LAYERED CHARACTER MODELS FOR FAST PHYSICS-BASED SIMULATION

DISSERTATION

zur erlangung des grades eines

doktors der naturwissenschaften

der technischen universität dortmund
an der fakultät informatik

von

MARTIN KOMARITZAN

dortmund
2022

Tag der mündlichen Prüfung:	29. August 2022
Dekan:	Prof. Dr. Gernot Fink
Gutachter:	Prof. Dr. Mario Botsch
	Prof. Dr. Klaus Hildebrandt

Dann schmolz die Wand, dann brach das Eisen
Und durch das Loch strömte das Licht
Ich spürte Lufthauch, einen leisen
Und hatte weithin klare Sicht.

- Walter Moers -

ACKNOWLEDGMENTS

The countless smaller and bigger steps leading to this thesis have been accompanied by many different people who supported me and shaped me to the person I am today. In the following list, which is definitely incomplete and has no meaningful order, I would like to say thank you.

I will start with my supervisor Prof. Dr. Mario Botsch who would tell me to omit his titles. He gave me the opportunity to pursue my PhD and always came up with helpful advice if I was stuck in my research. His extraordinary memory is a valuable source of information, both in terms of research in computer graphics as well as for telling interesting stories of his past. Furthermore, he cherished my passion for teaching and initiated me into the beauty of cleanly formatted equations. He has been the right combination of one-part friend and two-part supervisor I needed.

I am also grateful for having all the colleagues who spent the past five years with me. Their valuable feedback, our tea/coffee/ice-cream breaks and the friendly atmosphere in general did play an important role for my research. In particular, I would like to thank Stephan Wenninger for being my corona-lunch-mate, and Astrid Bunge for being my perfect office-match and fellow singer of Disney songs in times of joy and despair.

My family has been and still is a big source of inspiration and strength in my life. I thank my mother Julia for showing me that there can be more than one right answer, my father Steffen for not giving up, my grandparents Hannelore and Dieter for always being there when I needed them, my grandpa Detlef for supporting me and my interest in science from early on, my grandma Hanne for showing me that good results may take some time, and my sister Romy for being the person who understands me the most.

Lastly, I would like to thank my partner Maurice for ensuring a consistent supply of fresh air, water and delightful desserts. Moreover, he had been a great reference to the male body in different poses. Writing this thesis would have been much harder without his unconditional love and care.

ABSTRACT

This thesis presents two different layered character models that are ready to be used in physics-based simulations, in particular they enable convincing character animations in real-time. We start by introducing a two-layered model consisting of rigid bones and an elastic soft tissue layer that is efficiently constructed from a surface mesh of the character and its underlying skeleton. Building on this model, we introduce Fast Projective Skinning, a novel approach for physics-based character skinning. While maintaining real-time performance it overcomes the well-known artifacts of commonly used geometric skinning approaches. It further enables dynamic effects and resolves local and global self-collisions. In particular, our method neither requires skinning weights, which are often expensive to compute or tedious to hand-tune, nor a complex volumetric tessellation, which fails for many real-world input meshes due to self-intersections. By developing a custom-tailored GPU implementation and a high-quality upsampling method, our approach is the first skinning method capable of detecting and handling arbitrary global collisions in real-time.

In the second part of the thesis, we extend the idea of a simplified two-layered volumetric model by developing an anatomically plausible three-layered representation of human virtual characters. Starting with an anatomy model of the male and female body, we show how to generate a layered body template for both sexes. It is composed of three surfaces for bones, muscles and skin enclosing the volumetric skeleton, muscles and fat tissues. Utilizing the simple structure of these templates, we show how to fit them to the surface scan of a person in just a few seconds. Our approach includes a data-driven method for estimating the amount of muscle mass and fat mass from a surface scan, which provides more accurate fits to the variety of human body shapes compared to previous approaches. Additionally, we demonstrate how to efficiently embed fine-scale anatomical details, such as high resolution skeleton and muscle models, into the layered fit of a person. Our second model can be used for physical simulation, statistical analysis and anatomical visualization in computer animation or in medical applications, which we demonstrate on several examples.

CONTENTS

Acronyms **xi**

1 Introduction **1**

2 Fundamentals **5**
2.1 Skinning Basics 5
2.2 Projective Dynamics 8
2.2.1 Basic Constraint Types 10
2.2.2 Position- vs. Gradient-Based Strain Formulation 13
2.2.3 Hard Constraints in Projective Dynamics 16
2.3 GPU Programming 17

3 Fast Projective Skinning **23**
3.1 Related work 25
3.1.1 Geometric Skinning 25
3.1.2 Example-Based Skinning 26
3.1.3 Physics-Based Skinning 27
3.1.4 Position-Based Local-Global Solvers 28
3.1.5 Accelerating Projective Dynamics 28
3.1.6 Upsampling 29
3.2 Method 30
3.2.1 Generating a Volumetric Mesh 30
3.2.2 Coupling of Skeleton and Skin 35
3.2.3 GPU-Based Projective Skinning 37
3.2.4 Upsampling 43
3.2.5 Collision Handling 48
3.3 Implementation Details 54
3.4 Results & Discussion 56
3.4.1 Visual Results 56
3.4.2 Performance 61
3.5 Summary & Limitations 65

4 A Three-Layered Anatomical Model **67**
4.1 Related work 70
4.2 Method 72
4.2.1 Data Preparation 73
4.2.2 Generating the Volumetric Template 74
4.2.3 Estimating Fat Mass and Muscle Mass 79
4.2.4 Fitting the Volumetric Template to Surface Scans 82

4.3 Results and Applications 92
4.3.1 Evaluation on Hasler Dataset 94
4.3.2 Evaluation on CAESAR Dataset 94
4.3.3 Physics-Based Character Animation 97
4.3.4 Simulation of Fat Growth 99
4.4 Summary & Limitations 103

5 Conclusion 105

Bibliography 113

A Appendix 125

ACRONYMS

The following table lists all important acronyms that are repeatedly used throughout this thesis. The page on which each one is defined or used for the first time is also given. Acronyms and other abbreviations that are not defined in the text but instead used as names (e.g., OpenGL) are not included.

Acronym	Meaning	Page
ADMM	Alternating Direction Method of Multipliers	28
ARAP	As-Rigid-As-Possible	13
BF	Body Fat Percentage	71
BMI	Body Mass Index	73
CPU	Central Processing Unit	17
CRS	Compressed Row Storage	40
CoR	Skinning with Optimized Centers of Rotation	56
CUDA	Compute Unified Device Architecture	17
DQS	Dual Quaternion Skinning	25
FM	Fat Mass	73
FEM	Finite Element Method	26
FPS	Fast Projective Skinning	2
fps	Frames per Second	62
GPU	Graphics Processing Unit	17
ICP	Iterative Closest Point	74
LBS	Linear Blend Skinning	6
MAE	Mean Absolute Error	82
MLS	Moving Least Squares	44
MM	Muscle Mass	73
PBD	Position Based Dynamics	27
PCA	Principal Component Analysis	73
PCG	Preconditioned Conjugate Gradients	38
PD	Projective Dynamics	8
RBF	Radial Basis Function	91
SVD	Singular Value Decomposition	55
VAT	Visceral Adipose Tissue	89

1 INTRODUCTION

Virtual characters are ubiquitous in a wide range of graphics applications. In video games, we spend hundreds of hours with our animated virtual hero and run through massive virtual worlds full of animals, monsters and humans. In animated movies, characters are obviously virtually designed and posed, but even live action movies often use virtual clones for complicated scenes or special effects, and the audience can hardly detect any difference nowadays. However, virtual characters are not only found in the entertainment industry. Enabled by recent advances in 3D-scanning and character generation, realistic virtual avatars are also increasingly used in virtual reality applications, where they allow the user to act and interact in the virtual environment. This can be utilized in medicine (e.g., surgery simulations, psychotherapy) as well as in enterprises for virtual conferences. In this rapidly growing field of applications, the steadily improving fidelity of character appearance increases the demand for realistic character animation - while retaining interactive frame rates. There exist various approaches for animating a virtual character, however, the most realistic results are currently achieved by so-called example-based and physics-based approaches.

Example-based approaches try to learn the complexity of human motion from a large amount of data, typically consisting of 3D-scans of numerous different motions taken in small time increments. This is utilized to optimize parameters of an animation model such that it reproduces the input data. The resulting model works best if it remains close to the training animations, and can therefore effectively be used in environments where the set of motions is well defined (e.g., games). In cases requiring a general model that supports multiple characters, data-based approaches need to be trained on even more examples of different moving characters. This data is typically captured through 3D-photogrammetry scanning, requiring expensive scanning hardware. Using a pre-trained model can be a cheap alternative, however, it may not be flexible enough to support the desired target character. For instance, a change in the skeletal structure or the mesh topology usually requires to retrain the model.

Physics-based approaches build another main category for realistic character animation. Here, inspired by the real world, energies and forces are defined to determine the behavior of body compartments like muscles, fat and bones. These are applied to a volumetric discretization of the character's body, as opposed to data-based approaches, which often only model the character's surface. Physical simulations are capable of producing very realistic animations including advanced effects like collision handling, gravity, wobbling of soft tissue and volume preservation. But this usually comes to the cost of high computation times (multiple seconds per frame are not unusual). Therefore,

those approaches are typically applied for offline animation like in movie production, where quality is more important than performance. Another disadvantage is the cumbersome construction process of the volumetric model, prohibiting novice users from using physics-based animation approaches.

The goal of this thesis is to extend the range of applications for physics-based character simulation by overcoming its two downsides. Our *Fast Projective Skinning* (FPS) approach allows to construct an easy-to-use volumetric model for virtual characters from minimal input. The model consists of two layers, a rigid bone and a soft flesh layer that are ready to be used in a physical simulation. Moreover, our method is able to animate this model in real-time while supporting dynamic jiggling effects and handling arbitrary contact cases (e.g., hand touches belly).

Although our FPS approach can be applied to a wide range of input models like humans, animals or even imaginary creatures, its two simple volumetric layers are just a rough approximation compared to the inside of a *real* living organism. For some applications, like medical surgery simulations, a detailed and accurate representation of the bodies' anatomy is inevitable, and determining the shape of anatomical structures usually requires expensive volumetric imaging techniques. For the second half of this thesis, we therefore sacrifice some generality of our FPS models by focusing on *human* virtual characters. Exploiting the knowledge we have about human skeletal and muscular structures enables us to develop a more accurate three-layered volumetric model (bones, muscles, fat). While the anatomy of different people is similar but not identical, one main challenge is to transfer the layers of a base (template) human to a specific subject. Especially the amount of fat and muscles can vary drastically between different people and must be taken into account. Our approach takes a surface scan of the target model as input and requires no volumetric imaging technique to create anatomically plausible, personalized models. Moreover, it is fast and fully automatic, and can thereby reduce hours of manual work to a few seconds of computation time.

The main contributions of this thesis are:

- An approach for creating a two-layered, simple volumetric model that allows fast physics-based animations.
- A novel skinning method based on this model that is the first real-time skinning approach supporting arbitrary collision handling.
- A three-layered, anatomically plausible model that can efficiently be transferred to various human shapes and is ready for physical simulations.

More detailed lists of contributions can be found in each chapter.

This thesis is based on the following publications:

Komaritzan, M. and Botsch, M. (2018). "Projective Skinning." *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1.

Komaritzan, M. and Botsch, M. (2019). "Fast Projective Skinning." In *Proceedings of ACM Motion, Interaction and Games*, 22:1–22:10.

Komaritzan, M., Wenninger, S. and Botsch, M. (2021). "Inside Humans: Creating a Simple Layered Anatomical Model from Human Surface Scans." *Frontiers in Virtual Reality*, 2.

The implementation of our skinning approach and the full accompanying video, which will be referenced throughout the thesis can be found here:

- `https://github.com/mbotsch/FastProjectiveSkinning`
- `https://youtu.be/_oxBTxTngN8`

2 FUNDAMENTALS

In this chapter, we describe some fundamentals that will be used throughout the thesis. To aid the reader to understand our Fast Projective Skinning (FPS) method (Chapter 3), we first start by introducing the basic concept of skinning. Second, we explain Projective Dynamics, a position-based solver for simulating both static and dynamic systems, which we will deploy and extend to animate our FPS models as well as to build and simulate the anatomical model (Chapter 4). Third, we give an introduction to GPU computing with CUDA concluding with a short list of optimization guidelines that will be applied to develop a custom-tailored GPU-version of our Fast Projective Skinning.

2.1 SKINNING BASICS

Virtual characters are usually discretized by surface meshes composed of thousands of vertices. To animate a character, we need to find new vertex positions for each animation frame, and defining these by hand would be an overly cumbersome process. If one pictures a simple motion of a human body, like raising an arm, many parts of the skin do not move individually. Instead, the skin roughly follows the motion of its underlying bones. Inspired by this observation, the animation process of virtual models is commonly split into two parts. First, the animator defines the desired motion by posing a simplified skeleton of the character. This skeleton has much less degrees of freedom and is therefore easier to control than the mesh itself. Second, a so-called *skinning method* is applied to compute a plausible movement and deformation of the surface mesh (skin), based on the motion of the skeleton.

In Figure 2.1 (left) we show an example of a skeleton used to animate a virtual character. Its basic structure is similar to a biological skeleton but its geometry is simplified to a set of *joint* nodes connected by lines representing the bones. It is structured in a hierarchy, starting with the root joint J_1. All joints connected to the root joint by one bone are called its *child* joints. Analogously, the root joint is their respective *parent*. This structure can be continued for the complete rig until each joint $J_k, k \in \{2, 3, \ldots, J\}$, has exactly one parent $J_{P(k)}$ and a number of child joints. We will call the skeleton's graph *animation skeleton* or *(skeletal) rig*. Note that this structure does not support loops or loose parts in the graph. These can occur in some rigs but for the scope of this thesis we will focus on the most common case of a connected graph without loops.

To define the initial positioning of joints in 3D space and enable simple control over pose changes, a so-called *local* transformation $\mathbf{L}_k \in \mathbb{R}^{4\times4}$ is assigned to each joint. It holds the translation from the parent's position

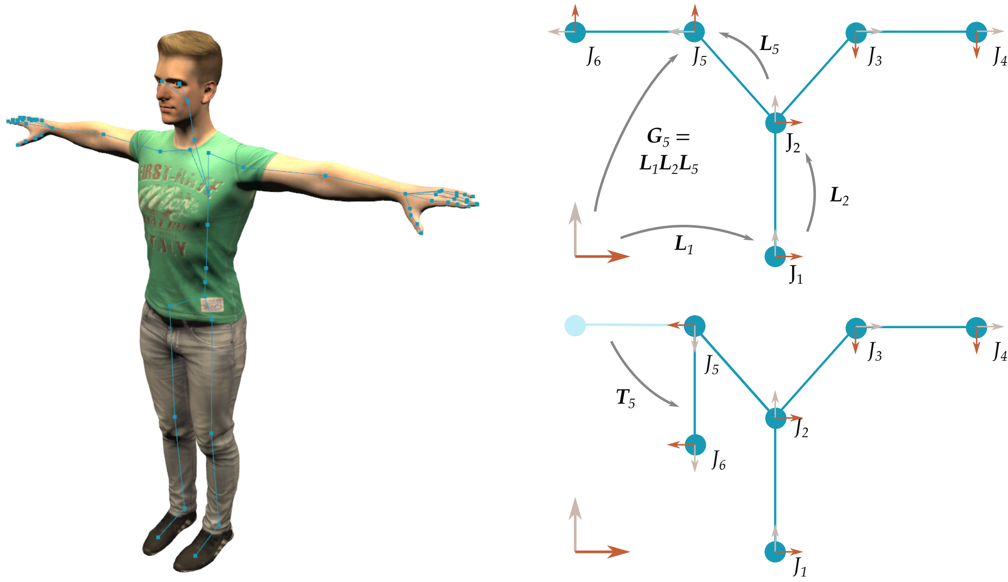


Figure 2.1: Left: a typical skeleton graph for a virtual character. Right: a 2D example of a skeleton graph. The different coordinate frames are depicted with small arrows. The joint's initial coordinate frame is rotated to align the gray y-axis with the child bone. The different mappings building the global transformation for the joint J_5 (right top) and its skinning transformation $\mathbf{T}_5$ (right bottom) are visualized. Note that arrow directions correspond to *active* transformations and the corresponding *passive* mapping between coordinate frames is directed inversely.

and, optionally, a rotation. In the initial state of the rig, the rotational part is often used to align a coordinate axis with a bone axis (see Figure 2.1, right). In case of the root joint, $\mathbf{L}_1$ holds the *global* translation and rotation $\mathbf{G}_1 = \mathbf{L}_1$ of the character. To compute the global transformation of a joint J_k, all local transformations in the kinematic chain of J_k are multiplied:

$$\mathbf{G}_k = \mathbf{L}_1 \cdots \mathbf{L}_{P(P(k))} \mathbf{L}_{P(k)} \mathbf{L}_k. \tag{2.1}$$

Each $\mathbf{L}_k$ transforms from the joint's to the parent's coordinate frame and $\mathbf{G}_k$ transforms from the joint's to the global coordinate frame respectively. We denote global matrices of the initial state by $\overline{\mathbf{G}}_k$.

To animate the character, different rotations are applied to the joint's local transformations $\mathbf{L}_k \leftarrow \mathbf{L}_k \mathbf{R}_k$, with $\mathbf{R}_k \in SO(3)$. The final transformation $\mathbf{T}_k \in \mathbb{R}^{4\times 4}$ of each joint maps from its initial to its current state $\mathbf{G}_k = \mathbf{T}_k \overline{\mathbf{G}}_k$, and can therefore be computed via

$$\mathbf{T}_k = \mathbf{G}_k \overline{\mathbf{G}}_k^{-1}. \tag{2.2}$$

These final mappings from the initial to the animated skeletal pose (Figure 2.1, right bottom) are typically used as input for skinning algorithms.

While there are many different skinning approaches, we will for now focus on the most basic one called Linear Blend Skinning (LBS), introduced by

Figure 2.2: Assigning each skin vertex to a single joint results in a rigid, block-like skinning (left). Blending multiple transformations allows for smoother transitions at joints (Linear Blend Skinning, right). The character's skinning weights that correspond to the hip bone are visualized on the right.

Magnenat-Thalmann et al. [1988]. It is still widely used and various other approaches can be interpreted as modifications of LBS. If the rest-pose skin mesh $\mathcal{S}$ is composed of S vertices at positions $\bar{\mathbf{x}}_i \in \mathbb{R}^3$, $i \in \{1, 2, ..., S\}$, the goal of any skeleton-based skinning approach is to find the animated vertex positions $\mathbf{x}_i$ based on the rig's transformations $\mathbf{T}_k$. A simple idea would be to assign each vertex to one bone (e.g., the closest one) and transform it with the transformation of the bone's corresponding parent joint. This leads to a kind of rigid skinning which can be desirable for animating robots or rigid parts of the character (e.g., an armor), however, the approach obviously fails at deformable regions of the skin, especially in proximity of joints (see Figure 2.2, left). The key idea of Linear Blend Skinning is, as hinted by its name, to linearly blend the joints' transformation matrices in those regions

$$\begin{pmatrix} \mathbf{x}_i \\ 1 \end{pmatrix} = \sum_k w_{ik} \mathbf{T}_k \begin{pmatrix} \bar{\mathbf{x}}_i \\ 1 \end{pmatrix} . \tag{2.3}$$

Here, w_{ik} are the blending weights, which should typically build a convex combination ($\sum_k w_{ik} = 1,\ w_{ik} \geq 0$) for each vertex. The process of defining these weights for a character is called *rigging*, and while automatic rigging approaches exist [Baran and Popović 2007; Jacobson et al. 2011], the best results are still achieved with the help of professional rigging artists that manually 'paint' the character's skinning weights. A LBS skinning result and an example of skinning weights is shown Figure 2.2. Once all w_{ik} are defined, LBS is very simple to implement, achieves real-time performance even for large scenes with multiple detailed characters and is therefore the most widely used skinning method in applications like games or VR-environments. As mentioned before, there exist numerous other skinning methods that will be (briefly) described and discussed in Section 3.1.

2.2 PROJECTIVE DYNAMICS

Methods for physics-based simulations of discretized deformable objects are generally based on Newton's second law of motion

$$m_i \ddot{\mathbf{x}}_i = \mathbf{f}_i, \tag{2.4}$$

where m_i is the mass associated to vertex $\mathbf{x}_i$ and $\mathbf{f}_i$ is the sum of all internal and external forces acting on it. Defining the velocity $\mathbf{v}_i = \dot{\mathbf{x}}_i$ and stacking the positions of all N simulated vertices into one matrix $\mathbf{z} \in \mathbb{R}^{N\times 3}$, as well as their velocities and forces $\mathbf{v}, \mathbf{f} \in \mathbb{R}^{N\times 3}$, results in

$$\dot{\mathbf{z}} = \mathbf{v} \quad \text{and} \quad \dot{\mathbf{v}} = \mathbf{M}^{-1}\mathbf{f}, \tag{2.5}$$

where $\mathbf{M} = \operatorname{diag}(m_1, m_2, ..., m_N)$ is a diagonal mass matrix. Implicit time-integration with time-step δt leads to the update rules:

$$\mathbf{z}_{t+1} = \mathbf{z}_t + \delta t\, \mathbf{v}_{t+1} \quad \text{and} \quad \mathbf{v}_{t+1} = \mathbf{v}_t + \delta t\, \mathbf{M}^{-1}\mathbf{f}_{t+1}, \tag{2.6}$$

where the subscripts t and $t+1$ denote the current and next time-step. Combining the two equations in (2.6) and assuming that forces are composed of constant external forces $\mathbf{f}_{\text{ext}}$ and position-dependent conservative internal forces induced by potentials $\mathbf{f}_{\text{int}}(\mathbf{z}) = -\sum_i \nabla W_i(\mathbf{z})$ yields

$$\frac{1}{\delta t^2}\mathbf{M}(\mathbf{z}_{t+1} - \mathbf{z}_t - \delta t\, \mathbf{v}_t) - \mathbf{f}_{\text{ext}} + \sum_i \nabla W_i(\mathbf{z}_{t+1}) = 0. \tag{2.7}$$

This is equivalent to solving the minimization problem

$$\mathbf{z}_{t+1} \;=\; \arg\min_{\mathbf{z}} \frac{1}{2\delta t^2} \left\| \mathbf{M}^{\frac{1}{2}} (\mathbf{z} - \mathbf{y}) \right\|_F^2 \;+\; \sum_i W_i(\mathbf{z}), \tag{2.8}$$

where $\mathbf{y}$ predicts the positions of the next time-step in absence of internal forces

$$\mathbf{y} = \mathbf{z}_t + \delta t\, \mathbf{v}_t + \delta t^2\, \mathbf{M}^{-1}\mathbf{f}_{\text{ext}}. \tag{2.9}$$

The first term of (2.8) therefore considers preservation of linear momentum and external forces, while the second term tries to minimize the internal potential energy.

The non-linear system involved in the minimization of (2.8) is typically solved with the help of Newton's method [Martin et al. 2011], which is computationally intensive due to the changing linear system, which has to be constructed and solved repeatedly in each time-step. Bouaziz et al. [2014a] introduce an alternative approach for solving (2.8), enabling much faster simulations of physical systems, which they call "Projective Dynamics" (PD). To

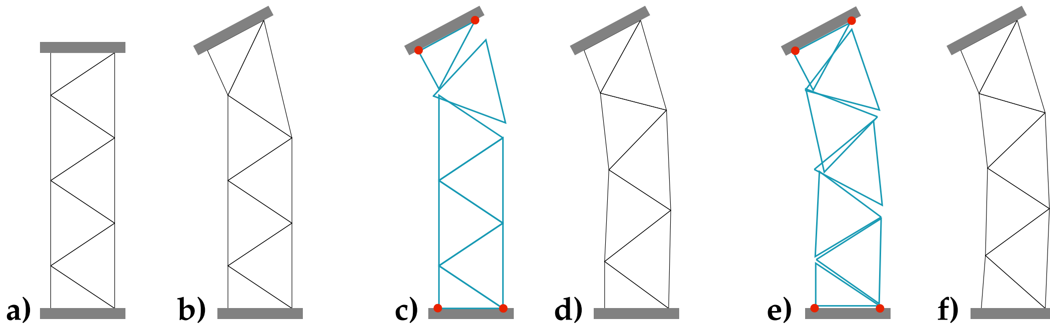


Figure 2.3: An example visualizing the local and global step of Projective Dynamics: A bar (a) shall be deformed by moving and rotating its top end (b). In the first local step (c), each element of the bar (triangles in 2D, or tetrahedra in 3D) will be projected to its closest undeformed state (shown in blue) while the top and bottom parts are projected to the positions at the boundary (red circles). The global step (d) determines vertex positions by combining the different projections. This process is iterated (e,f) for a fixed iteration count N_{PD}, where each iteration decreases the system's energy (2.8).

achieve this efficiency, they restrict the system to internal potentials of the following form

$$W_i(\mathbf{z}) \;=\; \frac{w_i}{2}\,\|\mathbf{Q}_i\mathbf{z}-\mathbf{p}_i\|_F^2\,. \tag{2.10}$$

Here, $\mathbf{Q}_i \in \mathbb{R}^{P_i\times N}$ has two functionalities: First, it extracts the positions contributing to the potential W_i (e.g., the two vertices of a spring or the four vertices of a tetrahedron). Second, it allows some processing of the selected vertices, like subtracting one from another to get a translation-invariant vector between two points. We will show different examples in the next section. $\mathbf{p}_i \in \mathbb{R}^{P_i\times 3}$ is the projection of $\mathbf{Q}_i\mathbf{z}$ to the closest allowed state of the potential, which will also become clearer in the next section. Analogously to the positions and velocities, the $\mathbf{p}_i$ and $\mathbf{Q}_i$ can be concatenated to a matrix $\mathbf{Q} \in R^{P\times N}$ and $\mathbf{p} \in R^{P\times 3}$, with $P=\sum P_i$. In that way, the accumulated internal potentials are expressed as

$$\sum_i W_i(\mathbf{z}) \;=\; \frac{1}{2}\left\|\mathbf{W}^{\frac{1}{2}}(\mathbf{Q}\mathbf{z}-\mathbf{p})\right\|_F^2, \tag{2.11}$$

where $\mathbf{W} \in R^{P\times P}$ is a diagonal matrix build from diagonal, P_i-sized blocks of the w_i. Inserting (2.11) into (2.8) yields

$$\mathbf{z}_{t+1} \;=\; \arg\min_{\mathbf{z}} \frac{1}{2\delta t^2}\left\|\mathbf{M}^{\frac{1}{2}}\,(\mathbf{z}-\mathbf{y})\right\|_F^2 \;+\; \frac{1}{2}\left\|\mathbf{W}^{\frac{1}{2}}(\mathbf{Q}\mathbf{z}-\mathbf{p})\right\|_F^2. \tag{2.12}$$

Note that the projections $\mathbf{p}$ depend on the current positions $\mathbf{z}$, meaning that the minimization still requires solving a non-linear system at this point. This is circumvented by alternatingly fixing either $\mathbf{p}$ or $\mathbf{z}$ and solving for the other. First, $\mathbf{z}$ is kept constant while solving for $\mathbf{p}$. Since each potential W_i typically depends on just a few different vertices, each $\mathbf{p}_i$ can be determined by solving

Algorithm 2.1: Time-step of the Projective Dynamics solver

1: $\mathbf{y} \leftarrow \mathbf{z}_t + \delta t\, \mathbf{v}_t + \delta t^2\, \mathbf{M}^{-1}\mathbf{f}_{\text{ext}}$
2: $\mathbf{z} \leftarrow \mathbf{y}$
3: **for** N_{pd} iterations **do**
4: $\quad \mathbf{p} \leftarrow \text{projectConstraints}(\mathbf{z})$
5: $\quad \mathbf{z} \leftarrow \text{solveGlobal}(\mathbf{z}, \mathbf{y}, \mathbf{p})$
6: **end for**
7: $\mathbf{z}_{t+1} \leftarrow \mathbf{z}$
8: $\mathbf{v}_{t+1} \leftarrow \mu\left(\mathbf{z}_{t+1} - \mathbf{z}_t\right) / \delta t$

multiple localized problems individually, which is called the *local* step. Second, $\mathbf{p}$ is kept constant transforming the minimization (2.12) into a *linear* system

$$\left(\frac{1}{\delta t^2}\mathbf{M} + \mathbf{Q}^{\mathsf{T}}\mathbf{W}\mathbf{Q}\right)\mathbf{z} = \frac{1}{\delta t^2}\mathbf{M}\mathbf{y} + \mathbf{Q}^{\mathsf{T}}\mathbf{W}\mathbf{p}, \tag{2.13}$$

which is solved in the *global* step of Projective Dynamics. Note that the system matrix $\frac{1}{\delta t^2}\mathbf{M} + \mathbf{Q}^{\mathsf{T}}\mathbf{W}\mathbf{Q}$ is sparse, symmetric, positive definite and does not depend on $\mathbf{z}$. Therefore, as long as masses are constant and no internal potentials have to be added or removed from the system, it can be pre-factorized using sparse Cholesky decomposition leading to an efficient global update.

The local and global steps are alternately iterated for N_{pd} iterations. Velocities are updated at the end of each time-step via

$$\mathbf{v}_{t+1} = \mu\frac{\mathbf{z}_{t+1} - \mathbf{z}_t}{\delta t}, \tag{2.14}$$

where $\mu \in [0, 1]$ can be used as damping parameter (0 corresponding to infinite damping and 1 to no damping). One time-step of Projective Dynamics is summarized in Algorithm 2.1. A visual example of two local-global iterations is shown in Figure 2.3.

Omitting the mass terms in (2.13) leads to the static (non-dynamic) version of Projective Dynamics which is equivalent to the previously published Shape-Up solver [Bouaziz et al. 2012]. The static systems do not involve inertial effects (i.e., dynamic 'wobbling') and thereby converge to a static state in fewer iterations. This enables simple, constrained-based energy minimization, which we will leverage in Chapter 4. In both versions of the solver, potentials can also be interpreted as a set of soft-constraints that should be satisfied by $\mathbf{z}$. In the following, we will therefore use the term *constraints* as a synonym for potentials.

2.2.1 Basic Constraint Types

In this section, we will define the constraint types of Projective Dynamics that will be used throughout the thesis. All constraint types have first been

introduced by Bouaziz et al. [2014a]. We will develop some specialized versions built on the base types shown here in later chapters.

Anchor Constraint

An anchor constraint is the simplest type of constraint, pinning a vertex to a specific location. Here, $\mathbf{Q_i}$ consists of a single row ($P_i = 1$), that selects the constrained vertex from $\mathbf{z}$, and the projection is performed by setting $\mathbf{p}_i$ to the desired location. Note that all potentials are handled as soft-constraints, meaning that in case of multiple competing constraints, the vertex is not guaranteed to stay at the anchor's position. We will solve this problem by reformulating anchors as hard constraints in Section 2.2.3.

Spring Constraint

A spring constraint keeps two vertices at a constant distance. Theoretically, this can be achieved by using two anchors, however, this would also circumvent any global translation of the spring. Instead, the constraint defines $\mathbf{Q}_i = \mathbf{B}_1\mathbf{S}_i$, where $\mathbf{S}_i \in \mathbb{R}^{2\times N}$ is a binary selector matrix selecting the two vertices and $\mathbf{B}_1 = \begin{pmatrix}-1 & 1\end{pmatrix}$ constructs the vector from the first to the second vertex. In this formulation, the constraint is translation-invariant, meaning that the global step can optimize the spring's translation in just one iteration. Therefore, a system of multiple constraints will converge much faster to the optimal solution. For a spring with rest length $\bar{L}$, the local update is

$$\mathbf{p}_i = \bar{L}\frac{\mathbf{Q}_i\mathbf{z}}{\|\mathbf{Q}_i\mathbf{z}\|}.$$

Note that the scaling can also be moved to the global step by $\mathbf{Q}_i \leftarrow \mathbf{Q}_i/\bar{L}$, but in this case, changing the desired rest length would affect the system matrix, resulting in a costly re-factorization. When splitting a spring into two or more smaller ones, the combined spring should not be stiffer or softer than the original. As derived by Bouaziz et al. [2014a], this resolution independence can be achieved by scaling each weight with the spring's rest length $w_i \leftarrow w_i\bar{L}$.

Tetrahedron Strain Constraint

A tetrahedron strain constraint penalizes the deformation of a tetrahedron from an initial state. Given a deformation $\mathbf{f}\colon \mathbb{R}^3 \to \mathbb{R}^3$ that maps the undeformed state to the deformed one, Chao and colleagues [2010] measure the elastic potential by the deviation of the deformation's differential $\mathrm{d}\mathbf{f} =: \mathbf{F}$ (also called deformation gradient) to the rotation group $SO(3)$. Discretizing with linear tetrahedral elements and casting the discrete strain into PD formalism yields

an elastic potential W for each element

$$W = \frac{1}{2} V \min_{\mathbf{R} \in \mathrm{SO}(3)} \|\mathbf{F} - \mathbf{R}\|_F^2 , \tag{2.15}$$

where V is the volume of the undeformed element, $\mathbf{F}$ the (constant) deformation gradient in this element, and $\mathbf{R}$ the rotation being closest to $\mathbf{F}$.

The state of a tetrahedral element can be defined by the edge matrix $\mathbf{E}_i = \mathbf{B}_3 \mathbf{S}_i \mathbf{z}$, where $\mathbf{S}_i \in \mathbb{R}^{4 \times N}$ is again the binary selector matrix and

$$\mathbf{B}_3 = \begin{pmatrix} -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{pmatrix}$$

builds the edge matrix by subtracting the first vertex from the other three vertices. The deformation gradient maps from the rest state $\bar{\mathbf{E}}_i$ to the current configuration of (column-wise) edges $\mathbf{E}_i^\top = \mathbf{F}_i \bar{\mathbf{E}}_i^\top$. As long as the initial rest tetrahedron is not degenerated, we get $\mathbf{F}_i = \mathbf{E}_i^\top \bar{\mathbf{E}}_i^{-\top}$.

The deformation gradient is translation-invariant but contains both the deformation $\mathbf{S}_i \in \mathbb{R}^{3 \times 3}$ and the rotation $\mathbf{R}_i \in SO(3)$ of the corresponding tetrahedron $\mathbf{F}_i = \mathbf{R}_i \mathbf{S}_i$, where $\mathbf{S}_i$ is a positive semi-definite, symmetric matrix. Transposing the matrices in (2.15) and inserting the definition of the deformation gradient yields the structure of a PD constraint

$$W_i = \frac{w_i}{2} \left\| \bar{\mathbf{E}}_i^{-1} \mathbf{B}_3 \mathbf{S}_i \mathbf{z} - \mathbf{R}_i^\top \right\|_F^2 , \tag{2.16}$$

with $\bar{\mathbf{E}}_i^{-1} \mathbf{B}_3 \mathbf{S}_i = \mathbf{Q}_i$ and $\mathbf{R}_i^\top = \mathbf{p}_i$. The weight is scaled by the volume for resolution independence $w_i \leftarrow w_i \, V_i = 1/6 \, w_i \, |\det(\bar{\mathbf{E}}_i)|$.

