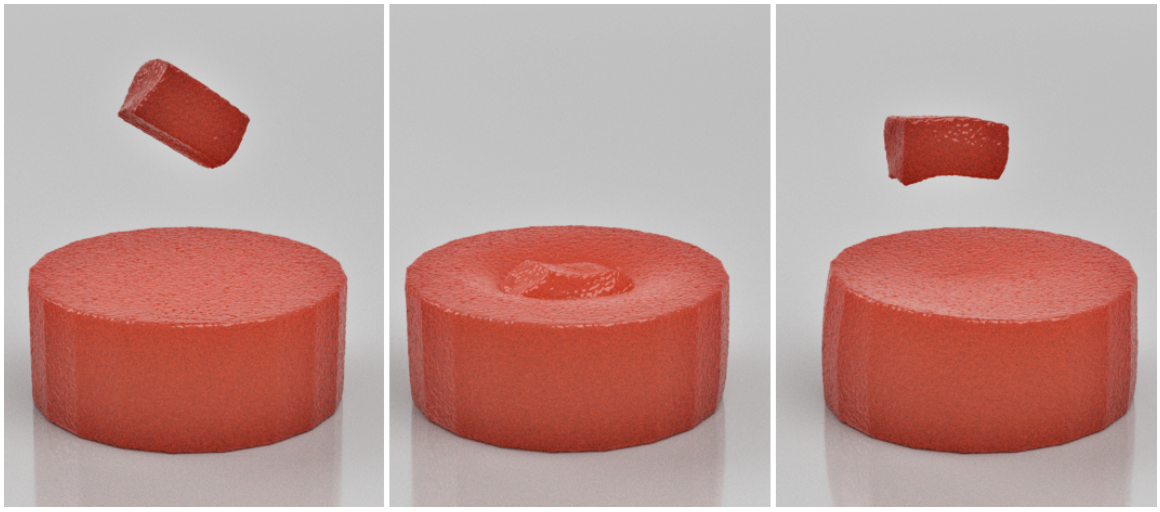


Fig. 7.10: Elasticity simulation of Gelatin bouncing off Gelatin with 6.9 million particles on a 512^3 grid at an average 6.72 seconds per $48Hz$ frame.

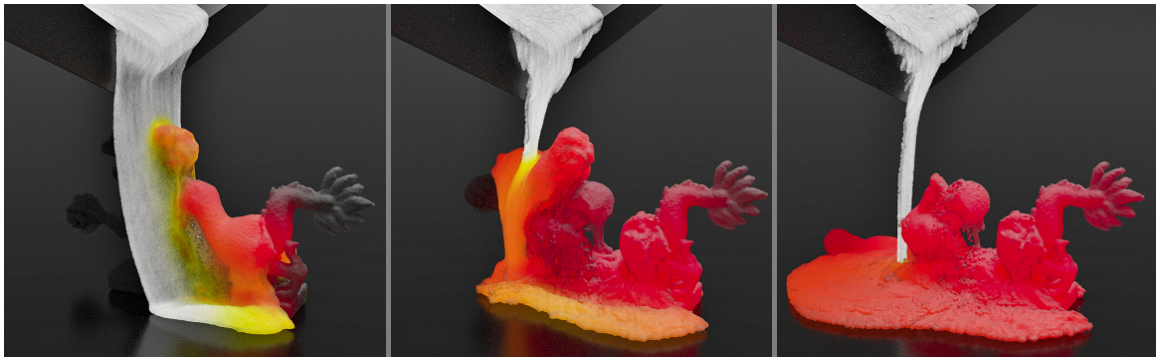


Fig. 7.11: How to melt your dragon. Melting an elastoplastic dragon with 4.2 million particles on a 256^3 grid using our GPU-optimized implicit MPM dynamics and heat solvers on a Nvidia *Quadro* P6000 GPU at an average 10.5 seconds per $48Hz$ frame.

Fig. 7.12: A level-level sweater with 0.5 million particles at an average 2 seconds per $48Hz$ frame on Nvidia Titan Xp.

Fig. 7.13: How to granulate your dragon. Granulated dragons fall on elastic ones. This simulation contains 6.7 million particles on a 512^3 grid at an average 39.4 seconds per $48Hz$ frame.

drastically minimizes the need for such atomics, improving parallel efficiency.

A number of the design choices that led to our demonstrated performance gains also carry some associated limitations that we consciously commit to. Our decision to mimic the design of the CPU-oriented SPGrid data structure in our GPU counterpart allows for implementations on the respective platforms to use similar semantics and maximize code reuse. However, the explicit use of the virtual memory system in the CPU version of SPGrid allows for computational kernels to be implemented (at reasonable, albeit not fully optimal efficiency) with computations performed at per-node granularity or, more realistically, SIMD-line granularity; for example, accessing a stencil neighbor of any individual grid node can be done at a reasonable cost, without any set-up overhead. On the GPU, however, such computations can only reach high efficiency if performed at a larger scale, e.g., at block granularity, since the overhead of fetching the neighboring blocks of the one being processed needs to be borne for each kernel invocation. This is not a prohibitive limitation, as performing computations at block granularity is by-and-large a necessity for efficiency for any similar GPU kernel.

GSPGrid apparently lacks a defining feature of CPU/SPGrid, its use of the virtual memory system and translational faculties, i.e., the TLB. One would hope that this can be a momentary shortcoming, and future GPUs and associated APIs will provide more direct

access to virtual memory and address translation, on the GPU. But even in the current version, our implementation, even if it feels more like a tiled grid, is trivially convertible from/to a CPU/SPGrid structure. Also, the main benefit that CPU/SPGrid draws from its affinity to the virtual memory system is its ability to deliver competitive performance, and even if one grid index is processed at a time – on the GPU, we are de facto forced to conduct operations at warp/block granularity, so the design of fetching a neighborhood of blocks prior to kernel application might have been the best performing implementation, even with a virtual memory-assisted data structure.

