Augmentation of Coarse Meshes with Wrinkles

by

Russell Gillette

B. Eng, McMaster University, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

August 2015

Abstract

Folds and wrinkles are an important visual cue in the recognition of realistically dressed characters in virtual environments. Wrinkles must, however, move dynamically within the context of an animation to retain much of this realism. Adding wrinkles to real-time cloth visualization proves challenging, as the animations used in games, pre-render visualization, and other such applications, often have no reference shape, an extremely low triangle count, and poor temporal and spatial coherence. I contribute approaches towards the persistence of wrinkles over time, and the creation and rendering of wrinkle geometry in a real-time context, towards a novel real-time method for adding believable, dynamic wrinkles to coarse cloth animations. With this method we trace spatially and temporally coherent wrinkle paths and overcomes the inaccuracies and noise in low-end cloth animation. We employ a two stage stretch tensor estimation process, first detecting regions of consistent surface behaviour, and then using these regions to construct a per-triangle, temporally adaptive reference shape and a stretch tensor based on it. We use this tensor to dynamically generate new wrinkle geometry on coarse cloth meshes through use of the GPU tessellation unit. Our algorithm produces plausible fine wrinkles on real-world data sets at real-time frame rates, and is suitable for the current generation of consoles and PC graphics cards.

Preface

The ideas and algorithms described in this thesis, unless otherwise stated below, were developed by myself in concert with Craig Peters, and in consultation with Dr. Alla Sheffer and Nicholas Vining. They were submitted in the research paper:

- R. Gillette, C. Peters, N. Vining, E. Edwards, and A. Sheffer. Real-time dynamic wrinkling of coarse animated cloth. In *Proc. Symposium on Computer Animation*, SCA '15, 2015 [15]

Chapter 4 describes an algorithm developed by Craig Peters in consultation with Dr. Alla Sheffer. The text from this chapter appears in the above publication and has been included here for the sake of completeness.

My contributions are found in Chapters 6, 7, and 8. Chapter 6 covers the exploration of shape contours of wrinkles, chapter 7 discusses rendering, and Chapter 8 describes the considerations for proximity tests on both the CPU and GPU. The comparison of tessellation and per fragment rendering approaches, as described in Chapter 9, may also be largely attributed to me.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to extend a heart-felt thank you to all the individuals who helped bring this thesis to fruition.

First and foremost, two individuals contributed more to the success of this thesis than I could ever have wished for. Thank you Craig, for being a great partner, for working through this project with me, and for the invaluable knowledge you brought, without which this thesis would never be what it is today. Thank you also Dr. Alla Sheffer, for your immeasurable help and focus. You provided us with experience, knowledge, and helped to navigate the endless pitfalls that would have otherwise been our demise.

I want to thank everyone in the lab: Mikhail, Nicholas, Essex, and I-Chao, for the tireless hours of work you put in to help get our paper published.

Lastly, I would like to thank Darcy, and again Essex, for always being available to bounce ideas off of, and further, for always being available to answer questions or explain concepts. I can't begin to list all the stuff you taught me, thank you so much.

Chapter 1

Introduction

Wrinkles and folds form at seams, bends, and other regions of cloth where material has been pushed or pulled from its original shape. This behaviour provides strong visual cues towards the recognition of shape and motion in garments, and plays a vital role in the representation of believable cloth in virtual environments such as film, animation, digital media or games [10]. Wrinkles can be considered fine details on cloth when compared to general draping movement or the size of the whole mesh. To capture this level of detail in a physical simulation requires a mesh of sufficient resolution to represent the wrinkles, much finer than is required to capture global draping behaviour. While simulation of large scale motion in cloth is often feasible in modern real-time applications, the fine resolution and subsequent computation required to simulate fine wrinkle details proves prohibitive to real-time applications. Offline computation of wrinkles is often not a suitable solution, as real-time applications may smoothly blend between animations to account for user input, or else use other dynamically generated content that therefore procludes the use of precomputed results. We introduce a new method for achieving fine detail wrinkling in real-time on low-resolution cloth animations, targeted towards games and virtual environments. An example is shown in Figure 1.1.

Cloth animation for games is not typically generated by a pure physics simulation; the shape of a garment on a character reacts to player input and is driven by a combination of coarse simulation, inverse kinematics, and animation blending from multiple sources [29]. Rather than trying to capture fine wrinkling in the

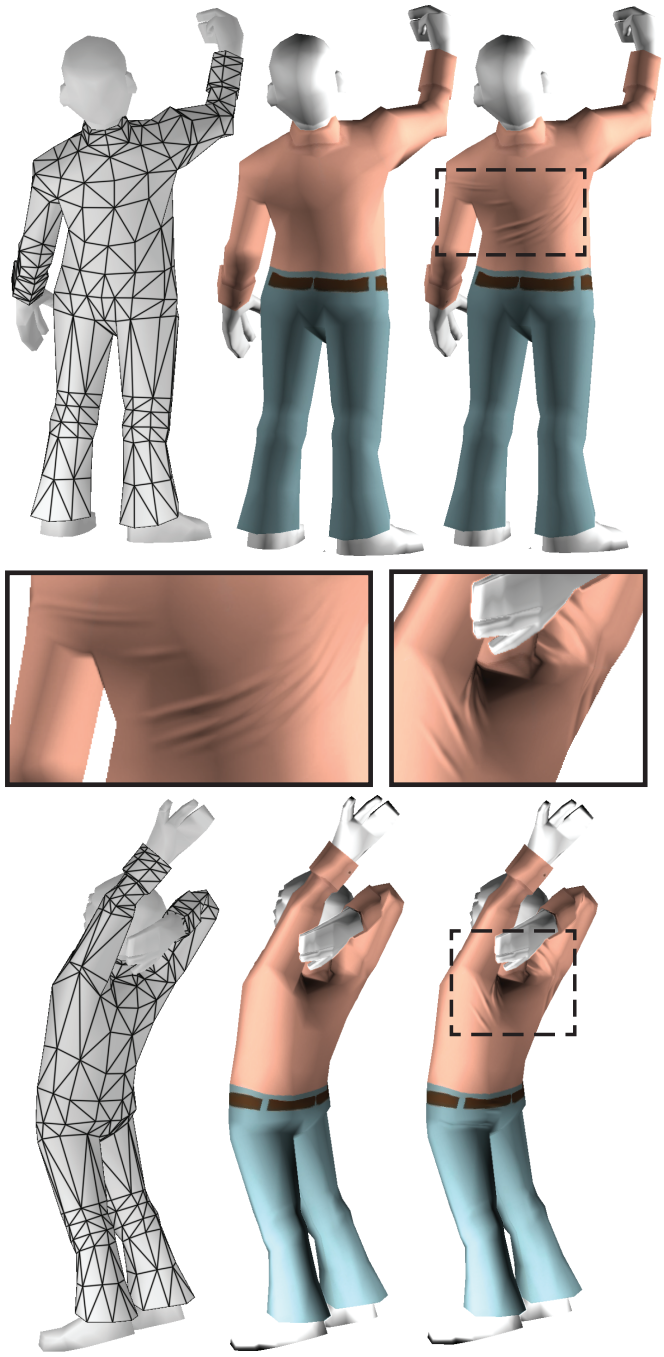

Figure 1.1: Typical example highlighting input mesh coarseness and finished results.

simulation, we are motivated by offline approaches for adding wrinkles and folds to cloth simulation as a post-process [24, 35, 43], as illustrated in Figure 2.1, top. These post-processing methods operate on the key observation that cloth is incompressible [21], and that a mesh without sufficient resolution to show wrinkles will appear to visually stretch and compress within a local, temoral window. This deformation is quantified as compression with respect to a known *reference shape* free of deformation in order to determine the amount of material lost due to lack of resolution. A refined, or user generated mesh of sufficiently high resolution to express the wrinkles is then mapped to the low resolution input so that areas of compression may be deformed to account for lost material by forming wrinkles normal to the mesh and orthogonal to the direction of compression.

Cloth meshes for video games are generated without a meaningful reference shape against which to measure compression [14]. The animations are non-physical, and thus a single meaningful reference shape free of deformation may not exist. We therefore construct an approximate frame of reference, in the form of a local *per-triangle* reference shape, which is accurate enough with respect to recently observed frames of animation to create temporally plausible wrinkles. This reference shape is allowed to change and evolve over time as animation is seen by the viewer, and is progressively updated as new frames are processed in response to user input and intrinsic changes in triangle shape are measured. In each frame, we compute the compression of the current triangle with respect to the per-triangle reference shape, and use this information to seed and modify wrinkles.

Creating wrinkles to resolve the loss of material indicated by our computed level of compression requires a mapping of geometry from the coarse input mesh to a high resolution counterpart. Performing this mapping on the CPU requires the generation or storage, deformation, and transfer to the GPU of high fidelity meshes. Transfer to the GPU for rendering results in a copy of the high resolution mesh existing on both the CPU and GPU and may prove a bottleneck for large meshes. We resolve these situations using an adaptive tesselation scheme on the GPU. The mesh is refined only in areas needing high precision, similar to Rohmer et al. [43], and a high resolution wrinkle mesh exists only on the GPU. In this manner we have removed the duplicate high resolution and spread computation of the refined mesh across the many specialized cores of the GPU.

We directly operate on the piecewise constant tensor data, tracing wrinkle paths across areas of high compression in our per-triangle compression field. We update existing wrinkle paths by solving for spatially and temporally smooth positions that are well aligned with the desired fold directions. We smooth our piecewise paths, as splines, on the GPU to provide sub-triangle precision and reduce data transfered to the GPU. We use programmable GPU tessellation to selectively refine and deform the mesh around wrinkle paths: adaptively generating wrinkle geometry. Normals are computed using a modified distance field and weighted by the height of each contributing wrinkle to allow visually smooth convergence and divergence of wrinkles, and to avoid discontinuities arising from euclidean distance calculations based on piecewise wrinkle paths. This approach enables the generation of cloth wrinkles across low-end animations at sustained framerates of 60 frames per second, making it ideally suited for game environments.

Our key technical contributions are three-fold. First, we introduce a method for generating a temporally adaptive reference shape for typical video game cloth animation sequences, consisting of low-triangle count, hand-animated meshes. Our approach makes no assumptions about cloth developability, does not rely on a parameterization of the underlying mesh, has no specific authoring requirements, and does not rely on training data sets that require updating for every new garment. Our second contribution is a method for dynamically seeding and evolving wrinkle paths on a coarse cloth mesh following a piecewise constant per-triangle compression field. Finally, we show how to generate and render smooth, plausible wrinkle geometry on cloth in real-time, from piecewise input data, with full use of the GPU shading and tesselation capabilites. This thesis focuses on the generation of wrinkle paths, and subsequent mesh refinement and rendering of wrinkles in this pipeline.

We demonstrate our method on a variety of animated garments taken from real-world video game titles. We validate our approach by comparing it to alternative approaches and artist drawn static texture-level folds.

Chapter 2

Background

Since being posed to the graphics community in works by Weil [52] and Terzopoulos et al.[46], the problem of accurately representing and simulating cloth has played a vital role in the modeling of virtual environments for movies, animation, advertisement, and games [9]. Simulation of cloth requires attention to many important concerns, such as the selection of an appropriate physical model and adequate handling of collisions; these concerns are outside the scope of this thesis. Our work falls in the category of "wrinkle augmentation" research, which attempts to refine a coarse cloth simulation with fine details.

The application of graphics hardware to problems in computer graphics has a diverse history. Modern increases in hardware capability have allowed for work and problems that were traditionally handled on the CPU to be offloaded to the GPU. In many cases the GPU is not only an extra computation resource, but also more efficient at particular tasks. Our work makes heavy use of these capabilities in a way not previously explored in research.

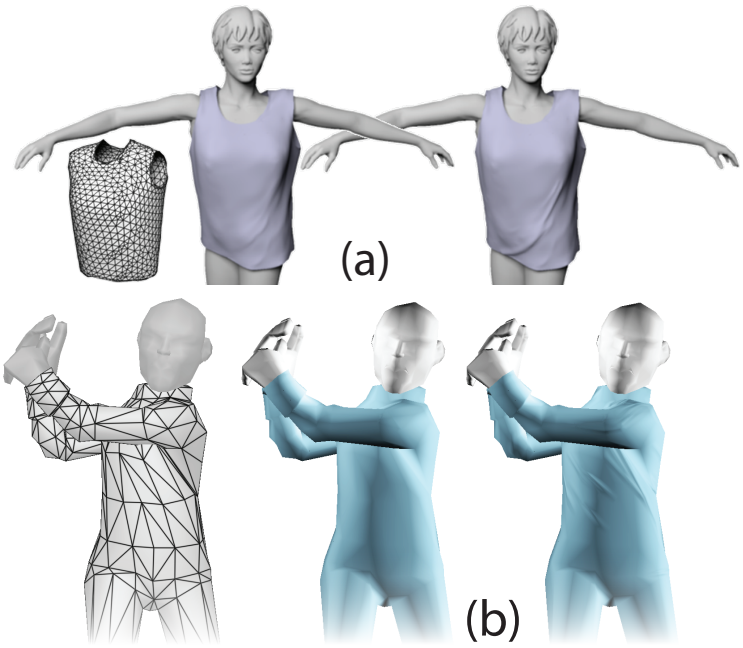

Figure 2.1: (a) Off-line wrinkling of simulated clothing using a user-provided reference shape [43]. (b) Our method adds wrinkles to a coarse cloth mesh typical of real-time applications, where no reference shape exists and where the animation is too coarse to capture fine intrinsic motion.

2.1 Physical Simulation

In mechanics, the bending outwards from a surface caused by compressive forces on a thin material is known as *buckling*. Buckling occurs when internal forces are overcome by applied compressive forces to result in a state of disequilibrium [30]. The wrinkling of fabrics is a buckling behaviour heavily dependant on both the physical properties and geometric structure of the fabric in question[28]. In woven textiles, for example, internal sheer of the weave structure may be more or less prevalent as a result of geometric weave strength and material friction between fibres [21, 28], and may allow for initial deformation inline with the surface prior to buckling. The physical simulation of cloth attempts to accurately approximate these properties and structural behaviour by modelling the internal forces they create [1, 9, 10].

The formulation of an accurate model for cloth simulation is difficult, and many

formulations have been given including: non-rigid body dynamics [8, 46], a non-continuum particle model [6, 11], elastic theory [50], and non-linear shell theory [12]. The difficulty arises in part due to solution stiffness and instability [9, 10], and in part due to cloth's non-linear and often hysteretic (previous-state dependant) behaviour[10, 21]. Cloth strongly resists stretch and compression, while easily succumbing to bending. Typical formulations of this in cloth simulation produce a stiff set of equations due to the strong penalty on stretch and compression that are both unstable and highly non-linear [9]. Baraff and Witkin [1] account for this stiffness, and allow for larger time steps in their solution, through the use of an implicit integration approach. Choi and Ko [9] build upon this work using a non-linear formulation of bending energy, allowing buckling to begin as soon as compression is applied, which overcomes the "post-buckling instability" of previous work.

The problem of buckling has led to novel approaches such as using non-conformal elements to eliminate stretch and compression during simulation [13] and building strain limits directly into a continuum-based deformation model [47]. A more in depth comparison of past approaches may be found in [10, 33].

2.2 Accelerating Simulation with the GPU

Many aspects of the problem of cloth simulation lend themselves well to efficient implementation on the GPU due to the separability and disjoint nature of its component problems. Vasilev et al.[48] use GPU parallelism to compute velocity, normal, and depth maps for an underlying model. Testing for collision with a point on the cloth is done efficiently in image space by referencing into the texture at the screen-space coordinates of the cloth and comparing depth with front- and back-rendered depth buffers. Tang et al.[44] offload the full computation of collision to the GPU, treating the GPU as a series of highly parallelizable data stream processors for the various stages of bounding volume hierarchy tests. Each stream processor is responsible for one of the underlying functional modules of the algorithm such as hierarchy update, bounding volume pairwise tests, and elementary tests. Li et al.[31] formulate the calculation of forces independently for each vertex and perform collision detection with a grid-based spatial subdivision algorithm originally proposed by Zhang et al. [54, 55]. Having posed both calculations as highly

parallelizable problems, they are able to efficiently compute forces, collisions, and resulting vertex positions on the GPU. They use the CPU to compute and correct the over-stretching that results from their choice of a mass-spring model.

Approaches for partial offloading of simulation to the GPU, such as those above, incur readback costs to the CPU, which impact performance. Alternative approaches have been suggested which avoid this cost by pushing the complete simulation to the GPU. Green [17] proposes a method based on Verlet integration [22] to simulate cloth in 4 rendering passes without explicitly storing velocities. The constraint pass used for collision detection however, relies on basic shapes and is unable to handle more complicated cases of intersection arising within or between geometry. Zeller [53] improves on the collision detection of this approach with iterative sets of draw calls to evaluate individual springs on each particle and apply collision constraints. Rodriguez et al. [41] extend the range of acceptable input meshes to other triangulations and quad meshes with the use of a variable number of springs per particle and augment the collision detection using a multiple-camera version of the work by Vassilev et al. [48]. Tang et al. [45] propose a more complete cloth simulation based on their own previous work [44] and work by Li et al. [31]. They augment their collision handling with the approach given by Bridson and Anderson [7], solving the major constraint of available GPU memory with deferred front bounding volumes and a compressed diagonal matrix format. This approach is limited to square or near square cloth meshes, owing to a compressed system matrix used to represent the chosen spring model.

A more comprehensive approach is proposed by Rodreguez and Susin [40], computing a finite element solution with linear shape functions, rather than the traditional polynomial, used for interpolation between nodes. This approach proves more suitable for the GPU, but requires factorization of the rotational part of the deformation gradient to compute the stiffness term for their Lagrangian formulation of forces. They provide an optimal GPU conjugate gradient method for solving this system, and use previous approaches for both self and external collision detection. This approach provides plausible results, but the many concessions made for mathematical stability may introduce error, and the total computational load is quite heavy.

Simulation of cloth on the GPU remains computationally expensive and suffers

from limitations imposed on the data representation and model complexity. Our approach takes advantage of the parallel nature and computational power of the GPU, runs at real-time rates, and does not suffer from restrictions on geometry.

2.3 Data-Driven Approaches

Data-driven approaches wrinkle cloth meshes with the use of information gained from offline physical simulations. Guan et al. [18] use a database of simulated fine garments and mannequins of different shape and pose to generate wrinkles for new shapes and poses. They construct a per triangle solution from linear transformations applied to garment and mannequin parametrizations and solve for a consistent solution using least squares. Kim et al. [26] construct a secondary cloth motion graph that does not explode in size with exploration by collapsing states of similar trajectories that share primary motion graph state. They efficiently explore the space of possibilities by expanding the collapsed nodes with the largest physical error between states. Their novel compression scheme greatly reduce the memory footprint of the secondary graph and enables real-time traversal of over 33 gigabytes of cloth training data.