In each local step, we need to extract the rotational part of the deformation gradients, which is known as *polar decomposition* [Shoemake and Duff 1992]. The most common approach for polar decomposition utilizes the singular value decomposition of the deformation gradient $\mathbf{F}_i = \mathbf{U} \mathbf{\Sigma} \mathbf{V}^\top$ (we omit the subscripts here), where $\mathbf{U}, \mathbf{V} \in O(3)$ are orthogonal matrices and $\mathbf{\Sigma} = \mathrm{diag}(\sigma_1, \sigma_2, \sigma_3)$ is a diagonal matrix holding the singular values. The rotational part of $\mathbf{F}_i$ is now $\mathbf{R}_i = \mathbf{U} \tilde{\mathbf{\Sigma}} \mathbf{V}^\top$, where $\tilde{\mathbf{\Sigma}} = \mathrm{diag}(1, 1, \det(\mathbf{U}\mathbf{V}^\top))$ guarantees that $\mathbf{R}$ is a true rotation (i.e., $\det(\mathbf{R}) = 1$) by flipping the axis corresponding to the smallest deformation for inverted elements.

Tetrahedron Volume Constraint

A tetrahedron volume constraint penalizes volume changes of a tetrahedron from its rest volume. The approach is similar to that of the strain constraint (2.16). The weighting and $\mathbf{Q}_i$ will be identical, but the local step projects to the closest configuration with identical volume $\tilde{\mathbf{F}}_i$.

The volume of a tetrahedron is proportional to the determinant of its edge matrix leading to $\det(\mathbf{E}_i^\top) = \det(\mathbf{F}_i)\det(\bar{\mathbf{E}}_i^\top)$. Thus, a change in volume $dV = \det(\mathbf{E}_i^\top)/\det(\bar{\mathbf{E}}_i^\top) = \det(\mathbf{F}_i)$ can be determined from the deformation gradient. With the help of singular value decomposition, this can be further simplified to $dV = \det(\mathbf{F}_i) = \det(\mathbf{U\Sigma V}^\top) = \sigma_1\sigma_2\sigma_3 \det(\mathbf{UV}^\top)$. Now, the goal is to find the smallest change in singular values such that $dV = 1$. Following the PD-implementation of Deuss et al. [2015], this can be achieved by a few iterations of gradient descent minimizing $(dV - dV_t)^2$ with $dV_t = 1$. The resulting singular values build up a new matrix $\tilde{\mathbf{\Sigma}}$ defining the projection $\mathbf{p}_i^\top = \tilde{\mathbf{F}}_i = \mathbf{U}\tilde{\mathbf{\Sigma}}\mathbf{V}^\top$. Note that we can vary the target volume change dV_t to different values. Thereby, the constraint will blow up or shrink the tetrahedron's volume dependent on whether the parameter is smaller or greater than one.

Surface Bending Constraint

A surface bending constraint penalizes local changes in curvature. This is achieved by restricting the vertex Laplacian to rotations from its initial state

$$W_i = \frac{w_i}{2} \left\| \Delta\mathbf{x}_i - \mathbf{R}_i \Delta\bar{\mathbf{x}}_i \right\|_F^2 , \tag{2.17}$$

where $\mathbf{x}_i$ and $\bar{\mathbf{x}}_i$ denote the vertex positions of the deformed and the initial surface, respectively. Determining the optimal rotation $\mathbf{R}_i \in SO(3)$ can be simplified to rescaling $\Delta\mathbf{x}_i$ to the length of $\Delta\bar{\mathbf{x}}_i$ in the local step. The constraint matrix is set to $\mathbf{Q}_i = \mathbf{L}_i\mathbf{S}_i$, where $\mathbf{S}_i \in \mathbb{R}^{n_i \times N}$ selects $\mathbf{x}_i$ and its $n_i - 1$ one-ring vertex neighbors from $\mathbf{x}$. $\mathbf{L}_i \in \mathbb{R}^{1 \times n_i}$ is composed of the cotangent weights and Voronoi area A_i to construct the cotangent Laplacian $\Delta\mathbf{x}_i$ [Botsch et al. 2010]. For resolution-independence, the weight should be scaled by the Voronoi area $w_i \leftarrow w_i A_i$. Note that the term 'bending' constraint can be misleading, since the constraint does also constrain the scaling and therefore stretching of the surface.

2.2.2 Position- vs. Gradient-Based Strain Formulation

To define strain constraints penalizing the difference between the deformed and undeformed state of a tetrahedral element, the formalism explained before is straight-forward and commonly used in real-time simulations. However, in case of *polyhedral* elements, Bouaziz et al. [2012] define a rigidity constraint which corresponds to the frequently employed as-rigid-as-possible (ARAP) energy of Sorkine and Alexa [2007]. This constraint is still applicable in the tetrahedral case but we noticed that it produces different results than the previously defined tetrahedral strain constraint. In the following, we will compare both approaches to explain this behavior and clarify the misconception of both formulations being equivalent [Müller et al. 2007; Myronenko and Song

2009]. To allow for a better comparison and discussion, we omit some term simplifications and describe both in the same formalism, while borrowing some of the variables we already used in the tetrahedral case and modifying their definitions to clarify the analogies.

To describe the state of arbitrary polyhedral elements consisting of N_p vertices, we define $\bar{\mathbf{E}} \in \mathbb{R}^{3\times N_p}$ containing its N_p mean-centered vertices as columns, as well as its deformed version $\mathbf{E}$. In case of $N_p > 4$ there is in general no linear transformation mapping $\bar{\mathbf{E}}$ to $\mathbf{E}$. The deformation gradient $\mathbf{F}$ is therefore the linear transformation that minimizes $\left\|\mathbf{F}\bar{\mathbf{E}} - \mathbf{E}\right\|_F^2$ and can be found by solving

$$\mathbf{F} \;=\; \mathbf{E}\bar{\mathbf{E}}^\top \left(\bar{\mathbf{E}}\bar{\mathbf{E}}^\top\right)^{-1}. \tag{2.18}$$

Its closest rotation $\mathbf{R}$ can be determined via polar decomposition $\mathbf{F} = \mathbf{RS}$ minimizing the potential

$$W\left(\bar{\mathbf{E}}, \mathbf{E}\right) \;=\; \frac{1}{2}\, V\left(\bar{\mathbf{E}}\right) \min_{\mathbf{R}\in SO(3)} \left\|\mathbf{F} - \mathbf{R}\right\|_F^2, \tag{2.19}$$

analogous to the tetrahedral case (2.15).

Another frequently used approach for measuring the deformation of a volumetric element is the already mentioned ARAP energy [Müller et al. 2005; Sorkine and Alexa 2007; Bouaziz et al. 2012; Deuss et al. 2015], which can be written as

$$W\left(\bar{\mathbf{E}}, \mathbf{E}\right) \;=\; \frac{1}{2} \min_{\mathbf{R}\in SO(3)} \left\|\mathbf{E} - \mathbf{R}\bar{\mathbf{E}}\right\|_F^2. \tag{2.20}$$

It measures the sum of squared differences of the deformed mean-centered points to the optimally-rotated undeformed points of the element. Finding the optimal rotation is the well known orthogonal Procrustes problem [Golub and Van Loan 2012] and is equivalent to

$$\frac{1}{2} \min_{\mathbf{R}\in SO(3)} \| \underbrace{\mathbf{E}\bar{\mathbf{E}}^\top}_{\mathbf{H}} - \mathbf{R} \|_F^2. \tag{2.21}$$

It can be solved via polar decomposition of $\mathbf{H}$.

Comparing (2.18) and (2.21) reveals that the two approaches decompose different matrices ($\mathbf{F}$ and $\mathbf{H}$) into their rotation ($\mathbf{R}$) and deformation ($\mathbf{S}$) components. If we assume that $\mathbf{E} = \mathbf{RS}\,\bar{\mathbf{E}}$, we get

$$\mathbf{F} = \mathbf{RS}\bar{\mathbf{E}}\,\bar{\mathbf{E}}^\top \left(\bar{\mathbf{E}}\bar{\mathbf{E}}^\top\right)^{-1} = \mathbf{RS} \quad \text{and} \quad \mathbf{G} = \mathbf{RS}\bar{\mathbf{E}}\,\bar{\mathbf{E}}^\top \neq \mathbf{RS},$$

i.e., decomposing $\mathbf{F}$ restores the original rotation $\mathbf{R}$, while decomposing $\mathbf{H}$ in general results in a different rotation. Müller et al. [2007] argue that the factor $(\bar{\mathbf{E}}\bar{\mathbf{E}}^\top)^{-1}$ is symmetric and therefore should be a part of $\mathbf{S}$ and *not* contribute to the rotation. They conclude that both variants result in the same rotation $\mathbf{R}$. Myronenko et al. [2009] also state both variants to be equivalent. However,

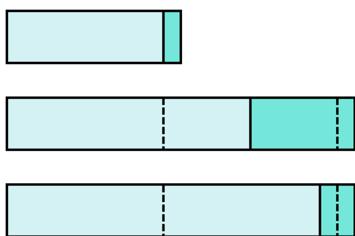


Figure 2.4: A two-element bar is stretched horizontally by a factor of two. When fitting positions (middle), edges minimize the *absolute* difference to their undeformed configuration. When fitting gradients, *relative* differences are minimized, preserving the size ratio of both elements (bottom).

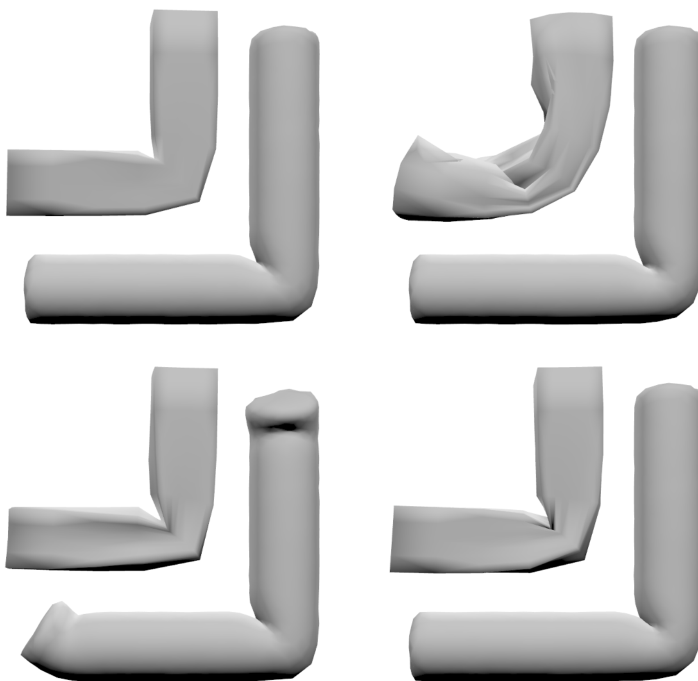


Figure 2.5: A high-resolution cylinder and a coarsely triangulated cuboid are deformed using our PD-based skinning (Chapter 3). Tetrahedral strain is handled with different combinations of: decomposing **F** (top) or **H** (bottom) and fitting gradients (left) or positions (right).

since the product of two symmetric matrices is in general not symmetric, this statement is not true, as pointed out earlier by Horn et al. [1988]. Moreover, **F** is independent of the element's shape while **H** is not, a drawback also mentioned by Chao et al. [2010].

Another interesting difference between the two approaches is that minimizing (2.19) tries to fit deformation gradients while minimizing (2.20) fits deformed positions/edges. The 2D example in Figure 2.4 shows the benefit of fitting gradients: when fitting positions/edges, the *absolute* differences between original and deformed edge lengths are minimized, while fitting gradients minimizes their *relative* differences. This difference can become more pronounced in case of irregular and inverted elements, where the changes in position can be small but gradients differ greatly [Chao et al. 2010]. An example is shown in Figure 2.5, which demonstrates that decomposing **F** and

fitting gradients yields better results in strongly deformed regions. Figure 2.5 also depicts that exchanging the computation of $\mathbf{R}$ between gradient- and position-fitting does not work at all.

2.2.3 Hard Constraints in Projective Dynamics

In the original Projective Dynamics [Bouaziz et al. 2014a], all internal potentials act as soft constraints and the global step computes a compromise between different projections concerning the same vertex. In case of an anchor constraint, the desired target position of a vertex is already known in advance, however, as long as there are concurrent constraints, the resulting vertex position will never perfectly coincide with the anchor's location.

We improve on this in a simple but effective manner: by replacing soft anchor constraints with *hard Dirichlet constraints*, we remove the N_b anchored vertices from the set of unknowns and thereby reduce the degrees of freedom from the previously $N_b + N_f$ simulated vertices to only the N_f unanchored ("free") vertices. To this end, we partition the vectors and matrices involved in (2.13) according to these two kinds of vertices:

$$\mathbf{z} = \begin{pmatrix} \mathbf{z}_f \\ \mathbf{z}_b \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} \mathbf{y}_f \\ \mathbf{y}_b \end{pmatrix}, \quad \mathbf{M} = \begin{pmatrix} \mathbf{M}_f & \mathbf{0} \\ \mathbf{0} & \mathbf{M}_b \end{pmatrix}, \quad \mathbf{Q} = \begin{pmatrix} \mathbf{Q}_f & \mathbf{Q}_b \end{pmatrix}.$$

Note that $\mathbf{Q}$, its sub-matrices and the projections $\mathbf{p}$ now no longer include anchor constraints, allowing the global system from (2.13) to be replaced by the reduced version

$$\left(\delta t^{-2}\mathbf{M}_f + \mathbf{Q}_f^\top \mathbf{W} \mathbf{Q}_f\right) \mathbf{z}_f = \delta t^{-2}\mathbf{M}_f \mathbf{y}_f + \mathbf{Q}_f^\top \mathbf{W} \left(\mathbf{p} - \mathbf{Q}_b \mathbf{z}_b\right).$$

For systems involving a high number of anchor constraints, the above matrix is considerably smaller than the original one. The absence of soft constraints moreover improves the matrix condition [Botsch and Sorkine 2008]. Furthermore, the anchor position is now perfectly preserved during simulation and we can skip the local step for all anchors. In the following, the number of simulated vertices N always denotes the number of unanchored vertices $N = N_f$.

2.3 GPU PROGRAMMING

There are typically two main types of processing units to execute the instructions of an algorithm: the *central processing unit* (CPU) and the *graphics processing unit* (GPU) . The CPU is equipped with a few powerful cores that are designed to handle a wide range of tasks in a very efficient manner. While the parallel execution of different instructions is possible, CPU-concurrency is limited to only a few tasks. Historically, the GPU is designed to compute the graphical output where the same operation must typically be performed several times (e.g., for each pixel of the screen, for each vertex/triangle of a mesh). For this purpose, it is equipped with hundreds or thousands of lightweight cores that are specialized to handle simple tasks in parallel. Nowadays, the GPU's extreme parallel potential is also leveraged for speeding up computations in many other fields apart from graphics, like physical simulations, machine learning or bitcoin mining.

Due to the historical connection to graphical tasks, GPU-communication is accomplished via shaders, small software modules specialized for the purpose of rendering a scene. Alternatively, there are more general frameworks like CUDA and OpenCL that are typically applied to more complex computational tasks since they support high-level programming languages and offer more flexibility than shaders. In this thesis, we decided to use CUDA due to its superior performance compared to OpenCL [Su et al. 2012]. CUDA (Compute Unified Device Architecture) was introduced by NVIDIA in 2006. They define it as a general purpose parallel computing platform and programming model for NVIDIA GPUs [NVIDIA 2021], allowing for the implementation and execution of algorithms on the GPU using C or C++ as programming language. To understand the design choices made during the implementation of our skinning algorithm on the GPU (Section 3.2.3), we will explain the basic concepts of GPU programming with CUDA in this section. For a more detailed CUDA guide, we refer to the CUDA C++ programming guide [NVIDIA 2021].

In order to run some instructions on the GPU, they must be placed into special CUDA functions called *kernels*. When launching (calling) a kernel, the number of assigned threads N_T must be set. Thereby, the kernel code will be executed N_T times in parallel (we will explain what 'in parallel' means later). Inside the kernel, an identifying index of the executing thread (thread ID) can be accessed to assign the correct portion of the parallel task to it. Below, we see a simple example kernel for adding arrays of float-values. Here, the thread ID is used to assign a specific addition to each thread:

```
__global__ void add(float* a, float* b, float* c)
{
    int id = threadIdx.x;
    c[id]  = a[id] + b[id];
}
```

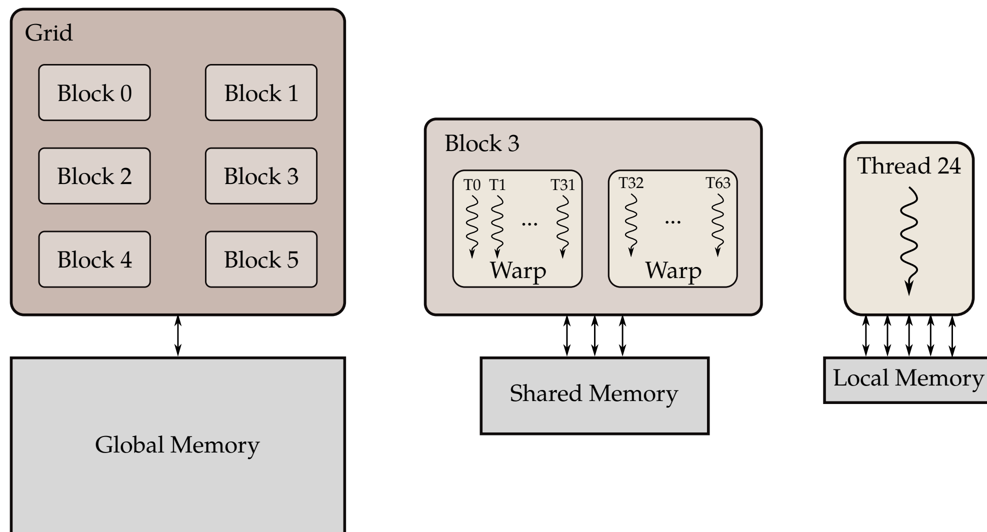


Figure 2.6: Example of CUDA's hierarchical thread and memory structure with $N_B = 6$ and $N_{\text{tpb}} = 64$.

Threads are structured in so-called *thread blocks* that are organized in a *grid* (see Figure 2.6). If a kernel is called, the number of threads per block N_{tpb} and the number of blocks N_B in the grid ($N_T = N_{\text{tpb}} \cdot N_B$) must be specified. These thread blocks are managed by one or more *streaming multiprocessors* (SM), where each of them is equipped with multiple CUDA cores to run different threads in parallel. The quantity of available SMs and cores per SM is dependent on the GPU. There is a limit to the maximum number of threads per block (currently 1024 for all existing NVIDIA GPUs) and also for the threads and blocks per multiprocessor that can be managed at the same time (dependent on the GPU's compute capability). The thread limit is typically higher than the number of cores per SM to allow switching between different threads (e.g., if one has to wait due to data access). If the number of assigned threads/blocks exceeds this limit, the SM will start executing its maximum quantity of blocks and load additional ones if a block is finished (see Figure 2.7). Each SM splits the resident blocks into smaller sub-blocks of 32 threads that are called *warps*. All threads in a warp execute the same instruction at the same time, meaning that no thread of the warp will start with the second instruction if another thread has not yet finished the first one (see Figure 2.7). If there is a branching instruction, each branch will be serialized within the warp, whereas different warps can execute different branches at the same time. Due to the division of blocks into warps, it is advisable to choose N_{tpb} to be a multiple of 32 and to avoid intra-warp branching if possible.

Calling a kernel does involve some overhead. Therefore, fusing multiple kernels into one is preferable over launching many smaller kernels. The only limiting factor is the need to synchronize threads to set some interim results of one thread as input for another one. While threads of a warp are always

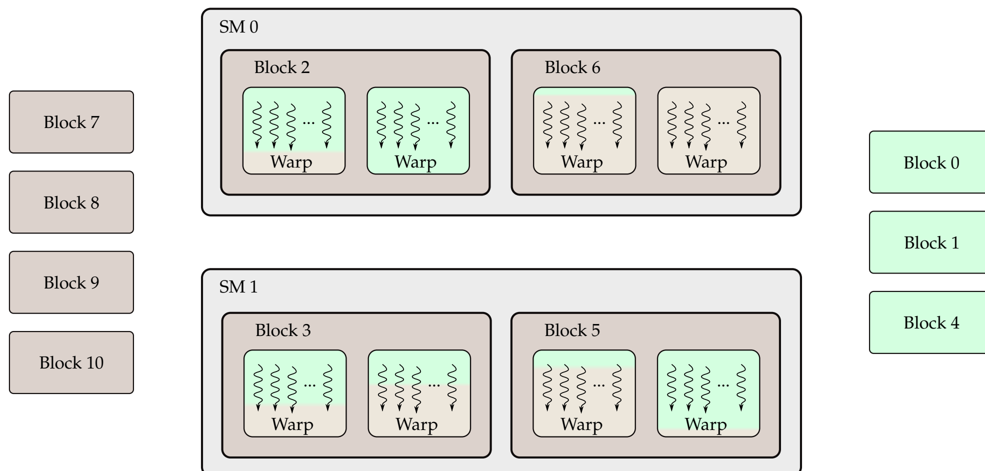


Figure 2.7: Example for parallel execution in CUDA with two streaming multiprocessors (SM) and a maximum of two resident blocks per SM. Each thread of a warp executes the same instruction in parallel. If all threads of a block have completed their tasks, the block is finished (right) and one of the unfinished blocks (left) can be processed by the SM. The order of warp-execution is determined by the SM.

synchronized, special synchronization points must be set to force threads of a whole block to wait for each other. Devices of compute capability 7.0 and higher also support thread *barriers* that allow GPU-wide or even system-wide (for multi-GPU setups) synchronization. On older devices, the only option to synchronize all threads in the grid is by ending the kernel and starting a new one. In order to not restrict our approach to very modern GPUs, we will use only block-wide synchronization in this thesis.

GPU-memory is also structured hierarchically. We will use the CUDA terms *host* (CPU) and *device* (GPU) in the following. The first important thing to note is that the GPU can just access device memory and not host memory. Therefore, data has to be allocated on the GPU and must be transferred between host and device. Since the data allocation is a costly operation, it should be performed just once at the beginning of a program if possible. The same holds for data transfers between host and device, which should also be minimized. In some cases it is preferable to execute some serial task on the device to avoid data transfers, even if the task itself would run faster on the CPU.

Device memory is divided into three basic types that differ in size and access speed. First, each thread has its own *local memory* called registers to store local variables. Read and write operations to registers are very fast but most devices are limited to 255 32-bit registers per thread (compute capability 3.2 and higher) and no thread can access data from another thread's register. For that purpose there is *shared memory*, which is shared across the complete thread block and limited to a few kilobyte per thread block (varying between 48 KB and 163 KB dependent on compute capability). Since shared memory is

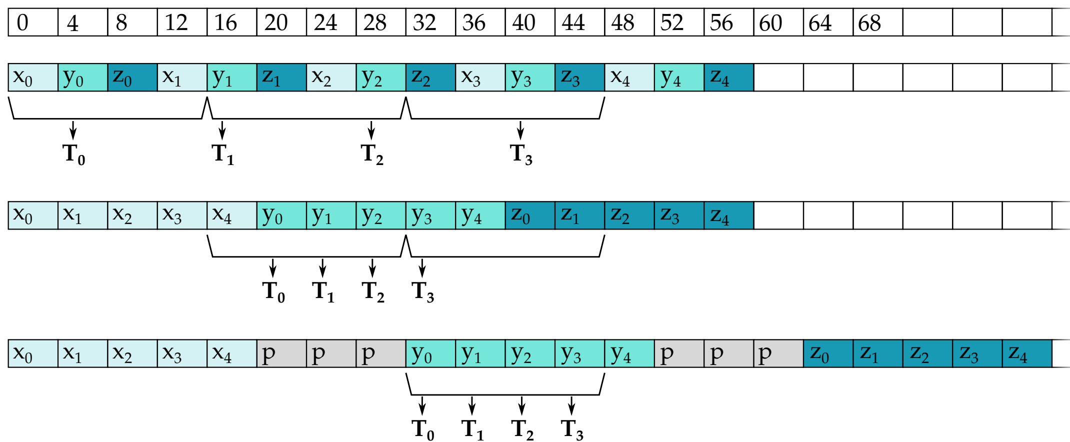


Figure 2.8: Example of memory coalescing with a reduced warp size of 4 threads (instead of 32) and 16 byte-sized cache-lines (instead of 128 byte). We want to access the y-values of a 4-byte float array storing five 3D-positions. Memory addresses are shown above. Top row: uncoalesced, three global reads. Middle row: partially coalesced, two global reads. Bottom row: fully coalesced (via padding), one global read operation.

also limited per multiprocessor, allocating a high amount of shared memory reduces the number of thread blocks that can be handled by the SM, which decreases performance. The majority of the device memory (some gigabytes) is so-called *global memory* that can be accessed from every thread. However, read and write operations are much slower here compared to the local and shared memory. Therefore, access to global memory should be minimized as much as possible.

When a warp accesses global memory, e.g., each thread reads a 4-byte float, the access occurs by reading whole cache-lines (typically 128-byte-sized) of data from the device memory. In the worst case, each float is located on a different cache-line leading to 32 read operations (an example is shown in Figure 2.8, top). In the best case, the floats are stored consecutively in device memory and the first float's address coincides with the start of a new cache-line. Here, just $1 = 128/(4 \cdot 32)$ read access per warp is sufficient (see Figure 2.8, bottom). This is called *memory coalescing*. Careful planning of the memory layout is needed to perform as many coalesced global read and write accesses as possible. This can also involve allocating additional memory (so-called padding) for optimal coalescing (compare Figure 2.8, middle and bottom).

In some cases the same data must be used in different parts of the algorithm, each with its own optimal access pattern. Here, memory coalescing can still be achieved with the help of shared memory. Shared memory is split into *memory banks*: if different threads of one warp access variables residing in the same bank, there is a so-called *bank conflict* and the access must be serialized. There are as many banks as threads per warp (32), and ideally, each thread

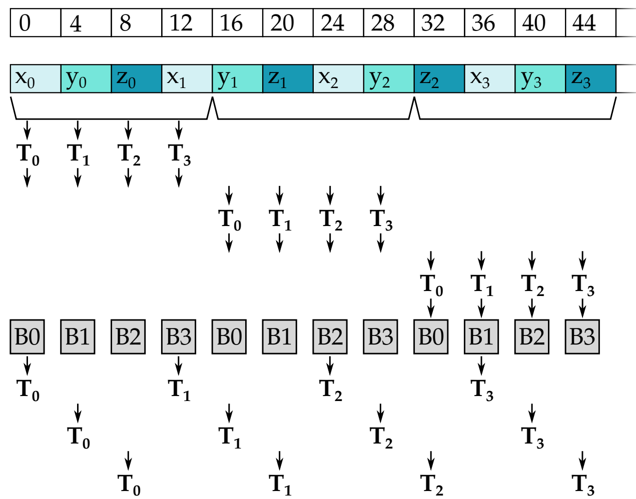


Figure 2.9: Example of transposing data via shared memory with a reduced warp size of 4 threads and 4 shared memory banks (B0, B1, B2, B3). Parallel operations are starting from the same height. First, the data is read from global memory using three coalesced memory accesses and written to shared memory (into a different bank for each thread). After synchronization, the values can be read coordinate-wise in parallel without bank conflicts.

accesses exactly one of them at the same time. As said before, accessing shared memory is much faster than global memory. Therefore, it can be advantageous to perform a coalesced read/write to global memory first, store the result in shared memory, synchronize the block and read in the desired access pattern. An example for transposing an array of 3D positions is shown in Figure 2.9.

There are also some specialized kinds of global memory called constant, texture and surface memory. Constant memory is read-only and provides fast access if all threads read the same address at the same time. Therefore, all kernel parameters that are passed by value, like sizes of arrays or constant parameters, will be stored in constant memory. Texture (read-only) and surface (read-write) memory use a special texture cache optimized for data access that provides some two-dimensional locality (e.g., parts of images or dense matrices). This texture cache can also be beneficial in case of unordered access patterns and is therefore an option if coalescing is not possible.

Summarizing the basic concepts of this section, the following steps are important to keep in mind while optimizing GPU CUDA code:

- minimize data transfers between host and device
- minimize the number of kernel executions
- minimize global memory accesses
- optimize data layout for global memory coalescing
- if the access must be uncoalesced, try using texture or surface memory

We will make use of these guidelines when optimizing our skinning code in Section 3.2.3.

3 FAST PROJECTIVE SKINNING

When it comes to real-time skinning in interactive applications, choosing between existing skinning approaches often means sacrificing one desired property for another. It seems that high performance, simplicity, a satisfying quality and a low amount of input data cannot be achieved at the same time. When building our own approach, we wanted to support advanced effects like dynamic soft tissue jiggling and collision handling without sacrificing real-time performance. Additionally, we aimed for a very simple, general and robust approach for both model construction and animation. Moreover, our method should yield convincing skinning results for a large variety of skin meshes without relying on capturing a lot of training data or on professional rigging expertise.

Our proposed approach, which we call *Fast Projective Skinning* (FPS), is able to meet these self-imposed requirements. The process of generating our volumetric model is very fast and requires just a minimum amount of input, namely the character's skin mesh and its embedded skeleton. These models can be animated using a physics simulation based on Projective Dynamics. Our method simulates convincing soft tissue deformations, provides secondary motion effects and can even resolve arbitrary global self-collisions while still maintaining real-time performance, even for detailed virtual characters. We achieve this by deriving a highly optimized GPU-implementation of Projective Dynamics that is able to dynamically add and remove collision constraints on demand. This required us to switch from the established factorization-based Cholesky solver to a custom-tailored GPU-based conjugate gradients solver, using a special matrix representation supporting fast matrix-vector multiplications. GPU-based FPS is therefore the first skinning approach handling arbitrary self-collisions in real-time. On the CPU however, where factorization-based solvers still achieve the best performance, we propose a more light-weight method for resolving local collisions in proximity of joints. Furthermore, we develop an upsampling technique to transfer the deformation from the lower-resolution simulation mesh to the high-resolution visualization mesh, which further improves the computational performance for both CPU and GPU simulations.

Fast Projective Skinning therefore enables real-time physics-based character animation even for novice users. While we aimed for a simple and general approach applicable for a large variety of input characters, we will also demonstrate several options for intuitive customization of the skinning result. To foster research in real-time physics-based character animation, our source code is freely available for research purposes on GitHub.

Individual Contribution *The author of this thesis developed the Fast Projective Skinning approach presented in this chapter. The whole process had been supervised by Mario Botsch. The main contributions of the method are:*

- *the simple volumetric mesh generation approach*
- *the skinning algorithm based on this model that is able to produce convincing, vivid animations in real-time*
- *a simple light-weight local collision handling approach*
- *the development of a highly optimized GPU implementation of the skinning method allowing to handle arbitrary global collisions in real-time*
- *an advanced, fast, MLS-based upsampling approach, which further increases the maximum level of detail for real-time skinning simulations*

Corresponding Publications *This chapter is based on the following publications:*

Komaritzan, M. and Botsch, M. (2018). "Projective Skinning." Proceedings of the ACM on Computer Graphics and Interactive Techniques, 1.

Komaritzan, M. and Botsch, M. (2019). "Fast Projective Skinning." In Proceedings of ACM Motion, Interaction and Games, 22:1–22:10.

3.1 RELATED WORK

In this section we discuss the most relevant related work on geometric, example-based, and physics-based skinning as well as on Projective Dynamics and upsampling. For more details on character skinning we refer the interested reader to the course notes of Jacobson et al. [2014].

3.1.1 Geometric Skinning

Magnenat-Thalman et al. [1988] introduced Linear Blend Skinning (LBS), which became the standard method to compute skeleton-driven skin deformations. While easy to compute, LBS suffers from artifacts like volume loss in joint regions or even collapsing joints in case of twisting bones. Many approaches try to overcome these drawbacks, for instance by using example poses [Lewis et al. 2000] or additional weights [Wang and Phillips 2002]. The artifacts in LBS are caused by the linear interpolation of rotation matrices, which, in general, does not result in a rotation. Dual Quaternion Skinning (DQS) [Kavan et al. 2008] uses dual quaternions instead of rotation and translation matrices to overcome this problem. However, DQS suffers from unnatural bulges in the presence of large rotations. More general skinning transformations [Jacobson et al. 2012; Kavan and Sorkine 2012] lead to better results than LBS and DQS, partly due to optimized skinning weights. Le and Hodgins [2016] reduce the artifacts of LBS and DQS by finding an optimal center of rotation for each vertex. Since these centers can be pre-computed, their performance is comparable to LBS and DQS.

A common drawback of geometric approaches is their dependence on high quality skinning weights, which either are complex to compute [Jacobson et al. 2011; Kavan and Sorkine 2012] or require hand-tuning by skilled artists. Automatic rigging methods avoid this problem, either by determining skeleton, joints and skinning weights through optimization [Baran and Popović 2007], or by transferring them from a high-quality template to the target model [Feng et al. 2015], or recently, with the help of neural networks trained for the rigging process [Xu et al. 2020]. While some methods require a closed two-manifold target model [Baran and Popović 2007; Xu et al. 2020], the auto-rigging method of Feng et el. [2015] can be applied directly to low-quality models, such as "dirty" raw 3D-scans. Mancewicz et al. [2014] introduce a skinning approach that does not depend on high quality skinning weights. They repair the artifacts of low-quality weights by smoothing the surface in rest pose, storing the offsets from smoothed to non-smoothed positions and applying them to the smoothed result of the deformed pose. This approach is able to produce convincing skinning results even for very simple weights (like the rigid binding shown in Figure 2.2, left) but is rather slow. Li et al. [2019] highly optimize

the approach and introduce two parameters that independently control the behavior for bone bending and twisting. Their Direct Delta Mush approach is able to skin a full detailed character in real-time but, like all the methods mentioned so far, it does not include collision handling or dynamic effects.

There are a few geometric approaches that support collision handling to a certain extent. Vaillant et al. [2013; 2014] tackle the problem by introducing Implicit Skinning. They describe the skin by an iso-surface of an implicit function and post-process the result of DQS or LBS by projecting vertices onto their initial iso-values. This approach can handle self-collisions, however, the projection produces unnatural results if the interpenetration is too deep. Though the resulting skin deformation looks very convincing and can even incorporate muscle bulging, it is rather slow and does not support dynamic skin deformation effects. The Steklov-Poincare Skinning of Gao et al. [2014] can handle collisions, but is also too slow and limited to static animations.

To enrich the geometric skinning result with dynamic secondary motion, Zhang et al. [2020] build a physical simulation on top of the skinned animation. They even incorporate collision forces but exceed the time-budget for real-time animations by some orders of magnitude. Rohmer et al. [2021] propose a more light-weight extension to a standard geometric skinning like LBS or DQS that supports different jiggling effects with just a small computational overhead. However, their method is more suitable for artistic, cartoon-like dynamic effects than for realistic skinning.

3.1.2 Example-Based Skinning

Example-based methods try to learn the skinning deformation from a given set of training examples. While some approaches learn corrective terms to an underlying geometric skinning method [Lewis et al. 2000; Weber et al. 2007; Li et al. 2021], other methods learn the whole skinning function [Anguelov et al. 2005a; Loper et al. 2015]. With the help of neural networks, even nonlinear soft-tissue dynamics can be learned from training data, resulting in highly dynamic animations [Casas and Otaduy 2018; Santesteban et al. 2020].