In addition, for a CPU implementation of SPGrid, accessing a grid node that has not been referenced before is an operation that can be done without any requisite setup or preprocessing (any page-faults that might occur are handled transparently). On the GPU, however, our need to explicitly allocate all active blocks necessitates that the set of all active indices be fully known before their data storage can be allocated and accessed. Once again, we regard this as a reasonable limitation, since the frequency at which the topology of the computational domain changes is small relative to the computational cost of operating on such data during MPM simulation. Also, since the maximum domain size is determined by the number of valid bits of a 64-bit offset and the resolution of each cell, it should be enough for most simulations.

Finally, GPU MPM simulations are still limited by the smaller amount of on-board memory, and it would be an interesting investigation to explore multi-GPU methods, or heterogeneous implementations to circumvent the size limitation.

While our optimization strategies greatly utilize computational resources on the GPU, high fidelity MPM simulations are still far from being real-time. This is largely due to the strict CFL restriction on time step sizes especially in high resolution. It would be interesting future work to further combine additional algorithmic acceleration of MPM time stepping with our GPU framework. We would also explore possibilities with spatially adaptive GSPGrid following Gao et al. [43] for superior performance.

7.7 Conclusion

We had presented an efficient method that ports the transfer kernels of the MPM pipeline to the GPU with the least amount of performance compromises. In addition to our data

representation via (G)SPGrid, our analysis identified the overhead associated with a scatter-approach as a significant hindrance, due to the fine-grained atomic operations necessary to avoid hazards. Rather than avoiding the scatter paradigm, we introduced an approach that drastically minimizes the need for such atomics, improving parallel efficiency. We demonstrated million of yarn particles can be simulated efficiently in a few second per frame on modern GPU.

CHAPTER 8

CONCLUSION

Knitting is a construction technique that starts with a long yarn and produces a surface using interlocked stitches. That particular constructing way allows the entire surface to be formed from a single piece of yarn, without introducing seams that lead to structural weakness or cuts that waste material. Interlocked stitches also make the surface elastic, even when the yarn material itself has high resistance to stretching. In addition to those advantages, knitting also has its special constraints to create lots of open-problems when digitalizing the entire knitting design pipeline, not only on the modeling side but also on rendering and simulation.

We have introduced a fully automatic method for converting arbitrary 3D shapes into knit structures, starting with quad-dominant mesh generation, followed by a two-step optimization process and topological operations that generate a valid stitch mesh [155]. We have demonstrated the effectiveness of our approach with complex knit models generated using our pipeline and the robustness of our method by processing a large number of different 3D models. The yarn-level models we produce are guaranteed to have valid knit topologies, and they are ready to be used with yarn-level simulations. To our knowledge, this is the first fully automatic method that can produce yarn-level knit model for arbitrary 3D shapes.

We have presented knittable stitch meshes that extend the powerful concept of stitch meshes into models that can be fabricated via knitting [156]. By introducing shift paths and properly handling mismatched knitting directions, we can convert any stitch mesh into a knittable structure. We have also introduced novel representations for handling shaping techniques that allow designing knit structures with unprecedented complexity. Finally, we have presented an algorithm that generates step-by-step instructions from a given knittable stitch mesh. We have shown a variety of example models with complex

topologies to demonstrate the effectiveness of our approach.

We have presented a new visual programming tool for computer-controlled knitting machines [107]. The core of our approach is an *augmented* stitch mesh data structure, which associates each face in a mesh with machine knitting instructions and maintains dependency information between the faces. Our system allows users to automatically create a machine-knittable augmented stitch mesh from a 3D model, edit this augmented stitch mesh while preserving machine-knittability, and transform the mesh into a knitting program for machine fabrication.

We have presented a real-time fiber-level cloth rendering framework [158, 159]. 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 an 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.

We have presented an efficient method that ports the transfer kernels of the MPM pipeline to the GPU with the least amount of performance compromises [44]. In addition to our data representation via (G)SPGrid, our analysis identified the overhead associated with a scatter-approach as a significant hindrance, due to the fine-grained atomic operations necessary to avoid hazards. Rather than avoiding the scatter paradigm, we introduced an approach that drastically minimizes the need for such atomics, improving parallel efficiency.

With the advancements presented in this dissertation, it is possible to digitalize the entire knitting design pipeline for computer graphic purposes and general-purpose fabrication with industrial knitting machines. In this dissertation, we have addressed various technical challenges of digitial knitting from computer-aided design to fabrication including modeling, rendering, and simulation.

REFERENCES

[1] M. Aanjaneya, M. Gao, H. Liu, C. Batty, and E. Sifakis, *Power diagrams and sparse paged grids for high resolution adaptive liquids*, ACM Trans. Graph., 36 (2017), p. 140.

[2] N. Adabala, N. Magnenat-Thalmann, and G. Fei, *Real-time rendering of woven clothes*, in Proceedings of the ACM Symposium on Virtual Reality Software and Technology, VRST '03, New York, NY, USA, 2003, ACM, pp. 41–47.

[3] E. Akleman, J. Chen, Q. Xing, and J. L. Gross, *Cyclic plain-weaving on polygonal mesh surfaces with graph rotation systems*, ACM Trans. Graph., 28 (2009), pp. 78:1–78:8.

[4] P. Alliez, M. Meyer, and M. Desbrun, *Interactive geometry remeshing*, ACM Trans. Graph., 21 (2002).

[5] M. Ashikmin, S. Premože, and P. Shirley, *A microfacet-based BRDF generator*, in Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '00, New York, NY, USA, 2000, ACM Press/Addison-Wesley Publishing Co., pp. 65–74.