Zurdo et al. [56] assume downscaled simulations exhibit similar large scale deformation and dynamic behaviour to their full resolution counterparts. They compute a physical simulation on low- as well as high-resolution meshes offline and preserve a small set of ”example” poses that best capture fine scale wrinkling. They compute correction vectors on given animations using an adapted pose-space deformation on edge distances to generate plausible results. Kavan et al. [24] learn regularization terms from input coarse and fine data sets to solve for upsampling operators that minimize low frequency differences between the meshes. They make use of a smart factorization with the offline computation of harmonic bases for input meshes to improve the computation of compute upsampling operators by two orders of magnitude. Wang et al. [51] index wrinkle patterns generated per joint on high resolution simulations of close fitted clothing. Wrinkle meshes for each joint are interpolated from the database and merged together to produce impressive results. This work is suitable only for tight-fitted clothing where collision and friction with the body constrain the range of effect for each joint, and does not

support loose clothing such as dresses or skirts.

Hahn et al. [20] compute full-space cloth simulations offline, saving a set of basis vectors for each cluster of poses generated by a principal component analysis of the results. At run time, they select a relevant subset of the basis vectors of neighboring nodes in pose space based on similarity to the gradient of the full-space equation of motion. From these basis vectors, they can compute the reduced-space hessian, and gradient, with which points in the animation may be updated. This approach gives impressive wrinkling and torsional folds, but is not fast enough for real-time applications; it requires close-fitting clothing rigged to a skeleton, a reference shape, and a set of training simulations.

The data-driven methods are most suited for wrinkling garment animations with similar design and motion to the training data. Adding a new piece of a garment typically requires the construction of new training sets. Our framework does not require training data or a reference shape.

2.4 Real-Time Cloth and Wrinkle Simulation

Real-time cloth simulation for games typically uses a mass and spring system on a coarse mesh [29], connecting vertices with simple springs designed to apply shearing and stretching forces while maintaining garment structure. These mass and spring systems form a series of differential equations that are typically integrated using a stable integration method, e.g. [49]. A detailed example of a cloth simulation in a commercially available video game is discussed by Enqvist [14]. Typical real-time cloth simulation is very coarse and crude (e.g. 600 to 800 vertices [51]), and does not address wrinkling or buckling in any part of the simulation.

Adding dynamic wrinkles in video games typically favours simple strategies. Oat [37] introduces artist-painted wrinkle normal-maps designed to fade-in and out within key areas of the mesh as cloth stretches and compresses. Wrinkle maps are easy to implement, with wrinkle regions delineated by texture masks, but are static and cannot simulate dynamic wrinkle evolution or react to movement not anticipated by the artist. Müller and Chentanez [35] attach a high-resolution "wrinkle mesh" to a coarse cloth simulation and run a static solver in parallel with the motion of the base mesh to animate the fine-grained wrinkles. This approach requires

a reliable reference shape and assumes the coarse cloth motion to be spatially and temporally smooth, which is rarely the case for game-type cloth animations.

2.5 Offline Wrinkle Augmentation

Wrinkle augmentation research attempts to sidestep the myriad of problems associated with high resolution physical simulation by adding wrinkling and fine detail to cloth after an initial simulation is complete. Bergou et al. [2] use constrained Lagrangian mechanics to add physically-based details such as wrinkles to an artist animated thin shell, such as cloth. They define their objective function as a weak formulation of equality (Petrov-Galerkin) between their low resolution input animation and high resolution output. Petrov-Galerkin equality specifies *test functions* which multiply each term and specify the constraints under which equality is satisfied. They design normalized, correlated test functions for the two animations, with buckling dictated by the distribution of test functions along the surface. They implement this distribution as the correlation between test functions of either animation to define a low-distortion mapping between the meshes. Rémillard and Kry [39] extend this work to simulate the detailed deformation of a thin shell surface on a deformable solid. They couple a high resolution thin shell to an attached, embedded mesh. Local positional constraints force the shell to match the interior surface only at low spatial frequency, thus allowing wrinkles without inhibiting large scale deformations.

Rohmer et al. [42, 43] compute a smooth stretch tensor field between animation frames and a given garment reference shape. A refined mesh is mapped back to the original mesh, and wrinkles are modelled along paths traced through the tensor field on contour lines of high compression. They assume the underlying animation to be sufficiently smooth in time and space to result in visually coherent wrinkles. We are inspired by this work in our use of a stretch tensor field on the mesh to measure deformation; however we do not assume the existence of a valid reference frame nor expect the garment motion to be smooth. In contrast to these offline approaches our method is fast enough to provide real-time performance for interactive applications.

2.6 GPU-based Wrinkle Augmentation

A common method of avoiding the difficulties and computational overhead of the above methods is to augment the animation of a coarse cloth mesh with wrinkles after the fact. Hadap et al. [19] compute the change of area for each triangle in an input animation with respect to a given rest pose. They note that cloth does not stretch or compress, and use the patterns within these scalar deformation values to give preference between different artist painted wrinkle textures. Loviscach [32] performs all calculation needed to augment a coarse mesh on the GPU, and thus requires the pre-calculation of adjacency information that is otherwise unavailable. Assuming the first available animation frame to be an undeformed reference shape, it creates a set of local orthogonal basis vectors in both pre- and post- deformation space, and solves for a tensor matrix minimizing their quadric error. This approximation is not resilient against noise, nor does it necessarily produce a smooth tensor field. It constructs wrinkles along this field to have constant width, and models the deformation of the tensors as a sinusoidal height field across the mesh. To produce a smooth field across vertices varying in height and deformation gradient he minimizes the phase errors between adjacent vertices using gradient decent. Loviscach does no mesh refinement or deformation, instead using a modified parallax mapping technique [23] to account for the difference of view perspective. He computes per-fragment normals from the height field and surface normals, resulting in periodic, uniform, aligned wrinkles, often an unrealistic behaviour.

Our approach constructs a smooth tensor field, resilient to noise, and does not assume or require any reference shape. Computing high compression paths through our field allows us to demonstrate wrinkle divergence behaviour that Loviscach does not achieve. Our computation of height values is more efficient, being polynomic rather than sinusoidal, and by tesselating the underlying mesh we create a more realistic silhouette, and do not need parallax computations.

Chapter 3

Overview

This thesis addresses the temporal coherence of wrinkle paths and the real-time generation and rendering of believable wrinkle geometry on top of low-quality animations of coarse meshes. In this setup we cannot rely on the surface animation to capture even the coarse intrinsic surface motion reliably per triangle, and have no expectation that the mesh will exhibit coherent, temporally and spatially smooth deformation with respect to some static reference frame [35, 42] (see Figure 3.3, top). Since we focus on believability rather than correctness, we note that humans largely rely on local movement when predicting wrinkle appearance on surfaces. Surfaces lacking wrinkles appear plausible as long as the intrinsic geometry is largely unchanged, and human observers expect wrinkles to show up when the surface visibly contracts and expect these to dissolve when a surface stretches following contraction. Real-life wrinkles are also persistent - appearing, moving and disappearing gradually. Humans anticipate similar behavior from virtual wrinkles - expecting them to evolve gradually, and to stay in place in the absence of underlying surface motion.

Following these observations, our paper is motivated to analyze the *local* intrinsic surface deformation of the mesh to predict wrinkle appearance. Since the input animation is noisy, our paper denoises this data using spatially and temporally local trends and constructs a per-triangle reference shape from which a stretch tensor feild may then be computed. The per-triangle reference shape is smoothly updated as new frames are rendered, to provide a temporally smooth tensor feild

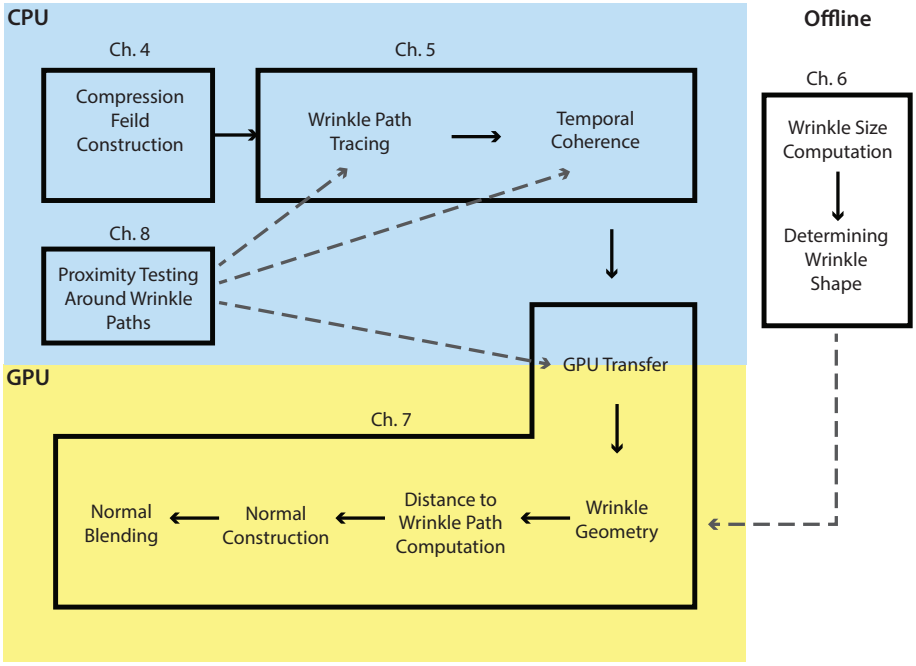

Figure 3.1: Thesis algorithm pipeline and chapter mapping

on which to further operate. The relevant sections of the paper have been included in this thesis, unmodified, as Section 4.

We trace wrinkle paths following the stretch component of the stretch tensor field, orthogonal to the direction of compression, along the surface of the input garment (Section 5, Figure 3.2, d). We generate both more, and deeper wrinkles through regions of higher compression. The computed stretch tensor feild is constant per-triangle, and thus cannot be leveraged to obtain temporally persistent and spatially smooth wrinkles. Due to the coarseness of our meshes, converting it into a piecewise-linear tensor field by averaging adjacent tensors at mesh vertices and then using barycentric interpolation (as suggested by [43], for instance) is undesireable due to loss of information (Figure 3.4, b). This thesis examines alternative approaches towards maintaining temporal persistence of wrinkles in both 3D and constrained to the 2D mesh surface (Section 5.2). Our final solution constrains the points of wrinkle paths to move only along mesh edges, and directly optimizes the

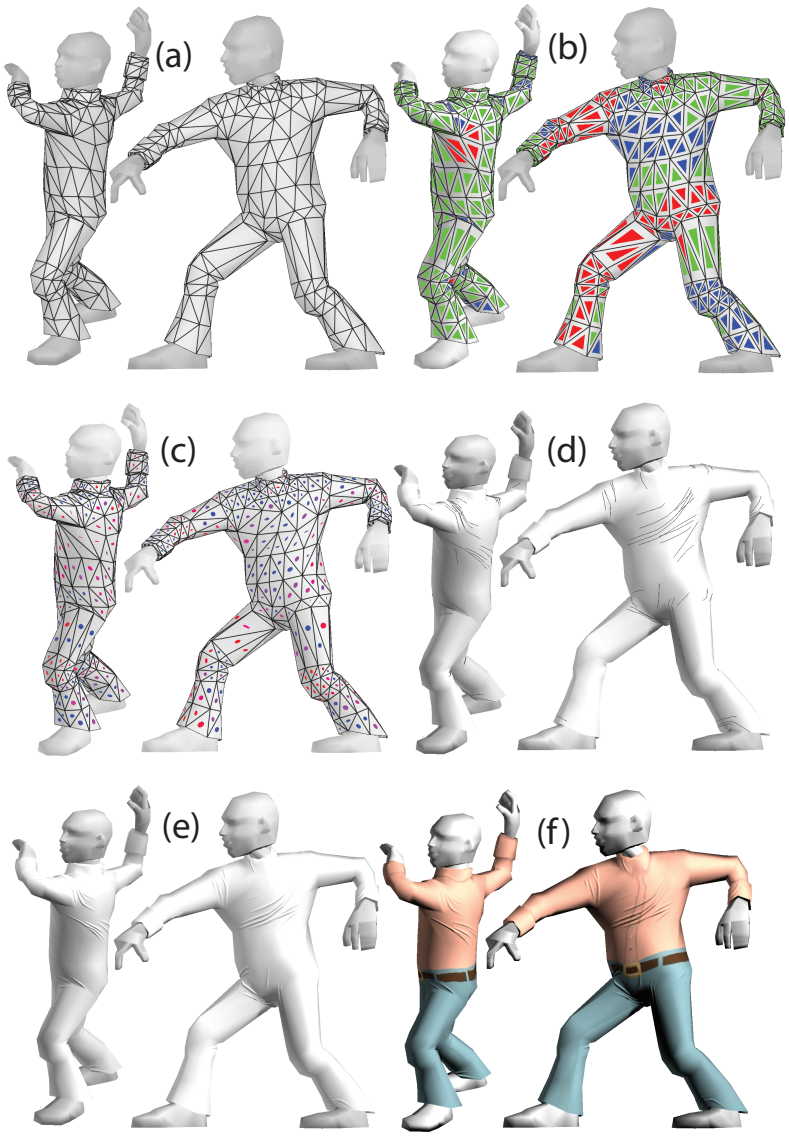

Figure 3.2: Algorithm components: (a) input animation frames; (b) compression(blue)/stretch(red)/neutral(green) labeling; (c) local stretch tensors shown by oriented ellipses; (d) temporally coherent wrinkle paths; (e) final wrinkled cloth rendered at real-time without and (f) with texture.

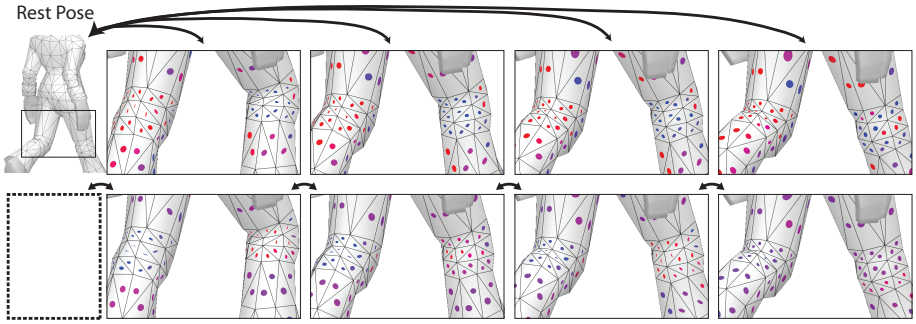

Figure 3.3: Typical game animation frames with intrinsic deformation (stretch) tensor computed with respect to the first (top) and previous (bottom) frame. The shape of the tensor shows compression stretch magnitude ratio and color reflects the larger between the eigenvalues (blue for compression, red for stretch). Using compression on the top tensor as cue for wrinkling, will generate no wrinkles on the left knee (an inverse reference pose will generate no wrinkles on the right). The local tensor (bottom) provides better, but noisy cues.

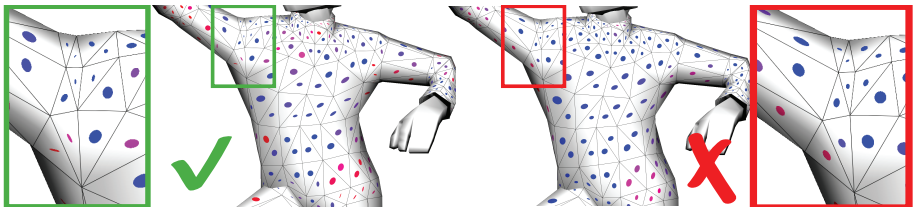

Figure 3.4: On coarse meshes smoothing the piecewise constant tensor field (left) to generate a piecewise linear one (right) leads to loss of details, such as the compression on the shoulder and across the chest (left) which are no longer distinguishable on the right.

shape of the individual wrinkle paths, balancing alignment with compression directions against spatial and temporal smoothness (Figure 5.2). The ouput wrinkle paths are piecewise-linear and both well-aligned with the compression field and persistent over time.

The construction of wrinkle geometry requires knowledge of the wrinkle height, width, and the contour shape that the wrinkle will exhibit along the previously constructed wrinkle path. We determine wrinkle width from the compression magnitude and material properties of the cloth and compute wrinkle height to perserve

the surface area of the input mesh (Section 6). The wrinkle contour is selected to visually blend smoothly with the surrounding mesh. In order to achieve sub-triangle precision for later calculations, we subdivide our piecewise-linear path by sampling from a spline of the original points. Performing this subdivision on the GPU allows us to minimize data transfer across the CPU-GPU bus (Section 6.2).

The real-time nature of our target domain requires careful consideration of the performance implications of data transfer to the GPU, optimizing data packing and minimizing coupling to mesh size (Section 7.2). Creating wrinkles on the GPU requires knowledge of both our generated wrinkle paths and the triangles of the mesh that each of these paths effects. Accurate computation of mappings between wrinkles and triangles is important to minimize both data transfer size and additional work done by the GPU. We consider the trade-offs of accurate computation, with additional CPU workload in Section 8.

Since our input meshes are coarse and not capable of capturing wrinkle detail, we use programmable GPU tessellation to selectively refine and deform the mesh around wrinkle paths and adaptively generate wrinkle geometry (Section 7, Figure 3.2, e). This is compared to existing methods on the CPU which directly modify the animated mesh and require either high resolution input meshes or on-the-fly mesh refinement to adequately capture the wrinkle details [35, 43]. We further improve realism through the computation of per fragment normals. Normals are computed using a modified distance field and weighted by the height of each contributing wrinkle to allow smooth convergence and divergence of wrinkles, and to avoid discontinuities arising from euclidean distances from piecewise wrinkle paths. This combined approach towards realism (Figure 3.2, f) enables the generation of cloth wrinkles in low-end animations at a sustained framerate of 60 frames per second.

Chapter 4

Compression Field Construction

The work presented in this section was developed by Craig Peters in consultation with Alla Sheffer and Nicholas Vining. It has been included without change for the sake of completeness.

4.1 Compression Pattern Extraction

The first step in computing a reliable stretch tensor field suitable for wrinkle tracing is to locate the regions on the surface which undergo noticeable compression or stretch (Figure 4.1). The first indication of such changes occurring is the purely local deformation of an individual triangle within any given frame with respect to the previous frame; we may classify this behaviour as compressing, stretching, or resting. The strongest cue for this classification is given by the stretch tensor of the affine transformation between the two triangles (Section 4.1.1); the indicated amounts of compression and stretch provide a local measurement of surface behavior. However, while real-life cloth deformation is typically both spatially and temporally smooth, meshes for video games are typically coarse and the animation may be noisy, with artifacts such as inter-surface penetration and jitter. On such inputs, these raw measurements are an unreliable indicators of global surface behavior (Figure 4.1, b). Therefore rather than using the stretch and compression magnitudes directly to classify the current state of the mesh triangles, we use this data as input to a more sophisticated labeling process which balances these

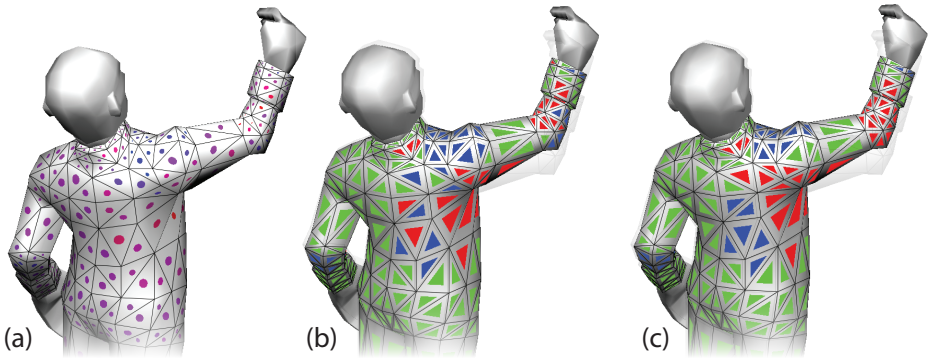

Figure 4.1: Left to right: per-triangle stretch tensor with respect to previous frame (red to blue shows stretch to compression ratio); raw deformation classification (blue - compression, red - stretch, green - rest); spatially and temporally coherent classification.

measurements against a preference for spatially and temporally continuous labels (Figure 4.1, c).

