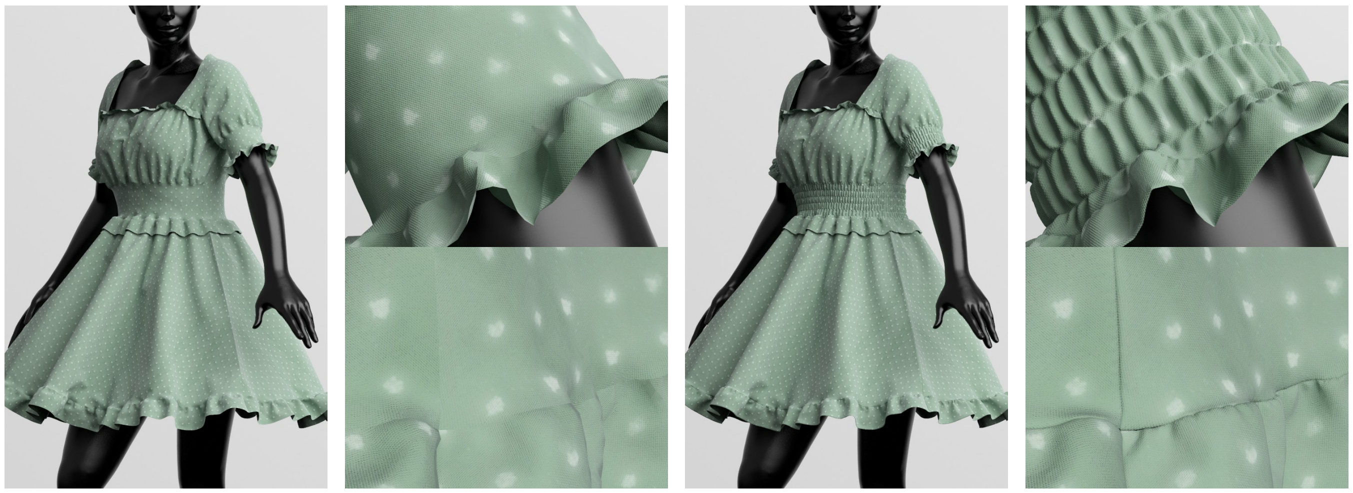
GPU-Based Simulation of Cloth Wrinkles at Submillimeter Levels

HUAMIN WANG, The Ohio State University, USA



(a) The coarse input (b) Input closeups (c) Our result (d) Result closeups

Fig. 1. A shirred dress example with 7.3M vertices and 14.6M triangles. In such a high resolution, we claim that it is more plausible to build a mesh with an underlying grid structure, which can benefit well from graphics hardware both computation-wise and memory-wise. Based on this idea, our system accomplishes fast simulation of fine wrinkle details on top of the coarse mesh input with 28K vertices in (a), under one second per frame. This is almost one order-of-magnitude faster than state-of-the-art GPU-based cloth simulators [Li et al. 2020; Wu et al. 2020].

In this paper, we study physics-based cloth simulation in a very high resolution setting, presumably at submillimeter levels with millions of vertices, to meet perceptual precision of our human eyes. State-of-the-art simulation techniques, mostly developed for unstructured triangular meshes, can hardly meet this demand due to their large computational costs and memory footprints. We argue that in a very high resolution, it is more plausible to use regular meshes with an underlying grid structure, which can be highly compatible with GPU acceleration like high-resolution images. Based on this idea, we formulate and solve the nonlinear optimization problem for simulating high-resolution wrinkles, by a fast block-based descent method with reduced memory accesses. We also investigate the development of the collision handling component in our system, whose performance benefits greatly from the grid structure. Finally, we explore various issues related to the applications of our system, including initialization for fast convergence and temporal coherence, gathering effects, inflation and stuffing models, and mesh simplification. We can treat our system as a quasistatic wrinkle synthesis tool, run it as a standalone dynamic simulator, or integrate it into a multi-resolution solver as an additional component. The experiment demonstrates the capability, efficiency and flexibility of our system in producing a variety of high-resolution wrinkles effects.

CCS Concepts: • **Computing methodologies → Physical simulation**; **Shape modeling**.

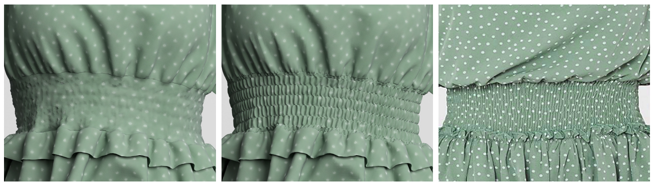
Additional Key Words and Phrases: cloth simulation, wrinkle synthesis, collision handling, parallel computing, GPU acceleration

ACM Reference Format:
Huamin Wang. 2021. GPU-Based Simulation of Cloth Wrinkles at Submillimeter Levels. *ACM Trans. Graph.* 1, 1, Article 169 (April 2021), 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Author's address: Huamin Wang, The Ohio State University, Columbus, OH, USA, whmin@cse.ohio-state.edu.

1 INTRODUCTION

Thanks to the boost of graphics hardware, researchers have significantly improved the runtime performance of physics-based cloth simulation in recent years. For instance, Wu et al. [2020] demonstrated the ability of their simulator in handling a cloth mesh with 297K vertices at 10 to 14 FPS on a single NVIDIA GeForce 2080 Ti GPU, and Li et al. [2020] accomplished the simulation of a cloth mesh with 825K vertices at 2.48 FPS on four NVIDIA Titan Xp GPUs. Unfortunately, there is still a noticeable gap between the simulation and the reality, in terms of fine wrinkle details as shown in Fig. 2a and 2c. An effective way of closing this gap is to increase the mesh resolution. But as the resolution increases, not only the number of vertices grows, but also the simulation system stiffens. In the past, researchers have considered the idea of adjusting the



(a) Coarsened result (4mm) (b) Our result (0.4mm) (c) Photograph

Fig. 2. The shirred dress results rendered at millimeter and submillimeter levels, in comparison with a photograph in (c). This figure demonstrates the necessity of using high-resolution meshes for modeling fine wrinkles.

mesh resolution adaptively [Li and Volkov 2005; Narain et al. 2012], but that introduces even more problems, such as the difficulty of formulating suitable adjustment criteria and the incompatibility with graphics hardware. In general, most of the cloth simulators are poorly scalable[1] to the mesh resolution.

In this work, we would like to address an interesting yet challenging problem: *can we develop a fast physics-based system for simulating very high-resolution cloth wrinkles on a GPU*? More specifically, we set the target resolution at submillimeter levels, which is sufficient for meeting the precision of human perception from a social distance. The use of such a system is multi-folded: it can synthesize high-resolution wrinkles quasistatically on top of existing coarse meshes, as discussed mostly in this paper; it can work as a standalone dynamic simulator for animation production; or it can serve as an extra component in a multi-resolution simulator for fine details. Fulfilling the development of such a system is not easy. Existing cloth simulators based on state-of-the-art techniques are just able to handle the mesh resolution at millimeter levels, in other words, approximately 10K to 1M vertices for normal-sized clothing. Increasing the resolution from millimeter levels to submillimeter levels means there are two orders-of-magnitude more vertices, way beyond the capability of existing simulation techniques.

The main idea behind our work comes from the observation that graphics hardware has a long and successful history of high-resolution image processing. The performances of GPU-based image processing algorithms rely heavily on the regular grid structure of image pixels, which is highly suitable for parallelization within 2D thread blocks. This observation motivates us to consider the use of regular grid meshes for high-resolution cloth simulation as well, which is not a common practice in computer graphics to our knowledge. The reason people choose unstructured meshes over structured meshes is simple: unstructured meshes provide more flexibility in representing cloth boundaries, especially in a low resolution. But since we are now dealing with high-resolution models, the boundary issue becomes less important and we are able to explore the full benefit of the regular grid structure.

Toward the development of our system, we make the following technical contributions.

- *Nonlinear optimization.* Based on the regular grid structure of our meshes, we derive a compact form of the nonlinear wrinkle optimization problem, with adjustable boundary conditions for sewing line effects, as shown in Fig. 6. Similar to the performances of other simulators, the performance of our simulator depends on both memory footprints and computational workload. While we cannot significantly reduce the workload, we invent an effective block-based descent method for fewer global and shared memory accesses.
- *Collision handling.* We investigate the handling of cloth-body and cloth self collisions, as a critical component in the aforementioned optimization process. Due to an enormous number of vertices in our meshes, we choose to use the popular GPU-based vertex repulsion method. We then focus our research on fast vertex proximity search, again relying on the regular grid structure.
- *Application-related topics.* To make our system practically useful, we study a variety of topics related to its potential applications. This includes the initialization approaches for fast convergence and temporal coherence, interactive modification to the reference mesh for gathering effects, inflation and stuffing models for multi-layered effects without modeling multiple layers, and quadtree-based mesh simplification for reducing the mesh size.

We implement our physics-based simulation system entirely on a GPU. We evaluate its main usage as a quasistatic high-resolution wrinkle synthesis tool for existing coarse meshes as shown in Fig. 1, but we also test its capability of functioning as a standalone dynamic simulator as discussed in Subsection 7.6. Our experiment shows the system is almost one order-of-magnitude faster than state-of-the-art generic cloth simulators, flexible enough to produce wrinkles for gathering and inflation effects as shown in Fig. 13, and adaptive to various simulation and synthesis tasks.

2 PREVIOUS WORK

Physics-based cloth simulation. Physics-based cloth simulation has been a fascinating topic for graphics researchers and developers for decades, thanks to its potential in entertainment and fashion applications. Since the seminal work by Terzopoulos et al. [1987], researchers have studied a variety of elastic models for representing cloth dynamics, including mass-spring models [Choi and Ko 2002; Liu et al. 2013], finite element models [Baraff and Witkin 1998; Narain et al. 2012] and yarn-based models [Cirio et al. 2014; Kaldor et al. 2010]. Regardless of these models, cloth simulation is always haunted by a major issue: the computational cost. The early effort made by researchers is to adopt implicit time integrators [Narain et al. 2012; Volino et al. 2009], instead of explicit ones [Bridson et al. 2003; Selle et al. 2009], so that a simulator can take large time steps with numerical stability. To stably simulate cloth with stiff behaviors, researchers have also explored the idea of replacing cloth elasticity by position-based techniques, such as strain limiting [Provot 1995; Thomaszewski et al. 2009; Wang et al. 2010b] and position-based dynamics [Müller 2008; Müller et al. 2014]. Liu et al. [2013] and Bouaziz et al. [2014] noticed the relationship between cloth elasticity and position-based techniques, and used that to build a new technique known as projective dynamics. Narain et al. [2016] and

[1]The simulators can still be linearly scalable to the number of vertices as long as the resolution stays the same, for example, by expanding the cloth area.

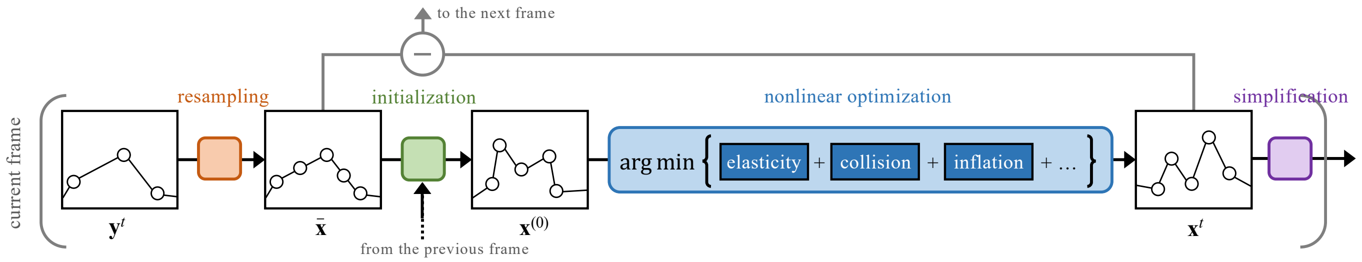


Fig. 3. The pipeline of our system as a quasistatic wrinkle synthesis tool. Given a coarse unstructured mesh input $\mathbf{y}^t$, the system resamples the reference space by a regular grid to build a high-resolution cloth mesh $\bar{\mathbf{x}}$. It then runs an initialization step to estimate wrinkles on the mesh, and solves a nonlinear optimization problem to generate wrinkle details. The system can also simplify the high-resolution output into a compact form for further processing.