[6] D. Baraff and A. Witkin, *Large steps in cloth simulation*, in Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '98, New York, NY, USA, 1998, ACM, pp. 43–54.

[7] L. Bavoil and M. Sainz, *Multi-layer dual-resolution screen-space ambient occlusion*, in SIGGRAPH 2009: Talks, 2009, pp. 45:1–45:1.

[8] S.-M. Belcastro, *Every topological surface can be knit: A proof*, J. Math. Art., 3 (2009), pp. 67–83.

[9] F. Berthouzoz, A. Garg, D. M. Kaufman, E. Grinspun, and M. Agrawala, *Parsing sewing patterns into 3D garments*, ACM Trans. Graph., 32 (2013), pp. 85:1–85:12.

[10] D. Bommes, T. Lempfer, and L. Kobbelt, *Global structure optimization of quadrilateral meshes*, Comp. Graph. Forum, 30 (2011), pp. 375–384.

[11] D. Bommes, B. Lévy, N. Pietroni, E. Puppo, C. Silva, M. Tarini, and D. Zorin, *Quad-mesh generation and processing: A survey*, Comp. Graph. Forum, 32 (2013).

[12] D. Bommes, H. Zimmer, and L. Kobbelt, *Mixed-integer quadrangulation*, ACM Trans. Graph., 28 (2009).

[13] S. Bouaziz, S. Martin, T. Liu, L. Kavan, and M. Pauly, *Projective dynamics: Fusing constraint projections for fast simulation*, ACM Trans. Graph., 33 (2014), pp. 154:1–154:11.

[14] J. Brackbill, *The ringing instability in particle-in-cell calculations of low-speed flow*, J. Comput. Phys., 75 (1988), pp. 469–492.

[15] D. Bradley, T. Popa, A. Sheffer, W. Heidrich, and T. Boubekeur, *Markerless garment capture*, ACM Trans. Graph., 27 (2008).

[16] D. E. Breen, D. H. House, and M. J. Wozny, *Predicting the drape of woven cloth using interacting particles*, in Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '94, 1994, pp. 365–372.

[17] R. Bridson, *Fast poisson disk sampling in arbitrary dimensions*, in ACM SIGGRAPH 2007 Sketches, 2007.

[18] R. Bridson, *Fluid simulation for computer graphics*, Taylor & Francis, 2008.

[19] R. Bridson, R. Fedkiw, and J. Anderson, *Robust treatment of collisions, contact and friction for cloth animation*, ACM Trans. Graph., 21 (2002), pp. 594–603.

[20] M. Cabral, S. Lefebvre, C. Dachsbacher, and G. Drettakis, *Structure-preserving reshape for textured architectural scenes*, Comput. Graph. Forum, (2009).

[21] M. Carignan, Y. Yang, N. M. Thalmann, and D. Thalmann, *Dressing animated synthetic actors with complex deformable clothes*, ACM SIGGRAPH'92, (1992), pp. 99–104.

[22] X. Chen, B. Zhou, F. Lu, L. Wang, L. Bi, and P. Tan, *Garment modeling with a depth camera*, ACM Trans. Graph., 34 (2015).

[23] W.-F. Chiang, M. DeLisi, T. Hummel, T. Prete, K. Tew, M. Hall, P. Wallstedt, and J. Guilkey, *GPU acceleration of the generalized interpolation material point method*, in Symposium on Application Accelerators in High Performance Computing, SAAHPC, 2009.

[24] K. Choi and T. Lo, *An energy model of plain knitted fabric*, Text. Res. J., 73 (2003), pp. 739–748.

[25] K. F. Choi and T. Y. Lo, *The shape and dimensions of plain knitted fabric: A fabric mechanical model*, Text. Res. J., 76 (2006), pp. 777–786.

[26] J. Chu, N. B. Zafar, and X. Yang, *A schur complement preconditioner for scalable parallel fluid simulation*, ACM Trans. Graph., 36 (2017), pp. 163:1–163:11.

[27] G. Cirio, J. Lopez-Moreno, D. Miraut, and M. A. Otaduy, *Yarn-level simulation of woven cloth*, ACM Trans. Graph., 33 (2014), pp. 207:1–207:11.

[28] G. Cirio, J. Lopez-Moreno, and M. A. Otaduy, *Efficient simulation of knitted cloth using persistent contacts*, in Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation, SCA '15, New York, NY, USA, 2015, ACM, pp. 55–61.

[29] G. Cirio, J. Lopez-Moreno, and M. A. Otaduy, *Yarn-level cloth simulation with sliding persistent contacts*, IEEE Trans. Vis. Comput. Graphics, 23 (2017), pp. 1152–1162.

[30] K. Crane, M. Desbrun, and P. Schröder, *Trivial connections on discrete surfaces*, Comput. Graph. Forum, 29 (2010).

[31] R. Daněřek, E. Dibra, C. Öztireli, R. Ziegler, and M. Gross, *DeepGarment : 3D garment shape estimation from a single image*, Comput. Graph. Forum, 36 (2017), pp. 269–280.

[32] DAVID, *David 3D scanner*, 2018. http://www.david-3d.com/.

[33] G. Daviet and F. Bertails-Descoubes, *A semi-implicit material point method for the continuum simulation of granular materials*, ACM Trans. Graph., 35 (2016), pp. 102:1–102:13.

[34] P. Decaudin, D. Julius, J. Wither, L. Boissieux, A. Sheffer, and M.-P. Cani, *Virtual garments: A fully geometric approach for clothing design*, Comput. Graph. Forum (Eurographics), 25 (2006), pp. 625–634.