4.1.1 Triangle Stretch Tensor

Given a pair of current and reference triangles we measure the stretch and compression of the transformation between them using the stretch tensor from continuum mechanics [3]. Given a current triangle with edge vectors $(\mathbf{u}_1 = (v_1 - v_0), \mathbf{u}_2 = (v_2 - v_0))$, and reference edge vectors $(\bar{\mathbf{u}}_1, \bar{\mathbf{u}}_2)$, we define the *Deformation Gradient* as,

$$F = [\mathbf{u}_1, \mathbf{u}_2][\bar{\mathbf{u}}_1, \bar{\mathbf{u}}_2]^{-1}. \tag{4.1}$$

The *stretch tensor* is then defined as

$$U = \sqrt{F^T F}. \tag{4.2}$$

The matrix U is symmetric positive definite, has eigenvectors pointing in the directions of maximal stretch and compression, and has eigenvalues λ_{max} and λ_{min} indicating the ratio of current length to rest length for stretch and compression respectively. We define and employ $\tilde{\lambda}_{min} = {}^1/_{\lambda_{min}}$ through the rest of the paper, as we find it more natural to work with. $\tilde{\lambda}_{min}$ is 1 when there is no compression, and

grows as the triangle compresses. Similarly, λ_{max} is 1 when there is no stretch, and grows as the triangle stretches.

4.1.2 Graph Cut Formulation

We formulate the problem of triangle labeling using a graph-cut framework. We solve for a label $l \in (C,S,R)$ per triangle where C indicates compression, S indicates stretch, and R indicates a neutral rest state. We construct a graph $G = (N,E)$ in which each node $i \in N$ is a tuple (f,t) where f is a face and t is a time step. Each node has three spatial neighbors (the three adjacent triangles in the mesh at time t) and two temporal ones (f in frames $t-1$ and $t+1$). Triangles along mesh boundaries as well as those in the first or last frames have fewer adjacencies.

For each node n we compute a unary cost for assigning it a particular label l_n, and for each pair of adjacent nodes we define a label-compatibility cost for each pair of label combinations assigned to them. We then find a labeling that minimizes the following discrete functional:

$$\sum_{i \in N} A(i,l_i) + \sum_{(i,j) \in E} B(l_i,l_j) \tag{4.3}$$

where $A(i,l_i)$ is the cost of assigning the label l_i to node i, E is the set of edges in G, and $B(l_i,l_j)$ is the binary cost of assigning labels l_i and l_j to nodes i and j.

Our unary costs are functions of the stretch tensor magnitudes $\tilde{\lambda}^i_{min}$ and λ^i_{max} over each triangle i (Figure 4.2). The rest label's cost is designed to be low when $\tilde{\lambda}_{min}$ and λ_{max} are both near 1, and grow as they move away from 1 and is set using a symmetric Gaussian function,

$$A(i,R) = 1 - e^{-\|(\tilde{\lambda}^i_{min}-1, \lambda^i_{max}-1)\|_1^2 / 2\sigma^2}. \tag{4.4}$$

The stretching and compressing costs are defined symmetrically using the auxiliary

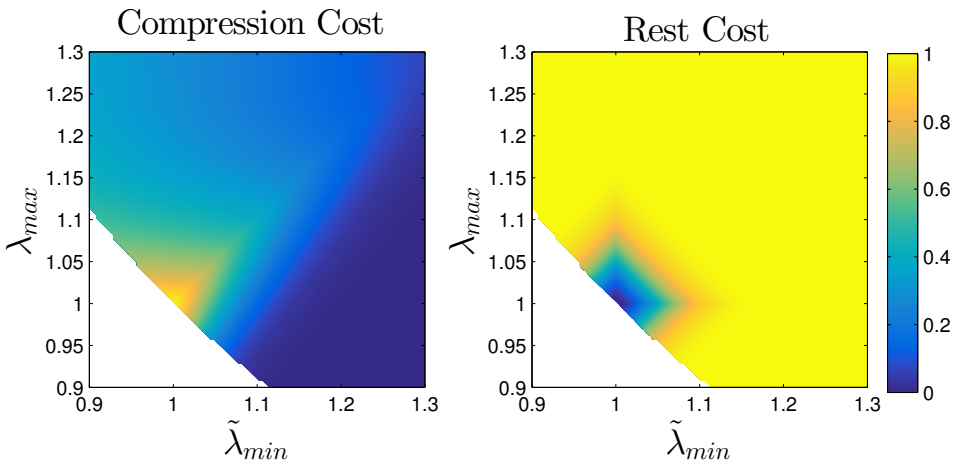

Figure 4.2: The unary cost functions for label assignment depend on $\tilde{\lambda}^i_{min}$ and λ^i_{max}. Left: $A(i,C)$, Right: $A(i,R)$. These eigenvalues measure deformation between two frames, not to the rest frame.

function $c(a,b)$:

$$c(a,b) = \begin{cases} (H-h)e^{-(\frac{1}{2}(a-b)^2)/2\sigma^2} + h, & a > b \\ \frac{b}{a}(H-1)+1, & a < b \end{cases} \tag{4.5}$$

$$H = e^{-\alpha(a-1)}, \qquad h = e^{-\beta a}$$

$$A(i,C) = c(\tilde{\lambda}^i_{min}, \lambda^i_{max}) \tag{4.6}$$

$$A(i,S) = c(\lambda^i_{max}, \tilde{\lambda}^i_{min}) \tag{4.7}$$

We empirically set $\sigma = 0.05$, $\alpha = 9$, $\beta = 90$. The motivation for this design of the cost function is as follows. We want the cost of the compression label to be large when $\tilde{\lambda}^i_{min} \leq 1$, and want it to decrease both when $\tilde{\lambda}^i_{min}$ increases and when it increasingly dominates λ^i_{max} (i.e. when $\tilde{\lambda}^i_{min} - \lambda^i_{max}$ grows). We want a symmetric behavior for the cost of the stretch label.

The goal of the binary term $B(l_i, l_j)$ is to penalize label changes between adjacent faces. It depends only on the labels and is zero when l_i is equal to l_j, and is positive otherwise. As we expect the animation to be gradual, triangles should not immediately transition from compression to stretch without at some point resting;

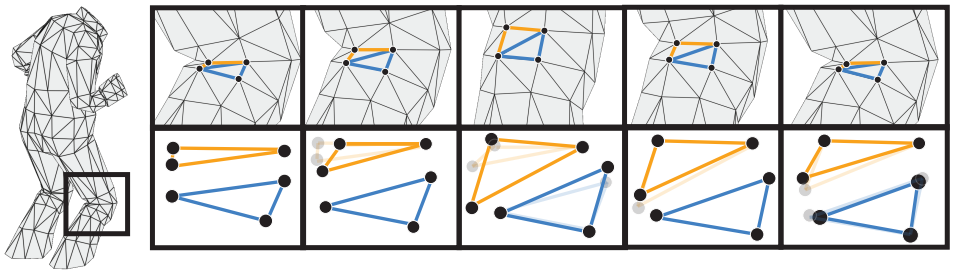

Figure 4.3: Evolution of reference triangles (bottom) throughout the animation process.

we therefore assign a higher cost of 0.4 for assigning *C* and *S* labels to adjacent triangles and a lower cost of 0.2 for assigning them the *R* and either *C* or *S* labels.We use the same costs for both temporal and spatial adjacencies. We solve the labeling problem using the solver of Boykov, Komolgorov, et al.[4, 5, 27], which efficiently minimizes Equation 4.3.

As designed, the framework can be applied to a frame sequence of any length. In a scenario where an entire coarse animation sequence is available in advance, we can therefore label it all at once. In a real-time application where the animation is on the fly, we apply this framework using a one-lookahead window: given each new animation frame we fix the labels for the previously displayed frame and solve the optimization problem for the current frame, while taking the binary cost across the temporal edges linking the current frame to the fixed, previous, one into account. For the first frame in the animation this binary cost is treated as zero. This strategy allows for real-time labeling update and is sufficient to overcome temporal noise in all the animations we tested our method on.

4.2 Local Reference Triangles

To generate a stretch tensor that guides our wrinkle formation we require a local reference shape. Since no such shape is provided a priori we compute one on the fly as the animation progresses. Intuitively, for a perfect incompressible cloth animation, for each individual triangle in the mesh its most stretched, or largest instance in the animation sequence provides the ideal reference shape. However,

in real-life data triangles can and do stretch due to noise and animation inaccuracy. Our on-the-fly rest triangle estimation (Figure 4.3) is designed around these observations.

In the first frame of the animation, we set the reference triangles to be identical to the current one, as we have no other sources of information as to the plausible reference shape of the garment. The reference triangles are then smoothly updated as more information becomes available, with the update strategy reflecting the triangle's intrinsic motion as reflected by our labeling.

If a triangle is labeled as compressed we theoretically could leave its reference triangle unchanged. However, we anticipate some noise in our reference triangle estimation, and in particular want to avoid it reflecting outlier stretched triangles. Thus we choose to dampen the reference triangle size, relaxing it toward the current mesh triangles as these undergo compression. In order for our tensor field to remain smooth and consistent with previous frames, this relaxation must partially preserve the tensor eigenvectors, while smoothly changing the eigenvalues. We therefore directly modify the eigenvalues of the stretch tensor, and then solve for reference triangles that generate the updated stretch tensor, as follows.

Let $F = A\Sigma B^T$ be the singular value decomposition of the deformation gradient (Equation 4.1) of the transformation from the reference to the current triangle. Then, the stretch tensor U can be written as $B\Sigma B^T$, with the eigenvectors encoded by B and the eigenvalues on the diagonal Σ. We compute new eigenvalues, that lie closer to one by setting $\Sigma' = 0.95\Sigma + 0.05I$, here I is the identity matrix. We construct a new Deformation Gradient $F' = A\Sigma' B^T$ and compute the new reference triangles as

$$[\bar{\mathbf{u}}_1', \bar{\mathbf{u}}_2'] = F'^{-1}[\mathbf{u}_1, \mathbf{u}_2] = B\Sigma'^{-1}A^T[\mathbf{u}_1, \mathbf{u}_2].$$

We expect a stretching label to reflect a dissolving compressed, or wrinkled, triangle. In this configuration if both eigenvalues of the stretch tensor from the reference to the current triangle are larger than one they indicate a stretch beyond the current reference triangle. We interpret this configuration as dissolving of previously undetected compression wrinkles. We consequently replace the previous reference triangle with this, new, less compressed one. Since we aim to enrich the rendered garments, we prefer to err on the side of overestimating the reference

triangle size, thus we use no smoothing or averaging in this scenario.

If a triangle is currently labeled as being in a rest state, we leave its reference triangle as is with no adjustment. This choice results in wrinkles that persist unchanged through periods of the animation with little deformation, such as a character holding a fixed pose.

Chapter 5

Wrinkle Path Tracing

The first step in creating wrinkles is to trace their paths on the mesh surface using the stretch tensor computed with respect to the local reference shape. Our computation builds on many of the ideas described by Rohmer et al. [43]; however, in contrast to their formulation that assumes that the tensor field is piecewise linear and smooth, we modify the framework to operate directly on a piecewise constant field. This results in wrinkle paths that are linear across triangle faces and need to be smoothed prior to rendering (as discussed in Section 6.2). The reason for the change is illustrated in Figure 3.4. Since the meshes we operate on are exceedingly coarse, converting the per-triangle tensors into a piecewise linear field (averaging the tensors at the vertices using tensor arithmetic [43] and then using barycentric coordinates to define values across triangles) smooths out critical details.

Following the logic of [43], a naive approach to wrinkle tracing would be to maintain a minimal distance between wrinkle paths while following high compression contour lines in the tensor field. This approach, however, fails to approximate the behaviour of real wrinkles which can merge or move arbitrarily close to each other. We instead hold distance constraints only on the points of high compression at which the wrinkles are seeded, allowing for natural wrinkle behaviour. Since our tensor field is piecewise constant, the converging and merging behaviour of the wrinkles cannot be handled implicitly by the field. We address the visual continuity of merging and overlapping wrinkles in chapter 7, and our approach to determining the proximity between wrinkles in chapter 8.

5.1 Wrinkle Initialization

Wrinkles are traced in areas of high compression and are placed orthogonally to the direction of maximal compression. At each time step in the animation we seed new wrinkles at random locations within triangles of compression ($\tilde{\lambda}^i_{min}$) greater than compression threshold τ. We initialize wrinkle paths from seed points by propagating a polyline along the mesh surface orthogonal to the maximal compression direction within each triangle, given by the tensor eigenvectors. Propagation terminates once the compression magnitude drops below the compression threshold τ, or if the propagated path intersects another wrinkle path. Wrinkle seeding proceeds per triangle from most compressed to least, holding seed points a minimum wrinkle width distance from previously generated paths. The parameter τ is set to reflect the desired fabric stiffness [43]; the expectation is that thinner fabrics, e.g. silk, will be more sensitive to compression and will also exhibit more and longer folds than thicker and stiffer materials such as wool or leather [21]. The computation of wrinkle width is further discussed in Section 6, and further discussion regarding proximity computation is found in Section 8.

A theoretical possibility is that the stretch tensor may have equal, high compression along both directions; this could occur when a surface region contracts simultaneously in all dimensions, in which case the choice of tracing direction is ill-posed. We have not encountered such situations in practice and expect them to be exceedingly rare. We believe that the best solution in this scenario would be to avoid seeding wrinkles in such triangles, and maintaining the previous wrinkle direction if a wrinkle reaches such a triangle during tracing.

While real wrinkles lie on the garment surface and hence follow the local curvature of this surface, they typically have low curvature in the tangential space of the garment. To mimic this behavior and eliminate undesirably sharp changes in wrinkle direction, we apply one iteration of tangential Laplacian smoothing to each path after tracing it, moving each intersection of a path with the mesh edges along this edge toward the shortest geodesic between the adjacent intersections (Figure 5.1).

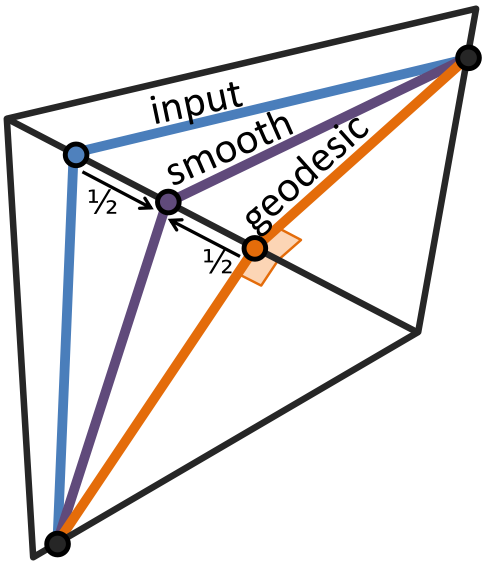

Figure 5.1: A single iteration Laplacian smooth over wrinkle paths to remove sharp changes

5.2 Temporal Persistence

Real-life wrinkles are persistent, changing their shape and position gradually over time. To replicate this behaviour we present three restrictions on generated wrinkle paths across consecutive frames that preventing jarring discontinuity, and allow easy association between the motion of a wrinkle and the actions of the animation:

1. A wrinkle path should not move very far between time steps
2. Any change of relative positioning of a wrinkle's constituent points should be small and in a similar direction to previous changes
3. A wrinkle's direction should be close to orthogonal to the compression of the surrounding region

A simple approach to wrinkle propagation is to reseed wrinkles from scratch each frame. This approach results in flickering due to the changing positions of seed points, failing to meet the second of our stated criteria. Rohmer et al. [43] extend this approach for wrinkle path updating by moving the wrinkle seed points

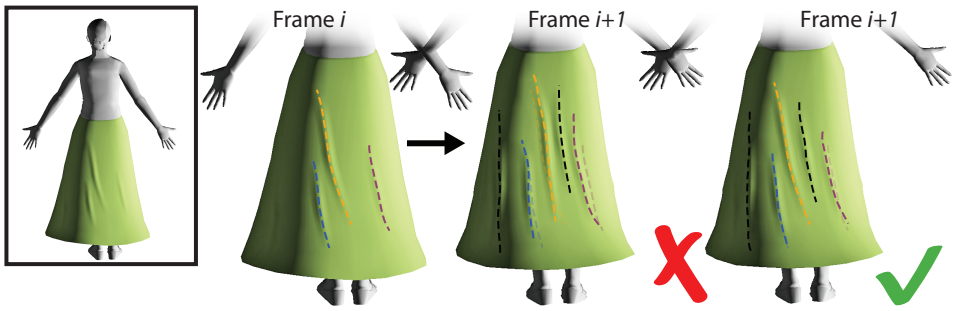

Figure 5.2: Wrinkle paths generated independently per frame vary significantly in both direction and length as highlighted when rendering both current and previous paths (center); our temporally coherent wrinkle paths change gradually across all modalities (right).

based on changes in the stretch tensor field. On a piecewise-constant field or even a diverging piecewise-linear field however, using this approach can drastically move the traced wrinkles following even minor directional changes in the field (Figure 5.3). The solution proposed by Rohmer et al. [43] is to restrict the movement of wrinkle seeds and subsequent propagation to be within a distance of the previous wrinkle. By adjusting existing wrinkles within this *corridor* to match their new eigenvectors, the discontinuity from repropagation within a piecewise field is avoided. For simplicity this approach requires a 2D parametrization, and our problem assumes that one is not given and may not exist. Without this parametrization, the approach proves quite complex; wrinkle point movement needs to be restricted in 3D, and a means for determining the best distribution of points within the corridor to match the tensor field is unclear.

Exploration of possible alternative approaches lead us to two solutions that meet our restrictions as listed above. We document them in the following sections.