Wang [2015] pointed out projective dynamics is basically a special optimization method solving the nonlinear cloth simulation problem. While this discovery alone does not resolve any computational issue, it opens the door to the development of many GPU-based cloth simulators [Fratarcangeli et al. 2016; Wang and Yang 2016; Wang et al. 2018; Wu et al. 2020] with large time steps.

Besides the cloth dynamics solver, collision handling of cloth, especially self collision handling, is another major contributor to the overall simulation cost. Most of the existing collision handling techniques, including both continuous [Bridson et al. 2002; Harmon et al. 2008; Provot 1997] and discrete ones [Baraff et al. 2003; Buffet et al. 2019; Volino and Magnenat-Thalmann 2006], depend heavily on sequential and reduction operations. Parallelizing and accelerating these techniques on GPUs [Lauterbach et al. 2010; Tang et al. 2016, 2018] is possible but difficult, and the accelerated performance still has room for improvement. Meanwhile, GPU-based cloth simulators [Fratarcangeli et al. 2016; Macklin et al. 2014; Stam 2009] often choose to use the vertex repulsion method, which is simple, efficient, but risky, due to no strict protection against penetrations. Wu et al. [2020] solved this issue by the use of dual state vectors and two collision phases, at the expense of extra computational workload and implementation effort.

In summary, graphics researchers have made substantial progress in physics-based cloth simulation lately, but state-of-the-art simulators are still barely capable of achieving real-time simulation performance even at millimeter levels, with approximately 10K to 1M vertices. We also note that previous simulators [Choi and Ko 2002; Selle et al. 2009] typically do not differentiate regular grid meshes too much from unstructured meshes, in terms of their algorithms. Villard et al. [2005] applied the adaptive remeshing idea to regular gird meshes. Goldenthal et al. [2007] chose grid meshes to address the locking issue in inextensibility enforcement. Tang et al. [2013] and Schmitt et al. [2013] used regular grid meshes to simplify matrix and connectivity representations. Pall et al. [2018] adopted red-black ordering of a quadrilateral mesh for fast Gauss-Seidel projective dynamics. As far as we know, no research has extensively explored the computational benefit of the regular grid structure in cloth simulation yet.

Data-driven wrinkle synthesis. Since cloth simulation is so expensive as the mesh resolution increases, a natural idea is to synthesize fine wrinkles on top of the coarse mesh input generated by simulation, sculpturing or skinning. Early techniques [Kim et al. 2013; Wang et al. 2010a; Xu et al. 2014] based on this idea are often known as data-driven, i.e., predicting wrinkles and secondary effects from a large data set, even on different deforming human bodies [Guan et al. 2012; Pons-Moll et al. 2017]. In recent years, researchers [Chentanez et al. 2020; Jin et al. 2020; Lähner et al. 2018] started becoming interested in using data to train deep neural networks for wrinkle synthesis. Some researchers [Neophytou and Hilton 2014; Patel et al. 2020; Santesteban et al. 2019; Vidaurre et al. 2020; Yang et al. 2018] have explored an even more ambitious idea: using deep neural networks to infer the whole clothing shapes with wrinkles. In general, these non-physics-based wrinkle synthesis techniques are good at interpolation, but not at extrapolation. To produce plausible results, they often need an enormous volume of data to cover the rich and complex clothing shape space, which poses many challenges in data acquisition. The acquisition issue becomes even more intractable, once we start to consider high-resolution data at submillimeter levels. Our system can work as a powerful data generation engine for future research in this field.

3 OVERVIEW

In most part of this paper, we will consider the main use of our physics-based simulation system to be an offline quasistatic wrinkle synthesis tool for existing coarse mesh inputs, similar to [Bergou et al. 2007; Müller and Chentanez 2010]. The key strength of this tool is that it places little restriction on the coarse mesh input and it allows an arbitrarily large time gap Δt between two consecutive frames. In our experiment, we perform wrinkle synthesis on each rendered frame selected from every three simulated frames for a faster playback speed. In other words, the time gap is three times the actual time step used by coarse simulation.

Fig. 3 illustrates the pipeline of our current system. The input to this system is a coarse unstructured triangular mesh $\mathbf{y}^t$ at time t, either simulated by a physics-based engine or keyframed by artists, with non-overlapping patches in the reference space. To construct the high-resolution mesh for wrinkle synthesis, we first overlay a high-resolution 2D grid on top of the reference patches, detect every grid vertex inside of the patches, and find the coarse patch triangle containing it. We then use barycentric interpolation to resample the position of every interior grid vertex in the deformed space. These vertices form a resampled mesh in both the deformed space and the

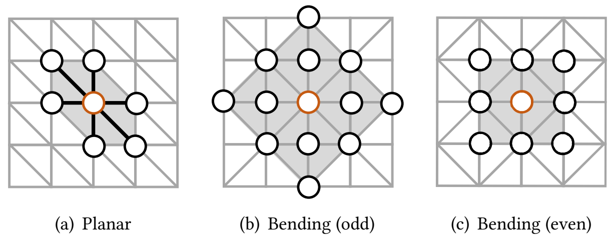


Fig. 4. Mesh tessellations used by planar and bending elastic models. Different from the tessellation used by planar elasticity in (a), the tessellation used by bending elasticity is interleaved to avoid biased bending resistance as shown in (b) and (c).

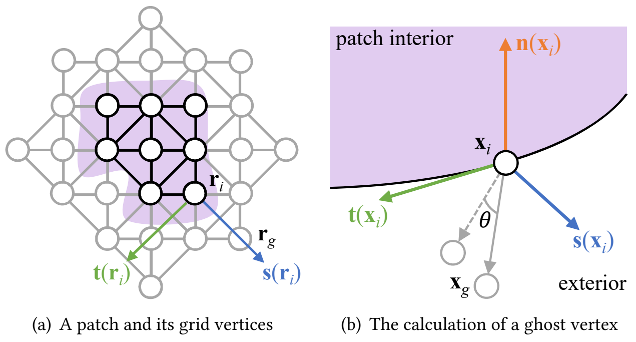

$\mathbf{r}_i$
$\mathbf{r}_g$
$\mathbf{t}(\mathbf{r}_i)$
$\mathbf{s}(\mathbf{r}_i)$
(a) A patch and its grid vertices
patch interior
$\mathbf{n}(\mathbf{x}_i)$
$\mathbf{x}_i$
$\mathbf{t}(\mathbf{x}_i)$
θ
$\mathbf{s}(\mathbf{x}_i)$
$\mathbf{x}_g$
exterior
(b) The calculation of a ghost vertex

Fig. 5. The initially resampled mesh contains only the grid vertices (in dark) inside of the patches (in purple), as shown in (a). To simulate sewing line details, we enrich the boundaries with ghost grid vertices (in gray), whose 3D positions are calculated to form a specified angle with the patch interior.

reference space. For simplicity, we name the originally resampled mesh in the deformed space *the original mesh* and we denote its stacked vertex positional vector by $\bar{\mathbf{x}} \in \mathbb{R}^{3N}$, in which N is the number of vertices. The system then runs an initialization step to transfer wrinkles from the result $\mathbf{x}^{t-\Delta t}$ at the previous frame to $\bar{\mathbf{x}}$ at the current frame, which produces an initialization $\mathbf{x}^{(0)}$. Given the initialization, the system solves a nonlinear optimization problem in the synthesis step to address a series of objectives, including elasticity, collision and inflation. The optimized result is a high-resolution mesh with synthesized wrinkle details. Finally, to make the output mesh size suitable for sharing, rendering and future processing, the system provides a quadtree-based mesh simplification step.

The key challenge in the development of our system comes from the sheer size of our high-resolution mesh. By default, our system uses a 4,096×4,096 grid and sets the grid cell size between 0.4 and 0.6mm for highly detailed wrinkles. This means the number of vertices, N, can be as large as 16M, and it is typically between 6M and 8M as shown in Table 1. Most of the existing simulation techniques have difficulty in handling so many vertices at such a high resolution level. Fortunately, since the mesh is built from a regular grid, we can explore the grid structure to optimize memory accesses and to reduce computational costs.

We note that the aforementioned pipeline provides only one use of our system. In subsection 7.6, we will show that our system is capable of working as a standalone dynamic simulator, by introducing additional terms into the optimization objective. Similarly, we can modify our system to become a component integrable into a multi-resolution simulator, such as geometric and algebraic multi-grids [Lee et al. 2010; Oh et al. 2008; Tamstorf et al. 2015] and non-linear multigrid with the full approximation scheme (FAS) [Wang et al. 2018].

4 NONLINEAR OPTIMIZATION

To begin with, we consider the simulation of cloth wrinkles as an unconstrained nonlinear optimization problem:

$$\mathbf{x} = \arg\min F(\mathbf{x}), \tag{1}$$

in which $\mathbf{x} \in \mathbb{R}^{3N}$ is the stacked positional vector of N vertices and $F(\mathbf{x})$ is the objective function. For simplicity, we consider $F(\mathbf{x})$ to include only elastic potentials in this section, which turns Eq. 1 into a quasistatic simulation problem. We will discuss more potentials for various effects and dynamic simulation in other sections.

4.1 Elastic Potentials

Given the original mesh constructed from a 2D regular grid, we build planar elasticity upon the positional relationship between each vertex and its six neighbors, as shown in Fig. 4a. Currently, our system supports two planar elastic models: the mass-spring model and the triangular finite element model. Under the mass-spring model, we calculate the rest lengths of the six edges for each vertex and store them as half floats into a list. Under the triangular finite element model, we store the six rest edge vectors into a list and use them to calculate the deformations of the six triangles.

To handle bending elasticity, our current system uses the quadratic bending model [Bergou et al. 2006], which relies on the relationship among the four vertices of two adjacent triangles. Although we can use the same triangle tessellation as in Fig. 4a, we choose not do so because it can cause biased bending resistance. Instead, we choose an interleaved tessellation as shown in Fig. 4b and 4c, in which an odd vertex is related to its twelve neighbors while an even vertex is related to its eight neighbors. We calculate the quadratic bending weights of these neighbors and store them as half floats into a list for every vertex.

If the mesh is unmodified in its reference configuration, the aforementioned lists storing elastic constants are the same for all of the vertices and there is no need to assign different lists to different vertices as in [Tang et al. 2013]. Instead we allow the same lists to be shared in the constant memory, resulting in significantly lower memory access costs. Having said that, we will show how a user can interactively modify the reference mesh in Subsection 6.2. In the modified regions, elastic constant lists are no longer the same and must be stored in the global memory.

4.2 Boundary Conditions

We consider two types of boundary conditions at cloth patch boundaries: *open* and *closed*.

According to the resampling procedure in Section 3, all of the patch boundaries are initially open: the mesh contains only those grid vertices inside of the patches, as shown in Fig. 5a. As a result, grid vertices near open boundaries do not have enough neighbors for the elastic models in Subsection 4.1. Instead of modifying elastic

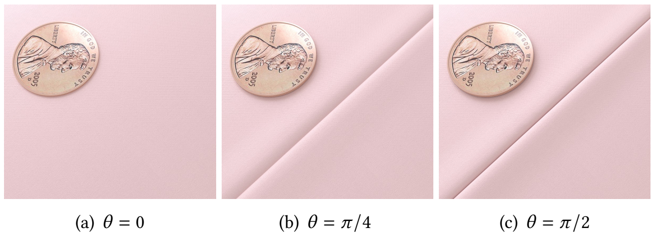


Fig. 6. The sewing lines formed by closed patch boundaries with different angle values. A greater angle value tends to produce curvy and bumpy boundaries, as shown in (c).

constant lists, we use the same lists and assume that non-existing neighbors are identical to the central vertex in the kernel function. We found this to be faster than using multiple versions of elastic constant lists.