In the last few years, different hybrid models have been introduced that are composed of a simulated FEM layer on top of a data-driven skinning animation. Here, material parameters of the physically animated layer can be optimized with the help of examples. The simulated layer allows for dynamic motion effects and physics-based interaction [Kim et al. 2017] and can even support nonlinear, anisotropic elasticity of the skin [Romero et al. 2020]. The hybrid approach proposed by Tapia et al. [2021] uses a subspace to reduce the simulation complexity and can thereby handle collisions with rigid geometric primitives in real-time. Romero et al. [2020] support even complex skin-cloth interactions but at high computational costs. Holden et al. [2019] show that

surface deformations due to collisions with external objects can be learned by a neural network. Nevertheless, there is currently no approach that handles global self-collisions in real-time.

Another common drawback of all example-based methods is the need for training examples, which either have to be modeled by a professional artist or are obtained from 3D-scanning using expensive equipment. Moreover, the trained model is specific to a certain topology of skeleton and skin mesh, and changing either of these usually requires a re-training of the model.

3.1.3 Physics-Based Skinning

Physics simulations have also been used to compute convincing skin deformations. The main drawback of physics-based approaches are the often associated high computational costs. For instance, Kavan and colleagues [Saito et al. 2015; Kadleček et al. 2016] simulate a biomechanical model of the human body, and McAdams et al. [McAdams et al. 2011] propose a multigrid skinning simulation that supports contact and collisions. Both give impressive results, but are far too expensive for real-time scenarios. Capell et al. [Capell et al. 2002, 2005] achieved interactive frame-rates, but only by using a rather coarse volumetric simulation mesh.

To simulate meshes of higher resolutions at interactive rates, Position Based Dynamics (PBD) [Müller et al. 2005; Bender et al. 2017] became a valuable method in the last decade. PBD simulates elasticity by individually decreasing local elastic energies and can be parallelized. Rumman et al. [2014; 2015] use PBD to enhance the result of LBS. Although their results overcome the artifacts of linear blending, they cannot handle self-collisions and their simulation produces non-smooth transitions near joints. Pan et al. [2017] extends the PBD skinning of Rumman et al. [2015] by solving *local* collisions. They initialize each pose with the result of linear blend skinning and use PBD constraints to enhance the result. Their method requires high-quality skinning weights as well as a local smoothing step in the vicinity of joints, resulting in a loss of detail in those regions. Bender et al. [2013] use a combination of PBD and Shape Matching [Müller et al. 2005] to simulate a three-layered character model, featuring individual stiffness parameters for bone, fat, and skin tissue. While their method produces convincing deformations and handles self-collisions, the technique is too slow for real-time skinning. Moreover, it also depends on high-quality skinning weights to construct the tissue layers and for an initial LBS step. Roussellet et al. [2018] combines PBD with the Implicit Skinning approach of Vaillant et al. [2013] to couple implicit muscles, rigid bones and deformable skin. Their approach is remarkably fast considering the number of handled collisions, but still too slow for interactive simulations of a full character.

3.1.4 Position-Based Local-Global Solvers

While PBD is very fast, it has the drawback that different constraints sharing the same vertex can alternatingly project to different goal positions, resulting in alternating jumps [Bouaziz et al. 2014a]. Furthermore, the order of individual constraint projections can influence the result. Macklin et al. extend the PBD approach to decouple the material stiffness from the time-step [2016] and to support nonlinear elasticity models [2021]. However, their approach still suffers from the above mentioned drawbacks. Bouaziz et al. [2014a] introduce Projective Dynamics (PD) and show that PBD is a special case of PD. Their method overcomes the artifacts of PBD and converges in fewer iterations, however, it needs to solve a global linear system. Fortunately, the global matrix is constant and hence can be pre-factorized in most cases. By decoupling the constraint minimization from the positional update, the PD approach is also easier to parallelize than PBD. Narain et al. [2017] and Liu et al. [2017] noticed that PD can be interpreted as a special case of the alternating direction method of multipliers (ADMM) or a quasi-Newton approach, respectively. Their methods can handle more general nonlinear elastic models, but are more complex to implement. Since Projective Dynamics shows clear advantages over Position Based Dynamics while still being efficient and simple to implement, we decided to build our skinning simulation on this method.

3.1.5 Accelerating Projective Dynamics

There have been different approaches to speed up Projective Dynamics solvers. Wang [2015] uses a Chebyshev semi-iterative approach that leads to faster convergence in case of large deformations. However, for numerical robustness their method is not used in the first ten iterations of the PD solver. Since our character simulations are highly constrained by the skeleton, ten or even fewer iterations are usually sufficient to converge, such that their approach is not applicable. Fratarcangeli et al. [2016] use a graph-coloring algorithm to parallelize their Gauss-Seidel solver for the PD linear system (strictly speaking, this results in a PBD system). But even with an optimal graph coloring, at least all matrix rows corresponding to the one-ring neighborhood of a vertex have to be processed sequentially, such that the GPU's potential cannot be fully utilized for medium-sized systems (like in our case). Peng et al. [2018] employ Anderson acceleration to optimize the convergence of PD simulations. Although this can lead to a drastic reduction of solver iterations in static simulations, the effect is less apparent in dynamic simulations (such as our Fast Projective Skinning), where less iterations are required. Li et al. [2019] accelerate PD-simulations of coupled rigid and soft body parts, but their

method does not apply to skinning applications, where bone transformations are given as input instead of being simulated.

Brandt et al. [2018] use model reduction to simulate highly detailed models in real-time, achieving a massive speed-up compared to a simulation of the original high-resolution mesh. Lan et al. [2020] use another reduction approach based on the medial axis transform that also accelerates collision handling. Xian et al. [2019] also leverages reduced models in a multi-grid approach that substantially increases performance of PD simulations without sacrificing accuracy. Model reduction can be considered complementary to our acceleration approach and could lead to further speed-ups for very large models if combined with FPS.

3.1.6 Upsampling

Most real-time physics-based animations perform the actual simulation on a coarse simulation mesh, and transfer the deformation to the high-resolution visualization mesh. Müller et al. embed the visualization mesh into a coarse tetrahedral [2004a] or hexahedral mesh [2004b] for simulation. However, transforming the visualization vertices via *piecewise* (tri-)linear interpolation can lead to visual artifacts. Using normal displacements of the simulation mesh [Botsch and Sorkine 2008] can again lead to artifacts due to the piecewise linearity, for instance in regions of high bending. We instead employ and extend the moving-least-squares approximation of Martin et al. [2010], which is guaranteed to be globally smooth and therefore leads to higher surface quality of the upsampled visualization mesh. While Martin and colleagues employ GMLS instead of MLS to avoid numerical problems, we achieve the same effect by combining quadratic MLS with matrix pseudo-inversion, which reduces both memory consumption and computation effort by a factor of ten.

3.2 METHOD

In the following sections, we explain our *Fast Projective Skinning* approach. Beginning with the generation of volumetric meshes from minimal input (Section 3.2.1), we will show how to animate the resulting models using Projective Dynamics on the CPU (Section 3.2.2) or, alternatively, with our much more efficient GPU solver (Section 3.2.3). The last two sections cover our approaches on upsampling (Section 3.2.4) and collision handling (Section 3.2.5), allowing simulations of even highly detailed character meshes in real-time while handling both local and global collisions.

3.2.1 Generating a Volumetric Mesh

As pointed out earlier, physics-based simulations require *volumetric* models, while characters designed by artists or acquired via scanning methods are usually *surface* models. In order to keep the input of our approach as simple as possible, we first construct the volumetric mesh from the character's skin surface and an embedded skeleton. This also enables us to optimize the volumetric models with regard to our skinning method as explained in the following.

Related methods typically compute a tetrahedral mesh that interpolates the skin surface as its outer boundary and the skeleton's bone lines in the interior [Rumman and Fratarcangeli 2014, 2015]. We decided against this approach since tetrahedral mesh generation (e.g., [Si 2015]) requires the surface mesh to be closed and intersection-free, which is often not the case. For example, virtual characters are often 3D-scans of real people, where difficult-to-scan concave regions (e.g., crotch, armpits) can suffer from holes or self-intersections. While the modern tetrahedralization approach of Hu et al. [2018] robustly produces volumetric meshes even for complicated input surfaces, self-intersecting regions in the input will be merged in the output (e.g., right upper leg sticks to the left one). Moreover, their approach alters the surface mesh which would require to re-compute all vertex attributes (e.g., texture coordinates). To avoid tedious mesh repair sessions we aim at a volumetric mesh generation technique that also works in such real-world cases.

The main problem with LBS and DQS is the unnatural behavior near joints in presence of large rotations. In reality, the skin is stretched around the joint while pressing against it, and, as a consequence, it slightly bulges out. Other approaches try to approximate this behavior by correcting an initial LBS result through additional constraints that preserve the distance between skin and bone-line [Deul and Bender 2013; Rumman and Fratarcangeli 2015, 2014]. In contrast, our method uses volumetric bones and joints that achieve this bulging in a more natural manner without any corrective constraints. Changing size

and placement of volumetric bones and joints enables intuitive control over the skinning behavior for both static poses and dynamic jiggling.

The resulting method we propose for volumetric mesh generation is specialized for skinning models, very fast to compute, and provides an intuitive correspondence between skin and skeleton. It consists of the following steps: We first inflate the joint positions and line segments of the input skeleton to true volumetric bones and joints. Second, we shrink the skin surface onto the bone surface, such that each skin triangle and its copy on the bone surface span a prismatic element, which we eventually split into tetrahedra.

Bones and Joints

We represent volumetric joints as spheres and volumetric bones as cylinders with spherical caps, which connect the spherical joints at both endpoints of the bone segment (see Figure 3.1b). In this representation both joints and bones are easily defined by a single radius, are simple to compute, and allow for efficient intersection testing. While some joints would more accurately be described by ellipsoids, like for example the elbow, we choose spherical joints due to their simplicity and computational efficiency. Compared to real bones in living organisms, where even the slightest movement involves complex contact scenarios of adjacent bones, our simple bone and joint structure avoids skeletal self-intersections for a wide range of joint rotations.

The user may specify the joint and bone radii directly. Alternatively, we can also derive them automatically from the skin mesh and the skeleton as follows: the radius of each bone is set to 75 % of the bone's closest distance to the skin surface, and each joint radius is the maximum of all radii of its incident bones. To ensure that two joints (radii r_1 and r_2) and the connecting bone (radius r_b) fit into one bone segment of length l_b, the condition $r_1 + r_2 + 2r_b \leq l_b$ must be satisfied. In case of equality this results in a spherical bone (see hip bones in Figure 3.1b). Wherever the volumetric bones do not fit the bone segment lengths, we can fix this by scaling the associated bone and joint radii by a factor of $l_b/(r_1 + r_2 + 2r_b)$. Unless otherwise mentioned, all FPS examples shown in this thesis have been produced using this automatic approach.

Skin Shrinking

The region around joints is typically the area where large deformations occur, especially at the gaps between neighboring bones. If we generated a tetrahedral mesh that perfectly encloses the skeleton, tetrahedra in these regions would be heavily stretched and inverted which causes numerical issues in physical simulations. In real bodies there is a fluid lubricant instead of an elastic material in the this area that ensures smooth joint motion. Therefore, we want our volumetric mesh to wrap the skeleton tightly while leaving some free

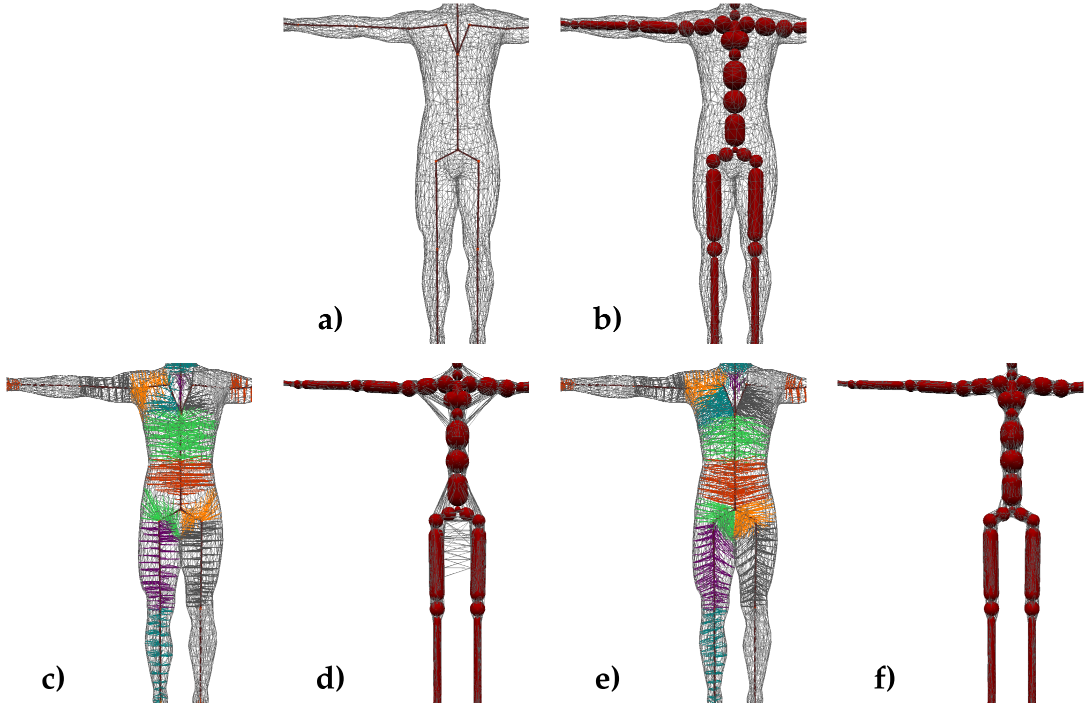


Figure 3.1: Given a skeleton and a skin mesh (a), we automatically compute volumetric bones and joints (b). Matching every skin vertex with its closest point on the skeleton (c) leads to artifacts when shrinking the skin to those vertices (d). Laplacian smoothing of the foot-points on the skeleton gives better correspondences (e), which leads to good shrinking results (f).

space at the connecting joint areas. In other words, we aim for something like a rubber-tube, enclosing the skeleton (see Figure 3.1f).

For this purpose, we shrink the vertices of the surface mesh down to the volumetric bones and joints. Given a skin surface $\mathcal{S}$ with S vertices $\mathbf{x}_1, \ldots, \mathbf{x}_S$, the shrinking process results in a skeleton surface $\mathcal{B}$ wrapping the bones and joints with S vertices $\mathbf{x}_1^{\mathcal{B}}, \ldots, \mathbf{x}_S^{\mathcal{B}}$. The goal is to find the vertex positions $\mathbf{x}_i^{\mathcal{B}}$ on the volumetric skeleton. To this end, we first find a point $\mathbf{s}_i$ on the initial (non-volumetric) input skeleton and then compute the intersection between the line segment from $\mathbf{x}_i$ to $\mathbf{s}_i$ and the volumetric skeleton, finally yielding $\mathbf{x}_i^{\mathcal{B}}$.

The trivial approach, setting each $\mathbf{s}_i$ to closest point on the input skeleton, is not sufficient, as depicted in Figure 3.1c,d. Incorporating vertex normals to determine the shrinking direction can improve the result, but fails for non-smooth skin surfaces. In failure cases neighboring vertices $\mathbf{x}_i$, $\mathbf{x}_j$ on the skin mesh map to very different positions $\mathbf{s}_i$, $\mathbf{s}_j$ on the skeleton.

The latter, however, can easily be corrected by Laplacian smoothing of the base points $\mathbf{s}_i$. We iterate

$$\mathbf{s}_i \;\leftarrow\; \mathbf{s}_i + \lambda \Delta \mathbf{s}_i \;=\; \mathbf{s}_i + \frac{\lambda}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} \left(\mathbf{s}_j - \mathbf{s}_i \right), \tag{3.1}$$

where $\mathcal{N}(i)$ denotes the one-ring neighbors (on the skin mesh) of vertex $\mathbf{x}_i$,

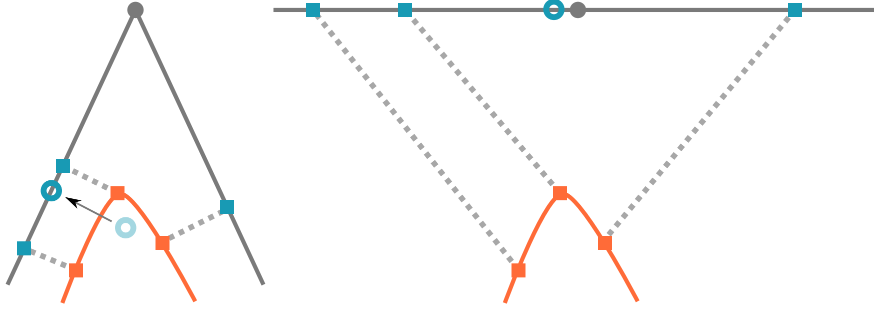


Figure 3.2: Left: when smoothing the bone-line points $\mathbf{s}_i$ (blue squares), the specific $\mathbf{s}_j$ corresponding to the vertex shown in the middle would be re-projected (black arrow) to the position indicated by the blue circle. Right: a better smoothing process can be achieved by first stretching out the skeleton graph, maximizing each angle between neighboring bones. Note that the skin (red) does just provide connectivity information and can stay unchanged. For simplification, we use $\lambda = 1$ in this example.

$|\mathcal{N}(i)|$ the number of neighbors of $\mathbf{x}_i$, and $\lambda \in [0, 1]$ a parameter to control the smoothing speed (we use $\lambda = 1/2$). Points at extremal leaf nodes (e.g., fingertips, toes) would move inwards during smoothing. However, the initial $\mathbf{s}_i$ is already the best correspondence in this case, and we can keep those fixed. Note that Equation (3.1) does not constrain the points $\mathbf{s}_i$ to lie on a bone-line after smoothing. We therefore project the updated $\mathbf{s}_i$ back to the skeleton after each iteration. If there is a small angle between two bone-lines (e.g., at the crotch in a T-Pose), the process can fail to produce a smoother result, as depicted in Figure 3.2, left. To solve this problem, we stretch-out all bone-lines to the maximal angle (meaning 180° for two connected bones, 120° for three, etc.) and move the initial $\mathbf{s}_i$ along with their bone-line. In that way, the distance of the back projection is minimized (see Figure 3.2, right). We also experimented with computing the smoothed result directly on the bone-line manifold. However, while this is computationally more expensive, we found no significant gain in quality compared to the iterative approach.

This smoothing process allows the base vertices to slide along the skeleton (also across joints) and is iterated until convergence. It yields a point distribution where neighboring skin vertices also have neighboring skeleton points (see Figure 3.1e) and we can now undo the limb-stretching step. The intersection between the line from $\mathbf{x}_i$ to $\mathbf{s}_i$ and the primitives of the volumetric skeleton defines the final points $\mathbf{x}_i^{\mathcal{B}}$ of $\mathcal{B}$. In the region between spherical joints and cylindrical bones, we test for intersection with a conical frustum connecting the two. Further details and illustrations of the intersection test can be found in the Appendix A. The resulting shrunken skin is depicted in Figure 3.1f.

Our choice of constructing *volumetric* bones/joints has important benefits. First, attaching the inner tissue layer to a volumetric bone naturally prevents the tissue from unnatural twisting around the skeleton. In contrast, when

Figure 3.3: A twisting motion is applied to the right leg bone to turn the knee (red circle) outwards. If we use non-volumetric bones, elements can twist around their bone-line and undo the transformation, as apparent on the left. Volumetric bones (right) naturally resolve this issue.

connecting the skin to a bone-line, as done for instance by Rumman [2014; 2015] and Capell [2002; 2005], a twisting of the bone-line is not noticed by connected elements, resulting in an unnatural deformation (see Figure 3.3), unless particularly prevented [Capell et al. 2005]. Second, effects like skin bulging and stretching at bent joints can be achieved in a natural manner, as described in Section 3.2.2.

Volumetric Tissue Mesh

The skin shrinking results in an inner skeleton surface $\mathcal{B}$ (for bones) and an outer skin surface $\mathcal{S}$ with identical vertex connectivity. The volumetric skeleton is wrapped by $\mathcal{B}$ (building the skeleton layer) and the soft tissue layer is enclosed between $\mathcal{B}$ and $\mathcal{S}$. Each outer triangle $(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k)$ on the skin surface together with its corresponding inner copy $(\mathbf{x}_i^{\mathcal{B}}, \mathbf{x}_j^{\mathcal{B}}, \mathbf{x}_k^{\mathcal{B}})$ spans a volumetric element, which we call *prism* with slight abuse of notation. Each prism can be split into three tetrahedra (Figure 3.4, top). The lateral surface of a prism consists of three quads. In the general case of non-planar quads, their segmentation into triangles can be accomplished in two ways, each resulting in a different surface. Therefore, we cannot tetrahedralize each prism individually, but must instead split the joined quad of neighboring prisms in the same direction. Ignoring this can lead to artificial gaps or overlaps in the terahedral mesh. Avoiding gaps is especially important for collision detection, which would fail at holes in the volumetric mesh. For this purpose, we must find a valid splitting configuration by defining splitting directions. We indicate these by arrows on the skin surface (one per edge). Each arrow points to the vertex connected to the quad's splitting diagonal (Figure 3.4, bottom left). Our goal is now to find a valid arrow-configuration. Due to the symmetries in the prisms' structure, the only two *invalid* states (per prism) are pure clock-wise (cw) or pure counter-clock-wise (ccw) orderings of the arrows. Since six of the eight configurations for a triangle are valid and each invalid configuration can be turned into a valid one by simply flipping one edge direction, there is an efficient solution to this problem: We start by choosing a random direction per

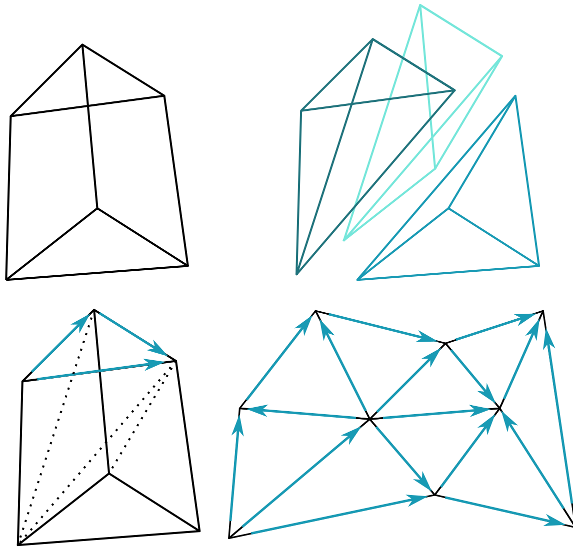


Figure 3.4: A volumetric prism element is spanned by each triangle of $\mathcal{S}$ and the corresponding triangle of $\mathcal{B}$ (top left). Each of these prisms can be split into three tetrahedra (top right). When dividing the prisms, we must split each pair of connected lateral surface quads in the same way. We indicate this splitting direction (dashed line) by an arrow on the top triangle (bottom left). Starting with random directions, we iteratively rearrange the arrows until we find a valid splitting configuration (bottom right).

edge. As long as there is an invalid (cw or ccw) triangle state, we randomly flip one of its edge directions until the complete mesh is in a valid state (Figure 3.4, bottom right). Guided by the resulting arrow configuration, we can compute a tetrahedral mesh without overlaps or holes. For a surface mesh with F triangular faces, this leads to a volumetric mesh with $2S$ vertices, F prism elements, and $T = 3F$ tetrahedra.

3.2.2 Coupling of Skeleton and Skin

The generated two-layered volumetric models are now ready to be animated through a Projective Dynamics simulation (see Section 2.2). This can be broken down into three simple steps:

- First, given the joint transformations of an animation frame, we rigidly transform the volumetric bones and joints of the skeleton.
- Second, the surface $\mathcal{B}$ is moved along with the new skeletal pose by attaching its vertices $\mathbf{x}_i^{\mathcal{B}}$ to the volumetric skeleton via anchor constraints. As shown in Section 2.2.3, we can also treat these as hard constraints.
- Third, the movement of the skeleton surface results in a deformation of the volumetric elements in the soft tissue layer. By penalizing this deformation via strain constraints, the skeletal motion finally results in the motion of the skin.

While the anchoring of the second step is straightforward for vertices located on cylindrical *bones*, vertices on spherical *joints* require special treatment to achieve realistic skinning results. When anchoring vertices of the skeleton

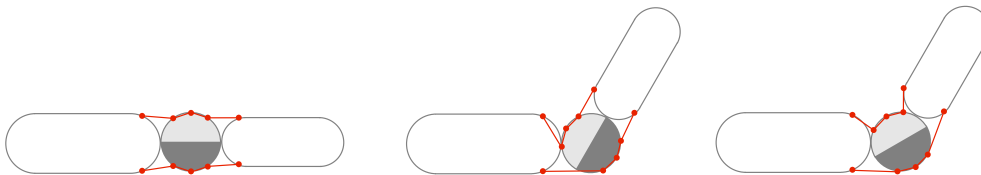


Figure 3.5: When joints rotate with the full transformation of their child bone, the shrunken skin (red vertices) experiences large stretching on one side (bottom left). Rotating the joint by the average of the incident bones' transformations avoids this problem (bottom right).

surface $\mathbf{x}_i^{\mathcal{B}}$ at spherical joints, the joint transformation should be chosen as the (quaternion-blended) average of the two incident bones' transformation, instead of the full transformation of the child bone. Otherwise large stretchings occur at one side of the joint that lead to artifacts in the skin deformation, as depicted in Figure 3.5.

However, a rigid motion just poorly mimics the natural behavior. The bending increases the area on one side of the joint. We would therefore expect that the skeleton surface stretches on this side and compresses on the other one. Furthermore, the stretching should affect a larger region around the joint. In a real body, muscles are attached to the bones via tendons and they compress or stretch to induce a bending of the joint. We want to inversely model this by compressing and stretching the soft tissue mesh at a bent joint.

To achieve this, we define virtual anchor points on the child and parent bones as well as on the joint (2D example in Figure 3.6, blue dots). In 3D, those anchor points are determined as rings of evenly sampled vertices around the perimeter of the bone/joint. Every vertex $\mathbf{x}_i^{\mathcal{B}}$ in between the bone and joint anchors (red points in Figure 3.6) is represented as a linear combination of four anchor points (two on the nearest bone and two on the joint). These anchors are rigidly transformed along with bones and the half-rotated joints, and their assigned vertices $\mathbf{x}_i^{\mathcal{B}}$ are transformed by re-applying the linear combination. The interpolated position can penetrate the volumetric skeleton, therefore we

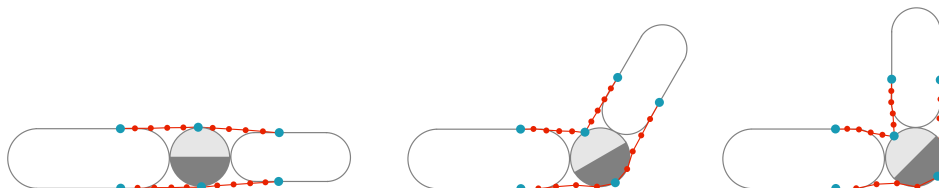


Figure 3.6: Vertices of the shrunken skin in the vicinity of joints (red dots) are transformed by linear interpolation of anchor points on the skeleton (blue dots), followed by a projection onto the volumetric bones/joints. In that way, the outer part is stretched, while the inner part is compressed.

Figure 3.7: Comparison of sticking vertices of the shrunken skin to bones and joints (left) and our skin sliding (right).

additionally project it onto the surface of the volumetric bones and joints. This nicely mimics the desired skin sliding behavior around joints and also gives better results for clothed models (see Figure 3.7).

3.2.3 GPU-Based Projective Skinning

In this section we will derive an efficient parallel GPU implementation of our skinning approach. For this purpose we summarize all steps contributing to one rendered frame of a skinned character in Algorithm 3.1, and note which elements can be processed in parallel for each step. As motivated in Section 2.3, data transfers between CPU and GPU should be minimized. Therefore, we move as many of the steps to the GPU as possible. Most of them are easy

Algorithm 3.1: Per frame FPS algorithm

```
 1: update transformations          # per joint
 2: update base points              # per vertex
 3: update PD momentum              # per vertex
 4: for N_pd iterations do
 5:     PD local                    # per constraint
 6:     PD global                   # per coordinate
 7: end for
 8: update PD velocity              # per vertex
 9: update face normals             # per face
10: update vertex normals           # per vertex
11: render frame
```

to parallelize: The updates of lines 2, 3, 8, 9 and 10 can be processed with one thread per element (vertex/face). The local step (line 5) can be computed using one thread per projection. Note that there are hybrids of OpenGL vertex array objects and CUDA arrays, which allow to directly render the kernels' results (line 11) without additional data transfers.

The remaining steps involve updating all joint transformations based on the current pose and solving the global linear system. The computation of global transformations $\mathbf{G}_k$ is highly recursive (see Equation (2.2)) and therefore hard to parallelize efficiently. Moreover, the low number of joints (typically less than a hundred) is further limiting the potential for parallelization. Thus, evaluating the $\mathbf{T}_k$ on the host and transferring them to the device has shown to be faster than a slightly parallel GPU implementation. The last critical point left is the global step (line 6), which we discuss in the following.

GPU Conjugate Gradients

The global step of Projective Dynamics involved in our skinning simulation requires solving a linear system (see Equation (2.13)). The system matrix $\mathbf{A} = \delta t^{-2}\mathbf{M} + \mathbf{Q}^{\mathsf{T}}\mathbf{W}\mathbf{Q}$ is sparse, symmetric, positive definite and *constant*, such that its sparse Cholesky factorization can be pre-computed. Moreover, we can solve for the three spatial coordinates in parallel, leading to an efficient global update on multi-core CPUs. This situation changes considerably if the matrix has to be updated frequently (e.g., due to collisions) and thus needs to be re-factorized or if the method is to be implemented on the GPU.

The forward and backward substitutions involved in the two triangular systems of a Cholesky solver are inherently sequential algorithms. While there exist some ideas for GPU parallelization [Naumov 2011; Liu et al. 2016], solving medium-sized, sparse, triangular systems is still faster on the CPU. To avoid the computational bottleneck, we employ a preconditioned conjugate gradients (PCG) solver instead. This iterative solver consists of a matrix-vector product and three dot products per iteration, which both can easily be parallelized. Furthermore, the solution of the previous FPS time-step can be used as an initial guess in the iterative solver, reducing the number of required iterations. We aim for a GPU-based PCG solver that is faster than the CPU-based multi-threaded pre-factorized sparse Cholesky solver of Eigen [Guennebaud et al. 2018], which is faster than Eigen's CPU PCG solver.

There already exist several implementations of general PCG solvers on GPUs [Bolz et al. 2003; Buatois et al. 2009]. CuSPARSE is part of the CUDA toolkit and provides all functions for building a GPU-based PCG solver, i.e., sparse matrix-vector multiplication, vector-vector multiplication, and vector addition. Unfortunately, this straightforward approach results in many CUDA kernel calls, which due to their computational overhead decreases performance. A considerably faster approach was proposed by Weber et al. [2013],

Algorithm 3.2: Solve $\mathbf{Ax} = \mathbf{b}$ with PCG using preconditioner $\mathbf{J}$

Input: *Initialize* $\mathbf{x}$ *with solution of previous time-step*

1: $\mathbf{r} \leftarrow \mathbf{b} - \mathbf{Ax}$
2: $\mathbf{d} \leftarrow \mathbf{Jr}$
3: $\gamma \leftarrow \mathbf{d}^\mathsf{T}\mathbf{r}$ } *Init-Kernel*
4: **for** N_{pcg} iterations **do**
5: $\eta \leftarrow \gamma$
6: $\mathbf{q} \leftarrow \mathbf{Ad}$
7: $\nu \leftarrow \mathbf{d}^\mathsf{T}\mathbf{q}$ } *Kernel 1*
8: $\nu \leftarrow \eta / \nu$
9: $\mathbf{x} \leftarrow \mathbf{x} + \nu\mathbf{d}$
10: $\mathbf{r} \leftarrow \mathbf{r} - \nu\mathbf{q}$
11: $\gamma \leftarrow \mathbf{r}^\mathsf{T}\mathbf{Jr}$ } *Kernel 2*
12: $\tau \leftarrow \gamma / \eta$
13: $\mathbf{d} \leftarrow \mathbf{r} + \tau\mathbf{d}$ } *Kernel 3*
14: **end for**

which uses just one initialization kernel and three kernels per PCG iteration. MAGMA [Anzt et al. 2014] is a linear algebra library that also provides a PCG solver with minimized kernel invocations. However, being optimized for *large-scale* linear systems, MAGMA is about two times slower than a CPU-based sparse Cholesky solver for our medium-sized matrices of a few thousand unknowns.

We therefore developed our own CUDA-based PCG solver that minimizes the number of kernel invocations and employs a special matrix format to optimized coalesced data access (as motivated in Section 2.3). The algorithm uses an initialization kernel and three kernels per PCG iteration similar to the approach of Weber et al. [2013], as shown in Algorithm 3.2. We use a Jacobi preconditioner $\mathbf{J}$, which is simply the inverse of the diagonal part of $\mathbf{A}$. Note that a further reduction to fewer kernels is not advisable, since all threads have to be synchronized after each inner product and a global thread synchronization within kernels is only supported by CUDA on very modern GPUs.

Matrix and Vector Storage

As explained in Section 2.3, the performance of modern graphics hardware is limited by memory bandwidth rather than by computing operations. Optimizing memory access is therefore the most important factor when aiming for

optimal performance. Read and write operations to global GPU (device) memory should be avoided where possible. The remaining accesses should be done in a *coalesced* way (see Figure 2.8). The most expensive step in Algorithm 3.2 is the sparse matrix-vector multiplication, where sparse matrix formats can reduce memory consumption, memory accesses, and computing operations.

The performance of the multiplication relies heavily on the way we store the sparse matrix. It is important to reduce the required number of read operations by either directly choosing a format that takes less memory or by coalescing the data-accesses. The compressed row storage (CRS) matrix format matches the row-wise access pattern of PCG and hence is frequently employed. It stores the non-zero matrix entries in a row-wise manner as an array of values, an array of corresponding column indices, and a third one containing the number of non-zeros per row. For CRS-based matrix-vector multiplication, a thread operates on a complete matrix row and thereby, each thread accesses a different number of elements in an uncoalesced way (see Figure 3.8, left). In order to achieve a constant number of elements per row, the ELLPACK (ELL) matrix format uses a per-row padding with zero elements. Additionally, the entries are stored in a column-wise manner resulting in a coalesced access (see Figure 3.8, middle). Here it is important to know how many non-zero elements per row typically occur in our FPS system matrix. This depends on the number of edges incident to a vertex, which in turn depends on its valence in the skin mesh as well as on the tetrahedralization of its incident tissue prisms. For a typical character mesh, this results in about 4–19 non-zeros per row. Using the ELL format, we would fill-up each row with zeros to get 19 elements, leading to many unnecessary data accesses.