5.2.1 Solution 1: Restriction to Edges

We represent each wrinkle path as a polyline whose vertices lie on the edges of the garment mesh. By restricting the movement of wrinkle points to the direction of edges, we simplify the problem of temporal coherence to one dimension and

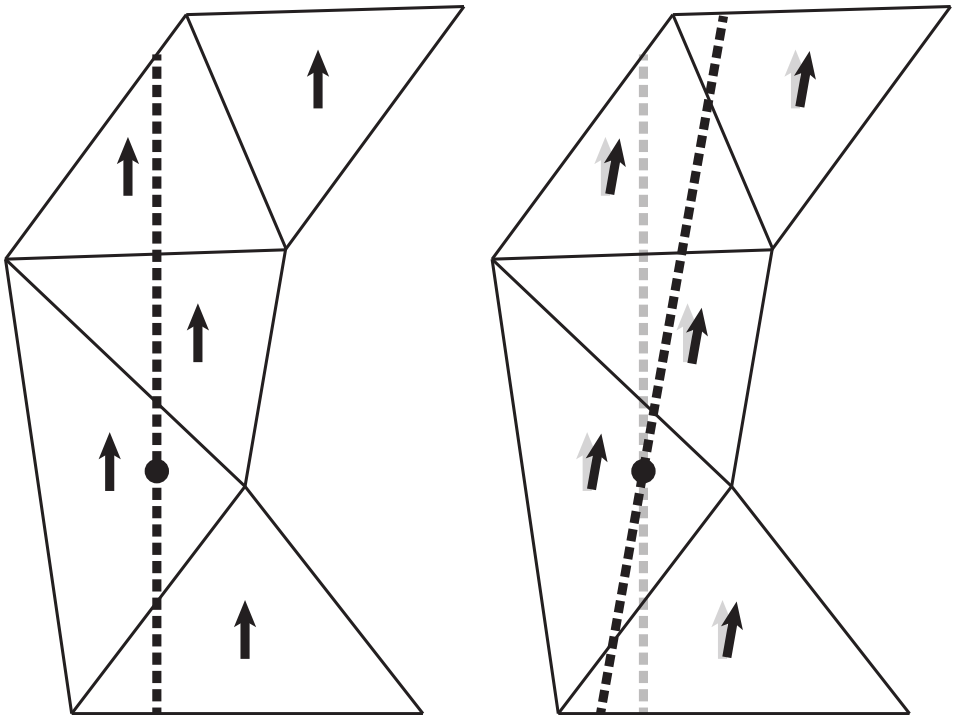

Figure 5.3: Drastic change in wrinkle due to repropagation from seed

restrict wrinkle point movement to the surface of the mesh. Continuous movement from the previous wrinkle location is easily enforced by restricting the distance travelled along the edge (accounting for the angle between the edge and the wrinkle). If the adjustment of a point following the direction of an edge crosses a vertex and thus leaves the mesh, the new wrinkle point will require projection to bring it back to the surface. Following projection, new edge intersections for the geometric neighbourhood of the crossed vertex must be computed, and the wrinkle updated. While this approach can be done in a single update pass of a wrinkle, the geometric update of a wrinkle's constituent points with respect to the eigenvectors of adjacent faces often over-corrects for changes in the compression field to result in unstable positions (jitter).

We frame our problem as an optimization on our three posed criteria at the beginning of the section. We encode our wrinkle paths as linear combinations of

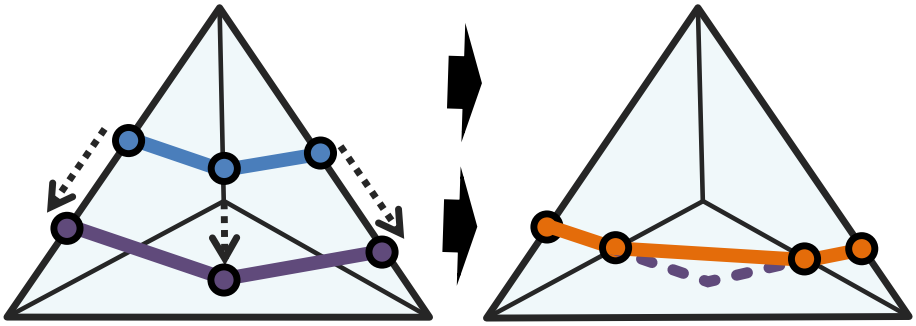

Figure 5.4: Recomputation of wrinkle path when wrinkle crosses a vertex

the edge end vertices:

$$p_i = v_i^1 t_i + v_i^2 (1 - t_i). \tag{5.1}$$

We optimize the shape of the path balancing three terms: orthogonality to compression direction, preservation of relative path vertex positions on the mesh, and wrinkle shape preservation:

$$E = \alpha \sum_{i=1}^{n-1} ((p_{i+1} - p_i) \cdot e_i)^2 + \sum_{i=1}^{n} \|t_i - \bar{t}_i\|^2 + \sum_{i=0}^{n} \|p_i - (p_{i-1} + p_{i+1})/2 - (\bar{p}_i - (\bar{p}_{i-1} + \bar{p}_{i+1})/2)\|^2 \tag{5.2}$$

$$p_{-1} \equiv p_1,\ p_{n+1} \equiv p_{n-1} \text{at endpoints}$$

where $\bar{p}_i$ and $\bar{t}_i$ encode the absolute and relative positions of the wrinkle control point from the previous frame, and e_i is the direction of maximal compression in the triangle shared by p_i and p_{i+1}. The optimization is over just the small number of t_i variables, as we represent p_i as a function of t_i. Since our computation is dominated by persistence we use a relatively small $\alpha = 0.4$. Persistence requires both position and shape preservation: preserving shape alone allows wrinkles to slide uncontrollably, while preserving positions alone leads to artifacts when the underlying mesh triangles undergo significant deformation.

While the solve constrains the vertices to lie on the straight line defined by their currently associated edges, we avoid explicitly constraining t_i to the $[0,1]$ interval

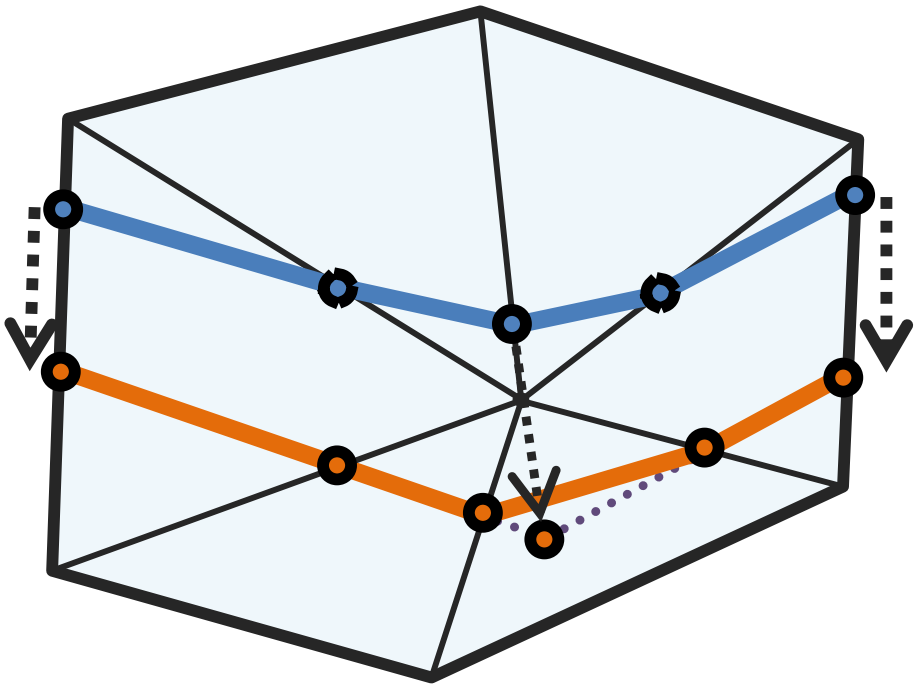

Figure 5.5: Recomputation of wrinkle path across multiple edges when crossing a vertex

as we want to allow wrinkle paths to slide along the mesh. If and when a computed t_i lands outside the $[0,1]$ interval, we locate the best position for the vertex on the mesh as follows, and then update the polyline accordingly. To compute the vertex position we parameterize the umbrella around the relevant edge endpoint (v_i^1 if $t_i < 0$ and v_i^2 if $t_i > 1$) and place p_i using the continuation of the edge projection using Equation 5.1 but constraining it to the umbrella triangles. We then compute the geodesic paths from the previous/next vertices (v_{i-1}, v_{i+1}) to the new location and use the path's intersections with the mesh edges as new wrinkle path vertices (Figure 5.4).

Lastly, if and when more than one wrinkle path vertex lies in the immediate vicinity of a single mesh vertex ($t_i < 0.1$ or $t_i > 0.9$) we use only one of these vertices in the optimization above and then use mesh geodesics to update the new set of intersection (Figure 5.5). Without this modification, the locations of the vertices become over-constrained as a movement in their individually preferred

directions would lead to path self-intersection.

5.2.2 Solution 2: 3D Projection

Restriction of wrinkle point movement to edges may simplify the problem, but it also restricts movement of wrinkles, creating difficulty maintaining alignment with the compression field. An alternative formulation of our restrictions on wrinkle movement with less restriction on direction of motion is given below:

$$E = \alpha \sum_{i=1}^{n-1} ((p_{i+1} - p_i) \cdot e_i)^2 + \sum_{i=1}^{n} \|p_i - \bar{p}_i\|^2 + \sum_{i=0}^{n} \|p_i - (p_{i-1} + p_{i+1})/2 - (\bar{p}_i - (\bar{p}_{i-1} + \bar{p}_{i+1})/2)\|^2 \quad (5.3)$$

$$p_{-1} \equiv p_1,\ p_{n+1} \equiv p_{n-1} \text{at endpoints}$$

This formulation restricts wrinkle point movement by displacement distance in 3-space rather than along an edge. Since the solution points are not guaranteed to lie on the surface, this approach requires a method to project the points back onto the surface and compute new edge intersections.

We notice that by enforcing temporal coherence we are in fact restricting the space where the new wrinkle points can move to a local region on the mesh. We make use of this spatial locality to quickly locate corresponding points of projection for each point given by our linear solve. Beginning at a known point on the wrinkle of the previous frame, the mesh is traversed in a greedy descent towards the first point given by our linear solve, and then between points of the linear solve until reaching the end of the wrinkle. Using euclidean distance, we have reached a vertex when any movement would send us past the closest point on the mesh to that vertex. Once the first point of the linear solve is reached, our new wrinkle path is constructed as the list of all subsequent edge crossings. We note that traversing edges of the mesh towards the points of the linear solve is technically susceptible to getting caught in a local minima (and thus not reaching the point). In practise however, since our wrinkles are restricted in movement and have the same resolution as the mesh, this is extremely unlikely in the time-step of a single frame.

Each step of mesh traversal tests edges of a current triangle against a known orientation towards a goal to determine which edge to cross, and the location of crossing. Selection of an appropriate algorithm proves important under certain traversal conditions where inconsistency in the edge crossing computed from different triangles may prevent reaching solution points given by the linear solve. In each of Figures 5.6 and 5.7, we depict the wrinkle from the previous frame as a path in blue, and the solution points from the linear solve as green spheres.

An initial approach to the computation of edge crossings uses the point of minimal euclidean distance on the line of travel, a line segment between a source and target, with a given edge (Figure 5.6 top). This approach generates inconsistent crossing points between adjacent triangles should the solve point lie directly above an edge (Figure 5.8 (a)). Two subsequent approaches, differing solely on algorithm, find the intersection of a line segment with a plane orthogonal to the incoming face. The first approach uses the plane through the source and target point and the line segment of the two edge vertices, and the second approach the reverse (Figure 5.6 bottom, Figure 5.7 top). In the case of the second approach, the intersection point must be projected along the plane back onto the mesh. These approaches rely on the normal of the incoming face, and thus generate similar inconsistencies to before (Figure 5.8 (b-c)). The solution is to divide space evenly into cells, finding a point of intersection in a similar manner to last two approaches, but using planes constructed as the average of the two adjacent-face normals.(Figure 5.7 bottom)

Consideration of our two approaches lead us to the choice of solution one. A comparison of the complexity in implementation is strongly in favor of the first solution, which except at vertices is a single dimensional problem. Furthermore, by parametrizing the curve as a combination of the mesh vertices, the problem size of the linear solve is greatly reduced to the benefit of performance. While the improvement in adhering to the compression tensor field may allow for smoother transitions in poor geometry, it is likely that in most cases it would be unnoticeable.

5.2.3 Merging and Length Update

As a cloth garment deforms, wrinkles both grow and shrink in length, and migrate

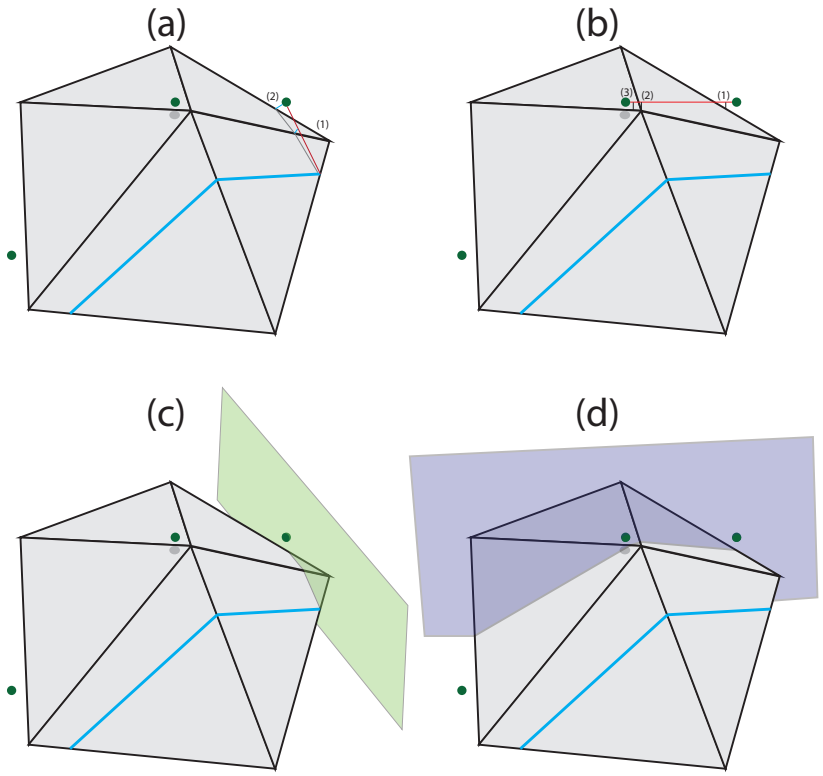

Figure 5.6: Methods of projecting points in 3-space back onto the mesh surface. Left Column: navigating the mesh to find the projection of the new first point of the wrinkle. Right Column: Navigating between points until the end of the wrinkle to compute all subsequent edge collisions. (a–b): finding the closest point between line segments. (c–d): finding intersections with the plane between start and end points.

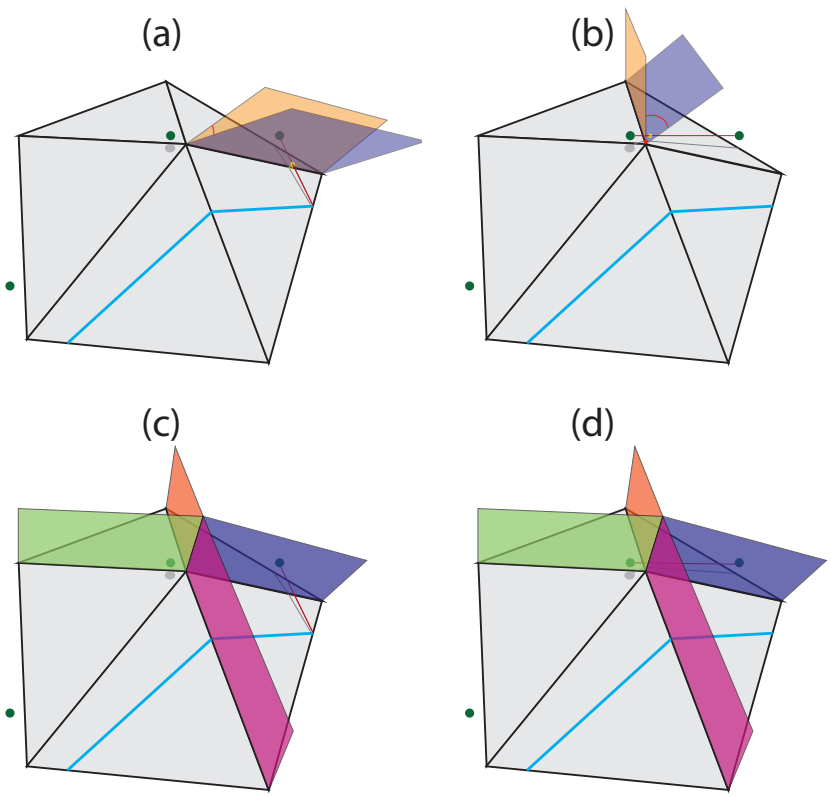

Figure 5.7: Methods of projecting points in 3-space back onto the mesh surface. Left Column: navigating the mesh to find the projection of the new first point of the wrinkle. Right Column: Navigating between points until the end of the wrinkle to compute all subsequent edge collisions. (a–b): finding the edge intersection as a projection along face normals. (c–d): finding the edge intersections as a projection along an average normal plane.

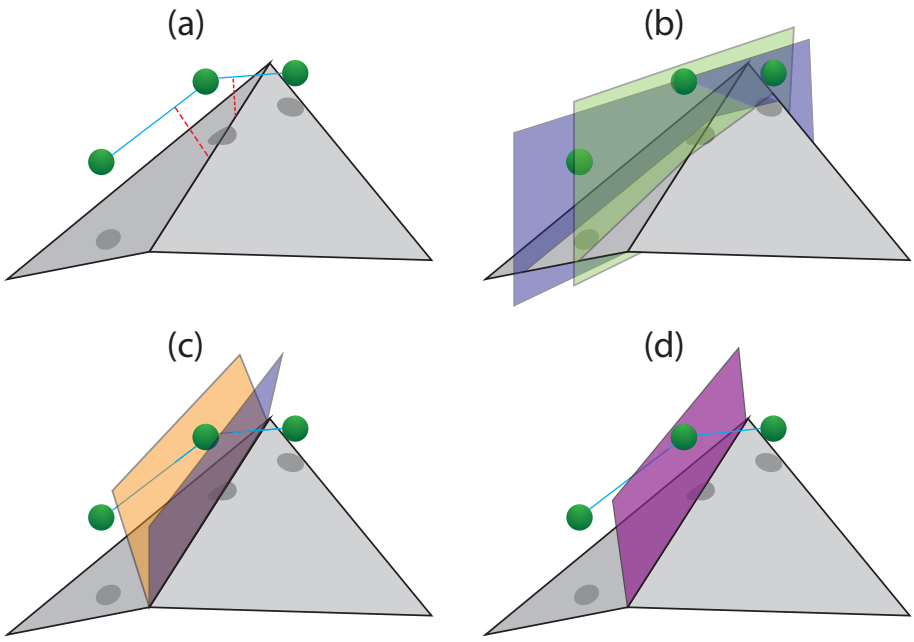

Figure 5.8: Discontinuity in the point chosen due to the method of projection can result in terminating the walk along wrinkle paths early. (a-c) show how such a situation arises, while (d) resolves this problem.

along the mesh. We replicate the change in length of wrinkles, for each frame in the animation, by extending or triming all existing wrinkles based on the magnitude of the underlying stretch field, using the same compression threshold τ as above. To maintain temporal continuity, we do not change the length of a wrinkle by more than 15% per frame. Migration or growth may result in wrinkle paths merging, at which point we truncate one path and allow the other to proceed. Computation of the point at which this occurs is covered in Chapter 8. The truncation of wrinkles both prevents the undesired crossing or passing of wrinkle paths, and reduces computation regarding the collision and proximity of duplicate paths.