We can turn a patch boundary into a closed one, by procedurally generating a narrow band of ghost vertices surrounding it. Doing this not only addresses the missing neighbor issue, but also enables the creation of sewing line details caused by different sewing methods. Specifically, for each ghost vertex $\mathbf{r}_g$ within the narrow band in the 2D reference space, we calculate its local coordinates[2] with respect to the local space of its closest interior vertex $\mathbf{r}_i$:

$$a_t = (\mathbf{r}_g - \mathbf{r}_i) \cdot \mathbf{t}(\mathbf{r}_i), \qquad a_s = (\mathbf{r}_g - \mathbf{r}_i) \cdot \mathbf{s}(\mathbf{r}_i), \tag{2}$$

where $\mathbf{t}(\mathbf{r}_i)$ and $\mathbf{s}(\mathbf{r}_i)$ are the tangent and the binormal at $\mathbf{r}_i$. We then compute the new vertex position $\mathbf{x}_g$ in 3D, which forms a specified angle θ at the closest interior vertex $\mathbf{x}_i$, as Fig. 5b shows:

$$\mathbf{x}_g = \mathbf{x}_i + \cos\theta\big(a_t\mathbf{t}(\mathbf{x}_i) + a_s\mathbf{s}(\mathbf{x}_i)\big) - \sin\theta\sqrt{a_t^2 + a_s^2}\,\mathbf{n}(\mathbf{x}_i), \tag{3}$$

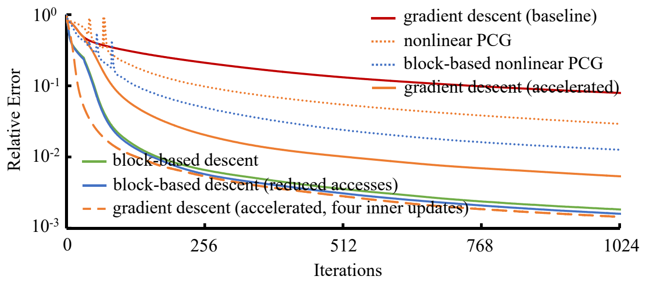
in which $\mathbf{n}(\mathbf{x}_i)$, $\mathbf{t}(\mathbf{x}_i)$ and $\mathbf{s}(\mathbf{x}_i)$ are the normal, the tangent and the binormal at $\mathbf{x}_i$. In our system, a user can either specify ghost vertices only once before wrinkle optimization, which makes the sewing lines unmovable, or update them from time to time. Fig. 6 shows the simulated results with different angle values.

We note that if patch boundaries at a sewing line are open, or if patch boundaries at a sewing line are closed with updated ghost vertices, we must apply position-based constraints on corresponding boundary vertices during wrinkle optimization, to keep patches stitched together.

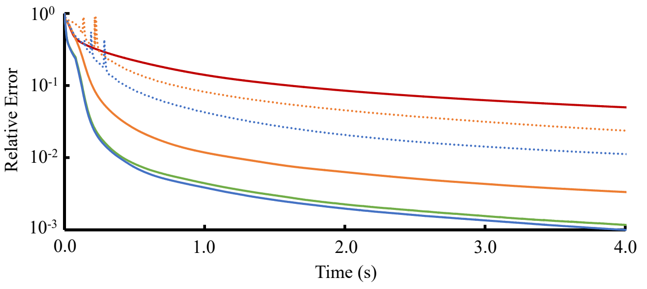
4.3 A Block-Based Descent Method

The main challenge involved in the development of our system is how to solve the optimization problem in Eq. 1. Fig. 7a shows that gradient descent preconditioned by diagonal Hessian [Wang and Yang 2016], serving as our baseline method, converges slowly as expected. We can apply two immediate improvements on gradient descent. First, we can adopt Chebyshev acceleration [Wang 2015], which improves the convergence rate as Fig. 7a shows. Second, we can use shared memory within each non-overlapping GPU thread block, naturally created from grid partitioning. However the experiment reveals that the use of shared memory increases the speed

[2]Theoretically, a_t should be zero if $\mathbf{r}_i$ is the exact closest point on the continuous patch boundary. But since $\mathbf{r}_i$, $\mathbf{t}(\mathbf{r}_i)$ and $\mathbf{s}(\mathbf{r}_i)$ are all discretely calculated, a_t may be not zero.



(a) Convergence rates of different methods



(b) Convergence speeds of different methods

Fig. 7. Convergence rates and speeds of different methods. Compared with the (preconditioned) gradient, the search direction calculated by our block-based approach reduces the optimization objective more effectively with a low computational overhead. Based on this approach, our block-based descent method outperforms gradient descent as shown in (a) and (b), both with Chebyshev acceleration. Our approach can also work as a preconditioner for improving the performance of nonlinear PCG. By default, we use the shirred dress example for convergence analysis in this paper. We define the relative error in the k-th iteration as: $\big(F(\mathbf{x}^{(k)}) - F(\mathbf{x}^*)\big)/\big(F(\mathbf{x}^{(k)}) - F(\mathbf{x}^*)\big)$, in which $\mathbf{x}^*$ is the exact solution estimated after a large number of iterations.

by only four percent. The reason shared memory fails to greatly improve the performance is because each block needs to load its part from global memory into shared memory in every single iteration, which compromises the benefit of shared memory. In contrast, if the block is able to cover all of the vertices, we can load shared memory once for all and execute the iterations inside of the kernel function. This practice is feasible only when the number of vertices[3] is small [NVIDIA 2021].

While we are unable to load millions of vertices into shared memory at once, we are intrigued by an interesting question: *what if we run L inner iterations just for the threads in the same non-overlapping GPU block, so that these iterations can reuse the shared memory data being loaded only once*? This idea is closely related to domain decomposition techniques [Widlund and Toselli 2004], especially the nonlinear additive Schwarz method [Brune et al. 2015]. Since we treat non-overlapping blocks as subdomains, we name this idea the *block-based* approach and we combine it with gradient descent to form a *block-based descent* method. Basically, the method updates all of the blocks in parallel, and the update of each block involves

[3]The total amount of shared memory per block is 49,152 bytes on an NVIDIA GeForce GTX 2080 Ti GPU, which is enough for storing 4,096 vertex positions at most.

L inner gradient descent iterations. Let $\mathbf{x}_B$ be the stacked vertex vector of block B, as a sub-vector of $\mathbf{x} = \{\dots, \mathbf{x}_{B-1}, \mathbf{x}_B, \mathbf{x}_{B+1}, \dots\}$. The update of $\mathbf{x}_B$ in the l-th inner iteration is:

$$\mathbf{x}_B^{[l+1]} = \mathbf{x}_B^{[l]} - \alpha \mathbf{P}_B^{-1}\left(\mathbf{x}_B^{[l]}\right) \mathbf{g}_B\left(\mathbf{x}_B^{[l]}\right), \tag{4}$$

in which α is the step size, $\mathbf{P}_B(\mathbf{x}_B) = \mathbf{P}_B\left(\dots, \mathbf{x}_{B-1}^{[0]}, \mathbf{x}_B, \mathbf{x}_{B+1}^{[0]}, \dots\right)$ is the positive definite preconditioner matrix of block B, and $\mathbf{g}_B(\mathbf{x}_B) = \nabla_B F\left(\dots, \mathbf{x}_{B-1}^{[0]}, \mathbf{x}_B, \mathbf{x}_{B+1}^{[0]}, \dots\right)$ is the gradient of block B. In our experiment, the preconditioner is the diagonal of the Hessian matrix.

Alg. 1 provides the pseudo code of this method. For simplicity, we omit the details related to Chebyshev acceleration, which effectively accelerates both inner and outer iterations. Here the block width is W and the width of the shared memory data is $W + 4$, because the update of each vertex needs the vertices in the 2-ring neighborhood according to Subsection 4.1.

To understand why the block-based descent method works, let us consider its difference from gradient descent. Assuming that the gradient and the Hessian are continuous, we know if:

$$\mathbf{x}_B^{[l]} = \mathbf{x}_B^{[0]} - l\alpha \mathbf{P}_B^{-1}\left(\mathbf{x}_B^{[0]}\right) \mathbf{g}_B\left(\mathbf{x}_B^{[0]}\right) + O(\alpha^2), \tag{5}$$

then

$$\begin{cases} \mathbf{P}_B^{-1}\left(\mathbf{x}_B^{[l]}\right) = \mathbf{P}_B^{-1}\left(\mathbf{x}_B^{[0]}\right) + O(\alpha), \\ \mathbf{g}_B\left(\mathbf{x}_B^{[l]}\right) = \mathbf{g}_B\left(\mathbf{x}_B^{[0]}\right) + O(\alpha), \end{cases} \tag{6}$$

which leads to

$$\mathbf{x}_B^{[l+1]} = \mathbf{x}_B^{[0]} - (l+1)\alpha \mathbf{P}_B^{-1}\left(\mathbf{x}_B^{[0]}\right) \mathbf{g}_B\left(\mathbf{x}_B^{[0]}\right) + O(\alpha^2). \tag{7}$$

Together, the overall update of $\mathbf{x}$ after applying L inner iterations to all of the blocks is:

$$\mathbf{x}^{[L]} = \mathbf{x}^{[0]} - L\alpha \mathbf{P}_{\text{diag}}^{-1}\left(\mathbf{x}^{[0]}\right) \mathbf{g}\left(\mathbf{x}^{[0]}\right) + O(\alpha^2). \tag{8}$$

in which $\mathbf{P}_{\text{diag}}(\mathbf{x}) = \text{diag}(\dots, \mathbf{P}_B(\mathbf{x}), \dots)$ is the positive definite block-based preconditioner matrix. When α is sufficiently small, $-\mathbf{P}_{\text{diag}}^{-1}\left(\mathbf{x}^{[0]}\right) \mathbf{g}\left(\mathbf{x}^{[0]}\right) + O(\alpha)$ must be a valid descent direction and the method must be convergent. In reality, $O(\alpha)$ is not always the trouble. Since we obtain the search direction from L inner iterations, we expect it to outperform the (preconditioned) gradient, but be outperformed by the joint direction from L (preconditioned) gradient descent updates, as shown in Fig. 7a.

An important question is how to decide L and α. We want a large L to minimize the transfer from global memory to shared memory, but we cannot make L too large, or the error will force α to decrease by step size adjustment. If we compare our search direction with the direction from L gradient descent updates, we see that the difference is due to block boundaries. This suggests L to be proportional to the block width W. In our experiment, when $W = 8$, $L = 8$ without inner Chebyshev acceleration or $L = 4$ with inner Chebyshev acceleration, we can use $\alpha = 0.4$ in most of the examples without triggering step size adjustment.