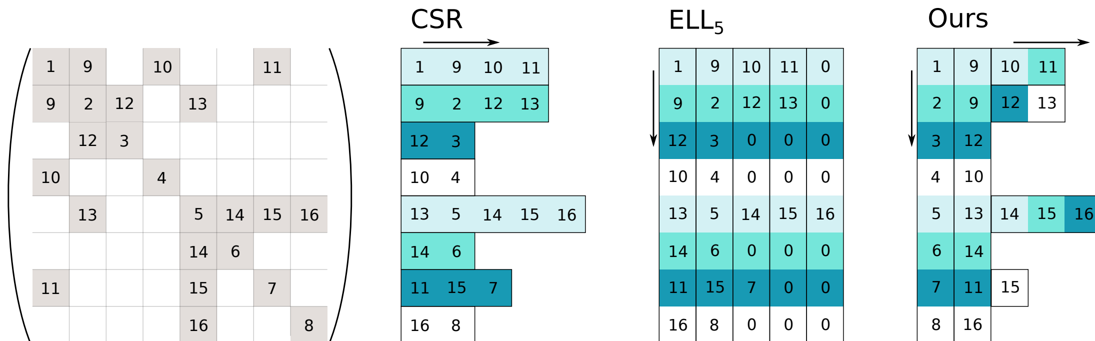


Figure 3.8: Access patterns of different matrix formats demonstrated with a simple example using 4 threads. In CRS sparse matrix format, data is stored row-wise. If we use one thread per matrix row, the access pattern will be uncoalesced (we use a different color per accessing thread). The ELL format uses zero padding, and column-wise storage to support coalesced access. Our format is a combination of both formats to reduce zero padding while still guaranteeing coalesced access. Furthermore, we store diagonal elements first since these are needed for our preconditioner.

Bell and Garland [2008] solve this problem by combining the ELL format with the COO (triplet) format, but their approach needs two kernels for one matrix-vector multiplication. Guo et al. [2016] use a hybrid CRS/ELL format also using a separate kernel for each part. Weber et al. [2013] instead employ the zero-padding for each thread block individually by computing the maximum number of non-zeros per row for each thread block. There are a lot of other proposed formats that optimize the ELL or CRS formats, like ELL-R [Vázquez et al. 2009], sliced-ELL [Monakov et al. 2010], as well as BCRS [Buatois et al. 2009], CRS-T [Yoshizawa and Takahashi 2012], and CRS SIC [Feng et al. 2011]. In all these approaches, performance benefits come at the price of additional data that has to be stored, the need to reorder matrix rows, or the usage of more than one thread per row or multiple kernels per matrix-vector product.

We propose a combination of ELL and CRS format that requires no additional zeros and supports a completely coalesced global memory access. The key idea is to utilize shared memory which can be accessed by all threads of a thread block and provides faster access than to global memory. Each row is processed by one thread. We use the ELL format for the first n_{ell} row entries, set to the minimum number of non-zeros per row, and store the remaining row elements in CRS format. These are read using a coalesced access pattern, multiplied by the vector element, and stored in shared memory. This enables us to employ the complete thread block for reading the data of all corresponding rows instead of just one thread per row (see Figure 3.8 right). The thread block processes as many rows as there are threads per block. While the number of values per row can differ largely between consecutive rows, this variance is much smaller for blocks of rows. After synchronizing the thread block, the products are read, summed up, and stored in global memory by the thread that is processing the corresponding row. The overhead produced by the usage of shared memory is compensated for by coalescing and distributing the global reads more equally. The major difference to the hybrid formats of Bell et al. [2008] and Guo et al. [2016] is that we first read the complete CRS part of a block into shared memory before adding it to the result instead of doing this in smaller chunks, allowing us to perform the complete matrix-vector multiplication in a single kernel.

So far, we focused on access patterns for the matrix $\mathbf{A}$ but not for the vector $\mathbf{x}$. Considering that the vector access patterns are hard to predict and exploiting that the vector is accessed read-only, we bind $\mathbf{x}$ to texture memory for the multiplication, as its cache accelerates random accesses. The three different spatial coordinates (columns of $\mathbf{x}$) could be processed separately, leading to $3N$ threads that could work in parallel to compute the N-dimensional matrix-vector product. However, in that case the same matrix elements have to be read three times, which for our matrix dimensions is slower than processing the three coordinates in a single thread.

Table 3.1: Timings (in µs) for sparse matrix-vector multiplication $\mathbf{Ax}$ using different matrix formats. $\mathbf{A}$ is the FPS skinning Matrix of a high-resolution character mesh (30k simulated vertices). The last four cases use texture memory to speed up the random access into $\mathbf{x}$. The CPU version is implemented with Eigen and parallelized using OpenMP. In the 3D case, we process all three spatial xyz-coordinates of $\mathbf{x}$ by one single thread. Hardware specifications can be found in Section 3.3.

	1D		3D	
Method	timing	speedup	timing	speedup
CPU	382	1	503	1
cuSPARSE	9.1	42	12.6	40
CRS	6.9	55	10.6	47
ELL	12.7	30	14.5	35
CRS + ELL	5.5	69	9.1	55
Ours	4.2	91	7.7	65

A performance comparison for the PCG matrix-vector multiplication is shown in Table 3.1. In the following we analyze the impact of the different optimizations. First, using the combined CRS/ELL format and handling spatial coordinates separately, the matrix-vector product $\mathbf{Ax}$ takes 18.7 µs. Handling all three coordinates in one thread then improves performance to 11.6 µs. Using texture memory to store the right-hand side yields 9.1 µs (corresponding to the 'CRS + ELL' entry), and using shared memory to distribute/coalesce the operations on the CRS part finally results in the 7.7 µs reported in Table 3.1.

A drawback of our format is that we have to allocate a high amount of shared memory to store all CRS-values of the thread block. Since we cannot specify an individual amount of shared memory for each block, we have to allocate the maximum of all blocks. Shared memory on our device (GTX 2080 TI, compute capability 7.5) can use up to 64 kB per block and the same amount per multiprocessor. Each multiprocessor can run up to 16 blocks with a maximum total of 1024 threads in parallel. Hence, even if we reduce the block-size, there is a limitation for the size of the CRS part of 16 non-zeros ($64\,\text{kB}/(4\,\text{B} \cdot 1024)$) per row on average in the CRS matrix. If we exceed this limit, less rows can be processed in parallel or we have to extend the ELL part of the matrix, which leads to additional zero-padding. This can be observed in Table 3.1 for the 3D timings. Here, we have to allocate three times as much shared memory, which leads to a smaller performance gain compared to the 1D case. Note that for small matrices, like we typically use for our character skinning (about 4k rows), the matrix-vector multiplication is dominated by the overhead of calling the kernel, and the performance difference between

the individual formats becomes very small. Using only one kernel is thus the most important factor and disqualifies other, more sophisticated matrix formats for our application. Our format turned out to be a good trade-off between performance, memory consumption, and simplicity.

For scalar products we use a standard approach. Each multiplication is processed by a different thread and the result is stored in shared memory. After a block-wide synchronization, the first thread of each block does the summation of the block's elements and adds the result to global memory by an atomic operation.

Like we already explained in Section 2.3, we can use shared memory to maximize the number of coalesced accesses. Besides our approach for matrix-vector multiplication, another important example of this is the local step, where we store and read 3×3 blocks of our $3 \times P$ matrix $\mathbf{P}$, holding the optimal rotations. Both column and row-major storage of $\mathbf{P}$ is not optimal in this case. We need to transpose the data to a $9 \times P/3$ format to maximize coalescing. On the other hand, $\mathbf{P}$ is part of the right-hand-side term of the global step, for which the original $3 \times P$ format is optimal. Here, we use shared memory to rearrange the data (see Section 2.3) as well as in many similar cases. For further details, we refer to our freely-available implementation of Fast Projective Skinning on GitHub.

3.2.4 Upsampling

Like many physics-based simulations, Fast Projective Skinning does not use the full high-resolution mesh for simulation. Instead, the simulation is performed on a coarse-resolution *simulation mesh* and then propagated or 'upsampled' to the original high-resolution *visualization mesh* (see Figure 3.9 and Video V.2). If we use a proper upsampling approach, the resulting deformation should be of similar quality as the result of a full-resolution simulation. To explain

Figure 3.9: FPS-simulations on a coarser version of the input mesh (left) allow for a much faster skinning (middle). Our highly efficient upsampling approach now transfers the fine-scale details from the detailed visualization mesh to the coarse posed result (right).

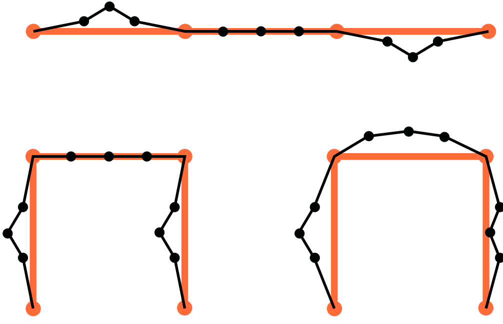


Figure 3.10: When upsampling the red mesh to the black one using precomputed normal displacements of the undeformed meshes (top), each vertex only takes the transformation of its assigned red segment into account (bottom left). Our MLS upsampling estimates deformations using multiple segments leading to smoother results (bottom right).

our upsampling strategies, we will use the index i for properties of the coarse simulation mesh (vertices $\mathbf{x}_i$, normals $\mathbf{n}_i$) and the index j for properties of the high-resolution visualization mesh.

For that purpose, we first tried using normal displacements [Botsch and Sorkine 2008]. Since we generate our decimated meshes by applying successive halfedge collapses on the original mesh using a quadric error metric [Garland and Heckbert 1997], each vertex $\mathbf{x}_j$ of the original fine-scale mesh is either at the same location as a vertex of the coarse mesh $\mathbf{x}_i$, or can be assigned to its closest coarse triangle. In rest-pose, we project $\mathbf{x}_j$ onto this triangle and store the projection via barycentric coordinates as well as the distance from the projection to $\mathbf{x}_j$. In order to compute the high-resolution result in a deformed pose, we re-apply the barycentric coordinates and normal displacements to the skinned coarse mesh. In regions where the deformation is close to a pure rotation and translation, this perfectly reproduces fine-scale details. However, using normal displacements can lead to artifacts in regions of strong bending, where normal displacements do not recover a smooth high-resolution mesh. This problem is sketched in Figure 3.10 and shown for the shoulder region in Figure 3.11d.

Inspired by Martin and colleagues [2010] we instead employ a moving-least-squares (MLS) approach to upsample deformations. We will explain our MLS-based upsampling in the following but refer to Fries and Matthies [2004] for more details on MLS interpolation in general.

Upsampling approaches operate on vertex displacements $\mathbf{u}(\mathbf{x}) = \mathbf{x} - \bar{\mathbf{x}}$, where the bar notation is again used to distinguish the rest pose position from the current one. The displacements of simulated vertices $\mathbf{u}_i = \mathbf{u}(\mathbf{x}_i)$ are known and can be upsampled to $\mathbf{u}(\mathbf{x}_j)$ by fitting a local affine transformation $\mathbf{A}_j \in \mathbb{R}^{3\times 4}$ through weighted least-squares minimization

$$\min_{\mathbf{A}_j} \sum_i \omega\left(\left\|\bar{\mathbf{x}}_j - \bar{\mathbf{x}}_i\right\|\right) \left\|\mathbf{A}_j \boldsymbol{\psi}(\bar{\mathbf{x}}_i) - \mathbf{u}_i\right\|^2, \tag{3.2}$$

where $\boldsymbol{\psi}(\mathbf{x}) = (1, x, y, z)^\mathsf{T}$ transforms a position $\mathbf{x} = (x, y, z)^\mathsf{T}$ into a vector of

linear monomials and $\omega : [0, \infty) \rightarrow [0, 1]$ is a smooth weighting function

$$\omega(r) = \begin{cases} (1 - r/\rho)^3 & r < \rho \\ 0 & \text{otherwise,} \end{cases} \tag{3.3}$$

with ρ defining the support radius of the MLS kernels. For short notation we define $\omega_{ij} = \omega(\|\bar{\mathbf{x}}_j - \bar{\mathbf{x}}_i\|)$ and $\boldsymbol{\psi}_k = \boldsymbol{\psi}(\bar{\mathbf{x}}_k),\ k \in \{i, j\}$. Minimizing (3.2) with respect to $\mathbf{A}_j$ results in

$$\mathbf{A}_j = \sum_i \omega_{ij} \mathbf{u}_i \boldsymbol{\psi}_i^\mathsf{T} \mathbf{K}_j^{-1} \quad \text{with} \tag{3.4}$$

$$\mathbf{K}_j = \sum_i \omega_{ij} \boldsymbol{\psi}_i \boldsymbol{\psi}_i^\mathsf{T}. \tag{3.5}$$

Applying the transformations $\mathbf{A}_j$ to the high-resolution vertex monomials results in the high-resolution displacements

$$\mathbf{u}_j = \mathbf{A}_j \boldsymbol{\psi}_j = \sum_i \omega_{ij} \mathbf{u}_i \boldsymbol{\psi}_i^\mathsf{T} \mathbf{K}_j^{-1} \boldsymbol{\psi}_j. \tag{3.6}$$

Realizing that everything but $\mathbf{u}_i$ only depends on rest-pose properties, we find

$$\mathbf{u}_j = \sum_i \mathbf{u}_i N_{ij} \quad \text{with} \tag{3.7}$$

$$N_{ij} = \omega_{ij} \boldsymbol{\psi}_i^\mathsf{T} \mathbf{K}_j^{-1} \boldsymbol{\psi}_j, \tag{3.8}$$

where the N_{ij} can be pre-computed. This results in a simple and perfectly parallel update rule (3.7) for the visualization vertices $\mathbf{x}_j$.

We deviate from the standard MLS interpolation in three ways. First, using a Euclidean distance metric for the weight function requires special treatment when surface parts (e.g., inner thighs) come close [Martin et al. 2010]. We therefore employ the *geodesic distance* w.r.t. the skin surface, computed through the approach of Kimmel and Sethian [1998].

Second, we switch from linear to quadratic polynomials, i.e.,

$$\boldsymbol{\psi}(\mathbf{x}) = (1, x, y, z, xx, yy, zz, xy, xz, yz)^\mathsf{T}.$$

This allows us to locally reproduce linear and quadratic transformation, which yields more faithful reconstructions in regions of strong bending (see Figure 3.11).

Third, in degenerate situations (Figure 3.11c) where $\mathbf{K}$ is not invertible (when the points $\bar{\mathbf{x}}_i$ lie on a plane or quadric) the system can either be under-constrained ($\mathbf{x}_j$ lies on the same plane or quadric) or has no solution. The first case can be solved by replacing the inverse $\mathbf{K}^{-1}$ by the Moore-Penrose pseudo-inverse $\mathbf{G}^+$. We further enhance the numerical robustness by mean-centering and scaling the points $\bar{\mathbf{x}}_i$ within the support radius ρ (as well as the point of evaluation $\mathbf{x}_j$) to the unit sphere. The second case is very rare

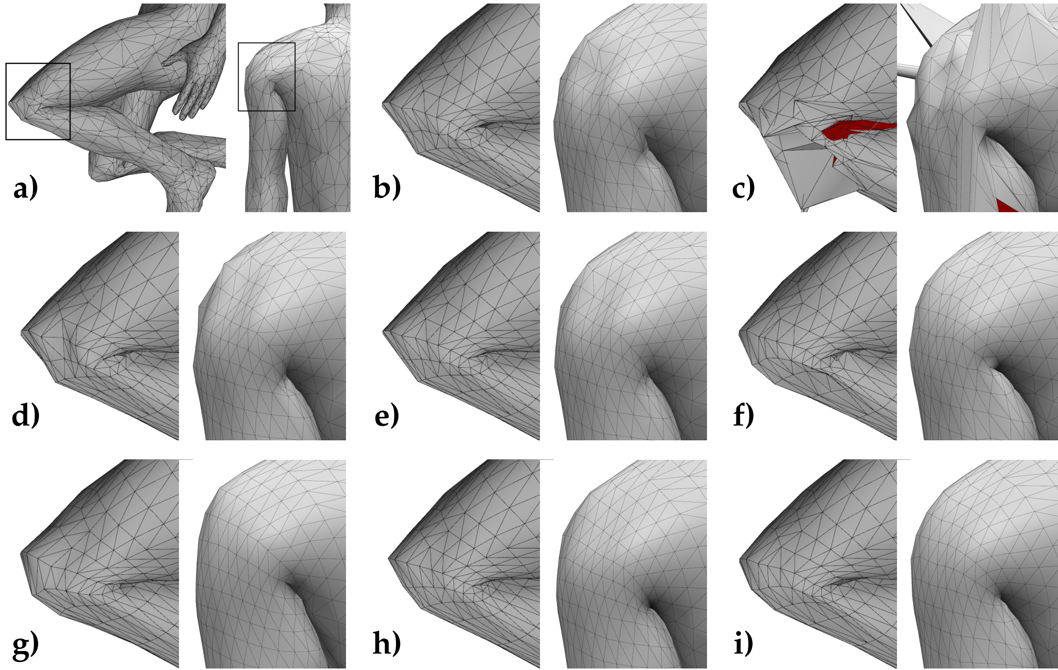


Figure 3.11: Different examples of our upsampling. Linear (2nd column) and quadratic (3rd column) MLS upsampling using $k = 8$ (b),(c), 16 (e),(f) and 32 (h),(i) vertices of the simulated low-resolution mesh (a). For small k-values, matrix $\mathbf{K}$ (3.5) can become ill conditioned (see (c)). Using a quadratic approximation leads to smoother result and less distortion of the simulated vertices. Normal displacements (d) are not able to produce a transition from a straight to a curved region. (g) shows the result of a direct simulation of the high-resolution mesh. This example corresponds to the model 'human low' in Table 3.2.

but can be provoked by using a unreasonably small support radius ρ and meshes with many regions of perfect planes or quadrics. In those cases, we fall back to the linear or constant representation of the $\boldsymbol{\psi}(\mathbf{x})$ at the price of sacrificing quadratic or linear precision. Since our upsampling is very efficient, increasing the size of ρ is usually the best option to solve these issues while the introduced overhead is negligible.

Due to the partition of unity and linear reproduction of MLS shape functions [Fries and Matthies 2004], the MLS upsampling weights N_{ij} perfectly reproduce the initial mesh $\bar{\mathbf{x}}_j = \sum_i \bar{\mathbf{x}}_i N_{ij}$. Similarly, we can derive that there is no need to compute the displacement field $\mathbf{u}(\mathbf{x})$. We can instead directly use vertex positions since

$$\mathbf{x}_j = \bar{\mathbf{x}}_j + \mathbf{u}_j = \bar{\mathbf{x}}_j + \sum_i \underbrace{(\mathbf{x}_i - \bar{\mathbf{x}}_i)}_{=\mathbf{u}_i} N_{ij} = \sum_i \mathbf{x}_i N_{ij} + \underbrace{\left(\bar{\mathbf{x}}_j - \sum_i \bar{\mathbf{x}}_i N_{ij}\right)}_{=\mathbf{0}}. \qquad (3.9)$$

The implementation of this upsampling approach in CUDA is simple. Working with positions (3.9) instead of displacements (3.7) is a massive reduction of

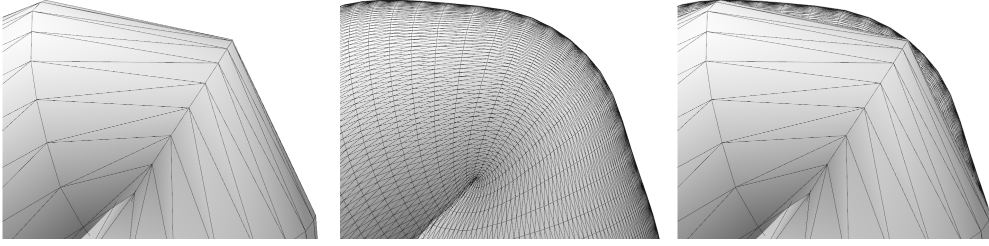


Figure 3.12: Our MLS-upsampling generates smooth deformations (middle) even in case of simulation meshes of very low-resolution (left). If we use quadratic polynomials to approximate deformations, the upsampling does not result in severe volume losses (right).

memory accesses. To simplify the kernel even more, we use a fixed number k of nearest neighbors $\mathbf{x}_i$ for each $\mathbf{x}_j$ (we found $k = 20$ to be sufficient). The radius ρ in (3.3) is set to the distance of the $(k+1)$st nearest neighbor for each vertex individually. Two neighbor indices can be fused into one 4-byte-integer as long as the simulation mesh has less than 2^{16} vertices to reduce memory reads even more. Additionally, the access to both indices and N_{ij} can be performed in a coalesced way. Reading the low-resolution positions is highly unordered and can thus be accelerated by using texture memory (see Section 2.3). Figure 3.11 compares quadratic and linear MLS upsampling with different values of k. Increasing k produces smoother results but in case of linear MLS it also leads to over-smoothing near joints, whereas quadratic MLS can better reproduce the deformation. In Figure 3.12 we use a very high-resolution visualization mesh to demonstrate that our MLS-upsampling converges to a smooth deformation even if the coarse deformation has very few faces in the deforming region.

Updating vertex normals of the deformed visualization mesh, typically as a weighted average of incident triangles' normals, is another computational bottleneck. We extend the approach to use MLS upsampling for normal vectors, simply by substituting $\boldsymbol{\psi}(\mathbf{x})$ with $\boldsymbol{\psi}(\mathbf{n})$ in (3.5) and (3.8). This results in different weights $\widetilde{N}_{ij}$ through which we can compute normals as

$$\mathbf{n}_j = \left(\sum_i \mathbf{n}_i \widetilde{N}_{ij} \right) \Big/ \left\| \sum_i \mathbf{n}_i \widetilde{N}_{ij} \right\|.$$

Our experiments revealed linear MLS to be sufficient for normal interpolation. The resulting interpolated normals differ slightly from re-computed normals in non-rigidly deformed regions, but the difference is visually negligible.

The MLS-based upsampling of vertex positions and normals produces a computational overhead of just 1–2% when using five times more vertices in the visualization mesh and $k = 20$ MLS neighbors. On the CPU, this upsampling approach is about twice as fast as using normal displacements. The GPU implementation of MLS upsampling is another ten times faster than its CPU version.

3.2.5 Collision Handling

Proper collision handling greatly improves the realism of any skinning method. It is commonly done in two steps: (i) detecting colliding vertices and (ii) moving these vertices to resolve the collisions. Despite the large body of research on fast collision detection and response, collision handling is still a very time consuming part of soft body simulations due to the high quantity of primitives that must be tested for collisions.

On the CPU, where the skinning simulation already takes up a large portion of the frame time budget, our idea was to pre-compute a conservative set of potentially colliding vertices. However, human anatomy provides way too many possibilities for collisions, which would result in a prohibitively large set of potential collisions pairs. In order to still enable real-time animations on the CPU, we distinguish two types of collisions: *local* collisions occurring in proximity of joints (e.g., elbow or knee region) and *global* collisions (e.g., hand touches belly). In contrast to the high quantity of possible global self-collisions, there are just a few important regions in which local collisions occur. Moreover, for a local collision at a specific joint, the same two skin patches collide each time (e.g., a part of the upper leg and one of the lower leg at the knee). In a pre-processing step, we can therefore find and store these potential collision pairs, consisting of one vertex from each patch, by simulating a set of extreme skeleton postures. Thereby, local collisions can efficiently be tested and resolved at run-time. For the latter we propose a method to enable/disable collision constraints in the PD simulation while avoiding the costly re-factorization of the global system matrix. This CPU-based local collision handling is explained in the first part of this section.

On the GPU however, the high efficiency of our Fast Projective Skinning simulation allows us to employ a full global collision handling instead. Furthermore, when using our iterative PCG solver, updating the global system matrix does no longer require a costly re-factorization but just a simple update of the system matrix. This global collision handling approach is detailed in the second half of this section.

In both cases, collision detection is done on the CPU using point-in-tetrahedron tests accelerated by spatial hashing [Teschner et al. 2003]. If we used the tetrahedra of our volumetric model, all collisions inside of the skeleton surface $\mathcal{B}$ would not be detected. To solve this issue, we construct a different set of tetrahedra between the skin surface $\mathcal{S}$ and the already computed base points $\mathbf{s}_i$ on the bone-lines. Thereby, the complete interior of the animated character is covered by tetrahedra and no collision will be missed. Furthermore, the $\mathbf{s}_i$ of many triangles lie on the same line, causing one of the three prism's tetrahedra to degenerate. We detect and exclude these from the point-in-tetrahedron tests, leading to an overall faster collision detection.

CPU-Based Local Collision Handling

In order to collect potential collision pairs, we simulate a set of extreme poses that have been manually designed to provoke different types of collisions in the regions of interest (elbows, shoulders, hip/crotch, knees). During these (offline) simulations we use standard collision detection as explained above.

Each of these individual simulations has two responsible bones and one joint (e.g., upper arm, lower arm, elbow joint). We denote the normalized vectors in directions of the bones starting from the joint by $\mathbf{b}_i$, $\mathbf{b}_j$ and the joint's position by $\mathbf{j}_{ij}$ (see Figure 3.13). We divide all colliding vertices into two groups $\mathcal{C}_1$, $\mathcal{C}_2$ corresponding to the two colliding skin patches. To this end, we first find the colliding skin vertex $\mathbf{x}_c$ that is closest to $\mathbf{j}_{ij}$. We compute a normal $\mathbf{n}_{ij} = (\mathbf{x}_c - \mathbf{j}_{ij}) \times (\mathbf{b}_i \times \mathbf{b}_j)$ and split the colliding vertices into $\mathcal{C}_1$ and $\mathcal{C}_2$ using the plane defined by $\mathbf{j}_{ij}$ and $\mathbf{n}_{ij}$. Note that the vertex $\mathbf{x}_c$ is identified in the extreme colliding pose, but the actual splitting is performed more robustly in a collision-free posture, typically the rest pose. For the two patches $\mathcal{C}_1$ and $\mathcal{C}_2$ we determine collision pairs $(\mathbf{x}_i, \mathbf{x}_j)$ by finding for each vertex $\mathbf{x}_i \in \mathcal{C}_1$ the closest vertex $\mathbf{x}_j \in \mathcal{C}_2$, now again in the colliding pose.

During the real-time skinning simulation, we test all stored potential collision pairs using a custom-tailored procedure aiming at high performance. Our approximate collision test for a specific potential collision pair entry $\{\mathbf{x}_i, \mathbf{x}_j, \mathbf{b}_i, \mathbf{b}_j, \mathbf{j}_{ij}\}$ works as follows. We project the points $\mathbf{x}_i$ and $\mathbf{x}_j$ into the plane spanned by $\mathbf{j}_{ij}$ and $\mathbf{b}_i, \mathbf{b}_j$, and denote the projecting function by $\pi_p(\mathbf{x})$. We report a collision based on the angles in the 2D planar configuration shown in Figure 3.13:

$$\delta c \;=\; \cos\alpha - \cos\beta \;=\; \frac{\mathbf{b}_i^\mathsf{T}\left(\pi_p(\mathbf{x}_i) - \mathbf{j}_{ij}\right)}{\left\|\pi_p(\mathbf{x}_i) - \mathbf{j}_{ij}\right\|} - \frac{\mathbf{b}_j^\mathsf{T}\left(\pi_p(\mathbf{x}_j) - \mathbf{j}_{ij}\right)}{\left\|\pi_p(\mathbf{x}_j) - \mathbf{j}_{ij}\right\|}.$$

If δc is negative, i.e., $\alpha > \beta$, we report a collision and setup an unilateral collision constraint as described below. Our approximate test might detect a collision in the projected 2D configuration where in 3D there is no collision. However, this false positive is not a problem, since it will be detected by

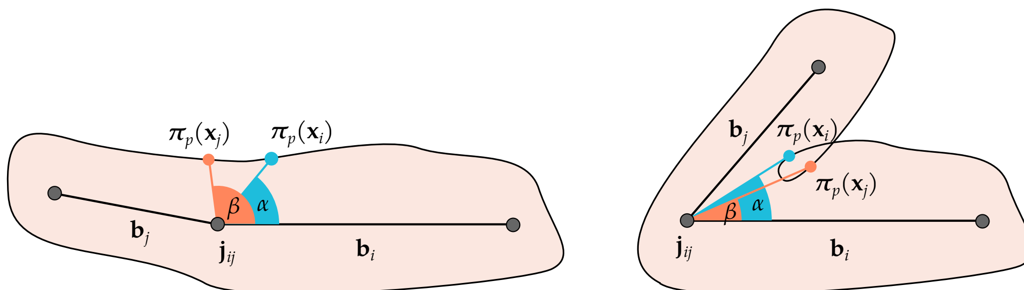


Figure 3.13: Our approximate collision detection reports a potential collision if in the projected configuration $\alpha > \beta$.

the unilateral collision constraint. The advantage of this approach is its high computational efficiency: the difference in frame-rates when adding our collision constraints to the simulation is just barely noticeable.

This test can also be generalized to situations where $\mathbf{b}_i$ and $\mathbf{b}_j$ are not incident, but connected by a chain of bones, as for instance the shoulder or hip/crotch region (see Figure 3.1). In such cases we choose the inner (w.r.t. the bone chain) endpoint of $\mathbf{b}_i$ as the joint $\mathbf{j}_{ij}$ and everything else stays the same.

A common approach for resolving collisions is to translate colliding vertices to the closest point on the object's surface. Bouaziz et al. [2014a] suggests applying unilateral constraints to colliding vertices, defined by the plane containing the closest face of the surface. This constraint projects the vertex onto the plane if it is below (i.e., colliding) and otherwise uses its current position as the local projection $\mathbf{p}_i$. In case of self-collisions, however, this projection is not the closest configuration in which the collision is resolved. Instead, it causes a gap between the two skin parts (see Figure 3.14, left). Moreover, the collision planes have to be updated due to the skin movement and another face might become the closest one to the colliding vertex.

In contrast, we aim at the minimal change in vertex positions to resolve the collision. For each collision pair $(\mathbf{x}_i, \mathbf{x}_j)$ reported by the collision detection we compute an "in-between plane", which is defined by the average position $(\mathbf{x}_i + \mathbf{x}_j)/2$ and the average normal $(\mathbf{n}_i - \mathbf{n}_j)/\left\|\mathbf{n}_i - \mathbf{n}_j\right\|$. For collision response, we simply project both vertices onto this plane if they are on the respectively wrong side. This approach leaves no gaps between the collision-resolved skin parts (see Figure 3.14, right) and it avoids the costly computation of the closest face for each colliding vertex. Note that this approach is just an approximation to an exact collision response, however, as shown in Figure 3.15, it is able to provide convincing results in real-time.

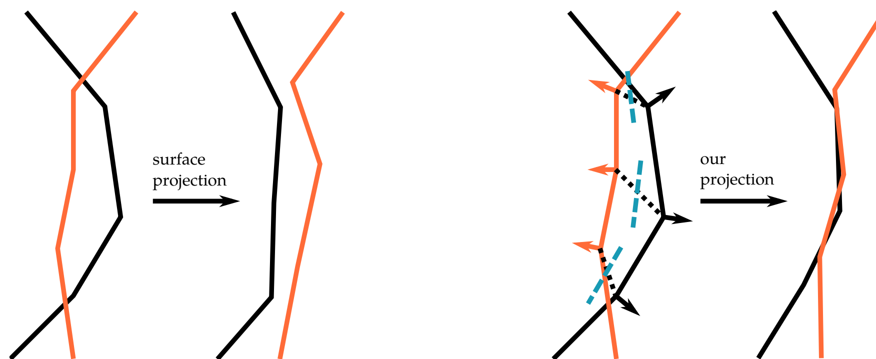


Figure 3.14: Common surface projection of colliding vertices imposes a gap after the projection step (left). Our collision response acts between a vertex pair and incorporates vertex normals. Colliding vertices are projected onto a mid-plane (blue dashed lines). While not resolving collisions perfectly, the result is visually plausible and fast, since it avoids the computation of a shortest mesh distance.

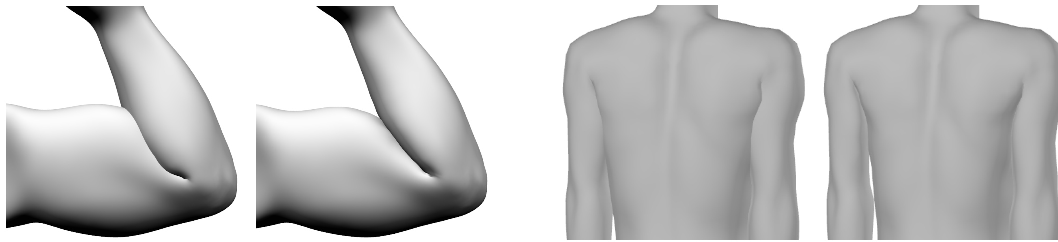


Figure 3.15: Comparison of simulations with our *local* collision constraints disabled (left) and enabled (right).

Collision constraints are in general unilateral constraints, meaning that the involved vertices will be projected only if a collision occurs. This behavior is difficult to implement efficiently in Projective Dynamics, since the global solve always requires one projection per constraint. Bouaziz et al. [2014a] propose to add a collision constraint to the solver when vertices collide and to remove it afterwards. However, this results in many re-factorizations of the global system matrix and would considerably slow down the CPU-simulation. Collisions could also be solved in a post-processing step, but that makes them invisible to surrounding vertices. For example, when one vertex is pushed in some direction because of a collision, neighboring vertices should also be affected due to elasticity. More sophisticated deformers for post-processing collision handling, like the one introduced by Brunel et al. [2021], are able to solve this issue but come with a computational overhead that prevent real-time skinning of a complete human character.

Other approaches always use an additional constraint per vertex that either projects the colliding vertices onto an intersection-free state or projects to the current position for non-colliding vertices [Lan et al. 2020]. However, the latter seemingly "do-nothing" constraint leads to artifacts: If other constraints, like the strain constraint, project to a different position, the collision-related "do-nothing" projection would prevent the involved vertices from following the strains' projection. The high weight typically used for collision constraints even emphasizes this effect. Increasing the iteration count of the alternating local/global steps reduces the artifacts (as done by Lan et al. [2020]) but slows down the simulation. Ichim et al. [2016] avoid these problems by using Lagrangian multipliers and the Schur complement to efficiently update the pre-computed factorization. They therefore just re-factorize a smaller matrix built from the colliding vertices which reduces the computational overhead when adding and removing collisions. However, this overhead can still become prohibitively large in situations with many collision constraints.

In our simulation we solve this problem by using two global matrices. The first one does not include collision constraints while the second one adds collision constraints for all potentially colliding vertex pairs. Assuming we would normally compute N_{pd} local/global PD iterations per frame, we now

perform $N_{\text{pd}}/2$ iterations with the first matrix to get a preliminary skinning result without collisions. Afterwards we do $N_{\text{pd}}/2$ iterations with the second global matrix to resolve possible collisions. This time, we have a good initialization for our "do-nothing" projection from the first solves without collisions. In that way, we can incorporate collisions in Projective Dynamics without the above-mentioned artifacts and without slowing down the simulation due to re-factorizations. For a full character with about 800 pre-computed collision pairs, the associated overhead is just about $2-4\%$. For comparison, re-factorizing the global matrix would result in an overhead of 300%. Figure 3.15 compares simulations with and without local collision handling. More examples can be found in Video V.3.

GPU-Based Global Collision Handling

While the method above works well for *local* collisions, extending it to *global* collisions is infeasible. By just considering all possible body-parts that can be reached with our hands, there are far to many possible collision scenarios that would have to be pre-computed. And even if we managed that, the number of potential collision pairs would lead to extremely many collision constraints.

However, matrix changes are no longer a performance problem for our iterative PCG solver, since no factorization has to be updated, and re-computing the diagonal Jacobi preconditioner $\mathbf{J}$ is trivial. Nevertheless, a complete rebuild of the system matrix would be prohibitive, due to our custom matrix format (see Section 3.2.3). We therefore do not update $\mathbf{A}$, but instead represent the system matrix as the sum $\mathbf{A}+\mathbf{A}_{\text{col}}$, where $\mathbf{A}_{\text{col}}$ contains the changing non-diagonal entries. Whenever collisions change, we just rebuild the highly sparse matrix $\mathbf{A}_{\text{col}}$ and update the diagonal entries of $\mathbf{A}$. We store $\mathbf{A}_{\text{col}}$ in CRS format and use our shared memory access pattern explained in 3.2.3. The computation of $\mathbf{Q}^{\mathsf{T}}\mathbf{W}\mathbf{p}$ in (2.13) is handled in a similar way.