Chapter 6

Wrinkle Shape Parameters

To generate actual wrinkle geometry from the traced wrinkle paths, we need to first assign a target wrinkle height and width to each point along the path. We then refine the path, smoothly interpolating heights, and generate appropriate cross-sectional geometry across it.

6.1 Wrinkle Contour

Our goal in the computation of wrinkle height and width is to minimize the change in surface area as compared to our temporally adaptive rest pose. Similar to Rohmer et al. [43], we use a formulation that accounts for both material properties and compression values along the wrinkle. The more flexible a material is, the higher we intuitively expect the fabric wrinkles to be.

Rohmer et al. represent wrinkle shape as a circle contour offset from the surface of the mesh. They define the radius of this circle to be a user defined material-minimum radius, scaled proportional to the compression on the mesh. They determine the offset of the circle from the mesh surface to retain an approximate surface area as compared to the rest pose (Figure 6.1).

We instead numerically pre-compute a set of wrinkle width-height ratios that exactly preserve a unit arc length on a circle. Given a minimum wrinkle width at run-time that accounts for material parameters, we scale and interpolate between these values to compute our desired height. We model our wrinkle shape as a

smooth b-spline contour, using the computed width and height to preserve cloth area. This approach is very fast, requiring a constant-time look up and linear-interpolation at run time. It offers precision based on sample size, and provides a more intuitive material parameter of wrinkle width, which compared to radius is a visible property on the surface of the mesh.

Pre-computation of height for given wrinkle widths must be independent of the user run-time-specified minimal wrinkle width L_{min}. Since we seek to maintain surface area, our arc length $s = L_{max}$, where maximal wrinkle width is computed simply as the arc length of the minimal width, $L_{max} = \pi L_{min}$. To construct a look-up table that is independent of the user specified L_{min}, we can assume our input width L is unit length, and scale all results at run time. We then need only numerically solve for the ratio L/s. Using standard trigonometry relationships we can derive the formula for ϕ as below:

$$\sin\phi - \frac{L\phi}{s} = 0 \tag{6.1}$$

We use Newton's method to pre-compute values of ϕ for given ratio L/s. Since L spans $[L_{min}, L_{max}]$, our sampling domain for this computation is $[\frac{1}{\pi}, 1]$. Since the height along a circular arc is sinusoidal, we space our samples across the input domain as a sinusoidal distribution so that our constructed lookup table has an even distribution of height. This creates a uniform level of precision for all input wrinkle widths.

At run time, we look up the index of our desired height through a binary search of a table of sampling ratios L/L_{max}. We interpolate linearly between the two closest height values given the height look-up index. We then scale the height by the current width to get our final result.

Our framework also differs from Rohmer et al.'s, in that the compression field we operate on is piecewise constant instead of piecewise linear. Using pointwise magnitudes on this input results in discontinuous wrinkle dimensions. Instead we use the maximal compression along a wrinkle path to obtain the maximal wrinkle height and corresponding width (Figure 6.2; here the width is L and the height is h). We then use these values as wrinkle parameters at its mid-point. At the wrinkle end points we set the width to be equal to the maximal wrinkle width L_{max}, and smoothly interpolate the width along the rest of the wrinkle path. We then use

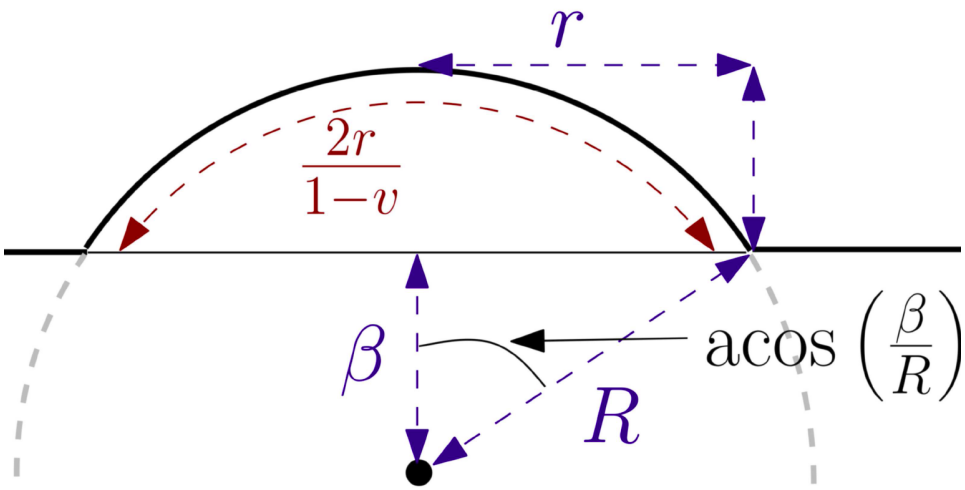

Figure 6.1: Estimation of wrinkle contour used by Rhomer et al.for calculation of height and width values that preserve area

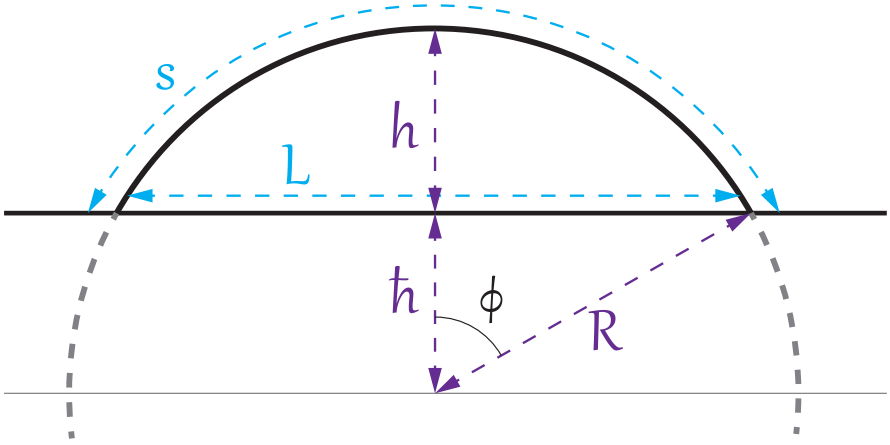

Figure 6.2: We numerically solve for the height h, knowing the maximal wrinkle width s, and current width L. Since s is defined as the maximal width, we note that h = 0 implies, L = s.

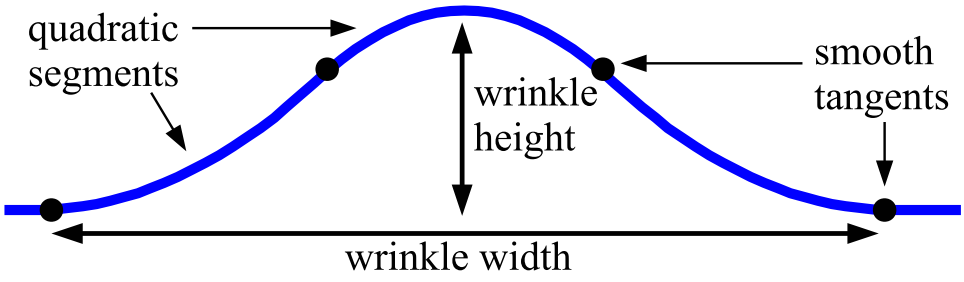

Figure 6.3: Wrinkle contour using three quadriatic b-splines

these width values to compute pointwise wrinkle height. Note that when the width is equal to the maximal width (at the end points) the height is, by construction, zero. This computation leads to smooth naturally dissolving wrinkle shapes along each path.

While the use of circular arcs to represent wrinkles is well suited for wrinkle height and width estimation, real-life wrinkle profiles, or cross-sections, deviate from this shape and smoothly blend with the surrounding surface. We concisely state these requirements as a set of boundary conditions on the first derivative of our cross-section function: $f'(\pm w) = 0 \wedge f'(0) = 0$. We offer for consideration four function families that satisfy these conditions: quartic polynomial, piecewise circular arcs, sinudsoidal, and piecewise b-spline (Figure 6.4). We empirically examine the rate of fall-off, simplicity of representation, and ease of calculation in our selection of a quadratic B-spline for our wrinkle contour.(Figure 6.3).

Quartic polynomials and sinusoidal waves are insufficient for our needs due to their high curvature at the boundary. Figure 6.5 a) and b) show that the high curvature results in the perception of a discrete transition due to the Mach band effect. Approaches to mitigate this effect, such as the addition of Perlin noise are only partially successful in removing this artifact, are very dependent on wrinkle scale in their implementation, and add unwanted computational overhead to the GPU. Piecewise circular arcs and b-splines do not suffer from this effect (Figure 6.5 b and d). The b-spline is the conceptually simpler, and more efficient of the two methods.

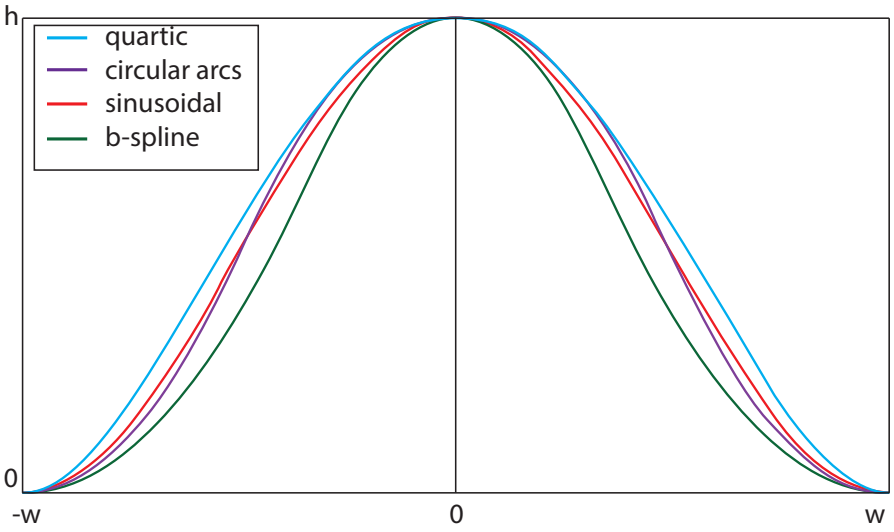

Figure 6.4: Cross-section contour possibilities.

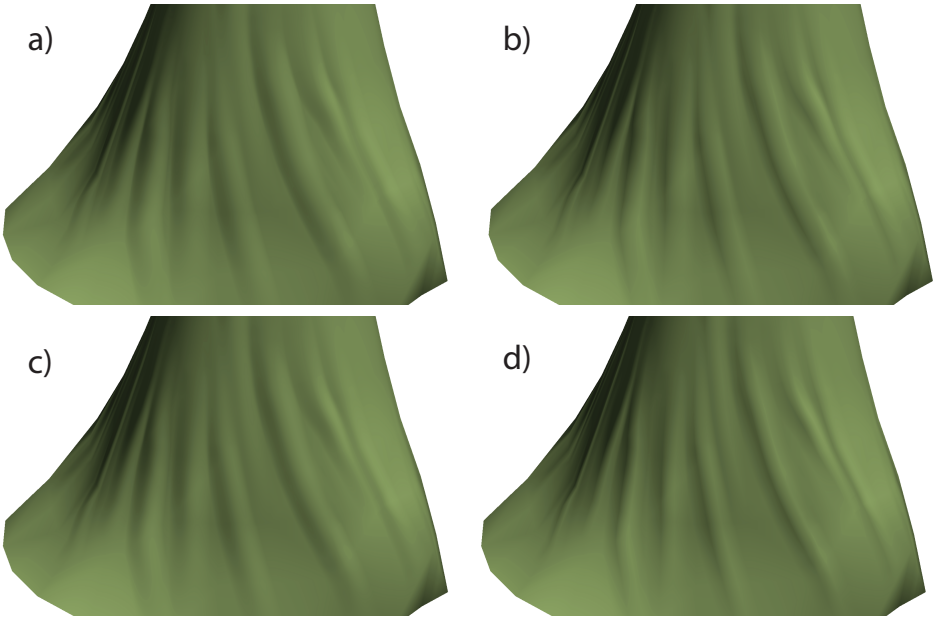

Figure 6.5: Example of each cross-section contour in use. a) quartic b) circular arcs c) sinusoidal d) b-spline

6.2 Path Smoothing

We smooth out the path of each wrinkle, replacing the linear segments within each mesh triangle with Bezier paths whose tangents across triangle edges are set to the average of the line segment tangents in the adjacent triangles. For efficiency this computation is done on the GPU in parallel per triangle, and each Bezier segment is discretized using a polyline.

Chapter 7

Fast GPU-Based Wrinkle Modeling

Embedding fine wrinkles in the actual garment mesh requires a very high mesh resolution. Storing and rendering such a mesh would significantly slow the animation display and gameplay. We avoid modifying the underlying surface mesh or animating a finer mesh in parallel by modeling wrinkles, at render-time, directly on the GPU; specifically, we use the OpenGL tessellation shader to create coarse wrinkle geometry in real time and use the fragment shader to compute per-pixel wrinkle normal maps to increase rendered wrinkle believability. The combined method creates realistic looking wrinkles and is very efficient, allowing on-the-fly wrinkle modeling at 60 frames per second, when the rendered characters occupy

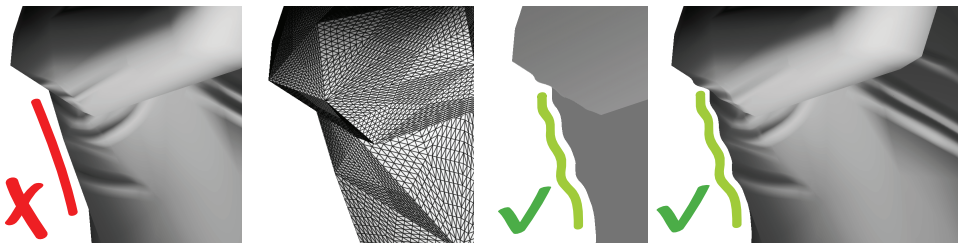

Figure 7.1: Left to right: wrinkles rendered using only normal maps; tessellated wrinkle region; wrinkles rendered using displacement (tessellation); wrinkles rendered using displacement and normal maps.

the majority of the screen on a standard computer monitor. While using normal maps alone is clearly even faster, the results do not appear as realistic as the rendered frames lack inter-wrinkle occlusions and characteristic changes in character contours (Figure 7.1). Our approach, which leaves the original mesh unchanged, allows for wrinkle rendering to be enabled or disabled as the character moves further away from the viewer.

7.1 GPU Rendering Pipeline

This section is a review of the standard GPU pipeline. Readers already familiar with this content may skip ahead to Section 7.2. More information may be found in OpenGL or DirectX online resources [25], [34].

The GPU processes data in a highly parallel pipeline designed to efficiently process the large amount of geometry that may be present in a scene. Data is transferred from the CPU to the GPU along a data bus of limited bandwidth, and stored in either buffers or textures allocated from global or texture/constant memory. On each render pass, the GPU reads data from user specified buffers, passing the data through a series of built in operations and compiled shader programs, the shader pipeline, before outputting the data. Each stage of the shader pipeline operates on the output of the previous stage, and is designed towards a specific task (Figure 7.2).

The vertex shader (VS) is run independently on each set of vertex information passed to the GPU and returns a single corresponding set of output data. The tessellation control shader (TCS) then specifies subdivision parameters on the geometry, defining for a given input patch the level of subdivision along the outer boundary, and the number of inner subdivision levels to add. The fixed-function tessellator generates new vertices for refined primitives from the input geometry and the TCS parameters. Each newly generated vertex is then placed by the tessellation evaluation shader (TES), which has access to the vertex information from which it was generated. The geometry shader (GS) operates on the subsequently generated primitives with access to the constituent vertex information. These primitives are then passed through a number of fixed-functionality (immutable) operations to generate the fragments on which the Fragment Shader (FS) will operate (Fig-

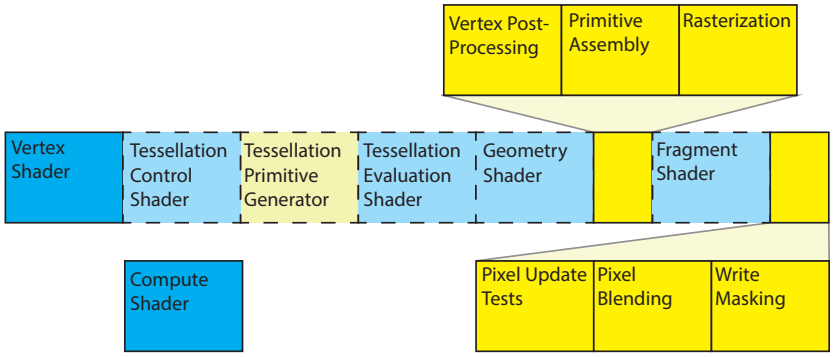

Figure 7.2: Summary of the rendering pipeline (fixed functionality in yellow)

ure 7.2).

Of the fixed function operations, vertex post-processing consists of transform-feedback and clipping. The transform feedback is unused in our framework, but allows for geometry generated up to this point to be saved for a future render pass. The clipping stage trims primitives to the viewing volume, before applying a transformation to account for perspective. The Primitive Assembly stage samples the scene to generate 2D primitives from the geometry, and may discard faces based on orientation. The generated primitives are rasterized to a series of fragments with values assigned as either one of, a linear interpolation between, or a perspective correct interpolation between seeding vertices. The generated fragments are operated on by the FS before undergoing a number of optional tests to decide if those pixels should be updated. These tests check against window focus and visibility (ownership test), fragment screen location (scissor test), fragment depth within the scene (depth test), and comparison of user provided values to the results of a user provided test (stencil tests). Fragments are finally combined in some combination with the existing render target values through blending. Specific write locations may be explicitly disabled by the user through a final 'masking' stage.

All stages of the rendering pipeline except the VS may be omitted, in which case they simply pass the data to the next stage. The rendering pipeline may be terminated prior to primitive assembly to perform only a transform feedback operation and the omission of a FS will still update the depth and stencil buffers.

An alternative render path is through just a Compute Shader (CS) which uses the GPU as a highly parallel computing path. The compute path allows the specification of both the number of compute groups, specified at invocation, and the local size of each group, specified within the shader. All compute invocations of the same group may access and synchronize on shared variables and memory, while separate groups are disjoint. Compute output may be stored to a local buffer and mapped back to the CPU, or else rendered to a texture from which data can be read.

7.2 GPU Performance Considerations

Bottlenecks in GPU performance often arise due to pipeline stalls as a result of resource transfer overhead, resource contention in shared memory, or synchronization of shader invocations. Management of these problems requires resource management and load balancing, and is of particular importance to real-time rendering, as a performance critical program.