ALGORITHM 1: A Block-Based Descent Method

```
Input: The original mesh $\bar{\mathbf{x}}$, the block width $W$, the number of outer
       iterations $K$, the number of inner iterations $L$.
$\mathbf{x}^{(0)}$ ←Initialize($\bar{\mathbf{x}}$, ...);
for $k = 0 \dots K-1$ do
    for each non-overlapping thread block B do
        __shared__ X[$W$+4][$W$+4];
        Global_to_Share_Transfer(X, $\mathbf{x}^{(k)}$, B, $W$);
        __syncthreads();
        $\mathbf{x}_B^{[0]}$ ← X[2:$W$+1][2:$W$+1];
        for $l = 0 \dots L-1$ do
            $\mathbf{x}_B^{[l+1]} \leftarrow \mathbf{x}_B^{[l]} - \alpha \mathbf{P}_B^{-1}(\mathbf{X})\mathbf{g}_B(\mathbf{X})$;
            X[2:$W$+1][2:$W$+1] ← $\mathbf{x}_B^{[l+1]}$;
            __syncthreads();
        end
        temp$_B$ ← X[2:$W$+1][2:$W$+1];
    end
    $\mathbf{x}^{(k+1)}$ ← temp;
end
return $\mathbf{x}^{(K)}$;
```

4.3.1 *Reduced shared memory accesses.*

So far we have discussed the use of inner iterations to reduce global-shared memory transfer, for maximizing the benefit of shared memory. An interesting follow-up question is: *can we further reduce shared memory accesses in every inner iteration*? Our answer to this question comes from the observation that out of the twelve (or eight) neighbors needed for each vertex update, the six (or two) solely related to bending are outside of the 1-ring neighborhood and their contributions are small, as shown in Subsection 4.1. Therefore, we propose to use their old positions at the beginning of the kernel function and formulate this technique as follows:

$$\nabla_i F^{\text{bend}}\left(\mathbf{x}_B^{[l]}\right) = \sum_{j \notin \mathcal{N}_i} w_{ij}\mathbf{x}_j^{[0]} + \sum_{j \in \mathcal{N}_i} w_{ij}\mathbf{x}_j^{[l]} - \sum_j w_{ij}\mathbf{x}_i^{[l]}, \tag{9}$$

in which $\nabla_i F^{\text{bend}}$ is the gradient of the bending potential F^{bend} at vertex i, w_{ij} is the quadratic bending weight between vertex i and j, and $\mathcal{N}_i$ is the 1-ring neighborhood. Since the leftmost term of Eq. 9 is constant in all of the inner iterations, we precompute it at beginning of the kernel with only six (or two) shared memory accesses. A little bit to our surprise, this technique improves both the convergence rate and the cost per iteration, although the overall improvement is not so significant as shown in Fig. 7.

4.3.2 *Nonlinear preconditioned conjugate gradient (PCG).*

We can also view our block-based approach as a preconditioner for other optimization methods, such as nonlinear preconditioned conjugate gradient. Fig. 7 shows such a block-based nonlinear PCG method outperforms the original nonlinear PCG method, both preconditioned by diagonal Hessian. Fig. 7 also shows nonlinear PCG runs slower than accelerated gradient descent, without and with the block-based approach, for two reasons. First, nonlinear PCG needs frequent restart to eliminate the divergence risk, especially in the first few iterations. In our experiment, we restart nonlinear PCG every 14 outer iterations. Second, nonlinear PCG needs one reduction per outer iteration, which causes 50 percent more cost. We note that these problems can possibly be addressed by more sophisticated schemes, but they are beyond the scope of this research.

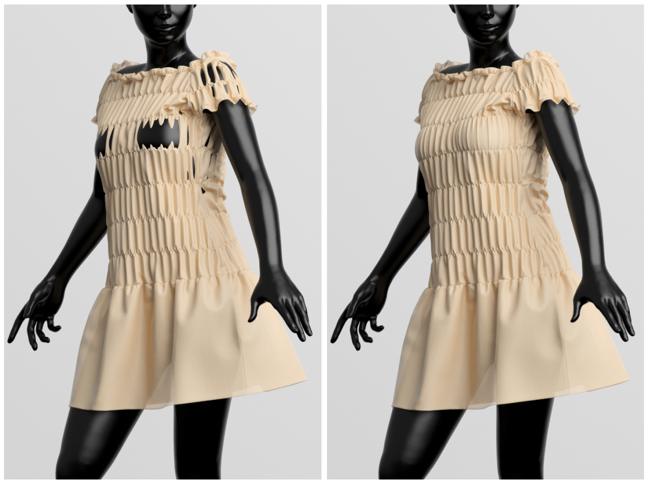

(a) Without cloth-body collision handling
(b) With cloth-body collision handling

Fig. 8. The mini dress example without and with cloth-body collision handling. Our system applies a fast way to handle cloth-body collisions, by using a half plane to approximate the local body around each cloth vertex.

5 COLLISION HANDLING

In this section, we will introduce the next important component: collision handling. Collision handling is notoriously known for its large cost, even for low-resolution meshes. Since we are dealing with meshes with millions of vertices, we do not have the luxury of using existing techniques developed for unstructured meshes. Instead we must explore the advantage of the underlying grid structure and find a balance between the efficiency and the accuracy.

5.1 Cloth-Body Collision Handling

Many existing GPU-based cloth simulators handle cloth-body collisions as a projection step at the end of every iteration, typically by the level set method. We would like to use the projection step at the end of every inner iteration as well, but we cannot afford running the level set method for millions of vertices. Instead we approximate the local body surface near $\mathbf{x}_i$ as a plane specified by $\hat{\mathbf{x}}_i$ and $\hat{\mathbf{n}}_i = \left(\mathbf{x}_i^{(0)} - \hat{\mathbf{x}}_i\right) \Big/ \left\|\mathbf{x}_i^{(0)} - \hat{\mathbf{x}}_i\right\|$, in which $\hat{\mathbf{x}}_i$ is the projection of $\mathbf{x}_i^{(0)}$ on the body surface by the level set method. We then formulate the projection step as:

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \max\Big(-(\mathbf{x}_i - \hat{\mathbf{x}}_i) \cdot \hat{\mathbf{n}}_i, 0\Big)\hat{\mathbf{n}}_i. \tag{10}$$

Under the assumption that vertices do not slide much during wrinkle optimization, our method has demonstrated its effectiveness as shown by the mini dress example in Fig. 8. Under the same assumption, we can further eliminate the projection step and its cost for any vertex originally far from the body, i.e., $\left(\mathbf{x}_i^{(0)} - \hat{\mathbf{x}}_i\right) \cdot \hat{\mathbf{n}}_i \gg 0$. We note that since $\hat{\mathbf{n}}_i$ and $\hat{\mathbf{x}}_i \cdot \hat{\mathbf{n}}_i$ are constant in Eq. 10, we can precompute them to lessen the memory access cost.

5.2 Cloth Self Collision Handling

Similar to other GPU-based simulators, our simulator uses the vertex repulsion method to push two vertices apart, if they are too close and they are not in each other's 1-ring neighborhood. Mathematically, we describe this as an additional potential to $F(\mathbf{x})$:

$$F_{ij}^{\text{self}}(\mathbf{x}) = \frac{\kappa^{\text{self}}}{2} \max\left(R^{\text{self}} - \left\|\mathbf{x}_i - \mathbf{x}_j\right\|, 0\right)^2, \tag{11}$$

for any i and j satisfying $i \notin \mathcal{N}_j$ and $j \notin \mathcal{N}_i$, in which κ^{self} is the collision strength coefficient and R^{self} is the collision distance threshold. The vertex repulsion method is known for two limitations. First, it cannot resolve existing self intersections. Second, it cannot strictly prevent self collisions, especially when vertices move too fast or triangles stretch too much. Fortunately, these limitations are not so problematic in our system as shown in Fig. 10, under the assumption that the coarse mesh input is intersection-free and the step size is sufficiently small.

5.3 Proximity Search

The bottleneck of the vertex repulsion method is to find those vertices close to each other, i.e., *proximity search.* To perform fast proximity search over millions of vertices, we design a novel GPU-based method specifically for our structured meshes. Our key idea is to use collision blocks, rather than vertices, as the basic elements of a spatial-partitioning grid, so that we can reduce the workload of spatial-partitioning to a tractable level. In our system, we define collision blocks as non-overlapping squares of 4×4 vertices. We calculate the bounding spheres of collision blocks, and construct the spatial-partitioning grid with its cell size equal to the maximal bounding sphere radius. In this way, we can easily perform broad-phase culling by checking every two blocks in adjacent grid cells, and store those block pairs with intersecting bounding spheres into a list. One issue is that the list can be too long to build by atomic operations on a GPU, given the fact that neighboring pairs are likely to have intersecting bounding spheres and there are millions of such pairs. To address this issue, we classify the pairs into two categories as shown in Fig. 9:

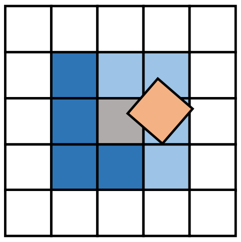
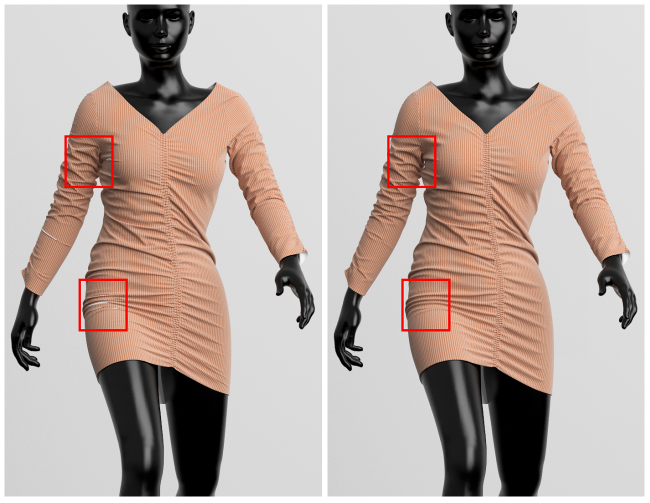


Fig. 9. Collision blocks. The central block (in gray) forms block pairs with other colored blocks, including itself, if their bounding spheres intersect.

- A block forms a *long-range* pair with another block (in orange), if their bounding volumes intersect and they are not neighbors.
- A block forms a *short-range* pair with another block (in blue), if they are neighbors. We view a block as its own neighbor, so it forms a short-range pair with itself as well.

During broad-phase culling, we consider only long-range pairs and we store them into the list. Compared with the number of short-range pairs, the number of long-range pairs is much smaller, typically under 10K in our experiment[4].

[4]Long-range pairs can be much more common in more complex collision cases, such as multi-layered clothing. But they should still be fewer than short-range pairs.


(a) Without cloth self collision handling (b) With cloth self collision handling

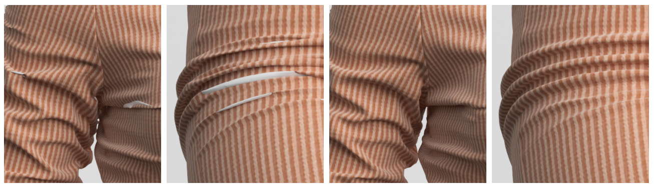

(c) Closeup (top) (d) Closeup (bottom) (e) Closeup (top) (f) Closeup (bottom)

Fig. 10. The body-con dress example without and with applying cloth self collision handling. Specifically designed for regular grid meshes, our GPU-friendly vertex repulsion method allows our system to protect its results from self intersections, as shown in (b), (d) and (f).

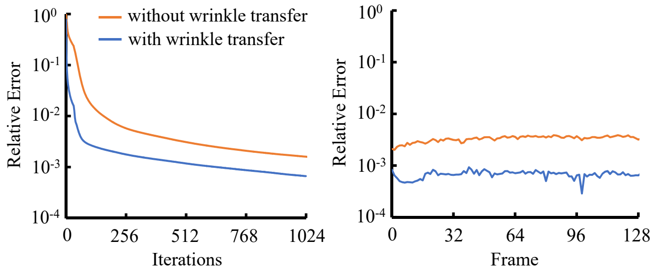
After broad-phase culling, we use two kernel functions to detect any two vertices whose distance is below the collision threshold R^{self}. The first kernel function tests every two vertices from the two blocks in each long-range pair, while the second kernel function tests every two vertices from each block and its neighbor, i.e., a short-range pair. Since the proximity relationship is mutual, we launch the second kernel for only half of each block's neighbors, including those dark blue ones in Fig. 9 and itself, which cuts the cost of the second kernel by half. We store each detected vertex pair into the proximity lists of the two vertices and use these proximity lists to provide collision handling in the optimization process.

6 APPLICATION-RELATED TOPICS

Given the basic system outlined in Section 4 and 5, we would like to discuss the remaining issues involved in the system pipeline in Fig. 3 next. These issues include initialization for fast convergence and temporal coherence (in Subsection 6.1), gathering effects (in Subsection 6.2), inflation effects (in Subsection 6.3) and GPU-based mesh simplification (in Subsection 6.4).