[35] A. Demiroz and T. Dias, *A study of the graphical representation of plain-knitted structures part I: Stitch model for the graphical representation of plain-knitted structures*, J. Text. I., 91 (2000), pp. 463–480.

[36] O. Diamanti, A. Vaxman, D. Panozzo, and O. Sorkine-Hornung, *Designing N-PolyVector fields with complex polynomials*, Comput. Graph. Forum, 33 (2014).

[37] S. Dong, S. Kircher, and M. Garland, *Harmonic functions for quadrilateral remeshing of arbitrary manifolds*, Comput. Aided Geom. D., 22 (2005), pp. 392–423.

[38] Y. Dong and J. Grabe, *Large scale parallelisation of the material point method with multiple GPUs*, Comput. Geotech., 101 (2018), pp. 149–158.

[39] Y. Dong, D. Wang, and M. F. Randolph, *A GPU parallel computing strategy for the material point method*, Comput. Geotech., 66 (2015), pp. 31–38.

[40] H.-C. Ebke, M. Campen, D. Bommes, and L. Kobbelt, *Level-of-detail quad meshing*, ACM Trans. Graph., 33 (2014).

[41] Y. Fang*, Y. Hu*, S. Hu, and C. Jiang, *A temporally adaptive material point method with regional time stepping*, in Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA '18, Eurographics Association, 2018. (*Joint First Authors).

[42] M. Gao, A. Tampubolon, X. Han, Q. Guo, G. Kot, E. Sifakis, and C. Jiang, *Animating fluid sediment mixture in particle-laden flows*, ACM Trans. Graph., 37 (2018).

[43] M. Gao, A. P. Tampubolon, C. Jiang, and E. Sifakis, *An adaptive generalized interpolation material point method for simulating elastoplastic materials*, ACM Trans. Graph., 36 (2017).

[44] M. Gao*, X. Wang*, K. Wu*, A. Pradhana, E. Sifakis, C. Yuksel, and C. Jiang, *GPU optimization of material point methods*, ACM Trans. Graph. (Proceedings of SIGGRAPH ASIA 2018), 37 (2018). (*Joint First Authors).

[45] X. Gao, W. Jakob, M. Tarini, and D. Panozzo, *Robust hex-dominant mesh generation using field-guided polyhedral agglomeration*, ACM Trans. Graph., 36 (2017), pp. 114:1–114:13.

[46] A. Garg and R. Tamassia, *On the computational complexity of upward and rectilinear planarity testing*, SIAM J. Comput., 31 (2001), pp. 601–625.

[47] T. Gast, C. Fu, C. Jiang, and J. Teran, *Implicit-shifted symmetric QR singular value decomposition of 3x3 matrices*, tech. rep., University of California Los Angeles, 2016.

[48] O. Goktepe and S. Harlock, *Three-dimensional computer modeling of warp knitted structures*, Text. Res. J., 72 (2002), pp. 266–272.

[49] R. Goldenthal, D. Harmon, R. Fattal, M. Bercovier, and E. Grinspun, *Efficient simulation of inextensible cloth*, ACM Trans. Graph., 26 (2007).

[50] N. K. Govindaraju, I. Kabul, M. C. Lin, and D. Manocha, *Fast continuous collision detection among deformable models using graphics processors*, Comp. Graph., 31 (2007), pp. 5–14.

[51] N. K. Govindaraju, D. Knott, N. Jain, I. Kabul, R. Tamstorf, R. Gayle, M. C. Lin, and D. Manocha, *Interactive collision detection between deformable models using chromatic decomposition*, ACM Trans. Graph., 24 (2005), pp. 991–999.

[52] E. Grinspun, A. N. Hirani, M. Desbrun, and P. Schröder, *Discrete shells*, in Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA '03, Aire-la-Ville, Switzerland, Switzerland, 2003, Eurographics Association, pp. 62–67.

[53] E. Groller, R. T. Rau, and W. Strasser, *Modeling and visualization of knitwear*, IEEE Trans. Vis. Comput. Graphics, 1 (1995), pp. 302–310.

[54] X. Gu, S. J. Gortler, and H. Hoppe, *Geometry images*, ACM Trans. Graph., 21 (2002).

[55] P. Guan, L. Reiss, D. A. Hirshberg, A. Weiss, and M. J. Black, *Drape: Dressing any person*, ACM Trans. Graph., 31 (2012), pp. 35:1–35:10.

[56] Q. Guo, X. Han, C. Fu, T. Gast, R. Tamstorf, and J. Teran, *A material point method for thin shells with frictional contact*, ACM Trans. Graph., 37 (2018).

[57] I. Gurobi Optimization, *Gurobi optimizer reference manual*, 2016.

[58] T. Harada, S. Koshizuka, and Y. Kawaguchi, *Smoothed particle hydrodynamics on GPUs*, in Comput. Graph. Int., vol. 40, 2007, pp. 63–70.

[59] D. J. Heeger and J. R. Bergen, *Pyramid-based texture analysis/synthesis*, in Proceedings of ACM SIGGRAPH, ACM, 1995, pp. 229–238.

[60] E. Heitz, J. Dupuy, C. Crassin, and C. Dachsbacher, *The SGGX microflake distribution*, ACM Trans. Graph., 34 (2015), pp. 48:1–48:11.

[61] A. Hertzmann and D. Zorin, *Illustrating smooth surfaces*, in Proceedings of ACM SIGGRAPH, ACM Press/Addison-Wesley Publishing Co., 2000, pp. 517–526.

[62] R. K. Hoetzlein, *Gvdb: Raytracing sparse voxel database structures on the GPU*, in Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics, The Eurographics Association, 2016.

[63] C. Horvath and W. Geiger, *Directable, high-resolution simulation of fire on the GPU*, ACM Trans. Graph., 28 (2009), pp. 41:1–41:8.