One common location for pipeline stalls is on the initial transfer of data to the GPU, making it important to minimize the amount of data transferred at any one time [36]. We refine the wrinkle path for sub triangle precision using a Compute Shader on the GPU so that only coarsely refined wrinkle path data needs to be transferred along the bus. Furthermore, we pack the data intelligently, using three dynamically-sized buffers to removing the need for fixed allocation sizes, and minimizing the coupling between data transfer size and mesh size (see Figure 7.3). With this scheme, the first buffer stores two values for each triangle: the number of wrinkles that affect it, and an offset into the subsequent array. Beginning at the offset into the second buffer, for each triangle with affecting wrinkles, is a set of start and end indices for each affecting wrinkle in the final buffer which stores wrinkle information. In this scheme, wrinkle data is passed only once and offsets into this data are required only for affected triangles.

GPU shaders are run synchronously on multiple shader cores, and may be stalled by conditional statements. In the case that both branches of a conditional statement are hit by synchronized shaders, both branches will be evaluated, and the correct result chosen for each instance [38] [36].

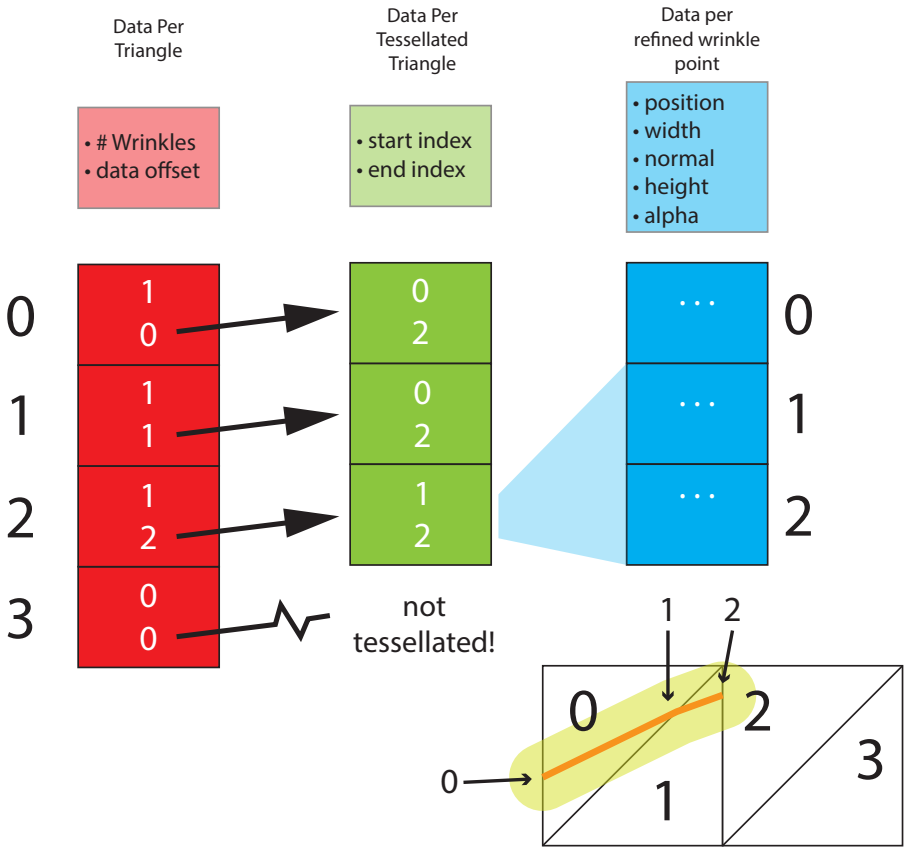

Figure 7.3: Wrinkle data is packed in three dynamically sized buffers to minimize data associated with each triangle (uncouple from problem size) and to minimize total data transferred over the CPU-GPU bus. Untessellated triangles require 8 bytes each, while tesselated ones require 16. This may feasibly be reduce to 4 and 8 using half-byte types.

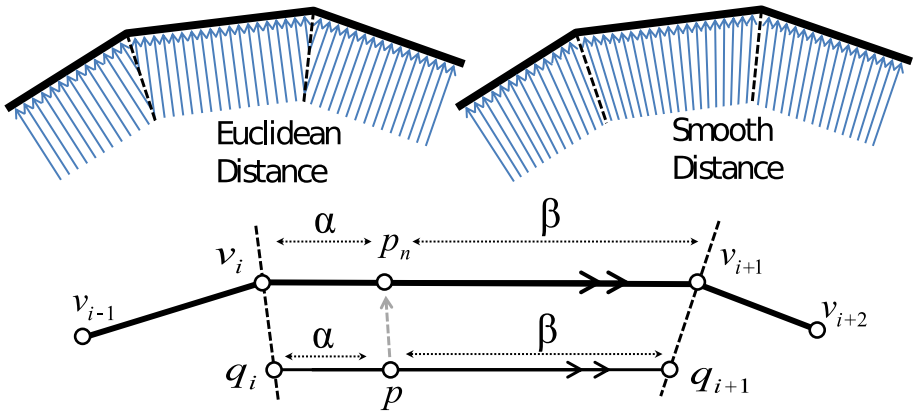

Figure 7.4: Projection of a point p to the winkle segment $v_i v_{i+1}$.

Global memory access suffers from a high latency and low bandwidth [36], proving to be one of the serious efficiency concerns of our program. The cost of access is difficult to mitigate due to the need for proximity checks on the GPU in computation of normals, and is one of the key areas in which our algorithm may be improved.

7.3 Wrinkle Geometry

We make use of the tessellation shader to subdivide each mesh triangle that falls within a proximity region of the wrinkle path (Figure 7.1, b, Section 8). We upload the list of wrinkles affecting each triangle for a given frame, and tessellate only those triangles with non-empty wrinkle lists. We use a uniform tessellation scheme, deciding on Tesselation Control Shader parameters to create a target edge length that is $\frac{3}{8}$ of L_{min}. This target edge length is derived from the observation that four triangles is a minimal requirement to represent a reasonable wrinkle contour. The rate that wrinkle height falls off over the length of a wrinkle allows for minor deviation towards coarser tessellation to improve efficiency. Since wrinkles merge smoothly with the mesh at their boundaries and fall off completely within tesselated regions, there is no concern over the introduction of T-junctions on the boundary of unrefined and tessellated regions. We offset each generated vertex in the direction of the triangle normal according to the cross-section height at that

point. To compute the height at a point p we project that point on to the wrinkle path, evaluate the wrinkle height and width at that point, then use the B-spline shape function to determine the offset at p.

Projecting to the wrinkle path using the Euclidean closest-point is a discontinuous operation in the vicinity of the medial axis of the path (Figure 7.4 top), leading to discontinuities in the offsets. We avoid such discontinuities by using a modified projection (Figure 7.4). To find the projection of a point p onto a wrinkle path, we compute an approximate geodesic Voronoi diagram of the path edges. Within each Voronoi cell, p is projected parallel to the wrinkle edge $v_i v_{i+1}$ onto the Voronoi facets to find points q_i and q_{i+1}. We find the barycentric coordinates α and β of p with respect to these projected points, and then construct the final projection p_n on to the wrinkle path using the same barycentric coordinates along the wrinkle edge.

To compute the projection efficiently on the GPU, we reformulate this problem as a single projection from p to p^n. We note that the direction of projection is a linear combination of the two Voronoi edges given by the intersection of our Voronoi facets with the mesh. We thus solve for the barycentric coordinate α along the edge directly, as the coefficient for which our interpolated Voronoi edge direction matches the vector from point p_n along the edge to our given point p. We need not formulate this as a minimization on the angle between the two vectors, as the Voronoi interpolation spans the full space of vector [p, p^n], and thus and exact solution will exist.

If we let s_0 and s_1 correspond to the average direction of the wrinkle (orthogonal to the Voronoi edge and easier to compute), then we formulate the problem as below:

$$0 = (v_i + \alpha * (v_{i+1} - v_i) - p) \cdot (\alpha * s_0 + (1 - \alpha) * s_1) \tag{7.1}$$

We can then simplify the problem by substituting $z_0 = v_{i+1} - v_i$ and $z_1 = v_i - p$, and solve for α.

$$0 = (z_1 + \alpha * z_0) \cdot (\alpha * s_0 + (1 - \alpha) * s_1) \tag{7.2}$$

$$0 = (z_0 \cdot (s_0 - s_1)) * \alpha^2 + (z_0 \cdot s_1 + z_1 \cdot (s_0 - s_1)) * \alpha + z_1 \cdot s_1 \tag{7.3}$$

7.4 Wrinkle Normals

In the fragment shader, we use the method shown above to obtain the smooth-distance projected points and corresponding widths on adjacent wrinkle paths. While this is a duplication of much of the work from the Tesselation Evaluation Shader, it is necessary to achieve per fragment precision in terms of wrinkle proximity. Making use of the interpolation across primitives during rasterization would avoid duplication, but is restricted to linear interpolation, and thus loses accuracy on our higher order deviation of normals. Interpolating distance rather than normals gives a good result on a single wrinkle, but results in inconsistent and innacurate results when affected by multiple wrinkles. Loss of accuracy in these cases results in visual artifacts that are visually unappealing.

Figure 7.5 elaborates on the interpolation across primitives using a signed distance. We see that between two wrinkle paths distance for either primitive will deviate linearly. Interpolation across a primitive may result in a good approximation to the real answer (top) or a completely incorrect result (bottom), depending on the orientation of the wrinkles, and thus the sign of their overlapping distances. Additional information may be passed to resolve this discrepancy (a non-signed distance), however the linear interpolation still fails to provide sufficient precision for visual fidelity. Furthermore, since vertex locations differ between primitives, interpolation boundaries will arise between primitives. Higher order interpolation would solve these problems, but is not yet available in consumer hardware. A visual comparison between the approaches may be seen in the results (Section 9).

Once an accurate distance to the wrinkle path is computed, we use the B-spline shape contour formula to obtain an analytic normal. We shade the wrinkles using the Phong shading model, evaluating the lighting equation directly, per-pixel for the computed normals. Other shading methods, such as Minneart shading and physical based rendering models, could be computed on the same normal data.

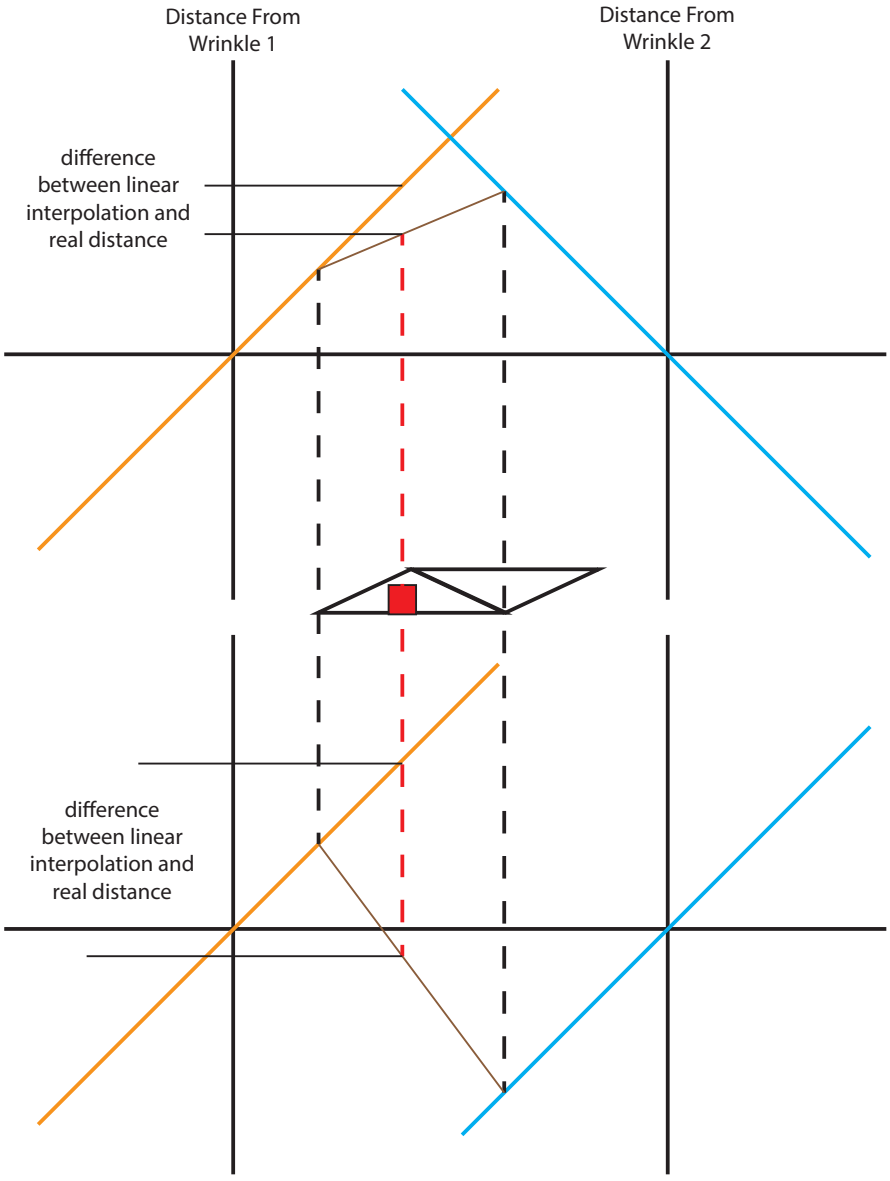

Figure 7.5: Given two wrinkles paths along theie respect z-axes, shown is the signed distance from each wrinkle (orange and blue), and the linear interpolation that would result across a primitive (brown). Wrinkle orientation may result in the overlap of similarly signed distance values (top), or opposite signed distance values (bottom).

7.5 Blending

One of the main challenges when modeling wrinkles is to believably handle cases where wrinkles overlap or merge. We found that a simple heuristic generates a feasible result while avoiding the computational overhead of more complicated blending operators (e.g., [16]). For each tessellation vertex within the region of influence of multiple wrinkle paths, we compute the height offset resulting from each path independently and select their maximum as the final offset. While this proves a good approximation of the deformed geometry, the discontinuous transition between contributing wrinkle paths results in an unrealistic, harshly shaded boundary when applied to normals. In the Fragment Shader we instead compute offsets and normals independently per path for fragment influenced by multiple paths. We generate the final normal as an average of these per-path normals weighted by their computed offsets (Figure 7.6).

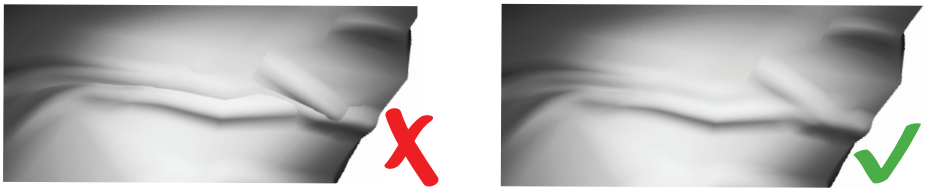

Figure 7.6: Result without (left) and (with) wrinkle blending.

Chapter 8

Proximity Testing Around Wrinkle Paths

With our wrinkles represented by their path, rather than as inherent constructs of a particle simulation or height field parametrization, it becomes very important to efficiently manage the computation of distance between wrinkle paths and from points on the mesh to these paths. This information is used to properly seed new wrinkles, to truncate wrinkle paths should they merge or cross (so no wrinkle paths are duplicated), and to determine the region each wrinkle path affects, so that triangles may be mapped to wrinkles for tessellation on the GPU. The algorithm chosen for each situation, and the precision it offers in computation of distance, play a large role in both the result and performance of our wrinkle computation on both the CPU and GPU.

8.1 Necessary Precision

We measure the precision of a test with respect to the error from the spline of the wrinkle path that will be computed on the GPU. The necessary precision for reasonable results varies for each of our use cases. Wrinkle seeding requires the most precision as we may have multiple seeds in each triangle, and want to ensure that the paths at each seed are disjoint and non-merging. In this situation full computation of the spline of each segment is unavoidable. Fortunately it need only

be done for one wrinkle for each comparison, as comparisons are being done with respect to the seed point.

Tests for wrinkle collisions or merging are less dependent on precision, as slight discrepancies are handled nicely by our normal blending. Truncating wrinkles upon merging helps reduce the number of wrinkle segments processed on the GPU, and crossing within the precision difference of a spline from its polyline is non-perceptible. It is sufficient in this case to merely compare shortest distances between the wrinkle paths.

In designing the test on proximity of triangles to a given wrinkle path it is not sufficient to test triangles against a bounding a box, as later refinement of the wrinkle path may curve the path outside of the box, resulting in un-tessellated regions begin deformed, and tears forming in the mesh. However with the sheer number of comparisons being done it is computationally heavy to compute the spline for each wrinkle; therefore we instead compute bounding ellipses that encapsulates the maximum possible deviation from the path that spline will take. Accuracy plays as much an effect on performance for this application as the algorithm efficiency, since accuracy effects the amount of data that will be passed to the GPU and subsequently processed.

8.2 Bounds Testing

In all comparison cases it is necessary to narrow down the search space prior to performing more accurate bounds or distance tests. In the case of wrinkle seed and wrinkle-wrinkle comparisons, we maintain a look-up table for wrinkles and their mapped triangles, allowing us to easily avoid the comparisons of wrinkles not passing through the same triangle. Each segment of a wrinkle path also maintains the triangle in which it resides, allowing for simple boolean checks for collision on each segment of a wrinkle prior to performing an actual distance check.

To determining the triangles affected by a given wrinkle for the GPU we walk radially from each wrinkle segment, as their triangles are known, and test each triangle we pass against a rough bounding ellipse for distance. We record each segment affecting a triangle and pass to the GPU a list of affecting wrinkles for each triangle, and the start and end segment indices within that wrinkle that affect

the triangle. In the control shader, any triangle with affecting wrinkles is tesselated.

Chapter 9

Results

We demonstrate our method on a range of meshes and animation sequences, most of which are taken from currently shipping video game titles. Our method adds believable, temporally coherent wrinkles to low-resolution character meshes and geometry (Fig. 9.6, 9.7, 9.8 and elsewhere in the paper.). The skirt example (Fig. 9.8) was provided courtesy of Kavan et al.[24] and is created using coarse physics-based simulation.

9.1 Parameters

Our system makes use of two tunable parameters closely related to expected material properties. Compression sensitivity τ is linked to the ease with which a modeled fabric bends, and minimal wrinkle width L_{min} is linked to how tightly compressed a material can to become. Compression sensitivity may also depend on the quality of the animation – one may choose to use a higher sensitivity threshold if the animation is more noisy and exhibits more spurious tangential motion. Figure 9.1 shows the impact of changing the parameter values. Table 9.1 list the numbers used for the animation sequences shown in the paper.

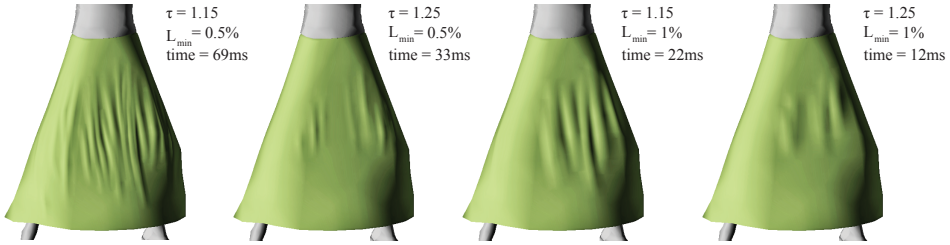

Figure 9.1: Impact of different choices of parameter values τ and L_{min}.