6.1 Initialization

In this subsection, we will discuss the initialization approaches for the wrinkle optimization process. A good initialization $\mathbf{x}^{(0)}$ serves two purposes: to help the optimization reach its convergence fast; and to achieve temporal coherence when the system handles a sequence of simulated coarse meshes.



(a) Convergence rates (b) Errors over frames

Fig. 11. The convergence behaviors of the wrinkle optimization process, without and with using the temporal wrinkle transfer approach. The figure shows that the use of this approach allows the optimization to reach a smaller residual error after the same number of iterations.

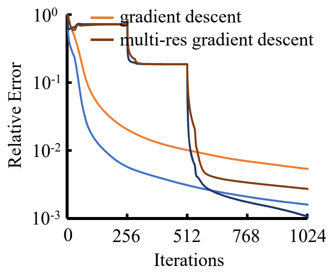
6.1.1 Temporal wrinkle transfer. When the system handles a simulated coarse animation sequence, it is natural to consider transferring synthesized wrinkles from the previous frame to the current frame as an initialization. Let $\bar{\mathbf{x}}^{t-\Delta t}$ and $\mathbf{x}^{t-\Delta t}$ be the originally resampled mesh and the synthesized final mesh at the previous frame. A naïve idea is to simply apply their difference to the originally resampled mesh $\bar{\mathbf{x}}^{t}$ at the current frame: $\mathbf{x}^{(0)} = \bar{\mathbf{x}}^{t} + \mathbf{x}^{t-\Delta t} - \bar{\mathbf{x}}^{t-\Delta t}$. Unfortunately, this approach does not work well, since cloth often undergoes nonlinear motion, including rotation, especially when Δt is large.

Our solution is to initialize $\mathbf{x}^{(0)}$ using the local coordinate system instead. To begin with, we build the local coordinate system of every vertex in $\bar{\mathbf{x}}^{t-\Delta t}$ and use that to calculate the local displacement from $\bar{\mathbf{x}}^{t-\Delta t}$ to $\mathbf{x}^{t-\Delta t}$. We then apply this local displacement to $\bar{\mathbf{x}}^{t}$, using the local coordinate system of $\bar{\mathbf{x}}^{t}$ this time. The outcome $\mathbf{x}^{\text{est}}$ is an estimation of the wrinkled mesh at the current frame, but it may contain self intersections. To avoid self intersections, we borrow the idea developed by Wu et al. [2020] and formulate a simplified optimization process, starting from $\bar{\mathbf{x}}^{t}$:

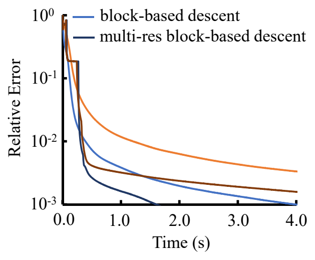
$$\mathbf{x}^{(0)} = \arg\min_{\mathbf{x}} \left\|\mathbf{x} - \mathbf{x}^{\text{est}}\right\|^2 + F^{\text{self}}(\mathbf{x}), \tag{12}$$

where $F^{\text{self}}(\mathbf{x})$ is the self collision potential given in Subsection 5.2. To solve Eq. 12, we apply the same solver developed for the wrinkle optimization process. Because $\mathbf{x}^{(0)}$ is an initialization, we do not need to calculate it exactly and we choose to run a fixed number of iterations with a small constant step size. Fig. 11 shows this wrinkle transfer approach improves the convergence of the following optimization process. The approach is also important to temporal coherence. Without initialization, the optimization tends to produce flickering wrinkle artifacts, caused by local minima corresponding to different quasistatic states.

6.1.2 A multi-resolution approach. To further speed up the convergence of the wrinkle optimization process, we construct a mesh hierarchy based on the underlying regular grid structure and solve



(a) Convergence rates



(b) Convergence speeds

Fig. 12. The convergence behaviors of the optimization process, without and with applying the multi-resolution approach. The multi-resolution approach is compatible with both gradient descent and block-based descent, for speeding up the convergence of the optimization process.

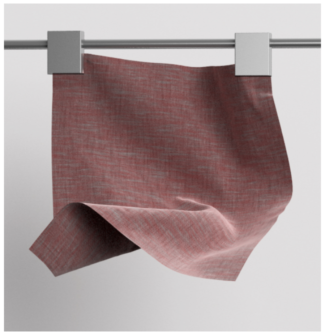
the optimization process in a multi-resolution fashion. Specifically, we formulate an optimization problem at each hierarchical level, with its elastic and collision constants adjusted to better approximate the original high-resolution problem. We then solve these problems from coarse to fine and upsample the coarser result as an initialization to the finer problem. Fig. 12 demonstrates the effectiveness of this multi-resolution approach by three hierarchical levels: 1,024× 1,024, 2,048×2,048 and 4,096×4,096. We choose not go any coarser than 1,024× 1,024, since patch boundaries would not be accurately represented.

To apply the two aforementioned initialization approaches together, we simply use the wrinkle transfer approach to initialize the wrinkled mesh at the coarsest level first, and then solve the optimization problems from coarse to fine. Doing this allows us to reduce the initialization cost associated with Eq. 12 as well. We note that upsampling does not cause any self intersection and we do not need to address it as in the wrinkle transfer approach.

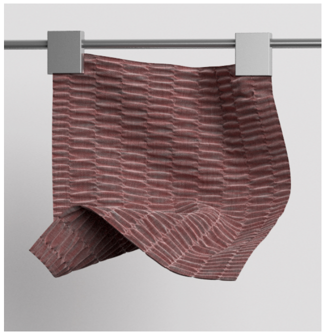
6.2 Gathering

Shirring and other gathering techniques are popularly used by fashion designers to create fine wrinkle details. Our system can immediately refine simulated coarse wrinkles caused by gathering, if the mesh input has already encoded the sewing relationship between two patch boundaries with unequal lengths. But if not, our system also provides an option for a user to interactively modify the reference mesh, so as to generate new gathering effects with flexibility and convenience, as shown in Fig. 13.

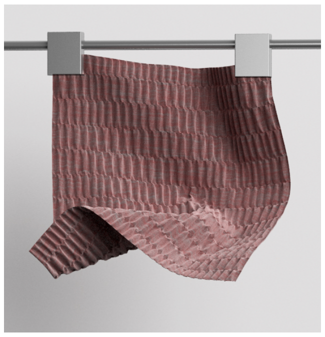
A naïve implementation of this option is to directly modify the reference mesh vertices and recalculate the elastic constants described in Subsection 4.1. However, we found this implementation to be not so user friendly in practice, especially after the mesh gets severely modified. Instead we implement a deformation tensor field that uses a 2D matrix $\mathbf{S}_i$ to describe the local transformation at vertex i from the original reference mesh to the desired reference mesh. From a different perspective, we can view $\mathbf{S}_i$ as the inverse of the deformation gradient from the desired reference mesh to the original one. Let $\mathbf{r}_i$ and $\mathbf{r}_j$ be the original reference positions of two vertices connected by an edge, and $\mathbf{S}_i$ and $\mathbf{S}_j$ be their deformation



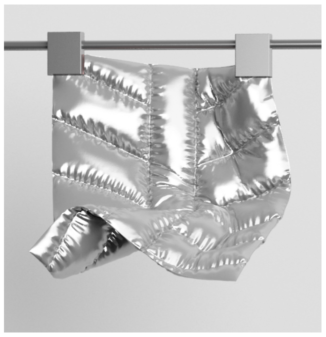
(a) The coarse input



(b) The vertically shirred result



(c) The horizontally shirred result



(d) The inflated result

Fig. 13. The square patch example. Given the coarse mesh input shown in (a), our system allows a user to interactively design shirring and inflation effects on the mesh with fine wrinkles, as shown in (b), (c) and (d).

tensors. We obtain the desired reference edge as:

$$\tfrac{1}{2}(\mathbf{S}_i+\mathbf{S}_j)(\mathbf{r}_j-\mathbf{r}_i). \tag{13}$$

We can then recalculate the desired reference edge length and use edge lengths to recompute the desired bending constants.

We allow a user to form such a deformation tensor field either by a procedural function, or by interaction. Initially, we set all of the tensors as identity, representing zero deformation. To interactively model the tensor field, a user can sketch a line segment from $\mathbf{r}_i$ to $\mathbf{r}_j$, which provides the following tensor:

$$\mathbf{S}=\left[\tilde{\mathbf{r}}_{ij}\ \ \mathbf{R}(90^\circ)\,\tilde{\mathbf{r}}_{ij}\right]\mathrm{diag}(s_0,s_1)\left[\tilde{\mathbf{r}}_{ij}\ \ \mathbf{R}(90^\circ)\,\tilde{\mathbf{r}}_{ij}\right]^{\mathsf{T}}, \tag{14}$$

in which $\tilde{\mathbf{r}}_{ij}=(\mathbf{r}_j-\mathbf{r}_i)/\left\|\mathbf{r}_j-\mathbf{r}_i\right\|$, $\mathbf{R}(90^\circ)$ is the 2D rotation matrix by 90°, and s_0 and s_1 are two user-controlled scaling variables. Intuitively, $\mathbf{S}$ scales the line segment direction by s_0 and its orthogonal direction by s_1. Given $\mathbf{S}$ being calculated, we blend it smoothly into the existing tensor field based on various factors, such as the distance from a vertex to the line segment. Fig. 13b and 13c show two gathered wrinkle results simulated by our system, thanks to the use of interactively modeled deformation tensor fields. To add irregularities, our system also provides random perturbation to the terms in Eq. 14, and Fig. 1 demonstrates such results.

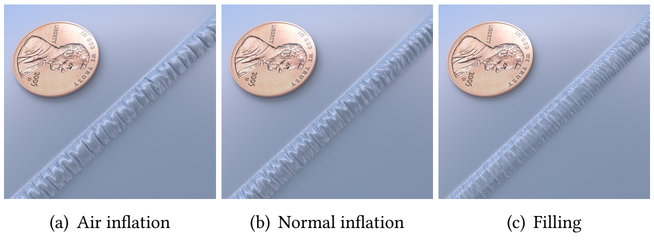


Fig. 14. The three models adopted by our system for modeling inflation and filling effects. The actual simulation of complex cloth appearance often involves the use of more than one models.

6.3 Inflation and Filling

Real multi-layered clothing can be filled with air or stuffing material among its layers. Our system offers three models for the simulation of such effects.

The first model, derived from the maximization of the enclosed volume, addresses air inflation. Its gradient at vertex i is:

$$\nabla_i F^{\mathrm{air}}(\mathbf{x}) = \kappa^{\mathrm{air}} \sum_{j,k \in \mathcal{N}_i} (\mathbf{x}_j - \mathbf{x}_i) \times (\mathbf{x}_k - \mathbf{x}_i), \tag{15}$$

in which κ^{air} is the air inflation strength coefficient, and j and k are two consecutive vertices in vertex i's neighborhood. Intuitively, this model tries to push vertex i in its area-weighted normal direction and it can be naturally implemented together with the triangular element model. But since it demands additional computational cost if it works with the mass spring model, we introduce the second model for similar air inflation effects with much less cost:

$$\nabla_i F^{\mathrm{norm}}(\mathbf{x}) = \kappa^{\mathrm{norm}} \bar{\mathbf{n}}_i, \tag{16}$$

where κ^{norm} is its strength coefficient and $\bar{\mathbf{n}}_i$ is the normal of vertex i by the original mesh $\bar{\mathbf{x}}$. Fig. 14 compares the results of these two models. In general, they are visually indistinguishable when strength coefficients are small.

Finally, the third model addresses filling effects when clothing is stuffed with soft material. This model assumes that the material tries to expand to a certain thickness D, and its resistance is linearly proportional to compression:

$$\nabla_i F^{\mathrm{fill}}(\mathbf{x}) = \kappa^{\mathrm{fill}} \max\left(D - (\mathbf{x}_i - \bar{\mathbf{x}}_i) \cdot \bar{\mathbf{n}}_i, 0\right) \bar{\mathbf{n}}_i, \tag{17}$$

in which κ^{fill} is its strength coefficient and $\bar{\mathbf{x}}_i$ is the original position of vertex i. Fig. 14c shows the effect of this model, in which cloth embeds a 3mm-thick 6mm-wide elastic band.