Collision detection is done on the CPU as explained before. The nearest surface point for a colliding vertex can be efficiently determined due to our volumetric tissue mesh construction: Each tetrahedron is uniquely associated with the skin triangle that its prismatic cell was built from. Thus, if we detect a collision in a tetrahedron, we project the colliding vertex onto this triangle's plane. While this is not always the closest point on the skin it is a very good approximation (see Figure 3.16 and Video V.4).

For skinning animations we have a lot of resting contacts. If we removed a collision constraint as soon as the corresponding vertex has left the tetrahedron, the strain constraints would push the vertex back into the colliding state in the next iteration. This causes the vertex to alternate between a colliding and a resolved state. To avoid this oscillating behavior, we retain all colliding triangle-vertex pairs that have a distance less than a threshold δ_{col}. We set this to 25% of the mesh's average edge length.

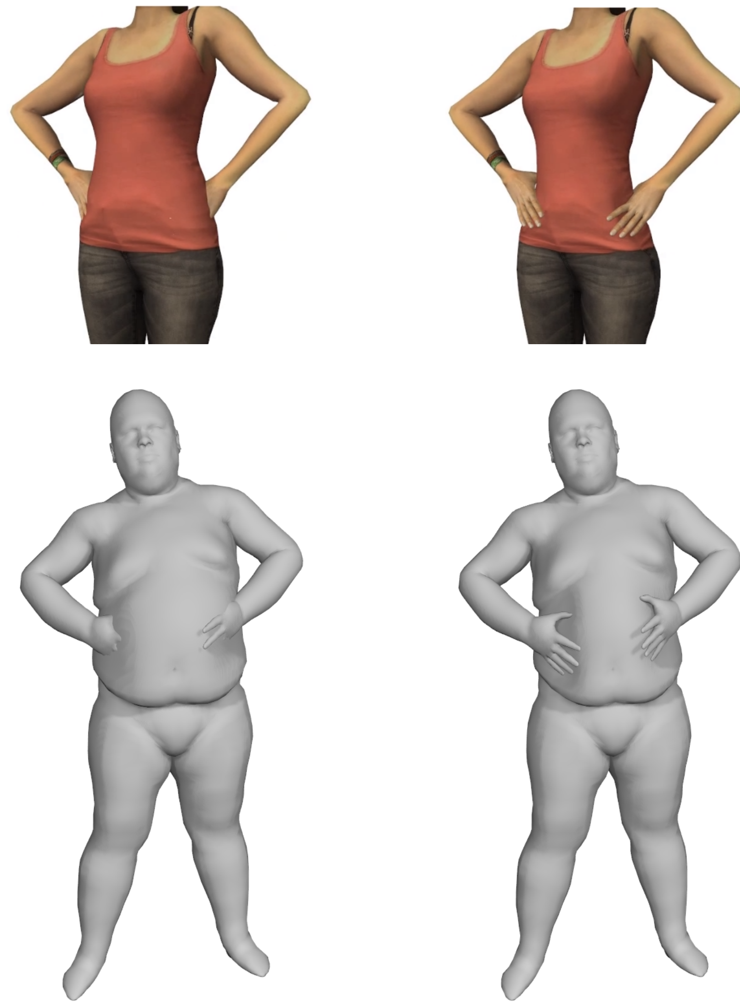


Figure 3.16: Comparison of simulations with our *global* collision constraints disabled (left) and enabled (right). The male model is part of the MPI Dynamic FAUST dataset [Bogo et al. 2017].

Resting contacts cause a second problem: As discussed earlier, using target positions in collision constraints can lead to unnatural dynamic effects. For example, if the character's skin collides between upper arm and forearm while jumping up and down, the global translation of the jumping motion will be transferred through the strain constraints, but all target projections of colliding vertices would still be untranslated, leading to local artificial damping. To solve this issue, we use translation-invariant collision constraints: Instead of using the *absolute* position of the collision-free state as target projection, we represent it relative to the corresponding skin triangle using three edge-strain constraints acting like springs between the colliding vertex and the triangle's vertices. The rest-length of each spring is set to the distance between a vertex of the triangle and the projection of the colliding point to the triangle plane.

3.3 IMPLEMENTATION DETAILS

We implemented our skinning method in C++ using OpenGL for rendering and Eigen [Guennebaud et al. 2018] for numerical linear algebra. We built our own Projective Dynamics solver inspired by Shape-Op [Deuss et al. 2015]. Our examples using Position Based Dynamics are based on the PDB framework of Bender et al. [2017]. All timings were measured on a workstation equipped with Intel Core i9-10900X CPU (10 cores, 20 threads $\times$ 3.7 GHz) and an Nvidia RTX 2080 TI (4352 CUDA cores, compute capability 7.5). For parallelization we used OpenMP on the CPU and CUDA 11.1 on the GPU.

Pre-processing & Parameters

Apart from the pre-computations required for our skinning simulation and upsampling, like mesh decimation, computation of MLS-weights and the volumetric mesh generation, we perform some special pre-processing for human characters. Here, we exclude all face vertices from the simulation, since it often contains complicated parts like separate eye and teeth meshes and is usually animated using blendshapes rather than skeleton-based skinning. For coupling the simulation to the movements of the face in connecting regions (i.e., at the neck), we use additional anchor constraints (either soft or hard) at the boundary.

If not otherwise mentioned, we use the following parameters in all our simulations: The weight of the tetrahedron strain constraint is set to 85 and the global and local collision weight to 150. We set a time-step of $\delta t = 0.2$, disable additional damping via $\mu = 1.0$ (implicit damping is sufficient) and perform $N_{pd} = 10$ local-global iterations per time-step. All human characters are scaled such that distances are measured in 1 m. For determining vertex masses, the user can set the total body weight of the character. This mass is divided among the vertices such that each sub-mass is proportional to the volume of all tetrahedra sharing the associated vertex.

Note that the frame rate also affects the dynamic behavior of the simulation: at lower rates the same motion is performed in fewer sub-steps leading to larger jumps of the skeleton from frame to frame. This is usually handled by coupling the time-step to the frame rate. However, since the time-step also controls the amount of implicit damping, there will still be a difference in the results. Therefore, we usually set the global mass parameter manually to adjust the intensity of dynamic jiggling.

Polar Decomposition

A performance bottleneck of our CPU simulation is the polar decomposition used in the local projection steps to extract the rotational part of the

elements' deformation gradients, which is typically done via singular value decomposition (SVD).

We increase performance by using faster alternatives to SVD for computing the polar decomposition, as proposed by Chao et al. [2010]: we only compute the SVD in case of $\det(\mathbf{F}) < 10^{-5}$, and use the Newton-based approach of Higham [1986] otherwise. An explanation of the algorithm can be found in Appendix A. Although we use the fast SVD implementation of Sifakis et al. [2011], this approach leads to a performance gain of the full CPU simulation by a factor of 1.9. On the GPU, we use the fast SVD implementation of Gao et al. [2018]. Here, opposed to the CPU case, the determinant-based branching shows no benefits. As explained in Section 2.3, a warp (32 threads) always performs the same operation in parallel and branching requires threads of one branch to wait for the others to finish their branch. That means warp execution can only be sped up, if all 32 threads follow the same (faster) branch.

Additionally, we implemented the iterative polar decomposition approach of Kugelstadt et al. [2018], which takes advantage of the rotation result from a previous local step. But opposed to reasonable speedups for CPU simulations, we found just minor improvements of a few percent compared to the direct SVD method on the GPU. We think that this is due to the additional cost of reading the matrix of the previous step. Moreover, we found the method to be less robust: if the deformation gradient includes high rotation angles ($> 60°$) or inversions, it often fails to extract a proper rotation, even when using many iterations. Therefore, we decided against using this approach.

3.4 RESULTS & DISCUSSION

In this section, we provide visual examples of simulations using our Fast Projective Skinning. We compare it to other skinning approaches and demonstrate its robustness and flexibility (Section 3.4.1). Additionally, we give details on the efficiency of our method regarding computational performance and memory consumption (Section 3.4.2).

3.4.1 Visual Results

By using a physics-based simulation, our method overcomes the artifacts of LBS (joint collapse) and DQS (bulging), as shown in Figures 3.17, 3.18 and in Video V.1. For the extreme pose shown in Figure 3.18 also the skinning with optimized centers of rotation (CoR) of Le and Hodgins [2016] suffers from bulging artifacts in the chest region, which is challenging since it is influenced by more than two bones. Moreover, the results of LBS, DQS and CoR all depend on high quality skinning weights and do not support collision handling or dynamic effects as opposed to our method. Examples of our

Figure 3.17: Three example poses. From left to right: Linear Blend Skinning, Dual Quaternion Skinning and our Fast Projective Skinning. The rightmost bottom image shows our FPS result with global collision handling enabled.

Figure 3.18: Comparison to other skinning approaches on a challenging pose. From top to bottom: LBS, DQS, CoR, skinning with PBD, our FPS.

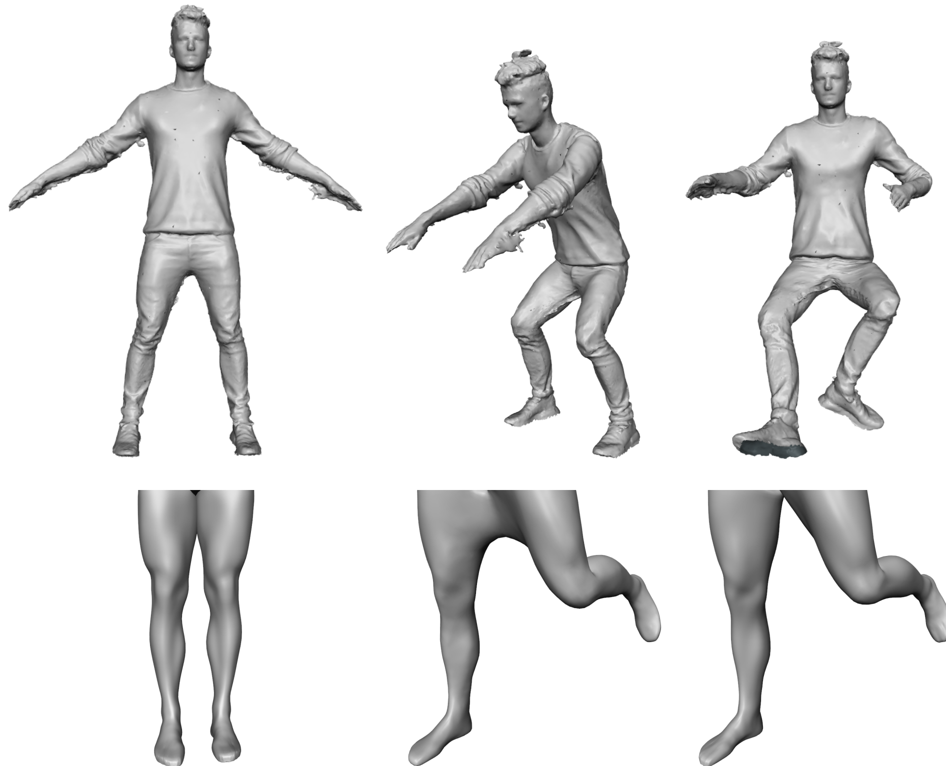


Figure 3.19: Top: raw scan obtained from Photoscan [Agisoft 2017], suffering from noise (arms) and holes due to missing data (feet). Bottom: Some input models suffer from self-intersections (e.g., in the crotch region). In this case, TetGen [Si 2015] fails to produce a volumetric model and TetWild [Hu et al. 2018] merges the intersecting regions leading to unwanted results (bottom middle). Our skinning approach robustly handles these challenging models (top/bottom right).

local/global collision handling can be found in Figure 3.15, 3.16, 3.17 and in Video V.3 (local) and V.4 (global). Especially in combination with dynamics, collisions provide much more believable motions, and FPS is the first approach that can detect and resolve global collisions for a full human character in real-time. Video V.5 shows some challenging motion-captured animations.

We also tried using Position Based Dynamics for our simulation, based on the implementation of Bender et al. [2017]. We add a spring constraint per tetrahedral edge and a volume-constraint, as proposed by Rumman and Fratarcangeli [2015]. An example is shown in Figure 3.18. Our skin shrinking step can lead to tetrahedrons with high aspect ratios, which are problematic for PBD, while our Projective Dynamics simulation works robustly. Moreover, PBD needs more iterations to converge than PD, and the PBD framework supports fitting positions only, causing the drawbacks discussed in Section 2.2.2. Since each constraint directly updates its corresponding vertices, it is more difficult to parallelize than the independent local steps of PD. Furthermore, the order in which the constraints are processed affects the solution and can lead to alternating jumps between different goal positions. PD's stability under

extreme motions and deformations, as demonstrated by Bouaziz et al. [2014a], is particularly important for skinning applications, where abrupt changes in joint angles can occur, e.g., due to noise and outliers in motion capture data.

Non-clean input meshes are another challenge for common skinning approaches. Here, automatic rigging is not able to produce proper skinning weights and traditional tetrahedral mesh generation approaches fail. Our method for volumetric mesh generation robustly handles non-clean meshes, as we demonstrate in Figure 3.19 by skinning a raw scan suffering from noise and holes. We also tried to apply the TetWild approach of Hu et al. [2018] to our inputs. While their method produces high-quality tetrahedral meshes for almost arbitrary input surfaces, it can take several minutes to compute, and the number of tetrahedra is much higher compared to our models, which negatively affects performance. In the bottom row of Figure 3.19 we further show that TetWild causes skinning artifacts by merging regions with initial self-intersections. Moreover, by not using our skin shrinking approach, the region between joints is also filled with tetrahedra. When bending the joint, these are exposed to extreme strain which negatively affects the skinning result. Our method can also handle challenging input skeletons. For instance, we can simulate the armadillo model with a skeleton consisting of just two bones in the torso. This can be desirable if the animator just wants to make some quick adjustments to the pose without constructing a full character rig including every bone. Note that collision detection and dynamics will not work properly in this case since some tetrahedra cover regions outside of the mesh.

When aiming for stylized animations, it may be desired to allow the bones to twist and stretch instead of being limited to rigid motions [Jacobson and Sorkine 2011]. This extension can be easily implemented in the current pipeline by simply twisting and stretching vertices of the skeleton surface $\mathcal{B}$ as demonstrated in Figure 3.20. We can also support unconnected skeletons

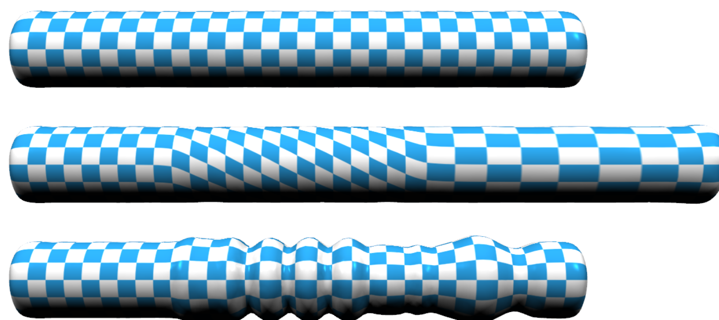


Figure 3.20: Our skinning algorithm is not limited to rigid motions of the skeleton. In this example, a cylinder mesh with three bones (top) is skinned using a twisting of its second and a stretching of its third bone (middle). In fact, we can incorporate even complex transformations of the bone layer if desired by the artist (bottom).

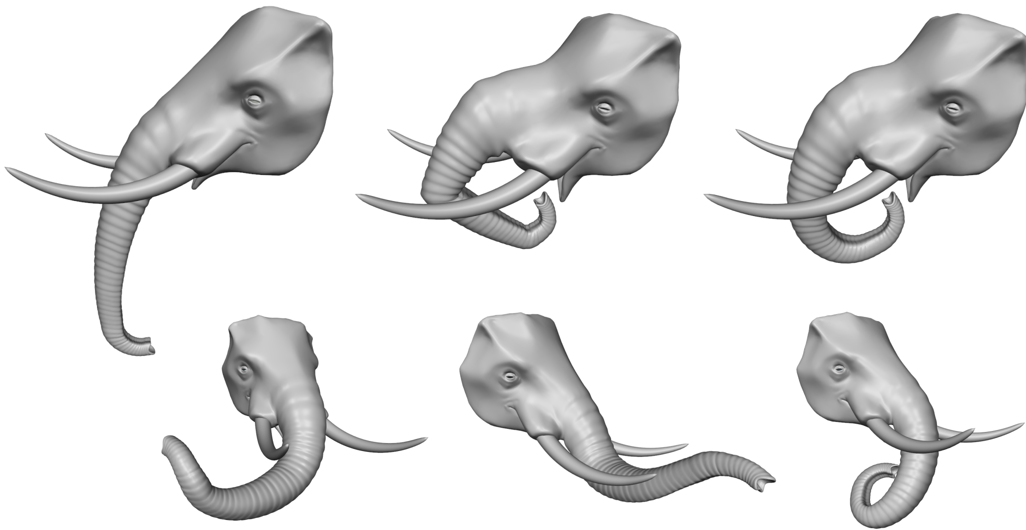


Figure 3.21: Skeleton-based skinning is not well-suited for soft extremities like the elephant's trunk. We can instead use virtual bones for the mid-part of the trunk, resulting in an unconnected skeleton. Top row (left to right): base pose, skinned pose with connected skeleton, skinned pose with unconnected skeleton. We are also able to simulate fully unconstrained extremities by setting virtual bones only. The bottom row shows some example frames of a rotating head motion, that dynamically shakes the soft trunk.

by temporarily connecting the individual parts using *virtual* bones for the volumetric mesh generation process. When building the FPS system matrix, we simply do not set anchor constraints in these regions. In the same way, soft extremities can be modeled. Examples of this are shown in Figure 3.21 and in Video V.7.

In Figure 3.22 we demonstrate the influence of joint positions and sizes. By choosing smaller joints that are closer to the surface, we get a sharper tip when bending the joint. On the other hand, setting a larger joint radius produces a bigger and smoother joint bulge. Both effects can be useful for different regions of the body. The former is better suited for finger and elbow joints, while the latter better approximates shoulder and knee joints.

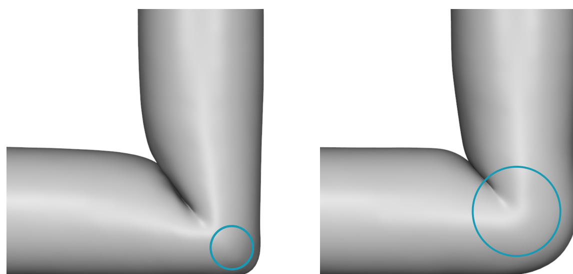


Figure 3.22: The position and size of joints influence the skinning result. The location of the bent joint is indicated with a blue circle. Left: small radius and close to the surface. Right: large radius and located at the center of the cylinder.

3.4.2 Performance

All the visual results shown so far can be simulated in real-time when using Fast Projective Skinning. In the following, we compare the performance of our method on both CPU and GPU for different character models. Note that while the GPU version of FPS exploits the iterative PCG solver discussed in Section 3.2.3, the CPU version employs Eigen's sparse Cholesky solver, since it is faster than the CPU-based PCG solver. Timing results are provided in Table 3.2.

Without global collision handling, even our CPU-FPS is able to achieve interactive frame rates. Due to the MLS-upsampling (Section 3.2.4) we do not need to use high resolutions for the simulated meshes. Therefore, the settings of the 'human low' model provide sufficient quality. Comparing CPU-FPS to GPU-FPS without collision handling, the latter is faster by an additional factor of 4–9, depending on model size. The timings for our MLS-upsampling are merely dependent on the number of high-resolution vertices H (compare the mid and low human character models in Table 3.2) as long as we use a constant number of neighbors for the upsampling. Note that the number of surface vertices S (and tetrahedra) of the elephant and 'human mid' models are similar. The difference in simulated vertices N comes from the decreased number of boundary constraints due to the virtual bones we use for the elephant's trunk.

Analyzing FPS *with global collisions*, the CPU version has to re-compute the Cholesky factorization, which the PCG solver of the GPU version avoids. Comparing the two, GPU-FPS is about $5\times - 8\times$ faster than CPU-FPS. Even with full collision handling, GPU-FPS is in general faster than CPU-FPS without collisions. Since both FPS versions perform collision detection on the CPU, the difference in t_{col} times is due to matrix re-factorization in the CPU case. Note that the collision timings refer to global collision handling. Our pre-computed local collision formulation comes with negligible overhead, hence, we can always handle at least local collisions in real-time on the CPU.

The t_{pre} timings listed in Table 3.2 do not include the computation of the upsampling weights which takes 10-20 seconds per model. These weights are pre-computed and stored. All other initialization steps for constructing the volumetric model from a skin surface and a skeleton graph are quite fast ($t_{\text{pre}} < 1\,\text{sec}$) and are therefore performed each time before the simulation starts. GPU-FPS is also memory efficient. Global memory on the GPU is limited, especially for video games, where most of the memory is reserved for highly detailed textures. Our typical human character models require only about 14 MB (8 MB simulation, 6 MB upsampling) of device memory to be skinned and are therefore able to fit within these restrictions.

Comparing the low-, mid-, and high-resolution human character results, one can observe that the simulation time scales linearly with the number of

Model	S	N	T	H	t_{pre}	CPU-FPS t_{sim}	t_{us}	fps	t_{col}	fps$_{col}$	GPU-FPS t_{sim}	t_{us}	fps	t_{col}	fps$_{col}$	M_{GPU}
Cylinder	810	874	4.8k	51k	97	1.5	0.74	410	2.0	210	1.2	0.07	630	0.5	450	16
Human low	3472	4151	21k	15k	253	6.5	0.22	145	8.9	63	1.6	0.02	530	1.1	330	14
Human mid	8482	9937	51k	15k	451	16.2	0.22	59	21	26	2.1	0.02	403	2.7	190	22
Human high	15k	18k	92k	15k	702	29.6	—	33	40	14	2.9	—	301	6.3	104	32
Armadillo	5189	6280	31k	173k	663	9.7	2.2	85	14	38	1.8	0.32	393	1.3	260	56
Elephant	8700	13k	52k	44k	939	21.5	0.58	44	21	23	2.3	0.05	370	2.7	185	32

Table 3.2: Benchmark results for several models with S input surface vertices, N simulated vertices, T simulated tetrahedra, and H high-res visualization vertices, comparing our CPU-based Fast Projective Skinning with the GPU version of our approach. We list the pre-computation time for constructing the volumetric mesh t_{pre}, the simulation time t_{sim} of one time-step (10 local-global iterations, each with 10 PCG iterations on GPU); the time t_{us} for upsampling to the visualization mesh; performance in frames per second (fps) for skinning, upsampling, and visualization; the time t_{coll} for collision detection and matrix update; performance in fps for skinning, upsampling, collision handling, and visualization (fps$_{coll}$). All timings are given in milliseconds. Lastly, M_{GPU} gives the amount of device memory that is used by our GPU solver in megabytes. Note that the armadillo, cylinder and elephant model can be found in different figures of this thesis but all human character models shown in this thesis are similar to the model 'human low' in regards to vertex counts and computation times. As explained before, the face region of the human character models is not simulated and contains about $6k$ vertices that are not included in H. Nevertheless, their rigid transformation and visualization still contributes to the timings.

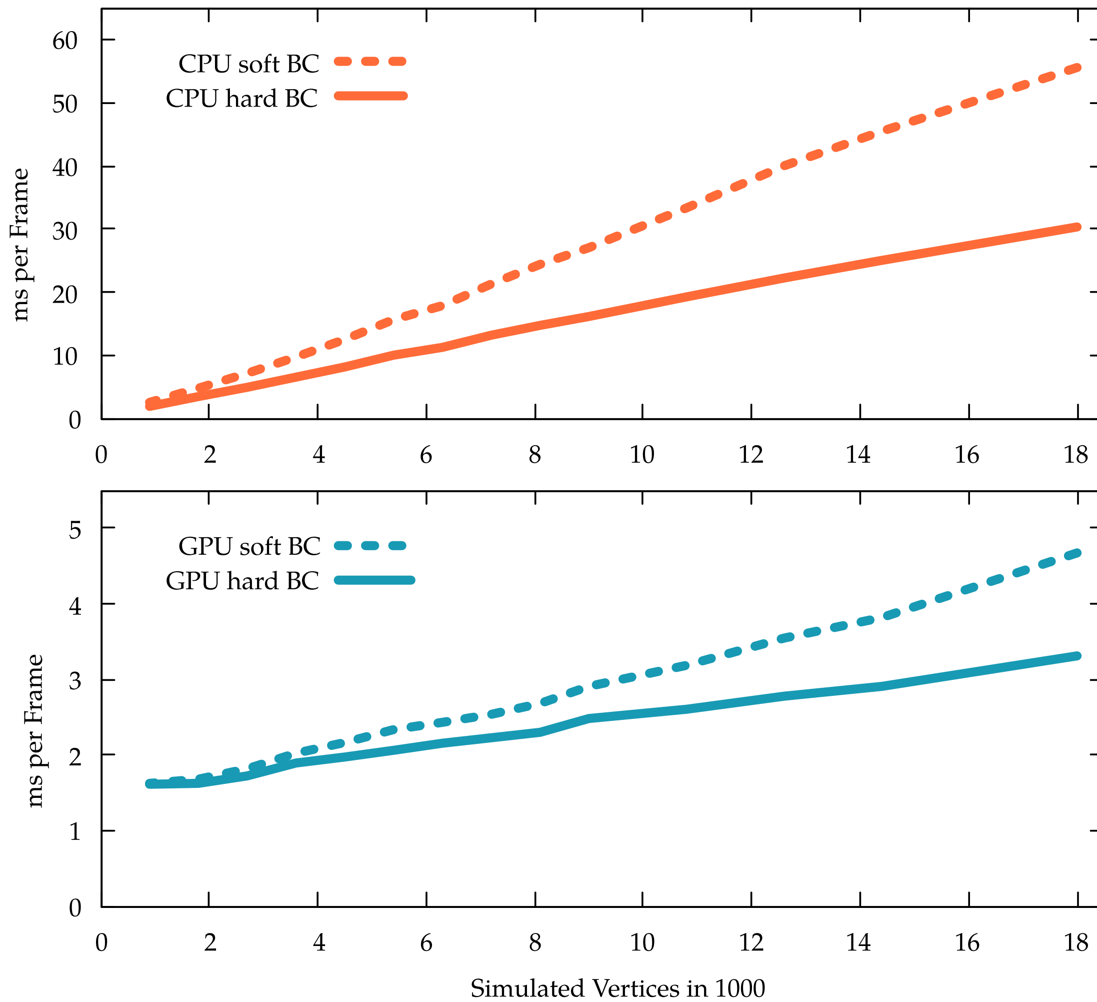


Figure 3.23: Performance comparison of Fast Projective Skinning with anchor constraints using Projective Dynamic's soft boundary conditions, and our hard Dirichlet boundary conditions on CPU (top) and GPU (bottom).

simulated vertices N on the CPU. On the GPU, we observe a sub-linear scaling due to overheads of kernel calls and a low GPU occupancy for smaller meshes. Note that in case of the simple cylinder model, this effect is predominant leading to a very small difference between GPU and CPU timings.

The influence of using hard constraints instead of soft constraints (Section 2.2.3) is analyzed in Figure 3.23. Hard constraints considerably reduce the size of the system matrix, since all constrained vertices are removed from the system. The absence of soft constraints furthermore improves the matrix condition [Botsch and Sorkine 2008], which improves convergence of our iterative solver. This leads to speed-up factors of 1.35 (for small meshes) up to 1.8 (for large meshes) on the CPU (Figure 3.23). On the GPU, the performance gain for small meshes is barely noticeable (only a few percent), because our 4352 CUDA cores can solve smaller systems completely in parallel, such that any further reduction just leaves some threads idle. In addition, the overhead of calling a kernel is more dominant for small meshes as explained before.

For larger systems however, the reduced formulation also yields an important speed-up of about 40% on the GPU.

When analyzing frames per second we include all steps of the pipeline contributing to one frame. In the following, we list more detailed timings for the armadillo model that should serve as a guideline for future optimization: The total frame-time of 2.54 ms (1000 ms/393 fps) splits up into 0.19 ms (7.5%) for computing the transformation matrices of joints and bones and sending these to the GPU, 1.75 ms (68.9%) for the Projective Dynamics time-step, 0.04 ms (1.6%) for updating the normals of the simulation mesh (needed for the upsampling) and 0.32 ms (12.6%) for the upsampling itself, lastly, the remaining 0.24 ms (9.4%) are used for rendering the visualization mesh. The 1.75 ms of the PD time-step can be further divided into the local step (0.62 ms), the global step (1.1 ms) and all other steps (0.03 ms) of the PD algorithm (2.1).

The distribution of the frame-time reveals that the local and global updates are still the major bottleneck of the algorithm. We found one additional option to increase performance even more if desired/needed: Looking at the cylinder model in Table 3.2, smaller models are barely able to reduce the frame-time further due to the overhead of calling the different kernels of the PD-algorithm ($N_{\mathrm{pd}} = 10$ updates, each with 10 PCG iterations require 310 kernel calls for the global steps in each time-step). We experimented with lower numbers of PD iterations and found that it can be further reduced to $N_{\mathrm{pd}} = 2$, without noticing major visual differences as shown in the Video V.6 (one can observe a slight difference at the fingers for abrupt hand motions). This reduces the simulation time by a factor of five and results in more than 1500 frames per second when using the low-resolution human character mesh with MLS upsampling and without collision handling, which is not much slower than standard Linear Blend Skinning. Here, other parts of the whole skinning pipeline like rendering and forward kinematics become increasingly relevant for the frame's time-budget. Moreover, we are currently detecting and updating global collisions for every frame. This can be reduced to about 60 updates per second (or even further, dependent on the motion) to accelerate FPS with global collision handling to more than 1000 frames per second.

Due to the high performance and moderate memory requirements, we can even animate multiple characters in real-time using Fast Projective Skinning. When simply running multiple FPS-solvers sequentially, frame-time scales linearly with the number of characters. We assume that it would be beneficial to fuse the corresponding kernels of different characters into one (e.g., one kernel for the local step of all characters instead of one per character) but we did not implement this for verification.

3.5 SUMMARY & LIMITATIONS

We have presented Fast Projective Skinning, a physics-based skinning approach that enables high quality skeleton-based character animation in real-time. Our volumetric skeleton and tissue model is computed automatically from an input skin mesh and a skeleton graph. In particular, our technique does not require any skinning weights, which makes it easy to setup and use for a large variety of skin meshes.

Our volumetric skeleton provides effects like joint bulges and skin sliding in a more natural way than existing approaches, which require additional constraints. Compared to methods based on Position Based Dynamics, our Projective Dynamics simulation is more robust and converges faster. This is crucial, since we do not just correct an initial geometric skinning, but instead compute the full deformation through a physics-based simulation, which in turn yields a more natural behavior.

For CPU skinning, we can efficiently detect and resolve local collisions by pre-computing potential collision pairs and through novel collision constraints that do not require computationally expensive re-factorization of the system matrix. Our GPU implementation of the Projective Dynamics solver yields not just an considerable speed-up, but also overcomes the dependence on a constant set of constraints throughout the simulation. By exploiting this feature, Fast Projective Skinning becomes the first skinning approach that is capable of detecting and handling arbitrary self collisions in real-time. The MLS-based upsampling of vertex positions and normals efficiently transfers deformations of the low-resolution simulation meshes to the high-resolution input meshes used for visualization.

Our method also has some limitations: By default, FPS is not suitable for meshes with overhanging/overlapping parts, like big bulges of fat or clothing. In those cases the skin shrinking can lead to tetrahedra that are partially outside of the character's volume, which can cause unnatural dynamics and erroneous collisions. We presented a workaround in 3.4.1, that involves defining additional, virtual bones inside of overhanging body parts. Moreover, FPS is not applicable if the skin mesh is composed of multiple unconnected parts or if the skeletons lies outside of the skin. A solution to the former could be to join the sub-meshes by temporary additional vertices/edges in a pre-processing step as proposed by Le and Lewis [2019]. Our skin shrinking process can produce tetrahedra of high aspect ratios, and in some cases (e.g., bulges, irregular triangulation), there can be a tangential shift between skin and skeletal triangles. In the next chapter, we will develop a more sophisticated (but also much slower) skin shrinking approach leading to layered meshes of higher quality. Nevertheless, we did not notice any major artifacts caused by this problem in our FPS simulations.

Another problem is that our collision constraints incorporate vertex and face normals. We observed cases where collision response and normal update lead to small oscillations at resting contacts. This can be prevented by using a higher mesh resolution in colliding regions. Employing more sophisticated elastic deformers in the projection step like those introduced by Brunel et al. [2021] could improve collision results in future. In general, realistic collision handling requires realistic input motions: if a hand is moved into the rib-cage, the result will most likely look implausible. Another promising avenue for future work is to investigate recent solvers for nonlinear elasticity, such as ADMM [Narain et al. 2017] or the hyper-elastic solver of Liu et al. [2017].

The physical/anatomical plausibility of our approach is limited by the simple one-layer tissue mesh spanned between skin and bones, which can be observed in Video V.8. Since our model lacks a realistic representation of the body's interior, collisions and dynamic jiggling effects can look too strong in some regions. This problem is most apparent at the upper torso and the hips that do not include the large rib-cage and pelvis bone-structure, or the lower arm that is in reality built from two bones and a lot of muscles. To an extent, we can imitate a realistic anatomy by manually defining additional bones (e.g., to mimic a rib-cage) and by varying the stiffness of the soft tissue layer locally. In the following chapter, we will build a more sophisticated tissue mesh including realistic representations of the skeleton and muscle layer for human characters.

4 A THREE-LAYERED ANATOMICAL MODEL

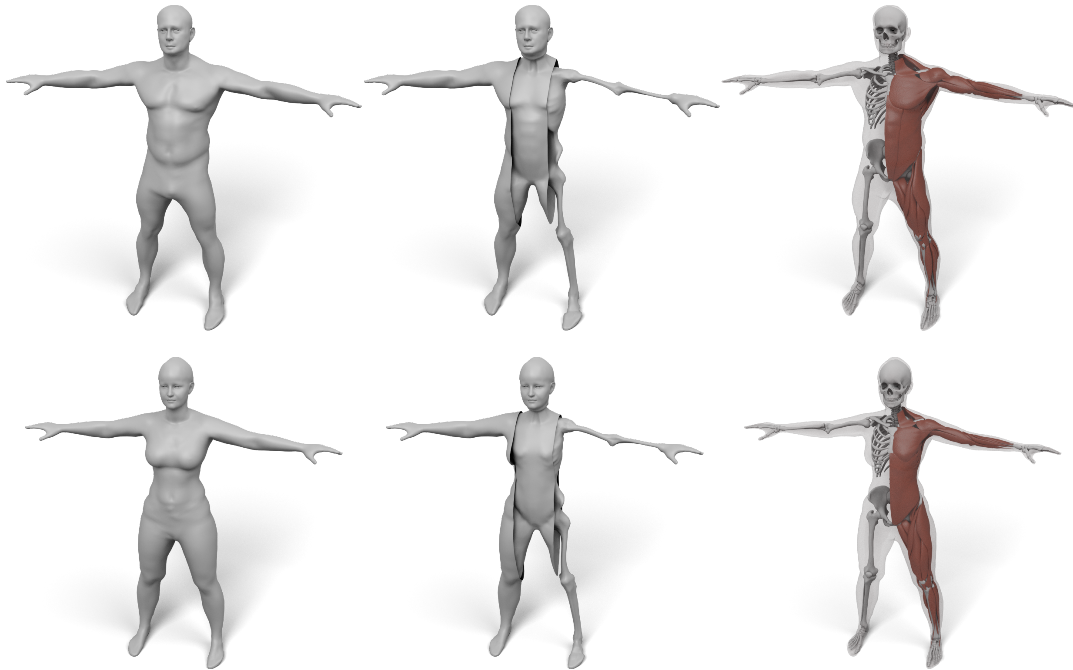


Figure 4.1: Starting from the surface of a human (left), we fit a three-layered model consisting of a skin, muscle, and skeleton surface (middle), which enables physical simulations in a simple and intuitive way. Interior structures, such as individual models of muscles and bones, can also be transferred using our layered model (right).