9.2 Performance

Table 9.1 shows the run-time performance of our algorithm on various meshes, as well as the mesh triangle counts and wrinkle parameters used. All times are computed on an Intel Core i7-3930K CPU running at 3.2 GHz, with 32 GB of RAM and an NVIDIA GeForce GTX 670 graphics card. Frames were rendered at a resolution of 1920x1080.

Figure 9.1 reports the impact of the choice of parameters on performance. As expected, performance decreases as the number of wrinkles increases (τ decreases) and their width increase; however, we maintain interactive performance throughout. The majority of our time per-frame is spent in the fragment shader, computing wrinkle normals in real time; in a real world scenario such as a first-person action game where models occupy smaller portions of the screen, performance increases accordingly.

An argument can be made against the choice of combining both tessellation and per-fragment normal computation in terms of performance and results. Examination of this trade-off shows performance using solely a tessellation scheme to be heavily reliant on the level of refinement and thus width of each wrinkle. Figures 9.3 and 9.4 use a large number of wrinkles to demonstrate the effect on performance. Refining the mesh to a degree that artifacts are not exhibited often brings the performance well below that of a fragment shader alternative (Figure 9.5). The fragment shader provides stable results at all levels of refinement, and thus proves the better choice.

Animation	τ	L_{min}	# tri.	avg # wrinkles	avg. ref. shape computation time
Fig. 15 (a)	1.4	0.5%	734	22.75	1.69ms
Fig. 15 (b)	1.3	0.5%	602	5.60	1.69ms
Fig. 15 (c)	1.4	0.5%	534	5.89	1.74ms
Fig. 15 (d)	1.4	0.5%	628	11.80	1.57ms
Fig. 15 (e)	1.2	1%	356	10.57	0.92ms
Fig. 15 (f)	1.4	0.5%	602	19.03	1.72ms

Animation	avg. path tracing time	avg. modeling time	total frame time
Fig. 15 (a)	5.33ms	8.30ms	12.45ms
Fig. 15 (b)	3.92ms	5.53ms	11.69ms
Fig. 15 (c)	2.46ms	4.68ms	7.39ms
Fig. 15 (d)	3.77ms	6.55ms	10.67ms
Fig. 15 (e)	4.84ms	8.95ms	13.19ms
Fig. 15 (f)	5.69ms	9.78ms	14.66ms

Table 9.1: Performance statistics for various models processed with our algorithm. L_{min} computed as percent of character height. All times in milliseconds.

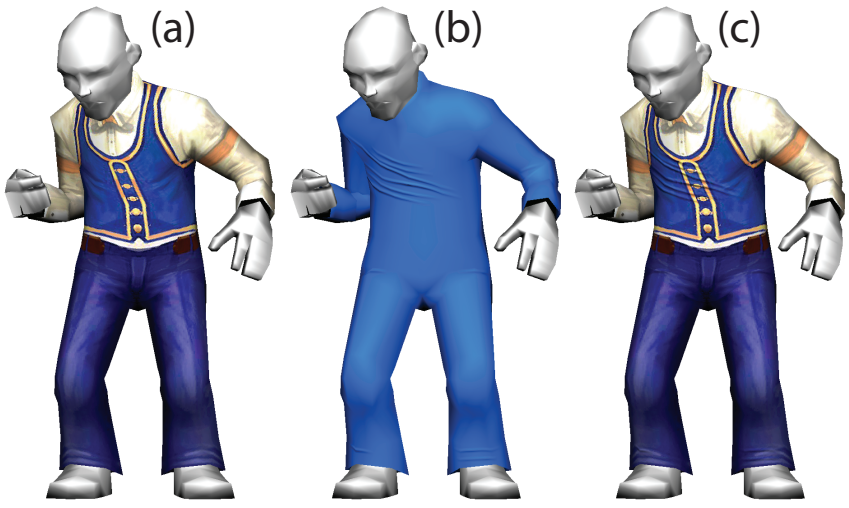

Figure 9.2: Left: Artist drawn static texture and our wrinkles without (center) and with same texture (right).

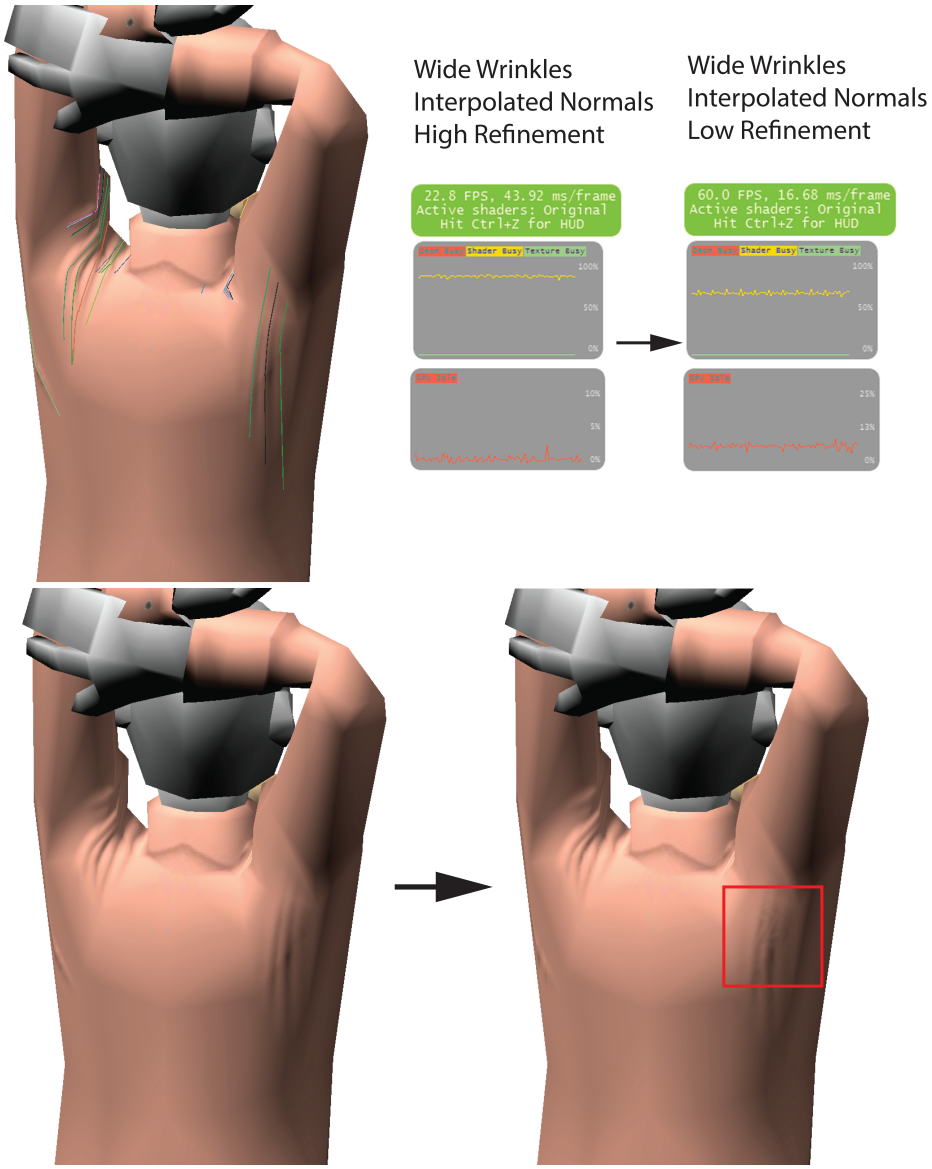

Figure 9.3: Use of interpolated normals may introduce artifacts for even wide wrinkles when reducing tessellation level to reach real time frame rates. From left to right the tesselation refinement is reduced, and we begin to see inconsistency in the normals of the highlighted area, resulting from insufficient granularity to determine the contributing wrinkle path.

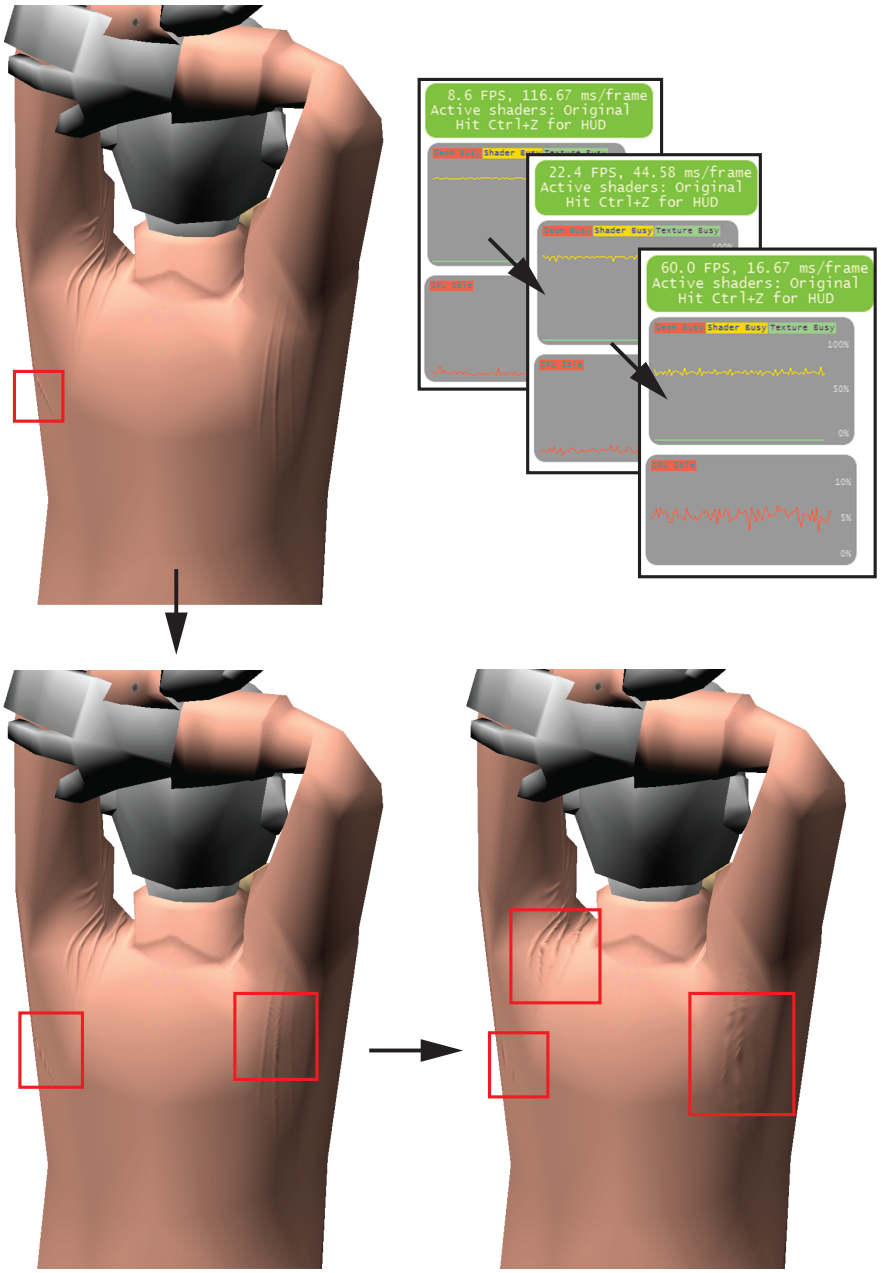

Figure 9.4: For large numbers of small wrinkles, the reduction of tessellation level results in visible artifacts well before reaching playable framerates. Three levels of tesselation with corresponding frame times are shown with normal inconsistencies highlighted in red.

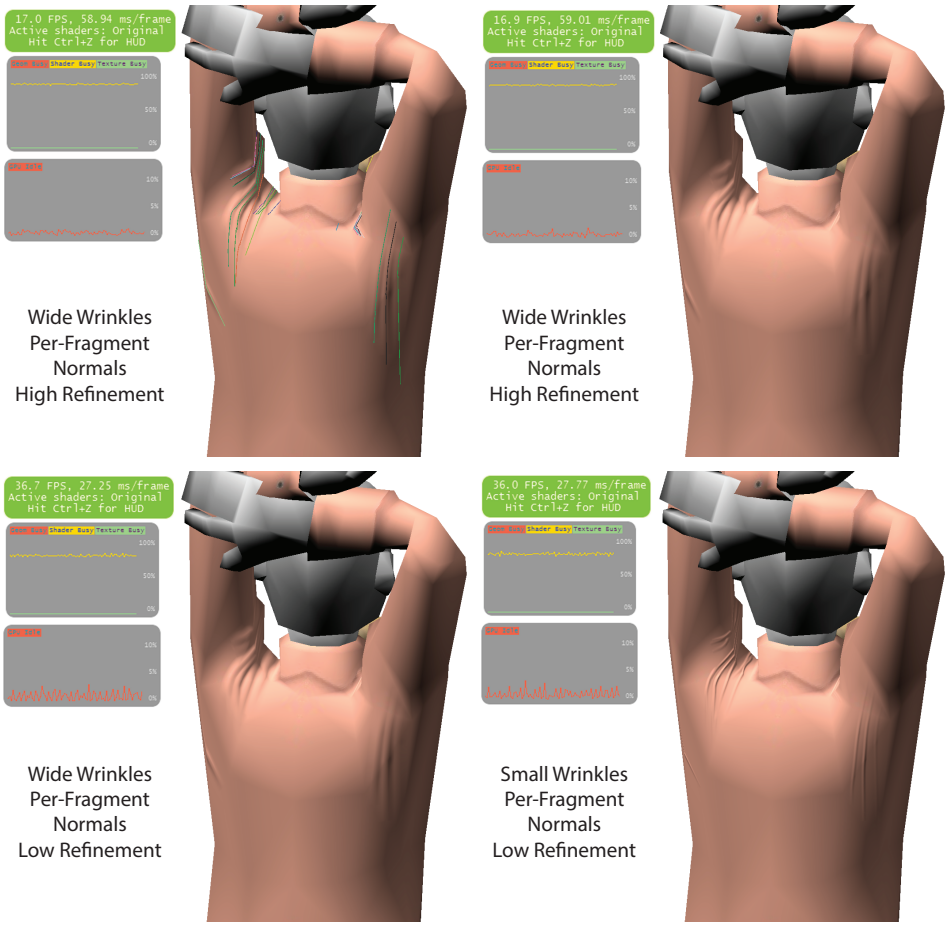

Figure 9.5: Fragment shader is not dependent on geometry tessellation for performance, and offers a stable, mid-level performance as compared to the range offered by a tessellation solution.

9.3 Comparison to Alternative Strategies

We validate our algorithmic choices against those used by previous work throughout the paper. Figure 3.3, (top) highlights the infeasibility of employing one of the frames in an animation sequence as a reference shape. Methods such as [35, 43] heavily rely on the existence of such reference shape. Figure 3.4 motivates our use of a path tracing strategy designed to operate on piecewise-constant instead of smoothed piecewise linear stretch tensor fields [43]. Our combined method is

dramatically faster than that of Rohmer et al.who report speeds of over a second per frame.

We also compare our method against the use of artist drawn wrinkles (Figure 9.2). We generate temporal motion-driven wrinkles, such as those on the chest, where static wrinkles would be meaningless, and place wrinkles at many concave joints (elbow, ankle, pelvis) where artist placed similarly shaped wrinkles. Since our method is motion based, if part of the anatomy remains fixed we will not generate wrinkles in that area; thus our method lacks wrinkles under the arms where the artist chose to place some.

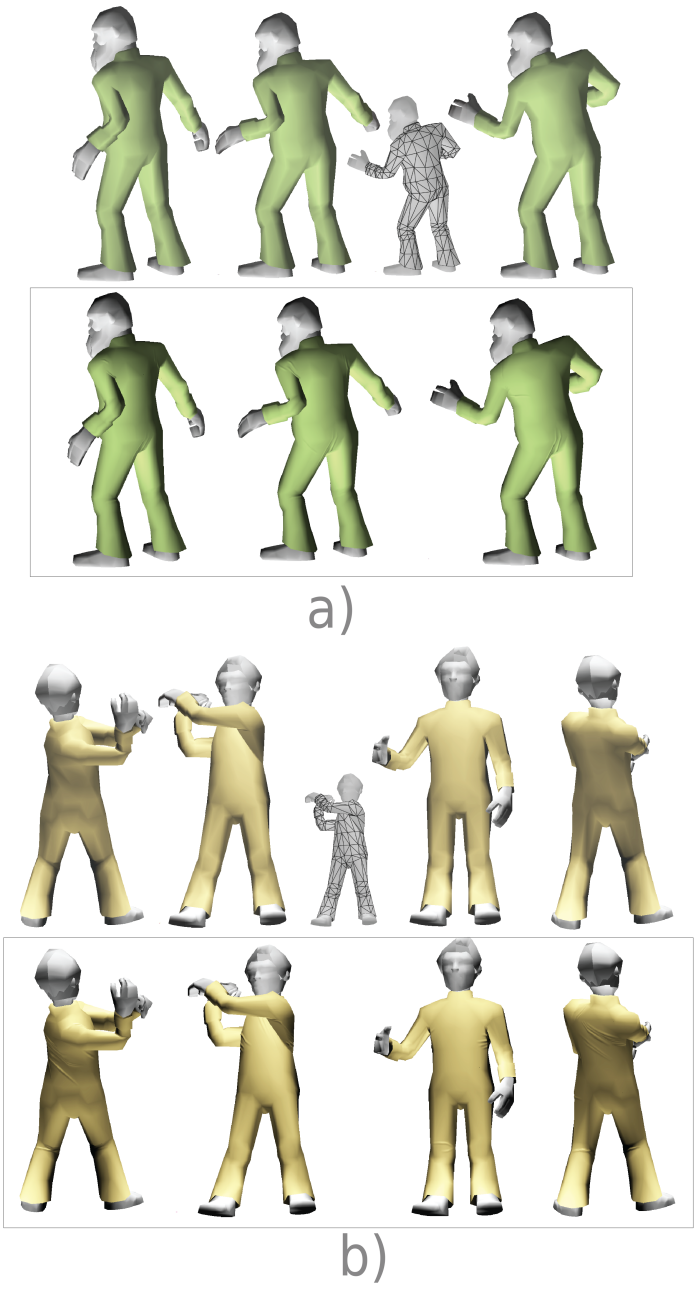

Figure 9.6: Some wrinkling results: input frames on top, wrinkled below.