[64] Y. Hu, Y. Fang, Z. Ge, Z. Qu, Y. Zhu, A. Pradhana, and C. Jiang, *A moving least squares material point method with displacement discontinuity and two-way rigid body coupling*, ACM Trans. Graph., 37 (2018).

[65] P. Huang, X. Zhang, S. Ma, and H. K. Wang, *Shared memory OpenMP parallelization of explicit MPM and its application to hypervelocity impact*, Comput. Model. Eng. Sci., 38 (2008), pp. 119–148.

[66] Z. Huang and T. Ju, *Extrinsically smooth direction fields*, Comput. Graph., 58 (2016), pp. 109–117.

[67] Y. Igarashi, T. Igarashi, and H. Suzuki, *Knitting a 3D model*, in Comput. Graph. Forum, 2008.

[68] ——, *Knitty: 3D modeling of knitted animals with a production assistant interface*, in Eurographics 2008 Annex to the Conference Proceedings, 2008.

[69] P. Irawan and S. Marschner, *Specular reflection from woven cloth*, ACM Trans. Graph., 31 (2012), pp. 11:1–11:20.

[70] W. Jakob, A. Arbree, J. T. Moon, K. Bala, and S. Marschner, *A radiative transfer framework for rendering materials with anisotropic structure*, in ACM SIGGRAPH 2010 Papers, SIGGRAPH '10, New York, NY, USA, 2010, ACM, pp. 53:1–53:13.

[71] W. Jakob, M. Tarini, D. Panozzo, and O. Sorkine-Hornung, *Instant field-aligned meshes*, ACM Trans. Graph., 34 (2015), pp. 189:1–189:15.

[72] C. Jiang, T. Gast, and J. Teran, *Anisotropic elastoplasticity for cloth, knit and hair frictional contact*, ACM Trans. Graph., 36 (2017), pp. 152:1–152:14.

[73] C. Jiang, C. Schroeder, A. Selle, J. Teran, and A. Stomakhin, *The affine particle-in-cell method*, ACM Trans. Graph., 34 (2015), pp. 51:1–51:10.

[74] T. Jiang, X. Fang, J. Huang, H. Bao, Y. Tong, and M. Desbrun, *Frame field generation through metric customization*, ACM Trans. Graph., 34 (2015).

[75] F. Kälberer, M. Nieser, and K. Polthier, *QuadCover – surface parameterization using branched coverings*, Comput. Graph. Forum, 26 (2007).

[76] J. M. Kaldor, D. L. James, and S. Marschner, *Simulating knitted cloth at the yarn level*, ACM Trans. Graph., 27 (2008), pp. 65:1–65:9.

[77] ——, *Efficient yarn-based cloth with adaptive contact linearization*, ACM Trans. Graph., 29 (2010), pp. 105:1–105:10.

[78] Y.-M. Kang, *Realtime rendering of realistic fabric with alternation of deformed anisotropy*, in Proceedings of the Third International Conference on Motion in Games, MIG'10, Berlin, Heidelberg, 2010, Springer-Verlag, pp. 301–312.

[79] A. Khodakovsky, N. Litke, and P. Schröder, *Globally smooth parameterizations with low distortion*, ACM Trans. Graph., 22 (2003).

[80] P. Khungurn, D. Schroeder, S. Zhao, K. Bala, and S. Marschner, *Matching real fabrics with micro-appearance models*, ACM Trans. Graph., 35 (2015), pp. 1:1–1:26.

[81] G. Klár, J. Budsberg, M. Titus, S. Jones, and K. Museth, *Production ready MPM simulations*, in ACM SIGGRAPH 2017 Talks, 2017, pp. 42:1–42:2.

[82] G. Klár, T. Gast, A. Pradhana, C. Fu, C. Schroeder, C. Jiang, and J. Teran, *Drucker-prager elastoplasticity for sand animation*, ACM Trans. Graph., 35 (2016), pp. 103:1–103:12.

[83] F. Knöppel, K. Crane, U. Pinkall, and P. Schröder, *Globally optimal direction fields*, ACM Trans. Graph., 32 (2013), pp. 59:1–59:10.

[84] ——, *Stripe patterns on surfaces*, ACM Trans. Graph., 34 (2015), pp. 39:1–39:11.

[85] A. Kurbak, *Geometrical models for balanced rib knitted fabrics part I: Conventionally knitted 1× 1 rib fabrics*, Text. Res. J., 79 (2009), pp. 418–435.

[86] A. Kurbak and T. Alpyildiz, *A geometrical model for the double lacoste knits*, Text. Res. J., 78 (2008), pp. 232–247.

[87] A. Kurbak and A. S. Soydan, *Geometrical models for balanced rib knitted fabrics part III: 2× 2, 3× 3, 4× 4, and 5× 5 rib fabrics*, Text. Res. J., 79 (2009), pp. 618–625.

[88] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick, *Graphcut textures: Image and video synthesis using graph cuts*, in ACM Trans. Graph., ACM, 2003, pp. 277–286.

[89] Y.-K. Lai, M. Jin, X. Xie, Y. He, J. Palacios, E. Zhang, S.-M. Hu, and X. Gu, *Metric-driven rosy field design and remeshing*, IEEE Trans. Vis. Comput. Graphics, 16 (2010), pp. 95–108.

[90] J. Leaf, R. Wu, E. Schweickart, D. L. James, and S. Marschner, *Interactive design of yarn-level cloth patterns*, ACM Trans. Graph. (Proceedings of SIGGRAPH Asia 2018), 37 (2018).

[91] N. Lei, X. Zheng, H. Si, Z. Luo, and X. Gu, *Generalized regular quadrilateral mesh generation based on surface foliation*, Procedia Eng., 203 (2017), pp. 336 – 348.