6.4 Quadtree-Based Mesh Simplification

Once we obtain the synthesized high-resolution mesh, we can use it either to create a high-resolution normal map for the original coarse mesh, or to replace the coarse mesh entirely in various applications. In the latter case, mesh simplification becomes an inevitable step, as the high-resolution mesh would be too large for sharing, rendering or processing by other tools. There exist many off-the-shelf mesh simplification techniques suitable for accomplishing this task, including some developed on GPUs [DeCoro and Tatarchuk 2007; Papageorgiou and Platis 2015]. But since the high-resolution mesh is built upon a regular grid, it becomes natural to implement a quadtree-based mesh simplification algorithm that can be well accelerated on a GPU.

Table 1. The statistics and the performances of our examples. To simulate each frame, the system solves the optimization problems from coarse to fine at three resolution levels: 1K, 2K and 4K. In total, we measure the cost spent on each frame, excluding the initialization cost and the output cost, by running 256 iterations at each level.

Name	Per-Iteration Cost (ms)			Total
(#Verts., #Tri., Ref.)	1K	2K	4K	(s)
shirred dress (7.3M, 14.6M, Fig. 1)	0.175	0.621	2.445	0.830
mini dress (6.0M, 12.0M, Fig. 8)	0.143	0.521	2.171	0.726
body-con dress (6.2M, 12.4M, Fig. 10)	0.165	0.596	2.403	0.810
down coat (7.6M, 15.2M, Fig. 16)	0.213	0.795	3.406	1.130
quilt dress (6.9M, 13.9M, Fig. 19)	0.188	0.693	2.818	0.947
shirred patch (1.0M, 2.0M, Fig. 13)	0.041	0.117	0.444	0.154
inflated patch (2.0M, 4.0M, Fig. 13)	0.067	0.188	0.722	0.250

Our mesh simplification algorithm involves three phases. In Phase 1, it checks every cell in the grid hierarchy from top to bottom, to see if any of its interior vertices is on patch boundaries or departs too much from bilinear interpolation of the four cell corners. If so, this cell contains details and must be labeled as a subdivided one. In Phase 2, the algorithm checks every unsubdivided cell in the hierarchy, to see if any of its eight cell neighbors at one lower level is subdivided. If so, we change such a cell to be subdivided as well, as Fig. 15 shows. The algorithm repeats the checks, until no unsubdivided cell can be changed. The purpose of Phase 2 is to ensure that an unsubdivided cell contains no more than one inner vertex within each of its four edges. Finally, in Phase 3, the algorithm triangulates a cell in the hierarchy, if it is unsubdivided while its parent at one upper level is subdivided. Thanks to Phase 2, the cell contains at most four inner vertices within its edges, which can produce at most six triangles after triangulation. We note that Phase 2 is not mandatory, but it greatly simplifies the triangulation process in Phase 3, making the algorithm balanced and suitable for GPU implementation.

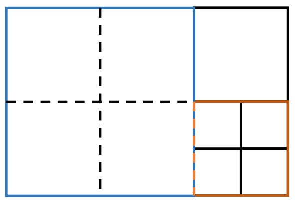
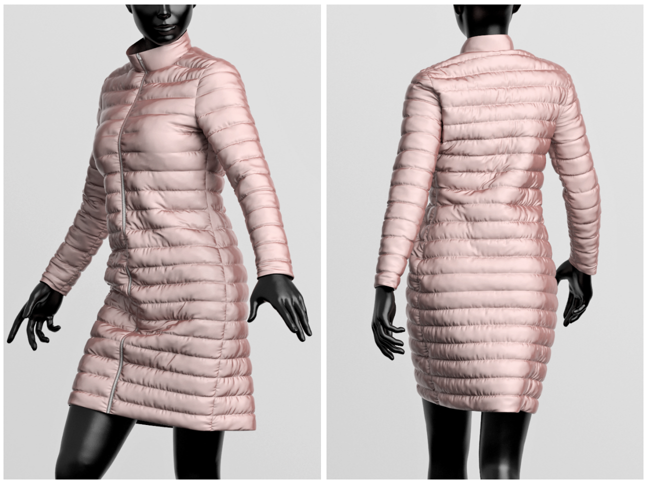


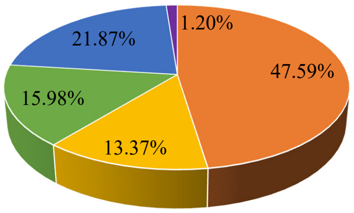
Fig. 15. Unsubdivided cells at three hierarchical levels. If an unsubdivided cell (in blue) is neighboring to a subdivided cell at one lower level (in orange), it needs to become subdivided as well in Phase 2.

7 RESULTS AND DISCUSSIONS

We develop the GPU implementation of our system by CUDA 11.1. We evaluate the system on an Intel Core i7-6700 3.4GHz CPU and an NVIDIA GeForce GTX 2080 Ti GPU. Table 1 summarizes the statistics and the runtime performances of the quasistatic wrinkle simulation examples in our experiment. The mesh sequence inputs to those examples are simulated by a GPU-based cloth simulation engine [Wu et al. 2020] with a fixed 1/100s time step. The sequence selects one frame every three time steps. In other words, the time gap Δt between two frames is 3/100s. By default, the system runs



(a) The front view (b) The back view



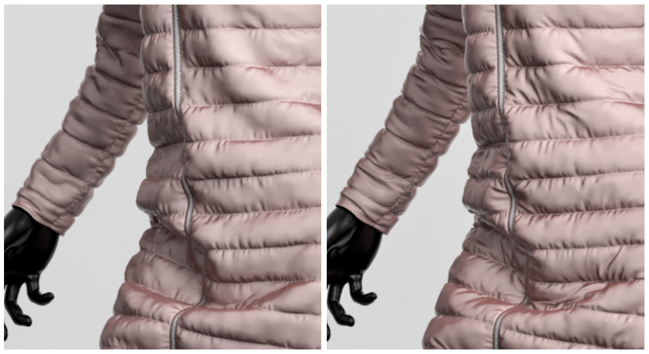
(c) The performance breakdown

Fig. 16. The down coat example and its performance breakdown. While the basic solver contributes the most to the overall cost, collision handling is also expensive and it can be even more expensive in complex collision cases.

256 iterations to solve the optimization problem at each resolution level, so the cost at the finer level is approximately four times the cost at the coarser level. In general, the computational cost depends on three factors: the number of vertices, the collision complexity and the effects to be simulated. For instance, the examples with inflation or stuffing effects are 20 to 30 percent more expensive, thanks to extra memory accesses and arithmetic operations. Table 1 shows that when the number of vertices decreases at the same resolution level, the computational time does not drop as fast as it should be. This is because we did not optimize our system enough for small examples and we plan to address this in the future.

7.1 Breakdown Analysis

Fig. 16 provides a breakdown of the computational cost spent on the simulation of the down coat example. It shows that the basic solver for handling the wrinkle optimization process with elastic potentials contributes the most to the overall cost. The second contributor is collision handling, including both cloth-body collision handling and cloth self collision handling, which together provides about 30 percent of the total cost. This is actually considerably lower than the contribution of collision handling in other GPU-based cloth simulators, which is approximately 45 to 60 percent [Tang et al. 2018; Wu et al. 2020]. The main reason is because the collisions in our examples, especially in the down coat example, are light and infrequent. We expect the collision cost to be much higher in more complex cases and we will study this issue further.



(a) The transferred result (b) The simulated result

Fig. 17. The down coat example handled by deformation transfer and quasistatic simulation. Instead of simulating all of the frames, we can also use our system to simulate a reference frame with a neutral body pose and transfer simulated wrinkles to other frames, such as the one performed by houdini Point Deform shown in (a).

7.2 Deformation Transfer

To reduce the computational cost, we can also use our system to simulate a single frame and then propagate simulated wrinkles to other frames by existing deformation transfer tools. Fig. 17 compares the transferred result with the simulated result in the down coat example. While the transferred result looks visually plausible, it does not contain the wrinkles caused by body motions and it cannot address existing artifacts in coarse mesh inputs. These problems are probably not so troublesome when gathered wrinkles are dense and tight, such as those in Fig. 1. In general, we still suggest to simulate all of the frames as long as computational resources permit.

7.3 A Double-Layered Case

Most of our examples, including the down coat example in Fig. 16 and the quilt skirt example in Fig. 19, are single-layered. But our system is able to handle simple multi-layered cases as well, such as the inflated patch example in Fig. 13d. In this example, we use two meshes facing in opposite directions to represent the two sides of an inflated patch. Thanks to the inflation effect, this example does not experience frequent collisions and we apply the same collision handling technique as before. Having said that, if it does experience frequent collisions, we may have to specifically address the proximity search between the two meshes later.

7.4 Sensitivity to the Input Resolution

Next we would like to evaluate the sensitivity of our system to the resolution of the coarse mesh input. Specifically, we provide two coarse mesh sequences to the mini dress example: a coarser one with 4K vertices and a finer one with 16K vertices. After running a fixed number of iterations in quasistatic simulation, Fig. 18 shows that the result using the input with 4K vertices contains more unnatural wrinkles, due to the locking issue in coarse cloth simulation.

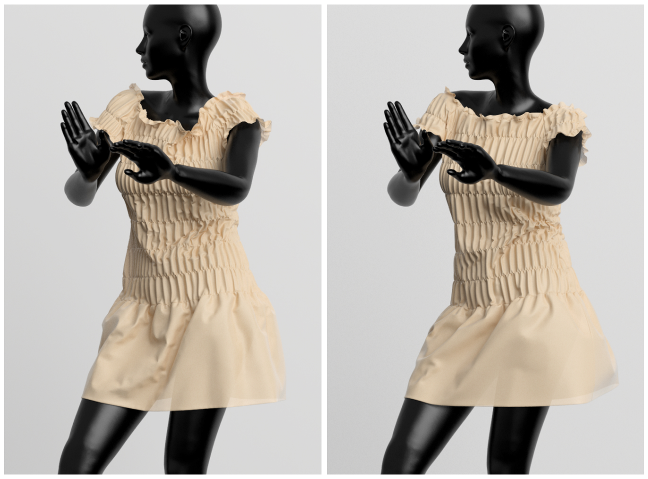

(a) Using an input with 4K vertices (b) Using an input with 16K vertices

Fig. 18. The mini dress example using coarse mesh sequence inputs at two resolutions. Compared with the result using an input with 16K vertices shown in (b), the result using an input with 4K vertices contains multiple unnatural wrinkles, as shown in (a).

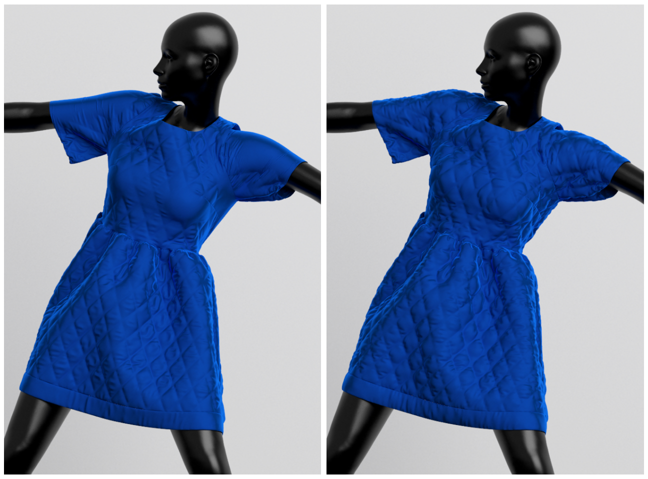

(a) Without applying existing stretching (b) With applying existing stretching

Fig. 19. The quilt dress example without and with applying existing stretching to the desired reference mesh. Existing stretching in the coarse mesh input can cause difficulty in synthesizing light wrinkles. Our system addresses this problem by calculating elastic constants upon existing stretching.