In the previous chapter, we presented a fast and simple approach for creating volumetric models of general virtual characters. However, our FPS model consists of a very simplified skeleton and just one uniform soft tissue layer. Therefore, it cannot be applied to simulate advanced effects like convincingly distributed dynamic jiggling or fat growth. In this chapter, we will focus on *human* characters to develop a model that is anatomically plausible, applicable for a wide range of body shapes, and ready to be used in physics simulations. Furthermore, we again aim for a very simple and efficient model generation approach requiring a low amount of input data.

If we look at a human, its appearance is mostly determined by everything we can directly see (skin, hair, cloth, etc.). Hence, it is not surprising that research has focused on capturing, analyzing, and animating *surface models* of humans. Consequently, there are numerous surface-based capturing approaches, suitable for almost every level of detail and budget: from complex multi-camera photogrammetry setups that capture finest-scale wrinkles of the human face [Riviere et al. 2020] over approaches that compute ready-to-

animate models from simple smart-phone videos [Wenninger et al. 2020] to machine learning approaches that reconstruct a virtual model from a single image [Weng et al. 2019]. As we already explained in the previous chapter, there are two different classes of approaches for creating convincing animations of those models. First, by collecting large amounts of 3D captured data to build sophisticated surface-based models [Loper et al. 2015; Anguelov et al. 2005b; Bogo et al. 2017], and second, with the help of physics-based simulations. The latter however, require models of *volumetric* virtual humans that incorporate (discrete approximations of) their interior anatomical structures. Although surface-based models might be sufficient for many applications, others (e.g., surgery simulation) require a volumetric model as an essential prerequisite.

While there exist detailed volumetric models of the human body [Ackerman 1998; Christ et al. 2009; Zygote 2020], they can be very tedious to work with. Since they usually consist of hundreds of different bones and muscles, merely creating a volumetric tetrahedral mesh for simulation purposes can be frustratingly difficult. Moreover, those models represent average humans and transferring their volumetric structure and anatomical details to a specific human (e.g., a scanned person) is not straightforward. Although there are a few approaches for creating personalized interior anatomy [Dicko et al. 2013; Kadleček et al. 2016], these methods either deform bone structures in a non-plausible manner [Dicko et al. 2013] or require a complex numerical optimization [Kadleček et al. 2016].

In this chapter we present a robust and efficient method for transferring an interior anatomy template into the surface mesh of a scanned person in just a couple of seconds. Inspired by our two-layered FPS models, we split the human body into three layers that are bounded by surfaces sharing the same triangulation: the *skin surface* defines the outer shape of the human, the *muscle surface* envelopes its individual muscles, and the *skeleton surface* wraps the subject's skeleton (see Figure 4.1 middle). The muscle layer is hence enclosed in between the skeleton and muscle surface, and the subcutaneous fat tissue by the muscle surface and skin surface. This layered template model is derived from the Zygote body model [Zygote 2020], which provides an accurate representation of both the male and female anatomy. We propose simple and fast methods for fitting the layered template to surface scans of humans and for transferring the high-resolution anatomical details [Zygote 2020] into these fitted layers (see Figure 4.1 right). Our method is robust, efficient, and fully automatic, which we demonstrate on about 1700 scans from the European CAESAR dataset [Robinette et al. 2002].

Our approach enriches simple surface scans by plausible anatomical details, which are suitable for educational visualizations and volumetric simulations. We note, however, that due to the lack of true volumetric information, it is not a replacement of volumetric imaging techniques in a medical context.

Individual Contribution *The approach for creating personalized, volumetric anatomical models presented in this chapter was developed in cooperation with Stephan Wenninger and supervised by Mario Botsch. Stephan Wenninger prepared the different datasets (pointclouds and surface meshes) by re-topologizing them to a common surface template. The author of this thesis created the novel approach for generating a layered volumetric template defined by skin, muscle, and bone surfaces, which all have the same triangulation, thereby making volumetric tessellation straightforward. Stephan Wenninger built the regressor that extracts the amount of muscle and fat mass of a subject from the skin surface only, thereby making manual specification of muscle and fat distribution unnecessary. By combining the former steps, the author of this thesis developed a robust and efficient method for transferring the layered volumetric template model into a given surface scan of a human in just a couple of seconds. The author further designed and implemented the different applications for physics-based simulation and fat growth.*

Corresponding Publication *This chapter is based on the following publication:*

Komaritzan, M., Wenninger, S. and Botsch, M. (2021). "Inside Humans: Creating a Simple Layered Anatomical Model from Human Surface Scans." Frontiers in Virtual Reality, 2.

4.1 RELATED WORK

Using a layered volumetric model of a virtual character has been shown beneficial compared to a surface-only model in multiple previous works. Deul and Bender [2013] compute a simple layered model representing a bone, muscle, and fat layer, which they use for a multi-layered skinning approach. Simplistic layered models have also been used to extend the SMPL surface model [Loper et al. 2015] in order to support elastic effects in skinning animations [Kim et al. 2017; Romero et al. 2020]. Compared to these works, our three layers yield an anatomically more accurate representation of the human body, while still being simpler and more efficient than complex irregular tetrahedralizations. Saito et al. [2015] show that a muscle enveloping layer yields more convincing muscle growth simulations and reduces the number of tetrahedral elements required in their computational model. They also demonstrate how to simulate different variations of bone sizes, muscle mass and fat mass for a virtual character.

When it comes to the generation of realistic personalized anatomical structures from a given skin surface, most previous works focus on the human head: Ichim et al. [2016] register a template skull model to a surface-scan of the head in order to build a combined animation model using both physics-based and blendshape-based face animation. Ichim et al. [2017] also incorporate facial muscles and a muscle activation model to allow more advanced face animation effects. Gietzen et al. [2019] and Achenbach et al. [2018] use volumetric CT head scans and surface-based head scans in order to learn a combined statistical model of the head surface, the skull surface, and the enclosed soft tissue, which allows them to estimate the head surface from the skull shape and vice versa. Regarding other parts of the human body, Zhu et al. [2015] propose an anatomical model of the upper and lower limbs that can be fit to surface scans and is able to reconstruct motions of the limbs.

There are just two former approaches for generating an anatomical model of the *complete core* human body (torso, arms, legs) from a given skin surface. In their pioneering work, Dicko et al. [2013] transfer the anatomic details from a template model to various humanoid target models, ranging from realistic body shapes to stylized non-human characters. They transfer the template's anatomy through a harmonic space warp and per-bone affine transformations, which, however, might distort muscles and bones in an implausible way. Different distributions of subcutaneous fat can be (and have to be) painted manually into a special fat texture. The work of Kadleček et al. [2016] is most closely related to our approach. They build an anatomically plausible volumetric model from a set of 3D-scans of a person in different poses. An inverse physics simulation is used to fit a volumetric anatomical template model to the set of surface scans, where custom constraints prevent

muscles and bones from deforming in an unnatural manner. We discuss the main differences between our approach and those of Dicko et al. [2013] and Kadleček et al. [2016] in Section 4.3.

Estimating the body composition from surface measures or 3D surface scans (like we do in Section 4.2.3) has been tackled before. There are numerous formulas for computing body fat percentage (BF), or body composition in general, from certain circumferences, skinfold thicknesses, age, gender, height, weight, and density measurements. Prominent examples are the 3-, 4-, and 7-site skinfold equations, or the Siri- and Brozek formulas [Jackson and Pollock 1985; Siri 1956; Brožek et al. 1963]. These formulas, however, either rely on anthropometric measurements that have to be taken by skilled personnel or on measuring the precise body density via expensive devices, such as BOD PODs [Fields et al. 2002]. Ng et al. [2016] compute BF based on a 3D body scan of the subject, but their formula is tailored towards body scans and measurements taken with the Fit3D Scanner [Fit3D 2021]. Even with the help of the authors we could not successfully apply their formulas to scans taken with different systems, since we could always find examples resulting in obviously wrong (or even negative) body fat percentages. Recently, Maalin et al. [2020] showed that modeling body composition through body fat alone is an inferior measure for defining the shape of a person compared to a combined model of fat mass and muscle mass. We therefore adapt their data to estimate fat mass and muscle mass from surface scans alone (Section 4.2.3). Incorporating these estimations into the volumetric fitting process allows us to determine the proportion of muscles in the soft tissue layer more plausibly than Kadleček et al. [2016].

4.2 METHOD

Our approach consists of three main contributions: First, the generation of the volumetric three-layered template, described in Section 4.2.2, where we derive the skin, muscle, and skeleton surfaces from the male and female Zygote model [Zygote 2020]. Second, the estimation of a person's body composition, i.e., how much of the person's soft tissue is described by muscles and fat (Section 4.2.3). By adapting the BeyondBMI dataset [Maalin et al. 2020] to our template, we derive this information from the surface scan alone. Third, by utilizing this information about body composition, we derive an efficient method for fitting the layered template (including all contained anatomical details) (in)to a given human surface scan (Section 4.2.4). Figure 4.2 shows

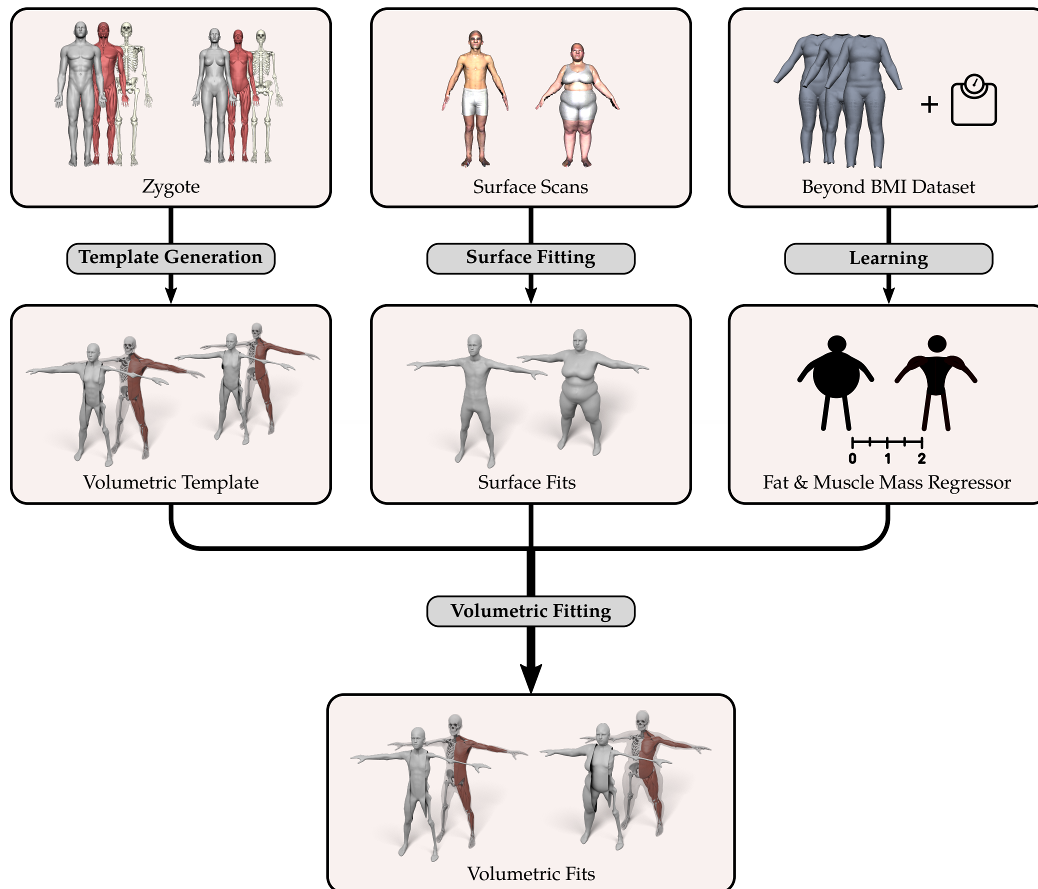


Figure 4.2: Overview of our volumetric template fitting approach. From the Zygote model [Zygote 2020], we build layered volumetric templates for the male and female anatomy. By adapting the BeyondBMI dataset [Maalin et al. 2020] we learn a model for estimating fat and muscle mass from a surface model. Given a person's surface scan, we then estimate its fat/muscle mass and use this information to fit the volumetric template (in)to the surface scan, which yields the personalized anatomical model.

an overview of the whole process, starting from the different input data sets, the template model and the muscle/fat regressor, to the final personalized anatomical fit.

4.2.1 Data Preparation

In our approach we make use of several publicly or commercially available datasets for model generation, model learning, and evaluation:

- *Zygote:* The Zygote model [Zygote 2020] provides high-resolution representations of the male and female anatomy. We use their skin, muscle, and skeleton meshes for building our layered template.
- *BeyondBMI:* Maalin et al. [2020] scanned about 400 people and additionally measured their fat mass (FM), muscle mass (MM), and body mass index (BMI) using a medical-grade eight-electrode bioelectrical impedance analysis. They provide annotated (synthetic) scans of 100 men and 100 women, each computed by averaging shape and annotations of two randomly chosen subjects. From this data we learn a regressor that estimates fat and muscle mass from the skin surface.
- *Hasler:* The dataset of Hasler et al. [2009] contains scans of 114 subjects in 35 different poses, captured by a 3D laser scanner. The scans are annotated with fat and muscle mass percentage as measured by a consumer-grade impedance spectroscopy body fat scale. We use this dataset to evaluate the regressor learned from the BeyondBMI data.
- *CAESAR:* The European subset of the CAESAR scan database [Robinette et al. 2002] consists of 3D-scans (with about 70 selected landmarks) equipped with annotations (e.g., weight, height, BMI) of about 1700 subjects in a standing pose. We use this data to evaluate our overall fitting procedure.

All these data sources use different model representations, i.e., either different mesh tessellations or even just point clouds. In a pre-processing step we therefore re-topologize the skin surfaces of these datasets to a common triangulation by fitting a surface template using the non-rigid surface-based registration of Achenbach et al. [2017].

This approach is based on an animation-ready, fully rigged, statistical template model. Its mesh tessellation (about 21k vertices), skeletal rig, and skinning weights come from the Autodesk Character Generator [Autodesk 2014]. It uses a 10-dimensional PCA model representing the human body shape variation, and we will call it the *surface template* in the following. In a pre-processing step we fit the surface template to all input surface scans to achieve a common triangulation and thereby establish dense correspondence. This fitting process is guided by a set of landmarks, which are either specified

manually or provided by the dataset. A nonlinear optimization then determines alignment (scaling, rotation, translation), body shape (PCA parameters), and pose (inverse kinematics on joint angles) in order to minimize squared distances of user-selected landmarks and automatically determined closest point correspondences in a non-rigid ICP manner [Bouaziz et al. 2014b]. Once the model parameters are optimized, a fine-scale out-of-model deformation improves the matching accuracy and results in the final template fit. For more details we refer to Achenbach et al. [2017].

4.2.2 Generating the Volumetric Template

We use the male and female Zygote body model [Zygote 2020] as a starting point for our volumetric model. Our volumetric template is defined by the *skeleton surface* $\mathcal{B}$ (for bones), the *muscle surface* $\mathcal{M}$, and the *skin surface* $\mathcal{S}$. The skeleton is enveloped by the skeleton surface, the muscle layer resides between the skeleton surface and the muscle surface, and the (subcutaneous) fat layer between muscle surface and skin surface, respectively. The soft-tissue layer is the union of the fat and muscle layers. In our layered model we exclude the head, hands, and toes. These regions will be identical to the skin surface in all layers. See Figure 4.3 for a visualization of the layered template.

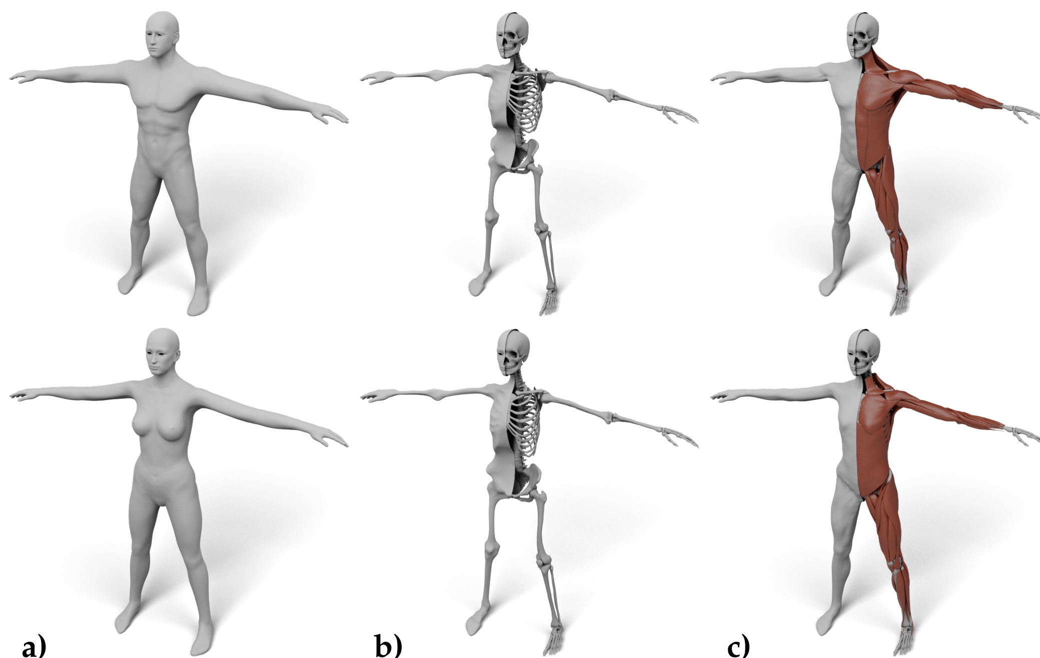


Figure 4.3: Our layered template for both male (top) and female (bottom): the skin surface (a), the skeleton surface enveloping the skeleton (b), and the muscle surface enveloping both muscles and skeleton (c). For (b) and (c) the left half shows the enveloping surface, the right half the enveloped anatomy.

Inspired by the two-layered volumetric models we used in our FPS method, the three surfaces $\mathcal{B}$, $\mathcal{M}$, and $\mathcal{S}$ will be constructed to share the same triangulation, providing a straightforward one-to-one correspondence between the ith vertices on each surface, which we denote by $\mathbf{x}_i^{\mathcal{B}}$, $\mathbf{x}_i^{\mathcal{M}}$, and $\mathbf{x}_i^{\mathcal{S}}$, respectively. Note that we differ slightly from the FPS notation by also denoting skin vertices with a superscript for easier identification in the different equations. Each two corresponding triangles $(\mathbf{x}_i^{\mathcal{S}}, \mathbf{x}_j^{\mathcal{S}}, \mathbf{x}_k^{\mathcal{S}})$ on $\mathcal{S}$ and $(\mathbf{x}_i^{\mathcal{M}}, \mathbf{x}_j^{\mathcal{M}}, \mathbf{x}_k^{\mathcal{M}})$ on $\mathcal{M}$ span a volumetric element of the fat layer. Similarly, the volumetric elements of the muscle layer are spanned by pairs of triangles $(\mathbf{x}_i^{\mathcal{M}}, \mathbf{x}_j^{\mathcal{M}}, \mathbf{x}_k^{\mathcal{M}})$ on $\mathcal{M}$ and $(\mathbf{x}_i^{\mathcal{B}}, \mathbf{x}_j^{\mathcal{B}}, \mathbf{x}_k^{\mathcal{B}})$ on $\mathcal{B}$. Analogously to the previous chapter, we call these elements, built from six vertices of two triangles, *prisms*, and will either use them directly in a simulation or split them into three tetrahedra each, resulting in a simple conforming volumetric tessellation.

The following two parts of this section describe how to generate the skeleton surface $\mathcal{B}$ and the muscle surface $\mathcal{M}$. The skin surface $\mathcal{S}$ is generated by fitting the surface template of Achenbach et al. [2017] to the skin of the anatomical model [Zygote 2020], as described in Section 4.2.1.

The Skeleton Surface

The skeleton surface $\mathcal{B}$ should enclose all the bones of the detailed skeleton model, as shown in Figure 4.3, center. We achieve this by shrink-wrapping the skin surface $\mathcal{S}$ onto the skeletal bones. To avoid problems caused by gaps between bones (e.g., rib-cage, tibia/fibula), we first generate a skeleton wrap $\mathcal{W}$, a watertight genus-0 surface that encapsulates the bones, and then shrink-wrap the skin surface to $\mathcal{W}$ instead. The wrap surface $\mathcal{W}$ can easily be generated by a few iterations of shrink-wrapping, remeshing, and smoothing of a bounding sphere in a 3D modeling software like Blender or Maya. This results in a smooth, watertight, and two-manifold surface $\mathcal{W}$ that excludes regions like the interior of the rib-cage and small holes like in the pelvis or between ulna and radius.

To generate the skeleton surface, we could apply our FPS skin shrinking approach from Section 3.2.1 by defining a skeleton graph inside the skeleton surface, finding nearest neighbors from each skin vertex to its bone-lines and smoothing those until convergence to get the base-points. Lastly, the vertex positions of the skeleton surface would be defined by the intersection between each line from a skin vertex to its base-point and $\mathcal{W}$. While this approach functioned properly for the simple tube-like skeletons of our FPS method, using the same strategy on realistic skeletons results in highly distorted triangles (see Figure 4.4) impairing the quality of the model itself and all simulations based on it.

Instead, we develop a more sophisticated skin shrinking approach that results in a high-quality skeleton surface $\mathcal{B}$ by starting from the skin surface

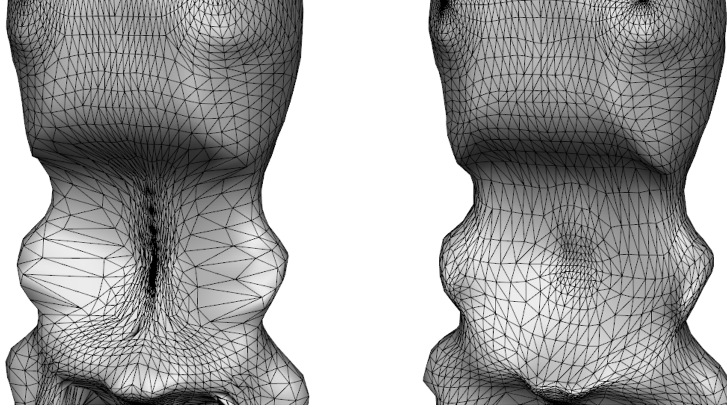


Figure 4.4: The skin shrinking approach we applied to generate the volumetric Fast Projective Skinning models (left) works well on cylindrical parts of the skeleton (e.g., rib-cage) but distorts the triangles in more complicated region (e.g., hips). Moreover, it does not take tangential shifting between skin and skeleton layer into account. For comparison, the result of our more sophisticated fitting approach is shown on the right.

$\mathcal{S}$, i.e., setting $\mathcal{X} = \mathcal{S}$, and then minimizing a nonlinear least squares energy. This energy is composed of a fitting term, which attracts the surface $\mathcal{X}$ to the bone wrap $\mathcal{W}$, and a regularization term, which prevents $\mathcal{X}$ from deforming in a physically implausible manner from its initial state $\overline{\mathcal{X}} = \mathcal{S}$:

$$\mathcal{B} \;=\; \arg\min_{\mathcal{X}} \; w_{\text{fit}} E_{\text{fit}}(\mathcal{X}, \mathcal{W}) + w_{\text{reg}} E_{\text{reg}}\big(\mathcal{X}, \overline{\mathcal{X}}\big)\,. \tag{4.1}$$

The regularization is formulated as a discrete bending energy that penalizes the change of mean curvature, measured as the change of length of the Laplacian:

$$E_{\text{reg}}\big(\mathcal{X}, \overline{\mathcal{X}}\big) = \sum_{\mathbf{x}_i \in \mathcal{X}} A_i \, \|\Delta \mathbf{x}_i - \mathbf{R}_i \Delta \bar{\mathbf{x}}_i\|^2\,, \tag{4.2}$$

where $\mathbf{x}_i$ and $\bar{\mathbf{x}}_i$ denote the vertex positions of the deformed surface $\mathcal{X}$ and the initial surface $\overline{\mathcal{X}}$, respectively. The matrix $\mathbf{R}_i \in SO(3)$ denotes the optimal rotation aligning the vertex Laplacians $\Delta \mathbf{x}_i$ and $\Delta \bar{\mathbf{x}}_i$, which are discretized using the cotangent weights and the Voronoi areas A_i [Botsch et al. 2010].

The fitting term penalizes the squared distance of vertices $\mathbf{x}_i \in \mathcal{X}$ from their target positions $\mathbf{t}_i \in \mathcal{W}$:

$$E_{\text{fit}}(\mathcal{X}, \mathcal{W}) = \sum_{\mathbf{x}_i \in \mathcal{X}} w_i A_i \, \|\mathbf{x}_i - \mathbf{t}_i\|^2\,. \tag{4.3}$$

The target positions $\mathbf{t}_i$ are points (not necessarily vertices) on the skeleton wrap $\mathcal{W}$ of either one of three types: closest point correspondences, fixed correspondences, or collision targets. The weight w_i is determined solely by the type of target position $\mathbf{t}_i$ (0.1 for closest point correspondences, 1 for fixed correspondences, 100 for collision targets). We define just one target $\mathbf{t}_i$ for each

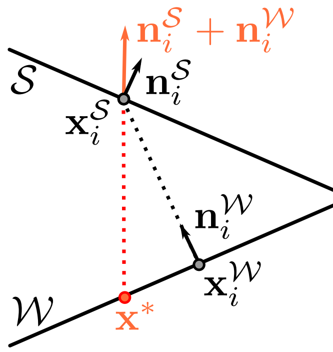


Figure 4.5: An example of minimizing the alignment energy (4.4) for the skin vertex $\mathbf{x}_i^{\mathcal{S}}$. Its closest position on the skeleton wrap is $\mathbf{x}_i^{\mathcal{W}}$, leading to a small minimal angle (between black dotted line and $\mathcal{S}$). The position $\mathbf{x}^*$ maximizes the minimal angle and minimizes the energy (4.4). It can be found by tracing a line from $\mathbf{x}_i^{\mathcal{S}}$ in negative direction of the average normal $\mathbf{n}_i^{\mathcal{S}} + \mathbf{n}_i^{\mathcal{W}}$.

vertex $\mathbf{x}_i$. The default is a closest point correspondence per vertex, which can be overridden by a fixed correspondence, and both of them will be overridden by the collision target in case of a detected collision. Below we explain the three target types.

Closest point correspondences are updated in each iteration of the minimization to match the closest position on $\mathcal{W}$ to the vertex $\mathbf{x}_i \in \mathcal{X}$, i.e., $\mathbf{t}_i = \arg\min_{\mathbf{t}\in\mathcal{W}} \|\mathbf{x}_i - \mathbf{t}\|$.

Near complicated regions, like the armpit or the rib-cage, the skin has to stretch considerably to deform toward the skeleton wrap. As a consequence, corresponding triangles $(\mathbf{x}_i^{\mathcal{S}}, \mathbf{x}_j^{\mathcal{S}}, \mathbf{x}_k^{\mathcal{S}})$ on the skin surface $\mathcal{S}$ and $(\mathbf{x}_i^{\mathcal{B}}, \mathbf{x}_j^{\mathcal{B}}, \mathbf{x}_k^{\mathcal{B}})$ on the eventual skeleton surface $\mathcal{B}$ will not be approximately on top of each other, but will instead be tangentially shifted. Misaligned triangles lead to heavily sheared prisms, which can cause artifacts in physical simulations. We define a per-vertex score penalizing misalignment of corresponding vertices $\mathbf{x}_i^{\mathcal{S}} \in \mathcal{S}$ and $\mathbf{x}_i^{\mathcal{W}} \in \mathcal{W}$ w.r.t. their common averaged normal $\mathbf{n}_i^{\mathcal{S}} + \mathbf{n}_i^{\mathcal{W}}$:

$$E_{\text{align}}\left(\mathbf{x}_i^{\mathcal{S}}, \mathbf{x}_i^{\mathcal{W}}\right) = \left| \frac{\left(\mathbf{n}_i^{\mathcal{S}} + \mathbf{n}_i^{\mathcal{W}}\right)^{\mathsf{T}} \left(\mathbf{x}_i^{\mathcal{S}} - \mathbf{x}_i^{\mathcal{W}}\right)}{\|\mathbf{n}_i^{\mathcal{S}} + \mathbf{n}_i^{\mathcal{W}}\| \cdot \|\mathbf{x}_i^{\mathcal{S}} - \mathbf{x}_i^{\mathcal{W}}\|} - 1 \right|. \tag{4.4}$$

A 2D example of minimizing this energy is shown in Figure 4.5.

Fixed correspondences are responsible for reducing these tangential shifts and thereby improving the prism shapes. We determine them for some vertices at the beginning of the fit as explained in the following, and keep them fixed throughout the optimization. Since the alignment error increases faster if the distance between skin surface and skeleton wrap is small, we specify fixed correspondences for vertices on $\mathcal{S}$ that have a distance less than 3 cm to $\mathcal{W}$. For each such vertex we randomly sample points in the geodesic neighborhood of $\mathbf{x}_i^{\mathcal{W}}$ and select the one that minimizes (4.4) as fixed correspondence. Normal vectors of sample points are generated using barycentric Phong interpolation. To avoid interference of spatially close fixed correspondences, we add them in order of increasing distance to the skeleton, and skip all vertices having a distance smaller than 5 cm to all previously selected ones. In that way, we get

Figure 4.6: Standard non-rigid registration from skin to skeleton (left) results in a bad tangential alignment of corresponding triangles, causing sheared prisms, which we visualize by color-coding the alignment error (4.4). Using fixed correspondences reduces this error (center). Shifting closest point correspondences with bad alignment reduces the error even further (right).

a well distributed set of fixed correspondences, favoring those with a small skin-to-skeleton distance. Figure 4.6, center, shows that this already reduces the alignment error by a large amount.

Closest point correspondences can also drag vertices to locations with high alignment error. In each iteration of the non-rigid ICP, we compute $E_{\text{align}}\left(\mathbf{x}_i^{\mathcal{S}}, \mathbf{x}_i\right)$ for each vertex on $\mathcal{S}$ and its counterpart on the current state of $\mathcal{X}$. If this error exceeds a limit of 0.01, which corresponds to an angle deviation of 8° from the optimal angle, we sample the one-ring neighborhood of vertex $\mathbf{x}_i$ on $\mathcal{X}$, set $\mathbf{x}_i$ to the sample with minimal alignment error and update its closest point correspondence $\mathbf{t}_i$ on $\mathcal{W}$. This strategy reduces the alignment error even further, as shown in Figure 4.6, right.

Collision targets prevent $\mathcal{X}$ from intersecting $\mathcal{W}$, ensuring that in the converged state the surface $\mathcal{X}$ (i.e., $\mathcal{B}$, due to (4.1)) fully encloses $\mathcal{W}$. We therefore detect these collisions during the optimization, resolve them, and define a collision target for each colliding vertex. We use the exact continuous collision detection of Brochu et al. [2012] and, in case of a collision, we back-track the triangles' linear path from the current $\mathcal{X}$ to the initial $\mathcal{S}$ to find the non-colliding state closest to $\mathcal{X}$. This state defines collision targets $\mathbf{t}_i$ for colliding vertices $\mathbf{x}_i$, which replaces the other two types of target positions in all following steps of the minimization. In case of multiple collisions for the same vertex $\mathbf{x}_i$, we determine all non-colliding states separately and choose the $\mathbf{t}_i$ that is closest to the initial skin surface $\mathcal{S}$.

For the minimization of (4.1) we use the Projective framework of Bouaziz et al. [2012, 2014a], implemented through an adapted local/global solver from the ShapeOp library [Deuss et al. 2015] (see also Section 2.2). E_{reg} corresponds to the bending constraint (Equation (2.17)) and E_{fit} can be implemented using anchor constraints. We first initialize $\mathcal{X}$ with $\mathcal{S}$ and set $w_{\text{reg}} = w_{\text{fit}} = 1$. When the minimization converges, we update the initial Laplacians $\Delta\bar{\mathbf{x}}_i$ in (4.2) to the Laplacians $\Delta\mathbf{x}_i$ of the current solution $\mathcal{X}$ and decrease w_{reg} by a factor of

0.1. This is repeated until w_{reg} reaches 10^{-7}. In order to speed up the fitting process, we first remove high frequency details of the skin surface (e.g., nipples and navels) by Laplacian smoothing [Botsch et al. 2010] before computing the initial Laplacians $\Delta\bar{\mathbf{x}}_i$. Since we exclude head, hands, and toes from the layered template, those regions are fixed throughout the whole process. The alignment error of the resulting skeleton surface $\mathcal{B}$ is shown in Figure 4.3b.

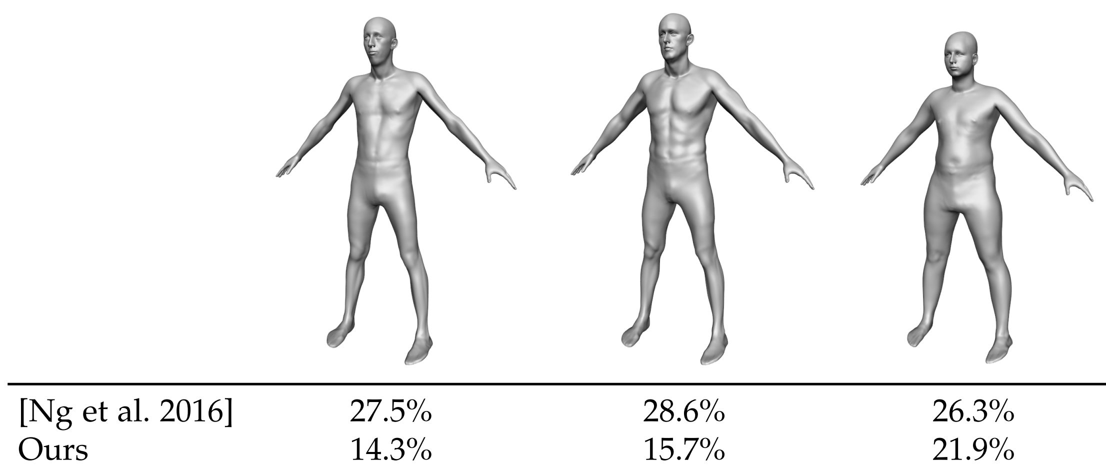
Due to the high resolution of the skeleton wrap and the continuous collision detection, the whole process is considerably slower (about three minutes) than the previous FPS-based skin shrinking approach but yields a much smoother result (shown in Figure 4.4). Note that this skin shrinking must just be executed twice, i.e., for the male and female template.