[92] J. Lin, V. Narayanan, and J. McCann, *Efficient transfer planning for flat knitting*, in Proceedings of the 2nd ACM Symposium on Computational Fabrication, SCF '18, New York, NY, USA, 2018, ACM, pp. 1:1–1:7.

[93] H. Liu, N. Mitchell, M. Aanjaneya, and E. Sifakis, *A scalable schur-complement fluids solver for heterogeneous compute platforms*, ACM Trans. Graph., 35 (2016), pp. 201:1–201:12.

[94] J. Lopez-Moreno, D. Miraut, G. Cirio, and M. A. Otaduy, *Sparse GPU voxelization of yarn-level cloth*, Comput. Graph. Forum, 36 (2017), pp. 22–34.

[95] F. Luan, S. Zhao, and K. Bala, *Fiber-level on-the-fly procedural textiles*, Comput. Graph. Forum, 36 (2017), pp. 123–135.

[96] J. Luitjens, *Faster parallel reductions on kepler*, Nvidia, (2014).

[97] Z. G. Luo and M. Yuen, *Reactive 2D/3D garment pattern design modification*, Comput.-Aided Des., 37 (2005), pp. 623–630.

[98] M. Marinov and L. Kobbelt, *A robust two-step procedure for quad-dominant remeshing*, Comput. Graph. Forum, (2006).

[99] A. McAdams, A. Selle, R. Tamstorf, J. Teran, and E. Sifakis, *Computing the singular value decomposition of 3× 3 matrices with minimal branching and elementary floating point operations*, University of Wisconsin Madison, (2011).

[100] J. McCann, *The "knitout" (.k) file format*. [Online]. Available from: https://textiles-lab.github.io/knitout/knitout.html, 2017.

[101] J. McCann, L. Albaugh, V. Narayanan, A. Grow, W. Matusik, J. Mankoff, and J. Hodgins, *A compiler for 3D machine knitting*, ACM Trans. Graph., 35 (2016), pp. 49:1–49:11.

[102] M. McGuire, B. Osman, M. Bukowski, and P. Hennessy, *The alchemy screen-space ambient obscurance algorithm*, in Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, HPG '11, ACM, 2011, pp. 25–32.

[103] M. Meisner and B. Eberhardt, *The art of knitted fabrics, realistic & physically based modelling of knitted patterns*, in Comput. Graph. Forum, vol. 17, Wiley Online Library, 1998, pp. 355–362.

[104] Y. Mori and T. Igarashi, *Plushie: An interactive design system for plush toys*, ACM Trans. Graph., 26 (2007).

[105] K. Museth, *Vdb: High-resolution sparse volumes with dynamic topology*, ACM Trans. Graph., 32 (2013), p. 27.

[106] V. Narayanan, L. Albaugh, J. Hodgins, S. Coros, and J. Mccann, *Automatic machine knitting of 3D meshes*, ACM Trans. Graph., 37 (2018), pp. 35:1–35:15.

[107] V. Narayanan*, K. Wu*, C. Yuksel, and J. McCann, *Visual knitting machine programming*, ACM Trans. Graph. (Proceedings of SIGGRAPH 2019), (2019). (*Joint First Authors).

[108] M. Nieser, J. Palacios, K. Polthier, and E. Zhang, *Hexagonal global parameterization of arbitrary surfaces*, IEEE Trans. Vis. Comput. Graphics, 18 (2012).

[109] J. Palacios and E. Zhang, *Rotational symmetry field design on surfaces*, ACM Trans. Graph., 26 (2007).

[110] D. Panozzo, E. Puppo, M. Tarini, and O. Sorkine-Hornung, *Frame fields: Anisotropic and non-orthogonal cross fields*, ACM Trans. Graph., 33 (2014), pp. 134:1–134:11.

[111] S. G. Parker, *A component-based architecture for parallel multi-physics PDE simulation*, Fut. Gen. Comp. Sys., 22 (2006), pp. 204 – 216.

[112] C.-H. Peng and P. Wonka, *Connectivity editing for quad-dominant meshes*, in Proceedings of the Eleventh Eurographics/ACMSIGGRAPH Symposium on Geometry Processing, SGP '13, Aire-la-Ville, Switzerland, Switzerland, 2013, Eurographics Association, pp. 43–52.

[113] J. Pérez, B. Thomaszewski, S. Coros, B. Bickel, J. A. Canabal, R. Sumner, and M. A. Otaduy, *Design and fabrication of flexible rod meshes*, ACM Trans. Graph., 34 (2015), pp. 138:1–138:12.

[114] G. Pons-Moll, S. Pujades, S. Hu, and M. J. Black, *ClothCap: Seamless 4D clothing capture and retargeting*, ACM Trans. Graph., 36 (2017), pp. 73:1–73:15.

[115] M. Popescu, M. Rippmann, T. Van Mele, and P. Block, *Automated Generation of Knit Patterns for Non-developable Surfaces*, Springer Singapore, Singapore, 2018, pp. 271–284.

[116] D. Ram, T. Gast, C. Jiang, C. Schroeder, A. Stomakhin, J. Teran, and P. Kavehpour, *A material point method for viscoelastic fluids, foams and sponges*, in Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation, SCA '15, New York, NY, USA, 2015, ACM, pp. 157–163.

[117] N. Ray, W. C. Li, B. Lévy, A. Sheffer, and P. Alliez, *Periodic global parameterization*, ACM Trans. Graph., (2006).

[118] N. Ray, B. Vallet, W. C. Li, and B. Lévy, *N-symmetry direction field design*, ACM Trans. Graph., 27 (2008).

[119] W. Renkens and Y. Kyosev, *Geometry modelling of warp knitted fabrics with 3D form*, Text. Res. J., 81 (2011), pp. 437–443.