Although we can address this problem by simply running more iterations, we think it is more computationally efficient to increase the resolution of the coarse mesh input instead.

7.5 Existing Stretching in the Input

Existing stretching in the coarse mesh input can also affect our result. While this is typically not a problem, it can cause trouble in the quilt skirt example, when the system tries to add light wrinkles by a procedural deformation tensor field in diamond shapes. In particular, the system is unable to synthesize wrinkles in chest and shoulder regions, where cloth gets stretched too much in the coarse mesh input. To address this issue, we replace Eq. 13 by:

$$\frac{1}{2}(\mathrm{S}_i + \mathrm{S}_j)(\mathbf{r}_j - \mathbf{r}_i)\frac{\|\bar{\mathbf{x}}_j - \bar{\mathbf{x}}_i\|}{\|\mathbf{r}_j - \mathbf{r}_i\|}. \tag{18}$$

Intuitively, Eq. 18 applies existing stretching in the original mesh $\bar{\mathbf{x}}$ to desired reference edges, so that the reference mesh gets expanded even further for wrinkles. Fig. 19b shows that after modifying desired reference edges like this, the system can now enrich chest and shoulder regions with realistic wrinkles.

7.6 Dynamic Simulator

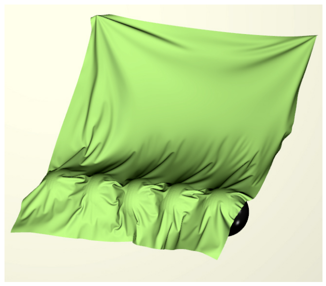
While we formulate our system mainly for offline quasistatic simulation of high-resolution wrinkles, we can also adjust it to function as a standalone dynamic simulator. To do so, we incorporate two more potentials into the overall optimization objective in Eq. 1:

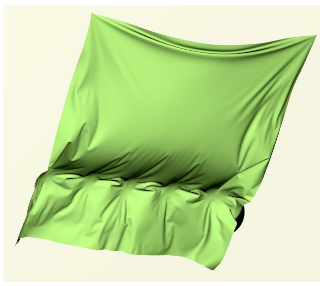
$$\begin{aligned} F^{\mathrm{mome}}(\mathbf{x}) &= \frac{1}{\Delta t^2}\left\|\mathbf{x} - 2\mathbf{x}^{t-\Delta t} + \mathbf{x}^{t-2\Delta t}\right\|^2_{\mathbf{M}}, \\ F^{\mathrm{grav}}(\mathbf{x}) &= (\mathbf{Mg})^{\mathrm{T}}\mathbf{x}, \end{aligned} \tag{19}$$

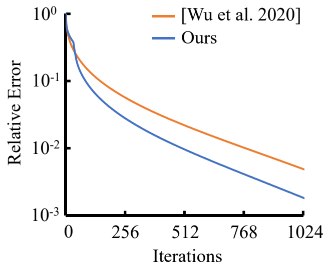
in which Δt is the time step, $\|\mathbf{x}\|^2_{\mathbf{M}} = \mathbf{x}^{\mathrm{T}}\mathbf{M}\mathbf{x}$, $\mathbf{M}$ is the mass matrix, and $\mathbf{g}$ is the stacked gravity acceleration. We are then able to compare the performance of our simulator with that of [Wu et al. 2020], which is one of the fastest GPU-based cloth simulators as far as we know. This comparison uses a square cloth example running at $\Delta t = 1/100$s. To make the comparison fair, we disable multi-resolution features and collision safety protection in [Wu et al. 2020]. Since the optimization problems solved by the two simulators are similar but not identical, we adjust their simulation variables so they converge at similar rates, as shown in Fig. 20c. Fig. 20e demonstrates that our simulator runs 8.43 times faster when the resolution is 512×512. As the resolution increases, the performances of both simulators drop. But our simulator still handles up to 16M vertices, while their simulator reaches memory limit with 800K vertices.

It is difficult to compare our simulator with P-Cloth [Li et al. 2020], another state-of-the-art simulator, since the two are using different simulation models and collision methods. Here we model the square cloth example with 600K vertices and roughly compare our performance with the flag example in [Li et al. 2020]. In our experiment, the simulator needs 0.285 seconds to solve 1,024 iterations at Δt =1/100s, while P-Cloth needs 0.855 seconds to solve fewer than 200 iterations at Δt =1/250s. Given the fact that more iterations are needed as the time step increases, we argue that our simulator is about $2.5 \times 0.855/0.285 \approx 7.50$ times faster. We note that P-Cloth is able to achieve large speed-ups by using multiple GPUs. We have not explored this idea yet, but our system should benefit significantly from multi-GPU acceleration as well.

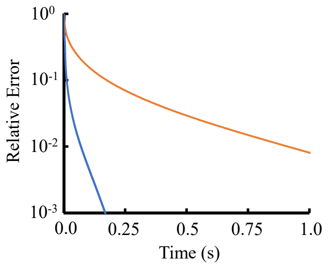
Finally, we would like to mention that our system can also work together with existing cloth simulators in a multi-resolution fashion. This should not be confused with the multi-resolution approach discussed in Subsection 6.1.2 for solving the optimization problem, which relies on the use of a grid-based mesh hierarchy. To use our system with other simulators handling unstructured meshes, we need restriction and prolongation operators between unstructured and structured meshes, which are beyond the scope of this paper.


(a) The result of [Wu et al. 2020]

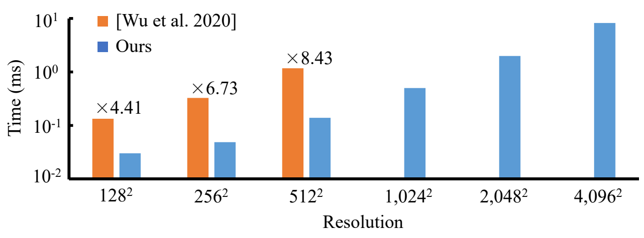

(b) Our result



(c) Convergence rates



(d) Convergence speeds



(e) Per-iteration costs at different resolution levels

Fig. 20. The square cloth example used for evaluating the performance of our system with those of other simulators. When the simulators are solving roughly the same optimization problem with similar convergence rates as shown in (c), our simulator runs 8.43 times faster and it is able to handle meshes with up to 16M vertices, as shown in (e).

7.7 Limitations

When our system is used for quasistatic wrinkle synthesis, its result quality depends on the quality of the coarse mesh input. In particular, the system is unable to resolve existing self intersections, or fix unnatural elastic or frictional behaviors. High-resolution wrinkles synthesized in a quasistatic fashion also contain no secondary dynamic effects, although they are hardly noticeable in reality. These issues do not exist, if our system runs as a standalone dynamic simulator. But this will require the use of smaller time steps and more computational costs, like other GPU-based simulators. For simplicity and efficiency, the implementation of our collision handling technique does not include any collision safety mechanism [Wu et al. 2020]. As a result, the output may contain self intersection artifacts, even though they are rare thanks to small step sizes. Currently, the computational cost of the collision handling technique is acceptable only in single-layered and simple double-layered cases, such as the inflated patch in Fig. 13d. As the collision complexity grows, we expect this cost to increase notably.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we claim that physics-based cloth simulation is now ready to enter the submillimeter era. For modeling and simulation in such a high resolution, it is reasonable to consider the use of regular grid meshes, which enables the exploration of the regular grid structure for GPU parallelization. Our research verifies the feasibility of this practice and demonstrates the effectiveness of our system in synthesizing highly detailed cloth wrinkles.

When comparing our results with real-world wrinkles, we notice that our results look more regular and structured, probably due to an insufficient amount of randomness in our models. We will address this issue first to produce irregular and realistic results. We then would like to experiment other effects that may also be simulated by our system, such as proper frictional contacts [Li et al. 2018; Ly et al. 2020], homogenized yarn-level elastic models [Sperl et al. 2020], fabric thickness, complex sewing lines and boundaries, and permanent creases and pleats. Since safe collision handling can be expensive in complex collision cases, we will develop safer and faster collision algorithms, especially for multi-layered clothing. Finally, we plan to explore the possibility of increasing the resolution even further, for simulating finer wrinkles in closeup views.

ACKNOWLEDGMENTS

The author thanks Liming Liu and Hangjie Jia for rendering. The author also thanks NVIDIA and Adobe for extra funding support.

REFERENCES

David Baraff and Andrew Witkin. 1998. Large Steps in Cloth Simulation. In *Proceedings of SIGGRAPH 98 (Computer Graphics Proceedings, Annual Conference Series)*, Eugene Fiume (Ed.). ACM, 43–54.
David Baraff, Andrew Witkin, and Michael Kass. 2003. Untangling Cloth. *ACM Trans. Graph. (SIGGRAPH)* 22, 3 (July 2003), 862–870.
Miklós Bergou, Saurabh Mathur, Max Wardetzky, and Eitan Grinspun. 2007. TRACKS: Toward Directable Thin Shells. *ACM Trans. Graph. (SIGGRAPH)* 26, 3 (July 2007), 50:1–50:10.
Miklos Bergou, Max Wardetzky, David Harmon, Denis Zorin, and Eitan Grinspun. 2006. A Quadratic Bending Model for Inextensible Surfaces. In *Proceedings of SGP*. 227–230.
Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Trans. Graph. (SIGGRAPH)* 33, 4, Article 154 (July 2014), 11 pages.
Robert Bridson, Ronald Fedkiw, and John Anderson. 2002. Robust Treatment of Collisions, Contact and Friction for Cloth Animation. *ACM Trans. Graph. (SIGGRAPH)* 21, 3 (July 2002), 594–603.
Robert Bridson, Sebastian Marino, and Ronald Fedkiw. 2003. Simulation of Clothing with Folds and Wrinkles. In *Proceedings of SCA*. 28–36.
Peter R. Brune, Matthew G. Knepley, Barry F. Smith, and Xuemin Tu. 2015. Composing Scalable Nonlinear Algebraic Solvers. *SIAM Rev.* 57, 4 (Nov. 2015), 535–565.
Thomas Buffet, Damien Rohmer, Loïc Barthe, Laurence Boissieux, and Marie-Paule Cani. 2019. Implicit Untangling: A Robust Solution for Modeling Layered Clothing. *ACM Trans. Graph. (SIGGRAPH)* 38, 4 (2019).
Nuttapong Chentanez, Miles Macklin, Matthias Müller, Stefan Jeschke, and Tae-Yong Kim. 2020. Cloth and Skin Deformation with a Triangle Mesh Based Convolutional Neural Network. *Comput. Graph. Forum* 39, 8 (Nov. 2020), 123–134.
Kwang-Jin Choi and Hyeong-Seok Ko. 2002. Stable but Responsive Cloth. *ACM Trans. Graph. (SIGGRAPH)* 21, 3 (July 2002), 604–611.
Gabriel Cirio, Jorge Lopez-Moreno, David Miraut, and Miguel A. Otaduy. 2014. Yarn-Level Simulation of Woven Cloth. *ACM Trans. Graph. (SIGGRAPH Asia)* 33, 6, Article 207 (Nov. 2014), 11 pages.
Christopher DeCoro and Natalya Tatarchuk. 2007. Real-Time Mesh Simplification Using the GPU. In *Proceedings of I3D*. 161–166.

Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. 2016. Vivace: A Practical Gauss-Seidel Method for Stable Soft Body Dynamics. *ACM Trans. Graph. (SIGGRAPH Asia)* 35, 6, Article 214 (Nov. 2016), 9 pages.

Rony Goldenthal, David Harmon, Raanan Fattal, Michel Bercovier, and Eitan Grinspun. 2007. Efficient Simulation of Inextensible Cloth. *ACM Trans. Graph. (SIGGRAPH)* 26, 3, Article 49 (July 2007).

Peng Guan, Loretta Reiss, David A. Hirshberg, Alexander Weiss, and Michael J. Black. 2012. DRAPE: DRessing Any PErson. *ACM Trans. Graph. (SIGGRAPH)* 31, 4, Article 35 (July 2012), 10 pages.