The Muscle Surface

We generate the muscle surface $\mathcal{M}$ by minimizing the same energy as in Equation (4.1), but using a different method for finding the correspondences $\mathbf{t}_i$ in Equation (4.3), which exploits that we already established correspondence between skin surface $\mathcal{S}$ and skeleton surface $\mathcal{B}$. We do not employ closest point correspondences, but instead set for each vertex $\mathbf{x}_i$ a *fixed* correspondence $\mathbf{t}_i$ to the first intersection of the line from skin vertex $\mathbf{x}_i^{\mathcal{S}}$ to skeleton vertex $\mathbf{x}_i^{\mathcal{B}}$ with the high-resolution muscle model [Zygote 2020]. If there is no intersection (e.g., at the knee), we set $\mathbf{t}_i = \mathbf{x}_i^{\mathcal{B}}$ and assign a lower weight w_i. When the minimization converges we decrease w_{reg}, and project the vertices of the current muscle surface $\mathbf{x}_i^{\mathcal{M}}$ to their corresponding skin-to-skeleton line from $\mathbf{x}_i^{\mathcal{S}}$ to $\mathbf{x}_i^{\mathcal{B}}$. Due to the collision handling, the resulting muscle surface $\mathcal{M}$ will enclose the high-resolution muscle model. To ensure that our volumetric elements always have a non-zero volume, even in regions where no muscles are located between skin and bone, we enforce a minimal distance of 1 mm to $\mathcal{B}$. The resulting muscle surface $\mathcal{M}$ is visualized in Figure 4.3c. Note that the muscle layer does not exclusively contain muscles: especially in the abdominal region, a large portion of the muscle layer is filled by organs. We therefore define a *muscle thickness map* that for each vertex i stores the accumulated length of the segments of the line $(\mathbf{x}_i^{\mathcal{S}}, \mathbf{x}_i^{\mathcal{B}})$ that are covered by muscles. This map will be used later in Section 4.2.4.

4.2.3 Estimating Fat Mass and Muscle Mass

Having generated the volumetric layered template, we want to be able to fit it to a given surface scan of a person. To regularize this under-determined problem, we first have to estimate how much of the person's soft tissue is explained by *fat mass* (FM) and *muscle mass* (MM), respectively. This is a challenging problem since we want to capture only a single surface scan of the person and



[Ng et al. 2016]	27.5%	28.6%	26.3%
Ours	14.3%	15.7%	21.9%

Figure 4.7: We applied the method of Ng et al. [2016] to estimate the body fat percentage of a skinny, a muscular and a moderately corpulent person (left to right). For comparison, we also give the results of our body fat estimator.

therefore cannot rely on information provided by additional hardware, such as a DXA scanner or a body fat scale. Kadleček et al. [2016] handle this problem by describing the person's shape primarily through muscles, i.e., by growing muscles as much as possible and defining the remaining soft tissue volume as fat. This strategy results in adipose persons having considerably more muscle mass than leaner people. Although there is a certain correlation between total body mass (and also BMI) and muscle mass – because the higher weight has a training effect especially on the muscles of the lower limbs [Tomlinson et al. 2016] – this general trend is not sufficient to define the body composition of a person. Like already mentioned in Section 4.1, there exist several methods for determining the body fat percentage from outer measures. While these approaches can work on certain target scans, we always found examples of obviously wrong BF estimations when we applied the different formulas to our scanned models of the CAESAR dataset [Robinette et al. 2002]. In Figure 4.7 we demonstrate this on three scans and their corresponding BF, determined through the method of Ng et al. [2016]. Here, the skinny person has a very high body fat prediction of 27.5%, even higher than the prediction of the moderately corpulent person. Moreover, the BF of muscular people is also often overestimated by existing methods that rely on outer measures.

Maalin et al. [2020] measured both FM and MM using a medical-grade eight-electrode bio-electrical impedance analysis and acquired a 3D surface scan. From this data, they built a model that can vary the shape of a person based on specified muscle or fat variation, similar to Piryankova et al. [2014]. Our model should perform the inverse operation, i.e., estimate FM and MM from a given surface scan. We train our model on their BeyondBMI dataset, which consists of scans of 100 men and 100 women captured in an approximate A-pose (see Figure 4.8), each annotated with FM, MM, and BMI.

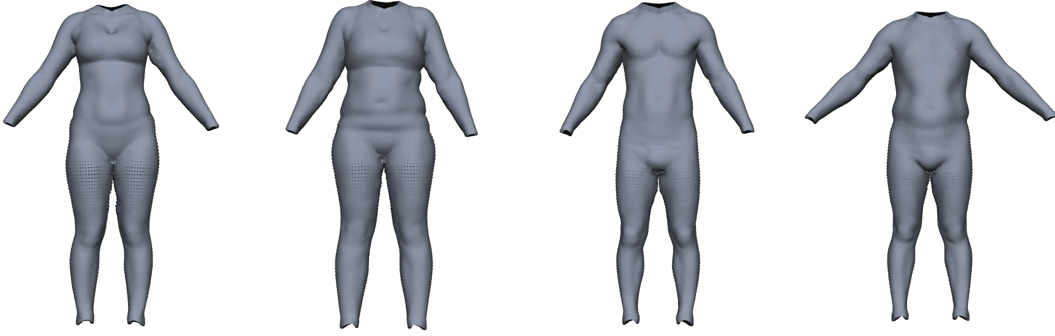


Figure 4.8: Examples for the BeyondBMI dataset provided by Maalin et al. [2020] consisting of scans of 100 men and 100 women, annotated with fat mass, muscle mass, and BMI. The scans lack geometric data for head, hands, and feet and are captured in approximate A-pose (with noticeable variation in pose).

By applying the surface fitting described in Section 4.2.1 to the BeyondBMI dataset, we make their scans compatible to our template and un-pose them to a common T-pose, thereby making any subsequent statistical analysis pose-invariant. After re-excluding the head, hands, and feet of our surface template, we are left with $M = 100$ surface meshes per sex that consist of $S = 7665$ vertices $\mathbf{x}_i$. We denote the jth training mesh by a vector of stacked vertex coordinates $\mathbf{X}_j \in \mathbb{R}^{3S}$ and perform PCA on the data matrix $\mathbf{X} = (\mathbf{X}_1, \ldots, \mathbf{X}_M) \in \mathbb{R}^{3S \times M}$. Let $\mathbf{P} \in \mathbb{R}^{3S \times M_{\text{sub}}}$ be the resulting basis of the subspace spanned by the first M_{sub} principal components and $\boldsymbol{\mu}$ the mean of the training data. Since the data is now pose-normalized, the dimensionality reduction can focus solely on differences in human body shape. As a result, our model only needs $M_{\text{sub}} = 12$ PCA components to explain 99.5 % of the data variance, while the original BeyondBMI dataset needs $M_{\text{sub}} = 24$ components to cover the same percentage due to noticeable variations in pose during the scanning process (see Figure 4.8).

We then perform linear regression to estimate FM and MM from PCA weights, as proposed by Hasler et al. [2009]. To this end, each scan $\mathbf{X}_j$ is projected to M_{sub}-dimensional PCA coordinates $\mathbf{c}_j = \mathbf{P}^\mathsf{T}(\mathbf{X}_j - \boldsymbol{\mu})$. We can now define the linear fat mass and muscle mass estimators

$$\text{FM}(\mathbf{c}) = \mathbf{a}_{\text{FM}}^\mathsf{T}\mathbf{c} + b_{\text{FM}} \tag{4.5}$$

$$\text{MM}(\mathbf{c}) = \mathbf{a}_{\text{MM}}^\mathsf{T}\mathbf{c} + b_{\text{MM}} \tag{4.6}$$

and determine its coefficients $\mathbf{a}_{\text{FM}}, \mathbf{a}_{\text{MM}} \in \mathbb{R}^{k_{\text{sub}}}$ and $b_{\text{FM}}, b_{\text{MM}} \in \mathbb{R}$ through linear least squares fitting of $\mathbf{c}_j$ to their annotated fat mass FM_j and muscle mass MM_j. For a new scan we are now able to predict fat mass and muscle mass by registering the surface template to the scan, yielding the stacked coordinate vector $\mathbf{Y}$, transforming it to PCA coordinates $\mathbf{c} = \mathbf{P}^\mathsf{T}(\mathbf{Y} - \boldsymbol{\mu})$, and applying the regressors (4.6), (4.5).

For a first evaluation of this model, we perform a leave-one-out test on the BeyondBMI dataset, i.e., excluding each scan once, building the regressors as described above from the remaining $M-1$ scans, and measuring the mean absolute error of the predictions. We again use $M_{\text{sub}} = 12$ PCA components, as this covers almost all the variance present in the dataset and gives the linear regression enough degrees of freedom. The leave-one-out evaluation yields a mean absolute error (*MAE*) of $\text{MAE}_{\text{FM}} = 1.20\,\text{kg}$ (± 0.93) and $\text{MAE}_{\text{MM}} = 1.01\,\text{kg}$ (± 0.79) for the female dataset, where the fat mass lies in the range 6.27 kg to 34.71 kg and the muscle mass in the range 21.59 kg to 31.63 kg. The linear regression shows an average R^2 score of 0.84, confirming that there is indeed a linear relationship between PCA coordinates and the FM/MM measurements. Performing the leave-one-out test on the male dataset shows similar values: $\text{MAE}_{\text{FM}} = 1.37\,\text{kg}$ (± 1.00) and $\text{MAE}_{\text{MM}} = 1.46\,\text{kg}$ (± 1.11), fat mass in the range 3.91 kg to 27.83 kg, muscle mass in the range 31.51 kg to 51.20 kg, and an average R^2 score of 0.88.

We compared the linear model to a support vector regression (using scikit-learn [Pedregosa et al. 2011] with default parameters and RBF kernels), but in contrast to Hasler et al. [2009] we found that for the BeyondBMI dataset this approach performs considerably worse: $\text{MAE}_{\text{FM}} = 2.98\,\text{kg}$ (± 2.85) and $\text{MAE}_{\text{MM}} = 1.24\,\text{kg}$ (± 1.02) with an average R^2 score of 0.64 for the female dataset, and $\text{MAE}_{\text{FM}} = 2.63\,\text{kg}$ (± 2.60) and $\text{MAE}_{\text{MM}} = 2.48\,\text{kg}$ (± 1.82) with an average R^2 score of 0.58 for the male dataset. We therefore keep the simpler and better-performing linear regression model.

Whenever we fit the volumetric model to a given body scan, as explained in the next section, we first use the proposed linear regressors to estimate the person's fat mass and muscle mass and use this information to generate the muscle and fat layers in Section 4.2.4.

4.2.4 Fitting the Volumetric Template to Surface Scans

Given a surface scan of a person, we create a personalized three-layered anatomical model through the following steps: First, we fit our *surface* template to the scan, which establishes one-to-one correspondence with the volumetric template and puts the scan into the same T-pose as the template (Section 4.2.1). After this pre-processing, we deform the *volumetric* template to match the scanned subject. To this end, we adjust global scaling and per-bone local scaling, such that body height and limb lengths of template and scan match, which is explained in the first part of this section. This is followed by a quasi-static deformation of the volumetric template that considers the skin surface $\mathcal{S}$ as hard constraint and yields the skeleton surface $\mathcal{B}$ through energy minimization. Given the skin surface $\mathcal{S}$, the bone surface $\mathcal{B}$, and the estimated fat mass and muscle mass from our regressors (Section 4.2.3), the muscle

surface $\mathcal{M}$ is determined, as detailed in the third part of this section. Having transferred all three layer surfaces to the scan we show how to warp the detailed anatomical structures of the template into the personalized model.

Global and Local Scaling

Registering the surface template to the scan allows us to put the latter into the same alignment (rotation, translation) and the same pose as the volumetric template. The next step is to correct the mismatch in scale by adjusting body height and limb lengths of the volumetric template.

This scaling does influence all three of the template's surfaces. Since the shape of the skeleton surface $\mathcal{B}$ will be constrained to the result after scaling, we have to keep the scaled bone lengths and bone diameters within a plausible range. The *length* of prominent bones, like the upper arm or the upper leg (humerus and femur), can be well approximated by measures on the surface of the model. However, finding the correct bone *diameters* is impossible without measurements of the subject's interior. In particular for corpulent or adipose subjects, the subcutaneous fat layer dominates the appearance of the skin surface, preventing us from precisely determining the bone diameters from the surface scan. It has been shown that there is a moderate correlation of bone length and bone diameter [Aydin Kabakci et al. 2017; Ziylan and Murshid 2002] and (obviously) a strong correlation of body height and bone length [Dayal et al. 2008]. We therefore perform a *global isotropic* scaling depending on body height (affecting bone lengths and diameters) as well as *local anisotropic* scaling depending on limb lengths (affecting bone lengths only).

The global scaling is determined from the height difference of scan and template and is applied to all vertices of the template model. It therefore scales all bone lengths and bone diameters uniformly. Directly scaling with the height ratio of scan and template, however, can result in too thin or too thick bones for extreme target heights. Thus, we damp the height ratio $d_h = h_{\text{scan}}/h_{\text{template}}$ by $d_h \leftarrow 0.5(d_h - 1) + 1$, which means that a person that is 20% taller than the template will have 10% thicker bones than the template. This heuristic results in visually plausible bone diameters for all our scanned subjects.

After the global scaling, the local scaling further adjusts the limb lengths of the template to match those of the scan. The (fully rigged) surface template has been fit to both the scan (Section 4.2.1) and the template (Section 4.2.2). This fit provides a simple skeleton graph for both models. We use the length mismatch of the respective skeleton graph segments to determine the required scaling for upper and lower arms, upper and lower legs, feet, and torso. We scale these limbs in their corresponding bone directions (or the spine direction for the torso) using the bone stretching of Kadleček et al. [2016]. As mentioned before, this changes the limb lengths but not the bone diameters.

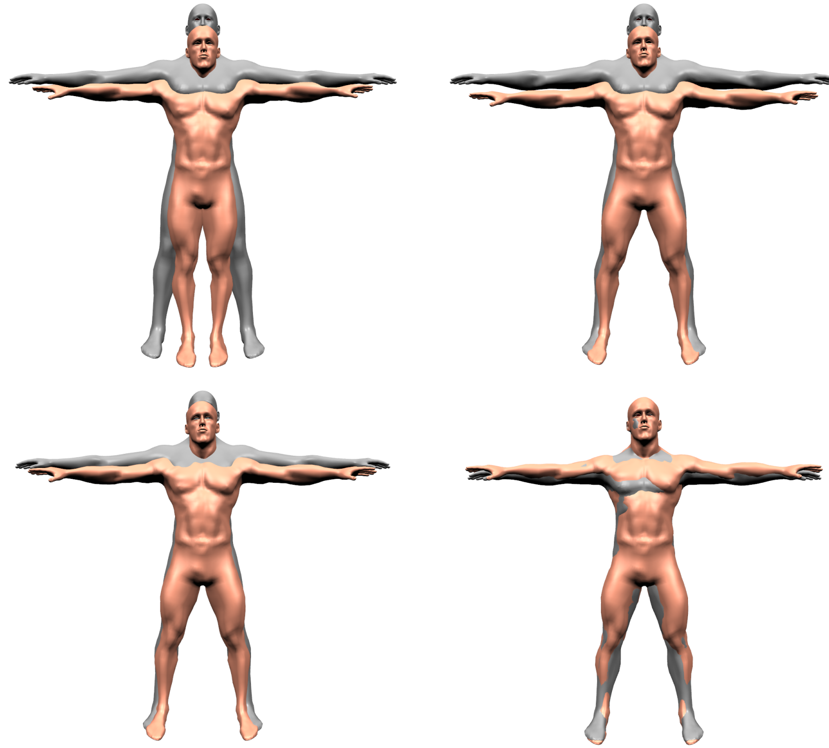


Figure 4.9: Matching template (grey) and scan (red): First, we put the scan into the same pose as the template (top right). Body height and limb lengths of the template are then adjusted by a global uniform scaling (bottom left), followed by local scaling for limbs and spine (bottom right).

This two-step scaling process is visualized in Figure 4.9. As a result, the scaled template matches the scan with respect to alignment, pose, body height, and limb lengths. Its three surfaces, which we denote by $\bar{\mathcal{S}}$, $\bar{\mathcal{M}}$, and $\bar{\mathcal{B}}$, provide a good initialization for the optimization-based fitting described in the following.

Skeleton Fitting

Given the scaled template of the previous step, we now fit the skin surface $\mathcal{S}$ and skeleton surface $\mathcal{B}$ by minimizing a quasi-static deformation energy. The template's skin surface $\mathcal{S}$ should match the (skin) surface of the scan and since both meshes have the same triangulation, we can simply copy the skin vertex positions from the scan to the template and consider them as hard Dirichlet constraints. It therefore remains to determine the vertex positions of the skeleton surface $\mathcal{B}$, such that the soft tissue enclosed between skin surface $\mathcal{S}$ and skeleton surface $\mathcal{B}$ (fat + muscles, which we call *flesh*) deforms in a physically plausible manner. This is achieved by minimizing a quasi-static energy consisting of three terms:

$$E(\mathcal{B}) = w_{\text{reg}} E_{\text{reg}}(\mathcal{B},\bar{\mathcal{B}}) + w_{\text{flesh}} E_{\text{flesh}}(\mathcal{B},\mathcal{S}) + w_{\text{coll}} E_{\text{coll}}(\mathcal{B},\mathcal{S}) \,. \tag{4.7}$$

The first term is responsible for keeping the skeleton surface (approximately) rigid and uses the same formulation as Equation (4.2), with $\bar{\mathcal{B}}$ and $\mathcal{B}$ denoting the skeleton surface before and after the deformation, respectively. We employ a soft constraint with high weight w_{reg} instead of deforming bones in a strictly rigid manner [Kadleček et al. 2016], since we noticed that for very thin subjects the skeleton surface might otherwise protrude the skin surface and therefore a certain amount of bone deformation is required. We also do not penalize deviation from rigid or affine transformations as Dicko et al. [2013] since this penalizes smooth shape deformation in the same way as locally flipped triangles, which we observed to cause artifacts in the skeleton surface. The discrete bending energy of Equation (4.2), with a suitably high regularization weight w_{reg}, allows for moderate *smooth* deformations and gave better results in our experiment.

The second term prevents strong deformations of the prism elements $p \in \mathcal{P}$, spanned by corresponding triangles $(\mathbf{x}_i^{\mathcal{S}}, \mathbf{x}_j^{\mathcal{S}}, \mathbf{x}_k^{\mathcal{S}})$ on the skin surface and $(\mathbf{x}_i^{\mathcal{B}}, \mathbf{x}_j^{\mathcal{B}}, \mathbf{x}_k^{\mathcal{B}})$ on the skeleton surface. While we penalize deformation of the top/bottom triangles, we allow changes of prism heights, i.e., anisotropic scaling in the direction from surface to bone, since otherwise the soft tissue layer cannot grow to bridge the gap from the skeleton surface to the skin surface. This behavior is modeled by the anisotropic strain limiting energy

$$E_{\text{flesh}}(\mathcal{B}, \mathcal{S}) \;=\; \frac{1}{2} \sum_{p \in \mathcal{P}} \left\| \mathbf{F}_p - \mathbf{R}_p \mathbf{B}_p \tilde{\mathbf{S}}_p \mathbf{B}_p^{\mathsf{T}} \right\|_F^2, \tag{4.8}$$

where $\mathbf{F}_p \in \mathbb{R}^{3\times3}$ is the deformation gradient of the element p, i.e., the linear part of the best affine transformation that maps the un-deformed prism $\bar{p}$ to the deformed prim p in the least squares sense (see Equation (2.18) from Section 2.2.2). Polar decomposition $\mathbf{F}_p = \mathbf{R}_p \mathbf{S}_p$ decomposes $\mathbf{F}_p$ into a rotation $\mathbf{R}_p$ and scale/shear $\mathbf{S}_p$ [Shoemake and Duff 1992]. $\mathbf{B}_p$ is a rotation matrix that aligns the z-axis with the surface normal of the prism's corresponding skin triangle, i.e., the direction in which we allow stretching. The matrix $\tilde{\mathbf{S}}_p$ represents the anisotropic scaling $\tilde{\mathbf{S}}_p = \operatorname{diag}(1, 1, \alpha)$, where $\alpha \in [\alpha_{\min}, \alpha_{\max}]$ allows to tune the amount of stretching in normal direction that should be allowed. We use $\alpha_{\min} = 0.2$ and $\alpha_{\max} = 5.0$ to allow stretching and compression of the element up to a factor of five before the energy of this element increases.

Third, we detect the set of collisions $\mathcal{C}$, defined as vertices of the skeleton surface $\mathcal{B}$ that are outside of the skin surface $\mathcal{S}$. For these colliding vertices we add a collision penalty term

$$E_{\text{coll}}(\mathcal{B}, \mathcal{S}) = \frac{1}{2} \sum_{\mathbf{x}_i \in \mathcal{C}} w_i \left\| \mathbf{x}_i - \boldsymbol{\pi}_{\mathcal{S}}(\mathbf{x}_i) \right\|^2, \tag{4.9}$$

where $\boldsymbol{\pi}_{\mathcal{S}}(\mathbf{x}_i)$ is the projection of the colliding vertex $\mathbf{x}_i$ to a position 2 mm beneath the closest triangle on the skin surface $\mathcal{S}$. The weight w_i is initialized

to 1 when the corresponding vertex collides for the first time, and is increased by 1 each time the minimization was not able to resolve the collision.

In order to perform the iterative minimization of (4.7), we again use a (static) PD solver (Section 2.2). The regularization energy can be implemented as a bending constraint and collisions via anchor constraints (Section 2.2.1). E_{flesh} can be incorporated by modifying the polyhedron strain constraint from Section 2.2.2: In the local projection step, we determine $\mathbf{F}_p$ via (2.18), compute its polar decomposition $\mathbf{F}_p = \mathbf{R}_p\mathbf{S}_p$ and determine the current amount of stretching $\alpha = (\mathbf{B}_p^{\mathsf{T}}\mathbf{S}_p\mathbf{B}_p)_{3,3}$. This is clamped to the range $[\alpha_{\min}, \alpha_{\max}]$ to determine the constraint's projection $\mathbf{R}_p\mathbf{B}_p\tilde{\mathbf{S}}_p\mathbf{B}_p^{\mathsf{T}}$. We set the weights $w_{\text{reg}} = 0.1$, $w_{\text{flesh}} = 0.01$, and $w_{\text{coll}} = 50$. The minimization is iterated until convergence. In the converged state, we detect collisions and add the corresponding collision constraints to the system. Minimization and collision detection are alternatingly repeated until no collisions are found in a converged solution. For all tested subjects, the skeleton fitting always reached this final stage in less than ten seconds on average.

Muscle Fitting

Having determined the skin surface $\mathcal{S}$ and skeleton surface $\mathcal{B}$, we now fit the muscle surface $\mathcal{M}$ in between $\mathcal{S}$ and $\mathcal{B}$, such that the ratio of fat mass (FM) and muscle mass (MM) resembles the values estimated by our regressors (Section 4.2.3). This muscle fitting is accomplished in three steps: First, we transfer the template's muscle surface to the fitted skin and skeleton surfaces, which we call *average muscle layer* in the following. Second, we grow and shrink the muscles as much as anatomically and physically plausible, yielding the *minimum* and *maximum muscle layers*. Third, we find a linear interpolation between these two extremes that matches the predicted fat mass and muscle mass as good as possible.

The average muscle surface is transferred from the *scaled* template $\overline{\mathcal{M}}$ (Section 4.2.4, Figure 4.9) by minimizing an energy consisting of two objectives:

$$E(\mathcal{M}) = w_{\text{reg}}E_{\text{reg}}(\mathcal{M},\overline{\mathcal{M}}) + w_{\text{line}}E_{\text{line}}(\mathcal{M},\mathcal{B},\mathcal{S})\,. \tag{4.10}$$

The first term tries to preserve the shape of the scaled template's muscle surface $\overline{\mathcal{M}}$ and is modeled using the regularization energy of Equation (4.2). The second term preserves the template's property that each muscle vertex $\mathbf{x}_i^{\mathcal{M}}$ resides on the line segment from its corresponding skeleton vertex $\mathbf{x}_i^{\mathcal{B}}$ to its skin vertex $\mathbf{x}_i^{\mathcal{S}}$, by penalizing the squared distance from that line:

$$E_{\text{line}}(\mathcal{M},\mathcal{B},\mathcal{S}) = \frac{1}{2}\sum_{\mathbf{x}_i\in\mathcal{M}} \left\| \mathbf{x}_i - \pi_L\left(\mathbf{x}_i, \mathbf{x}_i^{\mathcal{B}}, \mathbf{x}_i^{\mathcal{S}}\right) \right\|^2, \tag{4.11}$$

where $\pi_L\left(\mathbf{x}_i, \mathbf{x}_i^{\mathcal{B}}, \mathbf{x}_i^{\mathcal{S}}\right)$ is the projection of $\mathbf{x}_i$ onto the line $(1-\beta)\mathbf{x}_i^{\mathcal{B}} + \beta\mathbf{x}_i^{\mathcal{S}}$, $\beta \in [0,1]$. Minimizing (4.10) leads to flat abdominal muscles like in the

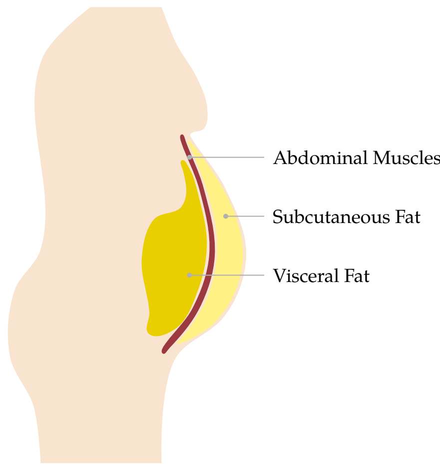


Figure 4.10: In addition to our fat layer holding the subcutaneous fat that resides between skin and muscle surface, there is visceral fat that accumulates below the abdominal muscles and between organs. Visceral fat pushes the muscles outwards and must therefore be taken into account when fitting the belly region of the muscles for obese targets.

template model, which is unrealistic for corpulent or adipose subjects, because the majority of body fat resides in two different fat tissues: the *subcutaneous fat*, which is located between skin and muscle surface, and the *visceral fat*, which accumulates in the abdominal cavity, i.e., under the muscle layer (see Figure 4.10). Since the bulging of the abdomen due to visceral fat causes a bulging of the belly, we inversely want the abdominal muscles in $\mathcal{M}$ to slightly bulge out in case of a belly bulge in the skin surface $\mathcal{S}$. The latter is a combined effect of visceral and subcutaneous fat in the abdominal region. We model this effect by adjusting E_{line} for each vertex $\mathbf{x}_i$ in the abdominal region. Instead of using the full interval $\beta \in [0,1]$, we adjust the lower boundary to $\beta_{\min} = \|\bar{\mathbf{x}}_i^{\mathcal{M}} - \bar{\mathbf{x}}_i^{\mathcal{B}}\| / \|\bar{\mathbf{x}}_i^{\mathcal{S}} - \bar{\mathbf{x}}_i^{\mathcal{B}}\|$, i.e., the parameter β where for the (scaled) template the muscle surface intersects the line. Note that E_{line} has to compete with the regularization E_{reg} and therefore, the muscle bulge in the abdominal region will be less dominant than $\beta_{\min}$ would suggest. This is especially important for targets with *multiple* belly bulges that are mainly caused by subcutaneous fat. The regularization prevents these higher frequency bulges from being present in $\mathcal{M}$.

For the iterative minimization of (4.10) we use PD bending constraints and modified anchor constraints performing the line projection in the local step. We initialize $\mathcal{M}$ with $\overline{\mathcal{M}}$ and set $w_{\text{reg}} = 0.01$, $w_{\text{line}} = 1.0$. When the minimization converges, we update the Laplacians in E_{reg} to those of the current solution and decrease w_{reg} by a factor of 0.5. This is iterated until the maximal distance of a vertex to its bone-to-skin line (see (4.11)) is less than 0.2 mm. Lastly, we project each vertex onto its corresponding bone-to-skin line to get a perfect alignment.

Having transferred the average muscle surface, we next grow/shrink muscles as much as possible in order to define the maximum/minimum muscle surfaces. Since certain muscle groups might be better developed than others, we perform the muscle growth/shrinkage separately for the major muscle groups, namely upper legs (including buttocks), lower legs, upper arms, lower

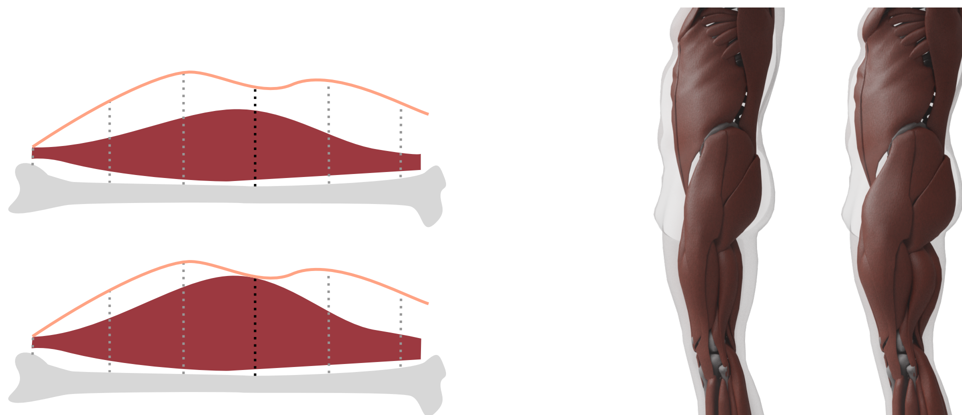


Figure 4.11: Left: When computing the maximum muscle surface, we move muscle vertices toward the skin by an amount proportional to their *muscle potential*, which for each vertex is the length of the dotted line intersected with the muscle. The vertex with the black dotted line defines the maximum allowed stretch in this example. Right: An example of our minimum and maximum muscle layers for the same target. These two define the lower and upper limit for the muscle mass and vice versa for the fat mass.

arms, chest, abdominal muscles, shoulders, and back. Muscles are built from fibers and grow perpendicular to the fiber direction. In all cases relevant for us, the fibers are approximately perpendicular to the direction from $\mathcal{M}$ to $\mathcal{S}$, thus muscle growth/shrinkage will move vertices $\mathbf{x}_i^{\mathcal{M}}$ along the line from $\mathbf{x}_i^{\mathcal{B}}$ to $\mathbf{x}_i^{\mathcal{S}}$. The amount of vertex movement along these directions is proportional to the muscle thickness map of the template (computed in Section 4.2.2). We determine how much we can grow a muscle before it collides with the skin surface in the thicker parts of the muscle (instead of close to its endpoints where it connects to the bone). Figure 4.11 shows an example, where the leftmost muscle vertex is already close to the skin and would prevent any growth if we took endpoint regions into account. For each muscle group, we also define an upper limit for muscle growth that prevents the muscles from increasing further, even if the skin distance is large (e.g., for adipose subjects). For determining the minimal muscle surface, we repeat the process in the opposite direction (towards the skeleton surface). To prevent distortions of the muscle surface, we do not set the new vertex positions directly, but instead use them as target positions $\mathbf{t}_i$ (using Equation (4.3)) and regularize with Equation (4.2). Figure 4.11 (right) shows an example of minimum/maximum muscle surfaces computed by this procedure.

We determine the final muscle surface $\mathcal{M}$ by linear interpolation between the minimum and maximum muscle surfaces, such that the resulting fat mass FM and muscle mass MM match the values predicted by the regressors (denoted by FM* and MM*) as good as possible. To this end, we have to compute FM and MM from an interpolated muscle surface $\mathcal{M}$. We can compute the volume V_{FL} of the fat layer (between $\mathcal{S}$ and $\mathcal{M}$) and the volume

V_{ML} of the muscle layer (between $\mathcal{M}$ and $\mathcal{B}$) and convert these to masses m_{FL} and m_{ML} by multiplying with the (approximate) fat and muscle densities $\rho_{\mathrm{F}} = 0.9\,\mathrm{kg/l}$ and $\rho_{\mathrm{M}} = 1.1\,\mathrm{kg/l}$, respectively.

The resulting masses require some corrections though: First, we have to add the visceral fat (VAT), which is not part of our fat layer but resides in the abdominal cavity. We estimate the VAT mass m_{VAT} by computing the difference of the cavity volumes of the scaled template and of the final fit, thereby assuming a negligible amount of VAT in the template. Second, we subtract the skin mass m_{skin} from the fat layer mass. We assume an average skin thickness of 2 mm, multiply this by the skin's surface area and the density ρ_{F}. Third, our fat layer includes the complete reproductive apparatus in the crotch region. This volume is even larger due to the underwear that was worn during scanning and incorrectly increases the fat layer mass by m_{crotch}. Our corrected fat mass is then

$$\mathrm{FM} = m_{\mathrm{FL}} + m_{\mathrm{VAT}} - m_{\mathrm{skin}} - m_{\mathrm{crotch}}. \tag{4.12}$$

We correct the muscle mass by subtracting the mass m_{abd} of the abdominal cavity, which is incorrectly included in the muscle layer. The remaining muscle mass is always too small even when using the maximum muscle surface, due to all muscles not considered in the muscle layer, such as heart, face, and hand muscles or the diaphragm. It is known that the lean body mass roughly scales with the squared body height [Heymsfield et al. 2011], which is the basis of the well known body and muscle mass indices. We analogously assume the missing muscle mass to be proportional to the squared height h of the subject, i.e., $m_h = \kappa h^2$, with a constant κ to be determined later. The corrected muscle mass is therefore

$$\mathrm{MM} = m_{\mathrm{ML}} - m_{\mathrm{abd}} + m_h. \tag{4.13}$$

There are other terms like the fat of head, hands, and toes, which could be added, or the volume of blood vessels and tendons, which could be subtracted. We assume those terms to be negligible.

Since the total volume of the soft tissue layer $V_{\mathrm{ST}} = V_{\mathrm{ML}} + V_{\mathrm{FL}}$ is constant, the muscle layer mass m_{ML} is coupled to the fat layer mass m_{FL} via $m_{\mathrm{ML}} = (V_{\mathrm{ST}} - V_{\mathrm{FL}})\rho_{\mathrm{M}}$. We want to compute the fat layer mass such that the resulting FM and MM minimize the least squares error to the values predicted by the regressor: $E_{\mathrm{comp}} = (\mathrm{FM} - \mathrm{FM}^*)^2 + (\mathrm{MM} - \mathrm{MM}^*)^2$. Inserting (4.12) and (4.13) into E_{comp}, rewriting m_{ML} in terms of m_{FL}, and setting the derivative $\mathrm{d}E_{\mathrm{comp}}/\mathrm{d}m_{\mathrm{FL}} = 0$ yields the optimal fat layer mass

$$m_{\mathrm{FL}} = \frac{\mathrm{FM}^* - m_{\mathrm{VAT}} + m_{\mathrm{skin}} + m_{\mathrm{crotch}} + \rho\left(V_{\mathrm{ST}}\,\rho_M - m_{\mathrm{abd}} + m_h - \mathrm{MM}^*\right)}{1 + \rho^2}, \tag{4.14}$$

with the density ratio $\rho = \rho_{\mathrm{M}}/\rho_{\mathrm{F}}$.