[120] C. Robson, R. Maharik, A. Sheffer, and N. Carr, *Context-aware garment modeling from sketches*, Comput. Graph. (SMI 2011), 35 (2011), pp. 604–613.

[121] K. P. Ruggirello and S. C. Schumacher, *A comparison of parallelization strategies for the material point method*, in 11th World Cong. on Comp. Mech., 2014, pp. 20–25.

[122] I. Sadeghi, O. Bisker, J. De Deken, and H. W. Jensen, *A practical microcylinder appearance model for cloth rendering*, ACM Trans. Graph., 32 (2013), pp. 14:1–14:12.

[123] I. Sadeghi, H. Pritchett, H. W. Jensen, and R. Tamstorf, *An artist friendly hair shading system*, ACM Trans. Graph., 29 (2010), pp. 56:1–56:10.

[124] K. Schroder, A. Zinke, and R. Klein, *Image-based reverse engineering and visual prototyping of woven cloth*, IEEE Trans. Vis. Comput. Graphics, 21 (2015), pp. 188–200.

[125] R. Setaluri, M. Aanjaneya, S. Bauer, and E. Sifakis, *SPGrid: A sparse paged grid structure applied to adaptive smoke simulation*, ACM Trans. Graph., 33 (2014), pp. 205:1–205:12.

[126] Shima Seiki, *Sds-one apex3*. [Online]. Available from: http://www.shimaseiki.com/product/design/sdsone$_{a}pex/flat/$, 2011.

[127] Soft Byte Ltd., *Designaknit*. [Online.], 1999.

[128] D. Sokolov, N. Ray, L. Untereiner, and B. Lévy, *Hexahedral-dominant meshing*, ACM Trans. Graph., 35 (2016), pp. 157:1–157:23.

[129] D. J. Spencer, *Knitting technology: A comprehensive handbook and practical guide*, vol. 16, CRC press, 2001.

[130] G. Stantchev, D. Dorland, and N. Gumerov, *Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU*, J. Par. Dis. Comp., 68 (2008), pp. 1339 – 1349.

[131] Stoll, *M1plus pattern software*. [Online]. Available from: http://www.stoll.com/$stoll_{s}oftware_{s}olutions_{e}n_{4}/pattern_{s}oftware_{m}1plus/3_{1}$, 2011.

[132] A. Stomakhin, C. Schroeder, L. Chai, J. Teran, and A. Selle, *A material point method for snow simulation*, ACM Trans. Graph., 32 (2013), pp. 102:1–102:10.

[133] A. Stomakhin, C. Schroeder, C. Jiang, L. Chai, J. Teran, and A. Selle, *Augmented MPM for phase-change and varied materials*, ACM Trans. Graph., 33 (2014), pp. 138:1–138:11.

[134] S. Sueda, G. L. Jones, D. I. W. Levin, and D. K. Pai, *Large-scale dynamic simulation of highly constrained strands*, ACM Trans. Graph., 30 (2011), pp. 39:1–39:10.

[135] D. Sulsky, S. Zhou, and H. Schreyer, *Application of a particle-in-cell method to solid mechanics*, Comp. Phys. Comm., 87 (1995), pp. 236–252.

[136] A. P. Tampubolon, T. Gast, G. Klár, C. Fu, J. Teran, C. Jiang, and K. Museth, *Multi-species simulation of porous sand and water mixtures*, ACM Trans. Graph., 36 (2017).

[137] M. Tang, Z. Liu, R. Tong, and D. Manocha, *PSCC: Parallel self-collision culling with spatial hashing on GPUs*, Proc. ACM Comput. Graph. Interact. Tech., 1 (2018), pp. 18:1–18:18.

[138] M. Tang, R. Tong, R. Narain, C. Meng, and D. Manocha, *A GPU-based streaming algorithm for high-resolution cloth simulation*, Comp. Graph. Forum, 32 (2013), pp. 21–30.

[139] M. Tang, H. Wang, L. Tang, R. Tong, and D. Manocha, *CAMA: Contact-aware matrix assembly with unified collision handling for GPU-based cloth simulation*, Comp. Graph. Forum, 35 (2016), pp. 511–521.

[140] V. Timonen, *Screen-space far-field ambient obscurance*, in Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, HPG '13, ACM, 2013, pp. 33–43.

[141] E. Turquin, M.-P. Cani, and J. F. Hughes, *Sketching garments for virtual characters*, in Proceedings of the First Eurographics Conference on Sketch-Based Interfaces and Modeling, SBM'04, Aire-la-Ville, Switzerland, Switzerland, 2004, Eurographics Association, pp. 175–182.

[142] Ultimaker, *Ultimaker 3*, 2018. https://ultimaker.com/en/products/ultimaker-3.

[143] N. Umetani, D. M. Kaufman, T. Igarashi, and E. Grinspun, *Sensitive couture for interactive garment modeling and editing*, ACM Trans. Graph., 30 (2011), pp. 90:1–90:12.

[144] J. Underwood, *The design of 3D shape knitted preforms*, PhD thesis, Fashion and Textiles, RMIT University, 2009.

[145] A. Vaxman, M. Campen, O. Diamanti, D. Panozzo, D. Bommes, K. Hildebrandt, and M. Ben-Chen, *Directional field synthesis, design, and processing*, Comput. Graph. Forum, (2016).

[146] P. Volino and N. Magnenat-Thalmann, *Accurate garment prototyping and simulation*, Comput.-Aided Des. Appl., 2 (2005), pp. 645–654.

[147] P. Volino, N. Magnenat-Thalmann, and F. Faure, *A simple approach to nonlinear tensile stiffness for accurate cloth simulation*, ACM Trans. Graph., 28 (2009), pp. 105:1–105:16.