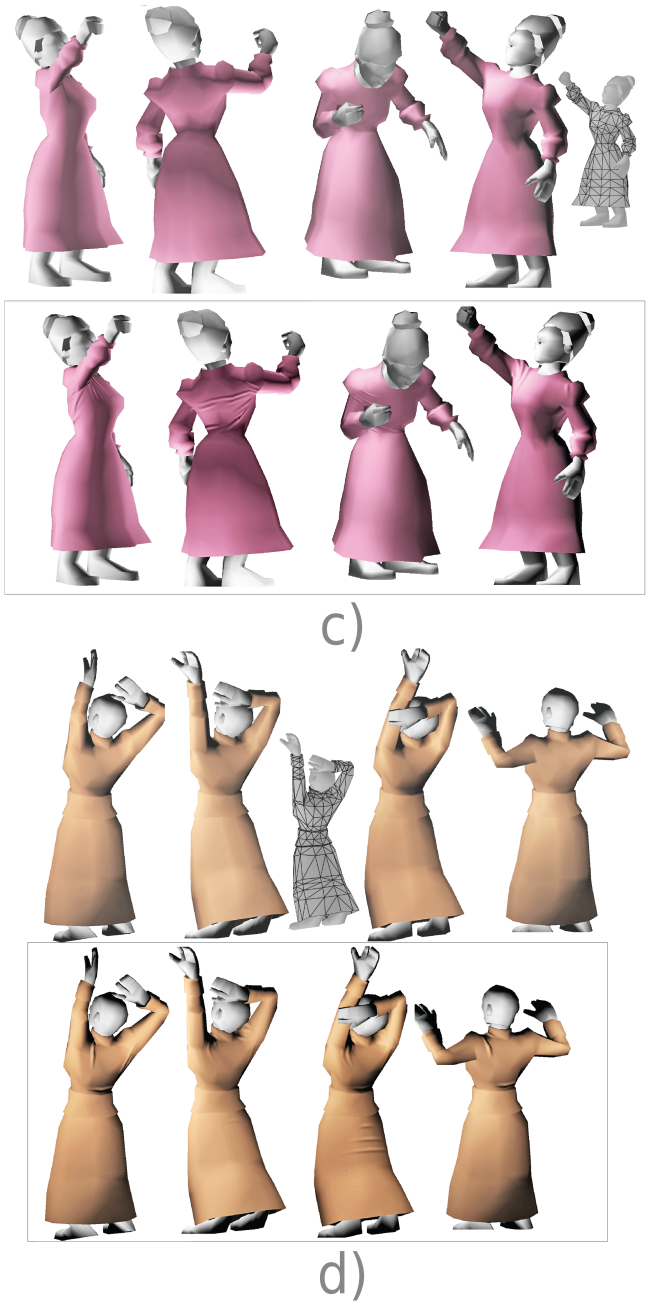

Figure 9.7: Some wrinkling results: input frames on top, wrinkled below.

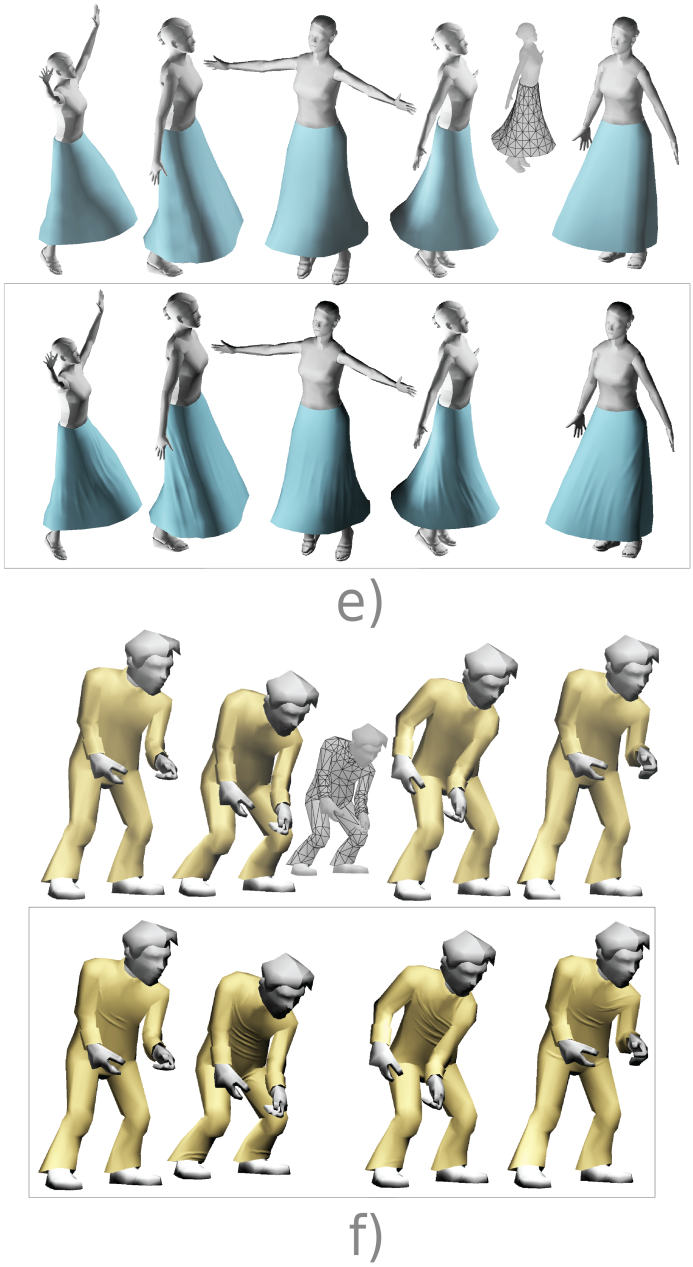

Figure 9.8: Some wrinkling results: input frames on top, wrinkled below.

Chapter 10

Conclusion and Future Work

We present a practical method for adding real-time wrinkling to cloth surfaces represented by coarse non-physically animated meshes. We have demonstrated the construction of a plausible garment reference shape from diverse mesh and animation inputs, and extracted from it a smooth tensor field through which to trace wrinkle paths. We generate wrinkle paths that are both spatially smooth and temporally consistant, and make full, efficient use of the GPU to tesselate, deform and render our wrinkles in real-time. Our method is suitable for interactive wrinkling of cloth and fabrics for video games on current-generation and next-generation consoles and hardware.

In the future we would like to explore improvements to both performance and rendering quality, as well as the application of our approach to other wrinkling surfaces, e.g., human skin. Our work addresses only dynamic wrinkles - i.e. those whose presence is hinted at by the cloth motion, and would benefit from the addressing of static wrinkles which humans often expect to be present in areas of negative Gaussian curvature.

Our method, like most previous work, generates outward bulging wrinkles - well suited for tight garments, but physically inaccurate on looser garments. Future work should explore automatic selection of wrinkle direction, or partial distribution of wrinkle deformation in both directions, to be consistent with human expectations. Additionally, our method does not currently handle stretch wrinkles, and incorporating this wrinkling is a source for future investigation.

Our method makes use of simple approximations for the falloff of wrinkle height towards its ends, and on the rate at which a wrinkle's length may change over time and as compared to compression values. Exploring the benefit of more detailed representations is a direction of future investigation. Further work may also explore approaches toward the persistence of wrinkles with less constraint on motion, or that avoid the need to reconstruct our wrinkle path in the neighbourhood surrounding a crossed vertex. Our current approach is restricted to moving vertices along edges; revisiting 3D walk approaches may offer smoother transition in a quickly changing discrete field.

A number of possibilities may be explored towards the further improvement of performance on the GPU. Global memory is accessed in 32, 64, or 128 byte memory transactions, and alignment of data to these boundaries times may significantly reduce the number of fetches required for each access [36]. Storage of wrinkles in memory location with faster access, such as texture memory or constant memory may also warrant examination. Both of these methods would require research into the access patterns and memory layout of wrinkles. An alternative is the complete reduction of required accesses and tests through the improvement of proximity testing around wrinkle paths.

The duplication of distance calculation to wrinkle paths between the Tesselation Evaluation Shader and Fragment Shader has proven unresolvable with the current graphics pipeline, however upcoming revisions to the specification propose changes that may expose the interpolation function during rasterization to the programmer. This would enable the exploration of higher order approximations for distance interpolation, or perhaps even height approximations.

Bibliography

[1] D. Baraff and A. Witkin. Large steps in cloth simulation. In *Proc. Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 43–54, 1998. → pages 6, 7

[2] M. Bergou, S. Mathur, M. Wardetzky, and E. Grinspun. TRACKS: Toward Directable Thin Shells. *ACM Transactions on Graphics (SIGGRAPH)*, 26 (3):50:1–50:10, 2007. → pages 11

[3] J. Bonet. *Nonlinear continuum mechanics for finite element analysis*. Cambridge university press, 1997. → pages 19

[4] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:1124–1137, 2004. → pages 22

[5] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:1222–1239, 2001. → pages 22

[6] 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*, pages 365–372. ACM, 1994. → pages 7

[7] R. Bridson, R. Fedkiw, and J. Anderson. Robust treatment of collisions, contact and friction for cloth animation. In *ACM Transactions on Graphics (ToG)*, volume 21, pages 594–603. ACM, 2002. → pages 8

[8] M. Carignan, Y. Yang, N. M. Thalmann, and D. Thalmann. Dressing animated synthetic actors with complex deformable clothes. In *ACM Siggraph Computer Graphics*, volume 26, pages 99–104. ACM, 1992. → pages 7

[9] K.-J. Choi and H.-S. Ko. Stable but responsive cloth. *ACM Trans. Graph.*, 21(3):604–611, 2002. → pages 5, 6, 7

[10] K.-J. Choi and H.-S. Ko. Research problems in clothing simulation. *Computer Aided Design*, 37(6):585–592, 2005. ISSN 0010-4485. doi:10.1016/j.cad.2004.11.002. URL http://dx.doi.org/10.1016/j.cad.2004.11.002. → pages 1, 6, 7

[11] B. Eberhardt, A. Weber, and W. Strasser. A fast, flexible, particle-system model for cloth draping. *Computer Graphics and Applications, IEEE*, 16(5): 52–59, 1996. → pages 7

[12] J. W. Eischen, S. Deng, and T. G. Clapp. Finite-element modeling and control of flexible fabric parts. *IEEE Computer Graphics and Applications*, 16(5):71–80, 1996. → pages 7

[13] E. English and R. Bridson. Animating developable surfaces using nonconforming elements. *ACM Trans. Graph.*, 27(3):66:1–66:5, 2008. ISSN 0730-0301. → pages 7

[14] H. Enqvist. The secrets of cloth simulation in Alan Wake. *Gamasutra*, Apr. 2010. → pages 3, 10

[15] R. Gillette, C. Peters, N. Vining, E. Edwards, and A. Sheffer. Real-time dynamic wrinkling of coarse animated cloth. In *Proc. Symposium on Computer Animation*, SCA '15, 2015. → pages iii

[16] O. Gourmel, L. Barthe, M.-P. Cani, B. Wyvill, A. Bernhardt, M. Paulin, and H. Grasberger. A gradient-based implicit blend. *ACM Transactions on Graphics (TOG)*, 32(2):12, 2013. → pages 52

[17] S. Green and N. Overview. Stupid opengl shader tricks. In *Advanced OpenGL Game Programming Course, Game Developers Conference*, 2003. → pages 8

[18] P. Guan, L. Reiss, D. Hirshberg, A. Weiss, and M. J. Black. Drape: Dressing any person. *ACM Trans. on Graphics (Proc. SIGGRAPH)*, 31(4): 35:1–35:10, July 2012. → pages 9

[19] S. Hadap, E. Bangerter, P. Volino, and N. Magnenat-Thalmann. Animating wrinkles on clothes. In *Proceedings of the Conference on Visualization '99: Celebrating Ten Years*, VIS '99, pages 175–182, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press. ISBN 0-7803-5897-X. URL http://dl.acm.org/citation.cfm?id=319351.319372. → pages 12

[20] F. Hahn, B. Thomaszewski, S. Coros, R. W. Sumner, F. Cole, M. Meyer, T. DeRose, and M. Gross. Subspace clothing simulation using adaptive bases. *ACM Trans. Graph.*, 33(4):105:1–105:9, 2014. ISSN 0730-0301. → pages 10

[21] J. Hu. *Structure and Mechanics of Woven Fabrics*. Woodhead Publishing Series in Textiles. Elsevier Science, 2004. → pages 3, 6, 7, 26

[22] T. Jakobsen. Advanced character physics. In *Game Developers Conference*, pages 383–401, 2001. → pages 8

[23] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, and S. Tachi. Detailed shape representation with parallax mapping. In *Proceedings of ICAT*, volume 2001, pages 205–208, 2001. → pages 12

[24] L. Kavan, D. Gerszewski, A. Bargteil, and P.-P. Sloan. Physics-inspired upsampling for cloth simulation in games. *ACM Transactions on Graphics (SIGGRAPH)*, 30(4):93:1–93:9, 2011. → pages 3, 9, 56

[25] Khronos. Opengl wiki, May 2015. URL https://www.opengl.org/wiki/Rendering_Pipeline_Overview. [Online; accessed 22-Aug-2015]. → pages 44

[26] D. Kim, W. Koh, R. Narain, K. Fatahalian, A. Treuille, and J. F. O'Brien. Near-exhaustive precomputation of secondary cloth effects. *ACM Transactions on Graphics*, 32(4):87:1–7, 2013. URL http://graphics.berkeley.edu/papers/Kim-NEP-2013-07/. Proc. SIGGRAPH. → pages 9

[27] V. Kolmogorov and R. Zabih. What energy functions can be minimized via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:65–81, 2004. → pages 22

[28] T. J. Lahey. Modelling hysteresis in the bending of fabrics. Master's thesis, University of Waterloo, 2002. → pages 6

[29] J. Lander. Devil in the blue-faceted dress: Real-time cloth animation. *Game Developer Magazine*, May 1999. → pages 1, 10

[30] T. Le Thanh and A. Gagalowicz. A new buckling model for cloth simulation. In A. Gagalowicz and W. Philips, editors, *Computer Vision/Computer Graphics Collaboration Techniques*, volume 6930 of *Lecture Notes in Computer Science*, pages 251–261. Springer Berlin Heidelberg, 2011. ISBN

978-3-642-24135-2. doi:10.1007/978-3-642-24136-9_22. URL http://dx.doi.org/10.1007/978-3-642-24136-9_22. → pages 6

[31] H. Li, Y. Wan, and G. Ma. A cpu-gpu hybrid computing framework for real-time clothing animation. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pages 391–396. IEEE, 2011. → pages 7, 8

[32] J. Loviscach. Wrinkling coarse meshes on the gpu. In *Computer Graphics Forum*, volume 25, pages 467–476. Wiley Online Library, 2006. → pages 12

[33] N. Magnenat-Thalmann and P. Volino. From early draping to haute couture models: 20 years of research. *The Visual Computer*, 21(8-10):506–519, 2005. → pages 7

[34] Microsoft. Microsoft dev center, May 2015. URL https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882. [Online; accessed 22-Aug-2015]. → pages 44

[35] M. Müller and N. Chentanez. Wrinkle meshes. In *Proc. Symposium on Computer Animation*, SCA '10, pages 85–92, 2010. → pages 3, 10, 13, 17, 61

[36] NVidia. Cuda toolkit documentation, March 2015. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#performance-guidelines. [Online; accessed 24-June-2015]. → pages 46, 48, 67

[37] C. Oat. Animated wrinkle maps. In *SIGGRAPH Courses*, SIGGRAPH, pages 33–37, 2007. → pages 10

[38] D. Rákos. Opengl developer forum, Sept 2011. URL https://www.opengl.org/discussion_boards/showthread.php/175672-shader-conditionals. [Online; accessed 24-June-2015]. → pages 46

[39] O. Rémillard and P. G. Kry. Embedded thin shells for wrinkle simulation. *ACM Transaction on Graphics*, 32(4):50:1–50:8, 2013. → pages 11

[40] J. Rodríguez-Navarro, A. Susín Sánchez, et al. Non structured meshes for cloth gpu simulation using fem. EUROGRAPHICS, 2006. → pages 8

[41] X. Rodriguez-Navarro, M. Sainz, and A. Susin. Gpu based cloth simulation with moving humanoids. In *Actas XV Congreso Español de Informática Gráfica (CEIG2005)*, pages 147–155, 2005. → pages 8

[42] D. Rohmer, S. Hahmann, and M.-P. Cani. Active geometry for game characters. In *Motion in Games*, volume 6459 of *Lecture Notes in Computer Science*, pages 170–181. 2010. ISBN 978-3-642-16957-1. → pages 11, 13

[43] D. Rohmer, T. Popa, M.-P. Cani, S. Hahmann, and A. Sheffer. Animation Wrinkling: Augmenting Coarse Cloth Simulations with Realistic-Looking Wrinkles. *ACM Transactions on Graphics (Proc. SIGGRAPH ASIA)*, 29(5), 2010. → pages vii, 3, 6, 11, 14, 17, 25, 26, 27, 28, 37, 61

[44] M. Tang, D. Manocha, J. Lin, and R. Tong. Collision-streams: fast gpu-based collision detection for deformable models. In *Symposium on interactive 3D graphics and games*, pages 63–70. ACM, 2011. → pages 7, 8

[45] M. Tang, R. Tong, R. Narain, C. Meng, and D. Manocha. A gpu-based streaming algorithm for high-resolution cloth simulation. In *Computer Graphics Forum*, volume 32, pages 21–30. Wiley Online Library, 2013. → pages 8

[46] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. In *ACM Siggraph Computer Graphics*, volume 21, pages 205–214. ACM, 1987. → pages 5, 7

[47] B. Thomaszewski, S. Pabst, and W. Straer. Continuum-based strain limiting. *Computer Graphics Forum*, 28(2):569–576, 2009. → pages 7

[48] T. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast cloth animation on walking avatars. In *Computer Graphics Forum*, volume 20, pages 260–267. Wiley Online Library, 2001. → pages 7, 8

[49] L. Verlet. Computer experiments on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical Review*, 159(1):98, 1967. → pages 10

[50] P. Volino and N. M. Thalmann. Implementing fast cloth simulation with collision response. In *Computer Graphics International Conference*, pages 257–257. IEEE Computer Society, 2000. → pages 7

[51] H. Wang, F. Hecht, R. Ramamoorthi, and J. F. O'Brien. Example-based wrinkle synthesis for clothing animation. *ACM Transactions on Graphics*, 29(4):107:1–8, 2010. URL http://graphics.berkeley.edu/papers/Wang-EBW-2010-07/. Proc. SIGGRAPH. → pages 9, 10

[52] J. Weil. The synthesis of cloth objects. *SIGGRAPH Comput. Graph.*, 20(4): 49–54, Aug. 1986. ISSN 0097-8930. doi:10.1145/15886.15891. URL http://doi.acm.org/10.1145/15886.15891. → pages 5

[53] C. Zeller. Cloth simulation on the gpu. In *ACM SIGGRAPH 2005 Sketches*, page 39. ACM, 2005. → pages 8

[54] D. Zhang and M. Yuen. A coherence-based collision detection method for dressed human simulation. In *Computer Graphics Forum*, volume 21, pages 33–42. Wiley Online Library, 2002. → pages 7

[55] D. Zhang and M. M.-F. Yuen. Collision detection for clothed human animation. In *Computer Graphics and Applications, 2000. Proceedings. The Eighth Pacific Conference on*, pages 328–337. IEEE, 2000. → pages 7

[56] J. S. Zurdo, J. P. Brito, and M. A. Otaduy. Animating wrinkles by example on non-skinned cloth. *IEEE Transactions on Visualization and Computer Graphics*, 19(1):149–158, 2013. URL http://www.gmrv.es/Publications/2013/ZBO13. → pages 9