David Harmon, Etienne Vouga, Rasmus Tamstorf, and Eitan Grinspun. 2008. Robust Treatment of Simultaneous Collisions. *ACM Trans. Graph. (SIGGRAPH)* 27, 3, Article 23 (August 2008), 4 pages.

Ning Jin, Yilin Zhu, Zhenglin Geng, and Ronald Fedkiw. 2020. A Pixel-Based Framework for Data-Driven Clothing. *Comput. Graph. Forum* 39, 8 (Nov. 2020), 135–144.

Jonathan M. Kaldor, Doug L. James, and Steve Marschner. 2010. Efficient Yarn-Based Cloth with Adaptive Contact Linearization. *ACM Trans. Graph. (SIGGRAPH)* 29, 4, Article 105 (July 2010), 10 pages.

Doyub Kim, Woojong Koh, Rahul Narain, Kayvon Fatahalian, Adrien Treuille, and James F. O'Brien. 2013. Near-Exhaustive Precomputation of Secondary Cloth Effects. *ACM Trans. Graph. (SIGGRAPH)* 32, 4 (July 2013), 7 pages.

Zorah Lähner, Daniel Cremers, and Tony Tung. 2018. Deepwrinkles: Accurate and Realistic Clothing Modeling. In *The European Conference on Computer Vision (ECCV)*. 698–715.

Christian Lauterbach, Qi Mo, and Dinesh Manocha. 2010. gProximity: Hierarchical GPU-Based Operations for Collision and Distance Queries. In *Proceedings of Eurographics*, Vol. 29. 419–428.

Yongjoon Lee, Sung-Eui Yoon, Seungwoo Oh, Duksu Kim, and Sunghee Choi. 2010. Multi-Resolution Cloth Simulation. *Comput. Graph. Forum (Pacific Graphics)* 29, 7 (2010), 2225–2232.

Cheng Li, Min Tang, Ruofeng Tong, Ming Cai, Jieyi Zhao, and Dinesh Manocha. 2020. P-Cloth: Interactive Complex Cloth Simulation on Multi-GPU Systems Using Dynamic Matrix Assembly and Pipelined Implicit Integrators. *ACM Trans. Graph. (SIGGRAPH Asia)* 39, 6, Article 180 (Nov. 2020), 15 pages.

Jie Li, Gilles Daviet, Rahul Narain, Florence Bertails-Descoubes, Matthew Overby, George E. Brown, and Laurence Boissieux. 2018. An Implicit Frictional Contact Solver for Adaptive Cloth Simulation. *ACM Trans. Graph. (SIGGRAPH)* 37, 4, Article 52 (July 2018), 15 pages.

Ling Li and Vasily Volkov. 2005. Cloth Animation with Adaptively Refined Meshes. In *Proceedings of ACSC*. 107–113.

Tiantian Liu, Adam W. Bargteil, James F. O'Brien, and Ladislav Kavan. 2013. Fast Simulation of Mass-Spring Systems. *ACM Trans. Graph. (SIGGRAPH Asia)* 32, 6, Article 214 (Nov. 2013), 7 pages.

Mickaël Ly, Jean Jouve, Laurence Boissieux, and Florence Bertails-Descoubes. 2020. Projective Dynamics with Dry Frictional Contact. *ACM Trans. Graph. (SIGGRAPH)* 39, 4, Article 57 (July 2020), 8 pages.

Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. 2014. Unified Particle Physics for Real-Time Applications. *ACM Trans. Graph. (SIGGRAPH)* 33, 4, Article 153 (July 2014), 12 pages.

Matthias Müller. 2008. Hierarchical Position Based Dynamics. In *Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS)*.

Matthias Müller and Nuttapong Chentanez. 2010. Wrinkle Meshes. In *Proceedings of SCA*. 85–92.

Matthias Müller, Nuttapong Chentanez, Tae Yong Kim, and Miles Macklin. 2014. Strain Based Dynamics. In *Proceedings of SCA*. 149–157.

Rahul Narain, Matthew Overby, and George E. Brown. 2016. ADMM $\supseteq$ Projective Dynamics: Fast Simulation of General Constitutive Models. In *Proceedings of SCA*. 21–28.

Rahul Narain, Armin Samii, and James F. O'Brien. 2012. Adaptive Anisotropic Remeshing for Cloth Simulation. *ACM Trans. Graph. (SIGGRAPH Asia)* 31, 6, Article 152 (Nov. 2012), 10 pages.

Alexandros Neophytou and Adrian Hilton. 2014. A Layered Model of Human Body and Garment Deformation. In *Proceedings of the 2014 2nd International Conference on 3D Vision*, Vol. 1. 171–178.

NVIDIA. 2021. NvCloth. https://gameworksdocs.nvidia.com/NvCloth/1.1/index.html.

SeungWoo Oh, Junyong Noh, and Kwangyun Wohn. 2008. A Physically Faithful Multigrid Method for Fast Cloth Simulation. *Computer Animation and Virtual Worlds* 19, 3 (2008), 479–492.

Pontus Pall, Oskar Nylén, and Marco Fratarcangeli. 2018. Fast Quadrangular Mass-Spring Systems Using Red-Black Ordering. In *Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS)*. 7 pages.

Alexandros Papageorgiou and Nikos Platis. 2015. Triangular Mesh Simplification on the GPU. *Vis. Comput.* 31, 2 (Feb. 2015), 235–244.

Chaitanya Patel, Zhouyingcheng Liao, and Gerard Pons-Moll. 2020. TailorNet: Predicting Clothing in 3D as a Function of Human Pose, Shape and Garment Style. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Gerard Pons-Moll, Sergi Pujades, Sonny Hu, and Michael J. Black. 2017. ClothCap: Seamless 4D Clothing Capture and Retargeting. *ACM Trans. Graph. (SIGGRAPH)* 36, 4, Article 73 (July 2017), 15 pages.

Xavier Provot. 1995. Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior. In *Graphics Interface*. 147–154.

Xavier Provot. 1997. Collision and Self-Collision Handling in Cloth Model Dedicated to Design Garments. In *Computer Animation and Simulation*. 177–189.

Igor Santesteban, Miguel A. Otaduy, and Dan Casas. 2019. Learning-Based Animation of Clothing for Virtual Try-On. *Comput. Graph. Forum (Eurographics)* 38, 2 (May 2019), 355–366.

Nikolas Schmitt, Martin Knuth, Jan Bender, and Arjan Kuijper. 2013. Multilevel Cloth Simulation using GPU Surface Sampling. In *Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS)*.

Andrew Selle, Jonathan Su, Geoffrey Irving, and Ronald Fedkiw. 2009. Robust High-Resolution Cloth Using Parallelism, History-Based Collisions, and Accurate Friction. *IEEE Transactions on Visualization and Computer Graphics* 15, 2 (March 2009), 339–350.

Georg Sperl, Rahul Narain, and Chris Wojtan. 2020. Homogenized Yarn-Level Cloth. *ACM Trans. Graph. (SIGGRAPH)* 39, 4, Article 48 (July 2020), 16 pages.

Jos Stam. 2009. Nucleus: Towards a Unified Dynamics Solver for Computer Graphics. In *11th IEEE International Conference on Computer-Aided Design and Computer Graphics*.

Rasmus Tamstorf, Toby Jones, and Stephen F. McCormick. 2015. Smoothed Aggregation Multigrid for Cloth Simulation. *ACM Trans. Graph. (SIGGRAPH Asia)* 34, 6, Article 245 (Oct. 2015), 13 pages.

Min Tang, Ruofeng Tong, Rahul Narain, Chang Meng, and Dinesh Manocha. 2013. A GPU-based Streaming Algorithm for High-Resolution Cloth Simulation. *Computer Graphics Forum* 32, 7 (2013), 21–30.

Min Tang, Huamin Wang, Le Tang, Ruofeng Tong, and Dinesh Manocha. 2016. CAMA: Contact-Aware Matrix Assembly with Unified Collision Handling for GPU-Based Cloth Simulation. *Comput. Graph. Forum (Eurographics)* 35, 2 (May 2016), 511–521.

Min Tang, tongtong wang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha. 2018. I-Cloth: Incremental Collision Handling for GPU-Based Interactive Cloth Simulation. *ACM Trans. Graph. (SIGGRAPH Asia)* 37, 6, Article 204 (Dec. 2018), 10 pages.

Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. 1987. Elastically deformable models. *Computer Graphics (SIGGRAPH 87)* 21, 4 (Aug. 1987), 205–214.

Bernhard Thomaszewski, Simon Pabst, and Wolfgang Straßer. 2009. Continuum-Based Strain Limiting. *Comput. Graph. Forum (Eurographics)* 28, 2 (April 2009), 569–576.

Raquel Vidaurre, Igor Santesteban, Elena Garces, and Dan Casas. 2020. Fully Convolutional Graph Neural Networks for Parametric Virtual Try-On. *Comput. Graph. Forum* 39, 8 (Dec. 2020), 145–156.

J. Villard and H. Borouchaki. 2005. Adaptive Meshing for Cloth Animation. *Eng. with Comput.* 20, 4 (Aug. 2005), 333–341.

Pascal Volino and Nadia Magnenat-Thalmann. 2006. Resolving Surface Collisions through Intersection Contour Minimization. *ACM Trans. Graph. (SIGGRAPH)* 25 (July 2006), 1154–1159. Issue 3.

Pascal Volino, Nadia Magnenat-Thalmann, and Francois Faure. 2009. A Simple Approach to Nonlinear Tensile Stiffness for Accurate Cloth Simulation. *ACM Trans. Graph.* 28, 4, Article 105 (Sept. 2009), 16 pages.

Huamin Wang. 2015. A Chebyshev Semi-Iterative Approach for Accelerating Projective and Position-Based Dynamics. *ACM Trans. Graph. (SIGGRAPH Asia)* 34, 6, Article 246 (Oct. 2015), 9 pages.

Huamin Wang, Florian Hecht, Ravi Ramamoorthi, and James F. O'Brien. 2010a. Example-Based Wrinkle Synthesis for Clothing Animation. *ACM Trans. Graph. (SIGGRAPH)* 29, 4, Article 107 (July 2010), 8 pages.

Huamin Wang, James F. O'Brien, and Ravi Ramamoorthi. 2010b. Multi-Resolution Isotropic Strain Limiting. *ACM Trans. Graph. (SIGGRAPH Asia)* 29, 6, Article 156 (Dec. 2010), 10 pages.

Huamin Wang and Yin Yang. 2016. Descent Methods for Elastic Body Simulation on the GPU. *ACM Trans. Graph. (SIGGRAPH Asia)* 35, 6, Article 212 (Nov. 2016), 10 pages.

Zhendong Wang, Longhua Wu, Marco Fratarcangeli, Min Tang, and Huamin Wang. 2018. Parallel Multigrid for Nonlinear Cloth Simulation. *Comput. Graph. Forum (Pacific Graphics)* 37, 7 (2018), 131–141.

Olof Widlund and Andrea Toselli. 2004. *Domain Decomposition Methods - Algorithms and Theory.* Springer.

Longhua Wu, Botao Wu, Yin Yang, and Huamin Wang. 2020. A Safe and Fast Repulsion Method for GPU-Based Cloth Self Collisions. *ACM Trans. Graph.* 40, 1, Article 5 (Dec. 2020), 18 pages.

Weiwei Xu, Nobuyuki Umentani, Qianwen Chao, Jie Mao, Xiaogang Jin, and Xin Tong. 2014. Sensitivity-Optimized Rigging for Example-Based Real-Time Clothing Synthesis. *ACM Trans. Graph.* 33, 4, Article 107 (July 2014), 11 pages.

Jinlong Yang, Jean-Sébastien Franco, Franck Hétroy-Wheeler, and Stefanie Wuhrer. 2018. Analyzing Clothing Layer Deformation Statistics of 3D Human Motions. In *The European Conference on Computer Vision (ECCV)*. 245–261.