[148] C. C. Wang, Y. Wang, and M. M. Yuen, *Feature based 3D garment design through 2D sketches*, Comput.-Aided Des., 35 (2003), pp. 659 – 672.

[149] H. Wang, *Proving theorems by pattern recognition II*, Bell Syst. Tech. J., 40 (1961), pp. 1–42.

[150] H. Wang, *Rule-free sewing pattern adjustment with precision and efficiency*, ACM Trans. Graph., 37 (2018), pp. 53:1–53:13.

[151] J. Wang, S. Zhao, X. Tong, J. Snyder, and B. Guo, *Modeling anisotropic surface reflectance with example-based microfacet synthesis*, ACM Trans. Graph., 27 (2008), pp. 41:1–41:9.

[152] X. Wang, M. Tang, D. Manocha, and R. Tong, *Efficient BVH-based collision detection scheme with ordering and restructuring*, in Comp. Graph. Forum, vol. 37, Wiley Online Library, 2018, pp. 227–237.

[153] R. Weller, N. Debowski, and G. Zachmann, *kDet: Parallel constant time collision detection for polygonal objects*, in Comp. Graph. Forum, vol. 36, 2017, pp. 131–141.

[154] J. Wretborn, R. Armiento, and K. Museth, *Animation of crack propagation by means of an extended multi-body solver for the material point method*, Comput. Graph., 69 (2017), pp. 131–139.

[155] K. Wu, X. Gao, Z. Ferguson, D. Panozzo, and C. Yuksel, *Stitch meshing*, ACM Trans. Graph. (Proceedings of SIGGRAPH 2018), 37 (2018), pp. 130:1–130:14.

[156] K. Wu, H. Swan, and C. Yuksel, *Knittable stitch meshes*, ACM Trans. Graph., 38 (2019), pp. 10:1–10:13.

[157] K. Wu, N. Truong, C. Yuksel, and R. Hoetzlein, *Fast fluid simulations with sparse volumes on the GPU*, Comp. Graph. Forum, 37 (2018), pp. 157–167.

[158] K. Wu and C. Yuksel, *Real-time cloth rendering with fiber-level detail*, IEEE Trans. Vis. Comput. Graphics, PP (2017), pp. 1–1.

[159] K. Wu and C. Yuksel, *Real-time fiber-level cloth rendering*, in ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D 2017), New York, NY, USA, 2017, ACM.

[160] R. Wu, H. Peng, F. Guimbretière, and S. Marschner, *Printing arbitrary meshes with a 5DOF wireframe printer*, ACM Trans. Graph., 35 (2016), pp. 101:1–101:9.

[161] Y.-Q. Xu, Y. Chen, S. Lin, H. Zhong, E. Wu, B. Guo, and H.-Y. Shum, *Photorealistic rendering of knitwear using the lumislice*, in Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '01, New York, NY, USA, 2001, ACM, pp. 391–398.

[162] Y. Yue, B. Smith, C. Batty, C. Zheng, and E. Grinspun, *Continuum foam: A material point method for shear-dependent flows*, ACM Trans. Graph., 34 (2015), pp. 160:1–160:20.

[163] W. Yuen, B. Wünsche, and N. Holmberg, *An applied approach for real-time level-of-detail woven fabric rendering*, J. World Soc. Comput. Graph., 20 (2012), pp. 117–126.

[164] C. Yuksel, J. M. Kaldor, D. L. James, and S. Marschner, *Stitch meshes for modeling knitted clothing with yarn-level detail*, ACM Trans. Graph. (Proceedings of SIGGRAPH 2012), 31 (2012), pp. 37:1–37:12.

[165] J. Zehnder, S. Coros, and B. Thomaszewski, *Designing structurally-sound ornamental curve networks*, ACM Trans. Graph., 35 (2016), pp. 99:1–99:10.

[166] H. Zhang, O. Van Kaick, and R. Dyer, *Spectral mesh processing*, in Comput. Graph. Forum, vol. 29, Wiley Online Library, 2010, pp. 1865–1894.

[167] Y. Zhang, B. Solenthaler, and R. Pajarola, *Adaptive sampling and rendering of fluids on the GPU*, in IEEE/ EG Symposium on Volume and Point-Based Graphics, The Eurographics Association, 2008.

[168] Y. Zhang, X. Zhang, and Y. Liu, *An alternated grid updating parallel algorithm for material point method using OpenMP*, Comput. Model. Eng. Sci., 69 (2010), pp. 143–165.

[169] S. Zhao, W. Jakob, S. Marschner, and K. Bala, *Building volumetric appearance models of fabric using micro CT imaging*, ACM Trans. Graph., 30 (2011), pp. 44:1–44:10.

[170] S. Zhao, F. Luan, and K. Bala, *Fitting procedural yarn models for realistic cloth rendering*, ACM Trans. Graph., 35 (2016), pp. 51:1–51:11.

[171] S. Zhao, L. Wu, F. Durand, and R. Ramamoorthi, *Downsampling scattering parameters for rendering anisotropic media*, ACM Trans. Graph., 35 (2016), pp. 166:1–166:11.

[172] B. Zhou, X. Chen, Q. Fu, K. Guo, and P. Tan, *Garment modeling from a single image*, Comput. Graph. Forum, 32 (2013), pp. 85–91.

[173] K. Zhou, X. Huang, X. Wang, Y. Tong, M. Desbrun, B. Guo, and H.-Y. Shum, *Mesh quilting for geometric texture synthesis*, ACM Trans. Graph., 25 (2006), pp. 690–697.

[174] Y. Zhu and R. Bridson, *Animating sand as a fluid*, ACM Trans. Graph., 24 (2005), pp. 965–972.