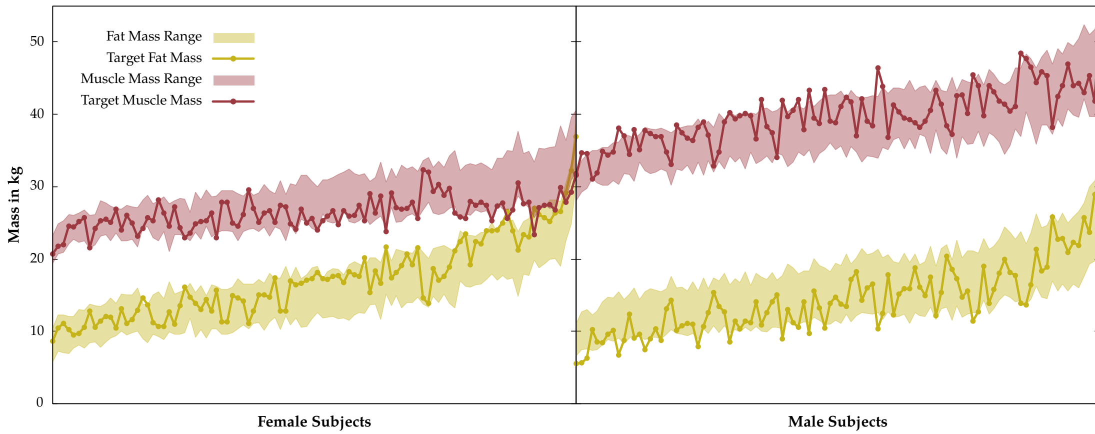


Figure 4.12: Measured muscle and fat masses provided by the BeyondBMI dataset for its female and male subjects, plotted on top of the possible ranges defined by our minimum and maximum muscle surfaces. Note that our minimal fat mass is coupled to the maximal muscle mass and vice versa.

The minimum/maximum muscle surface yields a maximum/minimum fat layer mass. The optimized fat layer mass is clamped to meet this range, thereby defining the final fat layer mass. We then choose the linear interpolant between the minimum and maximum muscle surface that matches this fat mass, which we find through bisection search.

We did this for the scans of 100 men and 100 women from the BeyondBMI dataset [Maalin et al. 2020], where we know the values for FM and MM from measurements, and optimized the value of κ for this dataset, yielding $\kappa_{\text{male}} = 1.5$ and $\kappa_{\text{female}} = 1.0$. This is plausible since women in general have a lower muscle mass. For instance, the average measured muscle mass of the male subjects in the dataset is indeed 50% higher than the average MM for the female subjects. The mean absolute errors (MAE) for the BeyondBMI dataset are $\text{MAE}_{\text{MM}} = 0.37\,\text{kg}\ (\pm 0.31)$, $\text{MAE}_{\text{FM}} = 0.46\,\text{kg}\ (\pm 0.38)$ for the female subjects and $\text{MAE}_{\text{MM}} = 0.46\,\text{kg}\ (\pm 0.39)$, $\text{MAE}_{\text{FM}} = 0.57\,\text{kg}\ (\pm 0.48)$ for the male subjects. Figure 4.12 shows how well our model can adjust to the target values of muscle and fat mass. All values are inside or at least close to the predicted possible range of minima and maxima. Moreover, in most cases the muscle/fat mass values for the same person split the two ranges at about an inverse point (e.g., close to maximum muscle and close to minimum fat), which leads to the low errors stated above.

Transferring Original Anatomical Data

After fitting the skin surface $\mathcal{S}$ to the scan and transferring the skeleton surface $\mathcal{B}$ and the muscle surface $\mathcal{M}$ into the scan, the final step is to transform the high-resolution anatomical details (Zygote's bone and muscle models in our case) from the volumetric template to the scanned subject. We implement

this in an efficient and robust manner as a mesh-independent space warp $\mathbf{d}\colon \mathbb{R}^3 \to \mathbb{R}^3$ that maps the original template's skin surface $\hat{\mathcal{S}}$, muscle surface $\hat{\mathcal{M}}$, and skeleton surface $\hat{\mathcal{B}}$ (all marked with a hat) to the scanned subject's layer surfaces $\mathcal{S}$, $\mathcal{M}$, and $\mathcal{B}$, respectively. All geometry that is embedded in between these surfaces will smoothly be warped from template to scan.

Dicko et al. [2013] also employ a space warp for their anatomy transfer, which they discretized by interpolating values $\mathbf{d}_{ijk}$ on a regular 3D grid constructed around the object. Their space warp is computed by interpolating the skin deformation $\hat{\mathcal{S}} \mapsto \mathcal{S}$ on the boundary and being harmonic in the interior (i.e., $\Delta\mathbf{d} = 0$), which requires the solution of a large sparse Poisson system for the coefficients $\mathbf{d}_{ijk}$.

We follow the same idea, but use a space warp based on triharmonic radial basis functions (RBFs) [Botsch and Kobbelt 2005], which have been shown to yield higher quality deformations with lower geometric distortion than many other warps (including FEM-based harmonic warps) [Sieger et al. 2013]. The RBF warp is defined as a sum of n RBF kernels and a linear polynomial:

$$\mathbf{d}(\mathbf{x}) \;=\; \sum_{j=1}^{n} \mathbf{w}_j \varphi_j(\mathbf{x}) + \mathbf{a}\mathbf{x} + \mathbf{b}, \tag{4.15}$$

where $\mathbf{w}_j \in \mathbb{R}^3$ is the coefficient of the jth radial basis function $\varphi_j(\mathbf{x}) = \varphi(\left\|\mathbf{x} - \mathbf{c}_j\right\|)$, which is centered at $\mathbf{c}_j \in \mathbb{R}^3$. As kernel function we use $\varphi(r) = r^3$, leading to highly smooth triharmonic warps ($\Delta^3\mathbf{d} = 0$). The term $\mathbf{a}\mathbf{x} + \mathbf{b}$ is a linear polynomial ensuring linear precision of the warp.

In order to warp the high-resolution *bone model* from the template to the scan, we setup the RBF warp to reproduce the deformation $\hat{\mathcal{B}} \mapsto \mathcal{B}$. To this end, we select 5000 vertices $\hat{\mathbf{x}}_i \in \hat{\mathcal{B}}$ from the template's skeleton surface by farthest point sampling. The corresponding vertices on the scan's skeleton surface are denoted by $\mathbf{x}_i \in \mathcal{B}$. At these vertices $\hat{\mathbf{x}}_i$ the deformation function $\mathbf{d}(\hat{\mathbf{x}}_i)$ should interpolate the displacements $\mathbf{d}_i = \mathbf{x}_i - \hat{\mathbf{x}}_i$. These constraints lead to a dense, symmetric, but indefinite $(n+4) \times (n+4)$ linear system, which we solve for the coefficients $\mathbf{w}_1, \ldots, \mathbf{w}_n, \mathbf{a}, \mathbf{b}$ using the LU factorization of Eigen [Guennebaud et al. 2018]. Note that all coefficients only depend on properties of the template and can therefore be pre-computed and stored. The resulting RBF warp $\mathbf{d}$ then transforms each vertex $\mathbf{x}$ of the high-resolution bone model as $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{d}(\mathbf{x})$. This process can trivially be parallelized over all model vertices, which we implement using OpenMP. We follow the same procedure to warp the high-resolution *muscle model*, but collect 7000 constraints from the vertices $\hat{\mathbf{x}}_i \in \mathcal{S} \cup \mathcal{M}$ of the skeleton and muscle surfaces, since these enclose the muscle layer.

4.3 RESULTS AND APPLICATIONS

Generating a personalized anatomical model for a given surface scan of a person consists of the following steps: First, the surface template is registered to the scanner data (triangle mesh or point cloud) as described in Section 4.2.1 following Achenbach et al. [2017]. After manually selecting 10–20 landmarks, this process takes about 50 sec. Fitting the surface template establishes dense correspondence with the surface of the volumetric template and puts the scan into the same T-pose as the volumetric template. Fitting the volumetric template by transferring the three layer surfaces (Section 4.2.4) takes about 15 sec. Transferring the high-resolution anatomical models of bones and muscles (145k vertices) takes about 0.5 sec (we pre-computed the warp coefficients for each template, which takes 4.5 sec). Timings were measured on a desktop workstation, equipped with an Intel Core i9 10900X CPU and a Nvidia RTX 2080 TI GPU except for the surface template registration (Section 4.2.1), which was carried out on a different machine equipped with an Intel Core i9 10850K and a Nvidia RTX 3070.

Dicko et al. [2013] and Kadleček et al. [2016] are the two approaches most closely related to ours. Dicko et al. [2013] also use a space warp for transferring anatomical details, but since they only use the skin surface as constraint, the interior geometry can be strongly distorted. To prevent this, they restrict bones to affine transformations, which, however, might still contain unnatural shearing modes and implausible scaling. Our space warp yields a higher smoothness due to the use of C^∞ RBF kernels instead of C^0 trilinear interpolation. It reduces unnatural distortion of bones and muscles by using the interior layer surfaces as constraints instead of the skin surface, and by optimizing these layers w.r.t. anatomical distortion. In Figure 4.13 we compare the result of a harmonic warp using only the skin surface as constraint to our

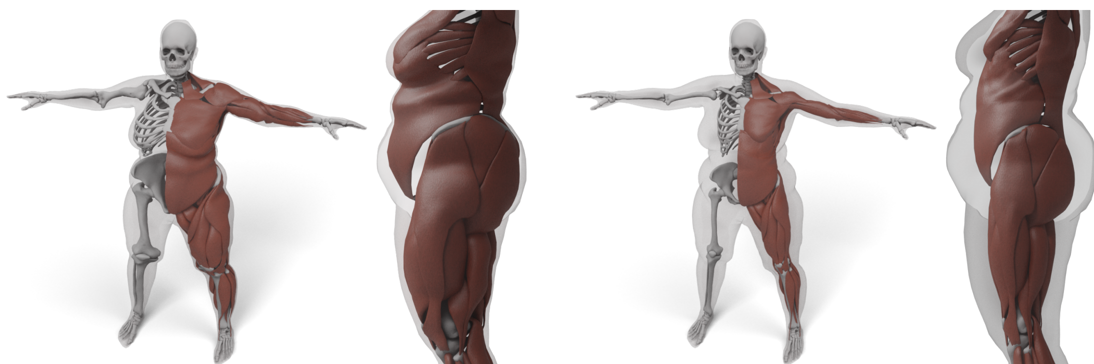


Figure 4.13: Result of transferring the anatomy by using just the skin layer and a harmonic basis (left). Here, both muscles and bones deform too much to fit overweight targets. We use the additional muscle and skeleton layer and a triharmonic basis (right) to prevent unnatural deformations.

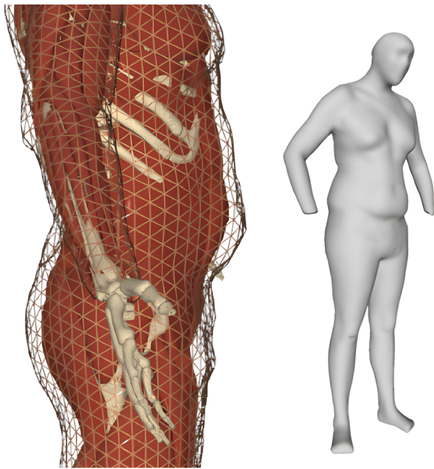


Figure 4.14: The approach of Kadleček et al. [2016] can result in intersections between skeleton and muscles as shown here at the ribcage, where the ribs cut through the muscle layer. By using our layered models, we can successfully avoid any interpenetration of the different layers (and therefore its contained structures) when fitting to various body types. The image was taken from Kadleček et al. [2016], Figure 13.

three-layered, triharmonic warp. The former shows drastic and unrealistic deformations of both muscles and bones while our approach solves these issues. Note that additionally restricting the bones to affine transformations like Dicko et al. [2013] would still produce unnaturally thick bones (e.g., the upper leg bone) and muscles.

Compared to Kadleček et al. [2016], we require just a single input scan, since we infer (initial guesses for) joint positions and limb lengths from the full-body PCA of Achenbach et al. [2017]. Putting the scan into T-pose prevents us from solving for bone geometry and joint angles simultaneously, which makes our approach much faster than theirs (15 sec vs. 30 min). Moreover, our layered model yields a conforming volumetric tessellation with constant and homogeneous per-layer materials, which more effectively prevents bones from penetrating skin or muscles. In their approach the bones (especially the ribcage) often intersect the muscle layer as mentioned by Kadleček et al. [2016] as a limitation of their work. We show one of their examples to demonstrate this problem in Figure 4.14. Furthermore, we automatically derive the muscle/fat body composition from the surface scan, which yields more plausible results than growing muscles as much as possible [Kadleček et al. 2016], since the latter tends to overestimate the amount of muscles for corpulent people. Our model extracts muscle and fat masses using data of real humans and can therefore adopt to the variety of human shapes (low FM and high MM, high FM and low MM, and everything in between). Finally, we support both male and female subjects by employing individual anatomical templates and muscle/fat regressors for men and women.

4.3.1 Evaluation on Hasler Dataset

In order to further evaluate the generalization abilities of the linear FM/MM models (Section 4.2.3) to other data sources, we estimate FM and MM for a subset of registered scans from the *Hasler* dataset [Hasler et al. 2009] and measure the prediction error. We select scans of 10 men and 10 women, making sure to cover the extremes of the weight, height, fat, and muscle percentage distribution present in the data.

For the female sample, the predictions show a mean absolute error of $MAE_{FM} = 0.65\,kg\ (\pm 0.44)$ and $MAE_{MM} = 4.39\,kg\ (\pm 1.71)$. For the male sample, the model shows a similar error for the MM prediction, but performs worse at predicting FM: $MAE_{FM} = 3.32\,kg\ (\pm 1.98)$ and $MAE_{MM} = 4.14\,kg\ (\pm 2.74)$. Compared to the leave-one-out tests on the BeyondBMI data, the average error increases noticeably, which can partly be explained by differences in the measurement procedure between the two datasets: while Hasler et al. [2009] used a consumer-grade body fat scale, Maalin et al. [2020] used a medical-grade scale, which should lead to more accurate measurements. Nevertheless, these results show that our regressor generalizes well to other data sources, providing a simple and sufficiently accurate method for estimating FM and MM from body scans.

Given the FM and MM values of a target from our regressor, we choose the optimal muscle surface between the minimal and maximal muscle surface as explained in Section 4.2.4. Comparing the final FM and MM of the volumetric model to the ground truth measurements of the *Hasler* dataset we get end-to-end errors of $MAE_{FM} = 0.70\,kg\ (\pm 0.52)$, $MAE_{MM} = 4.19\,kg\ (\pm 1.39)$ (female) and $MAE_{FM} = 3.49\,kg\ (\pm 2.02)$, $MAE_{MM} = 3.81\,kg\ (\pm 2.56)$ (male). This evaluation shows that the muscle fitting introduces just a very low additional error, or even reduces the error of the estimator.

4.3.2 Evaluation on CAESAR Dataset

In order to demonstrate the flexibility and robustness of our method, we evaluate it by generating anatomical models for all scans of the European Caesar data set [Robinette et al. 2002], consisting of 919 scans of women and 777 scan of men, with height range 131 cm to 218 cm for men and from 144 cm to 195 cm for women (we restricted to scans with complete annotation and taken in standing pose). Some examples for men and women of different body shapes can be seen in Figure 4.15 and Figure 4.16.

For the 1696 CAESAR scans, our muscle and fat mass regressors yield just one slightly negative value for the fat mass of the thinnest male (body weight 48 kg, height 1.72 m, BMI $16.14\,kg/m^2$). For all other subjects, we get values ranging from 3.5 % to 38.9 % body fat (mean 20.3 %) for male

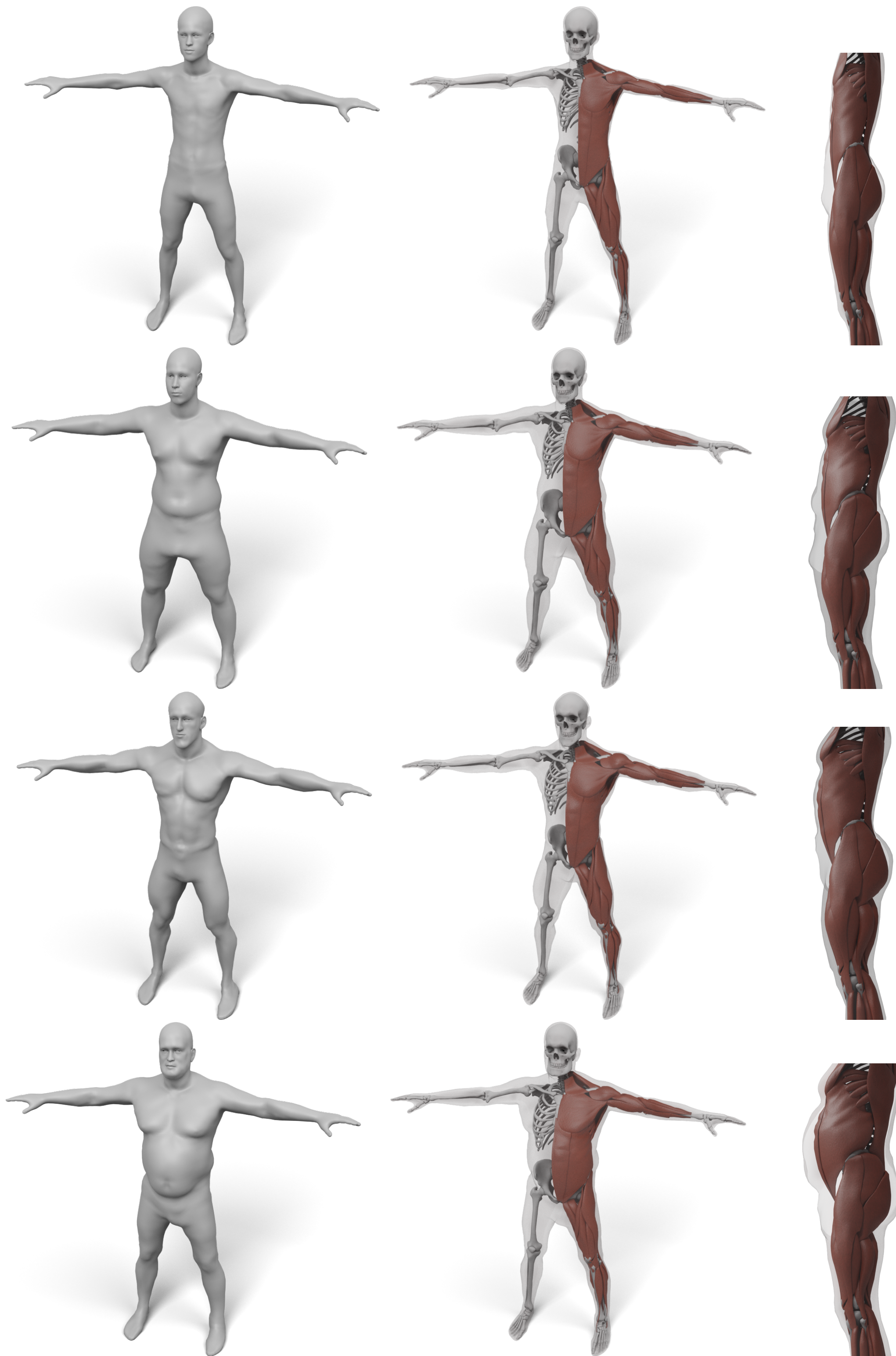


Figure 4.15: Some examples for various male body shape types. For each input surface the transferred muscles and skeleton are shown in front and side view.

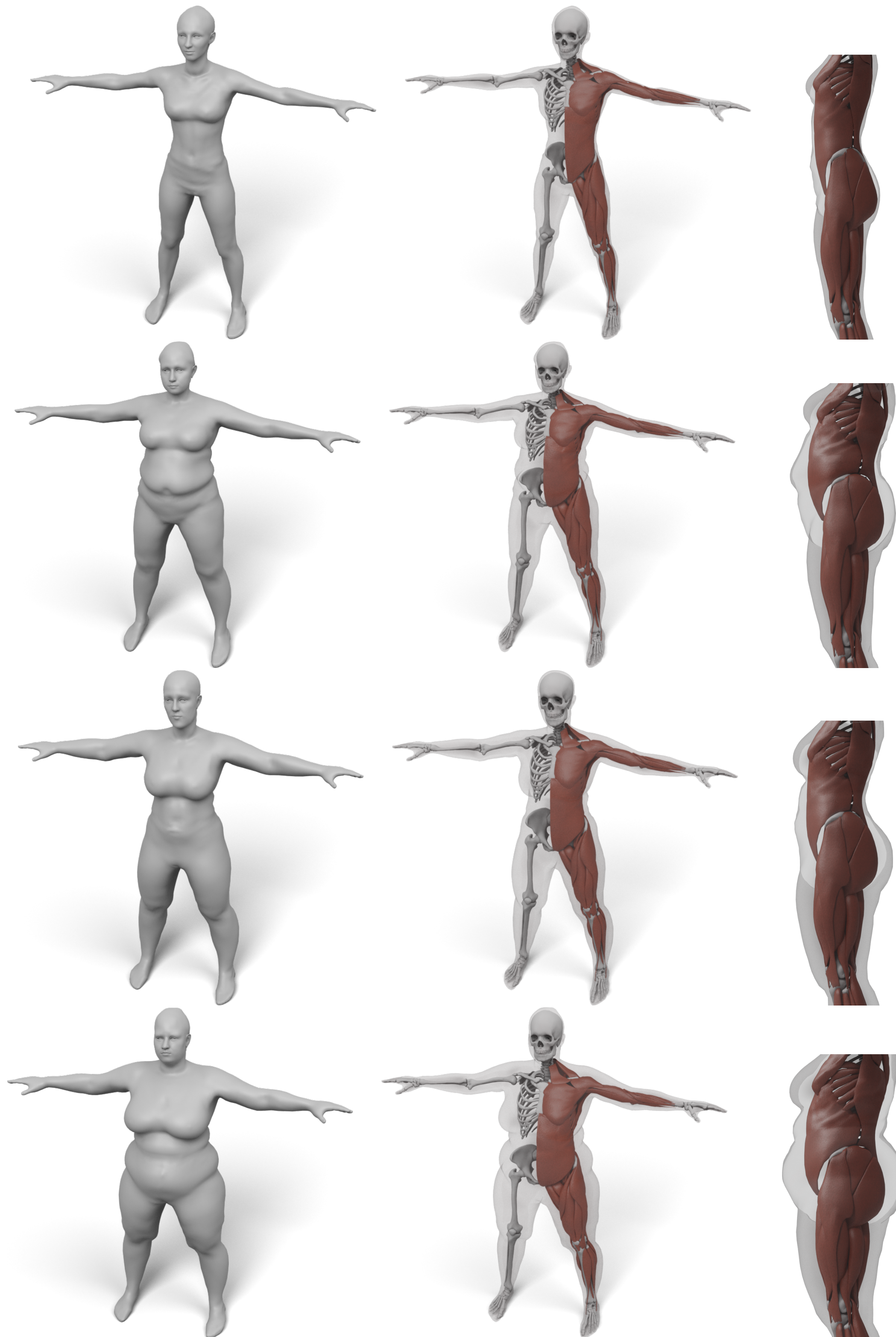


Figure 4.16: Some examples for various female body shape types. For each input surface the transferred muscles and skeleton are shown in front and side view.

subjects and 8 % to 45.3 % (mean 28.9 %) for female subjects. The range of predicted muscle masses is 24.9 kg to 57.8 kg (men) and 20.1 kg to 37.7 kg (women). When determining the optimal interpolation between the minimum and maximum muscle layer, we meet the estimated target values up to mean errors $\text{MAE}_{\text{FM}} = 1.08\,\text{kg}\ (\pm 0.9)$ (male), $\text{MAE}_{\text{FM}} = 1.41\,\text{kg}\ (\pm 1.35)$ (female) and $\text{MAE}_{\text{MM}} = 0.88\,\text{kg}\ (\pm 0.74)$ (male), $\text{MAE}_{\text{MM}} = 1.15\,\text{kg}\ (\pm 1.11)$ (female). Note that even the scan with predicted negative FM can be reconstructed robustly. In this case the muscle surface will be the maximum muscle surface, which in general is a suitable estimate for very skinny subjects.

The CAESAR dataset does not include ground truth data for fat and muscle mass of the scanned individuals. Thus, in order to further evaluate the plausibility of our estimated body composition, we compare it to known body fat percentiles. Percentiles are used as guidelines in medicine and provide statistical reference values. For instance, a 10th percentile of 20.8% body fat means that 10% of the examined population have a body fat percentage less than 20.8%. Assuming that the European CAESAR dataset is a representative sample of the population, the percentiles we get from our reconstructions of the CAESAR scans should match the percentiles of the European population. The total body mass of each subject is provided in the dataset. Therefore, we can compute the BF from the values produced by our fat mass regressor (Section 4.2.3), determine the percentiles on the CAESAR dataset and compare these to Kyle et al. [2001], who measured body fat using 4-electrode bio-electrical impedance analysis from 2735 male and 2490 female western European adults. Our body fat percentiles on the CAESAR dataset are very well in agreement with their results, as shown in the following table:

	Percentile	5th	10th	25th	50th	75th	90th	95th
Male	Our estimate	10.2	12.3	16.0	20.3	24.6	28.1	30.7
	Kyle et al. [2001]	10.9	12.6	15.7	19.2	23.5	27.0	29.2
Female	Our estimate	18.6	21.1	24.7	28.5	33.7	37.4	39.3
	Kyle et al. [2001]	18.5	20.8	23.8	28.1	32.6	37.5	40.5

4.3.3 Physics-Based Character Animation

As explained before, the complicated process of creating volumetric models is often prohibitive for physics-based character animation. Our Fast Projective Skinning approach (Chapter 3) provides a simple and efficient model generation but the volumetric skeleton built from spheres and cylinders is just a rough approximation to realistic anatomy. Moreover, FPS handles muscles and fat in one combined layer, taking no differences in body composition into

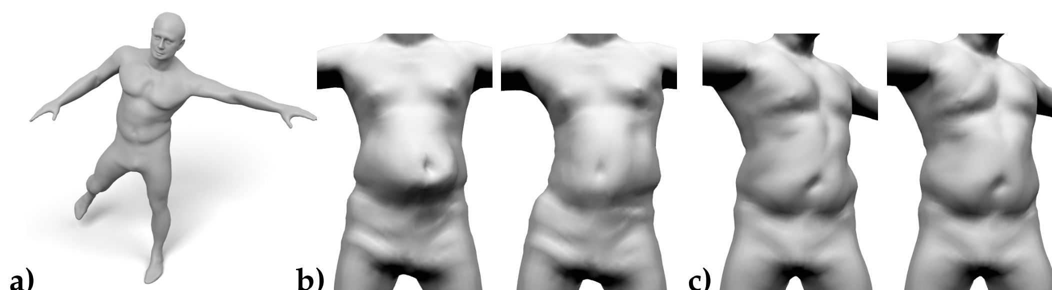


Figure 4.17: Our layered anatomical model can be animated using an extension of Fast Projective Skinning (FPS), as shown in (a). When the character performs a jump to the left (b), our realistic skeleton correctly restricts the dynamic jiggling to the belly region (b-left), while the original FPS deforms the the complete torso (b-right). For a static twist of the torso (c), the rib-cage of our layered model keeps the chest region rather rigid and concentrates the deformation to the belly (c-left). Without a proper anatomical model, the deformation of FPS is distributed over the complete torso (c-right).

account. Our three layered anatomical models overcome some limitations of the FPS approach while still being much simpler and faster to produce than previous models [Kadleček et al. 2016].

We demonstrate this by modifying our skinning method to make it compatible with the three-layered models. We build the skeleton graph by setting joints at locations of the real anatomical joints (the spine is approximated with only four joints). When changing the pose of the animation skeleton, the detailed skeleton mesh is animated using Linear Blend Skinning, where most of the volumetric bones just depend on one joint but a few (e.g., spine and scapula) are also influenced by multiple joints. We attach vertices of the skeleton surface $\mathcal{B}$ to their corresponding bone using hard constraints (see Section 2.2.3) such that they follow the skeletal motion.

The remaining steps of the skinning simulation stay unchanged. The FPS soft tissue layer is now split into our separate muscle and fat layers. This enables us to use different stiffness values for the fat and muscle layers (the latter being three times larger). Moreover, the skeleton surface of the three-layered model features a realistic rib-cage. As a result, our extended version of FPS yields more realistic results compared to vanilla FPS in particular in the torso and belly region, as shown in Figure 4.17 and 4.18 and in Video V.8. In other regions like arms and legs, where the bones' shape is already close to the cylinders of the simplified FPS skeletons, differences in the skinning result are more subtle. Note that the GPU simulation still allows real-time simulations of the full three-layered model. Using the same settings as in Table 3.2, we achieve about 250 fps with $N \approx 16.7k$ simulated vertices of our three surface (a large part of the skeleton surface can be set as boundary constraint) and collisions disabled.

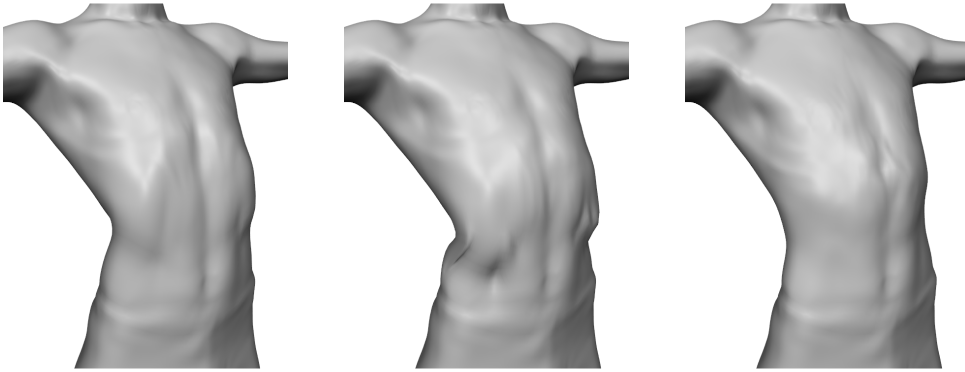


Figure 4.18: A bending-backwards posture with (left) and without (middle) defined sliding regions, where the skeleton surface can stretch. The right image shows the result of the simplified FPS-based volumetric model.

The FPS skin sliding approach explained in Section 3.2.2 is based on the simplified skeletal structure and is therefore not applicable when using a realistic skeleton. In order to accomplish a similar effect, we define some regions in proximity of joints, which we do not attach to bones by hard constraints. In these areas the skeleton surface can stretch and compress when the corresponding joint is bent. This is especially important in the abdominal region. Here, a rigid binding of the skeletal motion to different joints leads to artifacts as shown in Figure 4.18, middle. Allowing a stretching and compression in this area simulates the softer abdominal region more realistically (Figure 4.18, left). Note that the skeletal rig as well as all manual assignments have to be defined just once for the template as they can be transferred to all derived models.

4.3.4 Simulation of Fat Growth

Our anatomical model can also be used to simulate an increase of body fat, where its volumetric nature provides advantages over existing surface-based methods.

In their computational bodybuilding approach, Saito et al. [2015] also propose a method for growing fat. They, however, employ a purely *surface-based* approach that conceptually mimics blowing up a rubber balloon. This is modeled by a pressure potential that drives skin vertices outwards in normal direction, regularized by a co-rotated triangle strain energy. The user must specify a scalar field that defines where and how strong the skin surface should be "blown up", which is used to modulate the per-vertex pressure forces. Despite the strain-based regularization we sometimes noticed artifacts at the boundary of the fat growing region and therefore add another regularization through Equation (4.2). This approach allows the user to tune the amount of

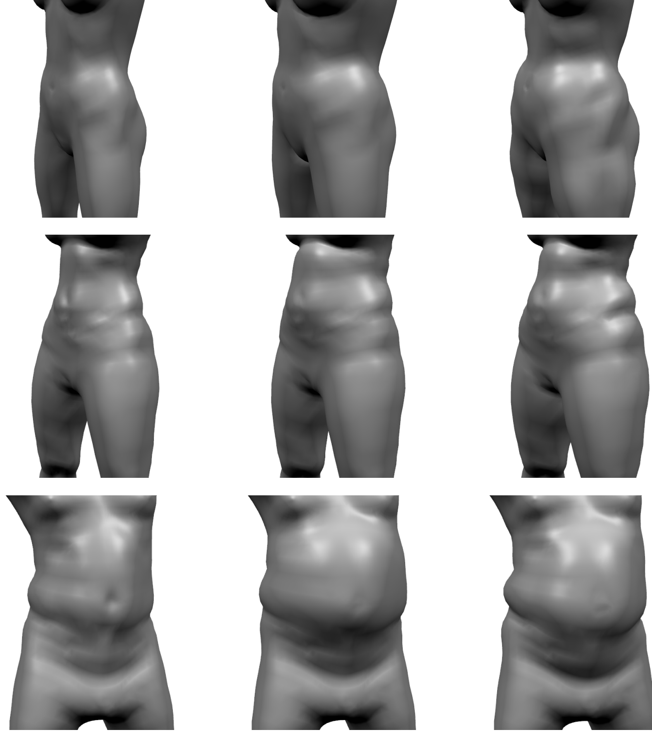


Figure 4.19: Given a reconstructed model (left), the pressure-based fat growth of Saito et al. [2015] leads to a more uniform increases in fat volume (center), while our volume-based fat growth increases the initial fat distribution.

subcutaneous fat, but unless a carefully designed growth field is specified, the fat growth looks rather uniform and balloon-like (see Figure 4.19, mid-top).

Every person has an individual fat distribution and gaining weight typically intensifies these initial fat depots. We model this behavior by scaling up the local prism volumes of our fat layer. Each fat prism can be split into three tetrahedra, which define volumetric elements $t_j \in \mathcal{T}$ with initial volumes $\bar{V}_j$. A simple uniform scaling $s \cdot \bar{V}_j$ achieves the desired effect that fat increases more in fat-intense regions. The growth simulation is implemented by minimizing the energy

$$E_{\text{grow}}(\mathcal{S}) = w_{\text{vol}} E_{\text{vol}}(\mathcal{S}) + w_{\text{reg}} E_{\text{reg}}\left(\mathcal{S}, \bar{\mathcal{S}}\right) + w_{\text{rest}} E_{\text{rest}}\left(\mathcal{S}, \bar{\mathcal{S}}\right) \tag{4.16}$$

with the Laplacian regularization of Equation (4.2), the displacement regularization

$$E_{\text{rest}}\left(\mathcal{S}, \bar{\mathcal{S}}\right) = \sum_{\mathbf{x}_i \in \mathcal{S}} A_i \left\|\mathbf{x}_i - \bar{\mathbf{x}}_i\right\|^2 \tag{4.17}$$

and the volume fitting term

$$E_{\text{vol}}(\mathcal{S}) = \sum_{t_j \in \mathcal{T}} \bar{V}_j \left(\text{vol}(t_j) - s \cdot \bar{V}_j\right)^2, \tag{4.18}$$

where $\bar{\mathcal{S}}$ and $\mathcal{S}$ denote the skin surface before/after the fat growth and s is the global fat scaling factor. The energy minimization can again be performed by

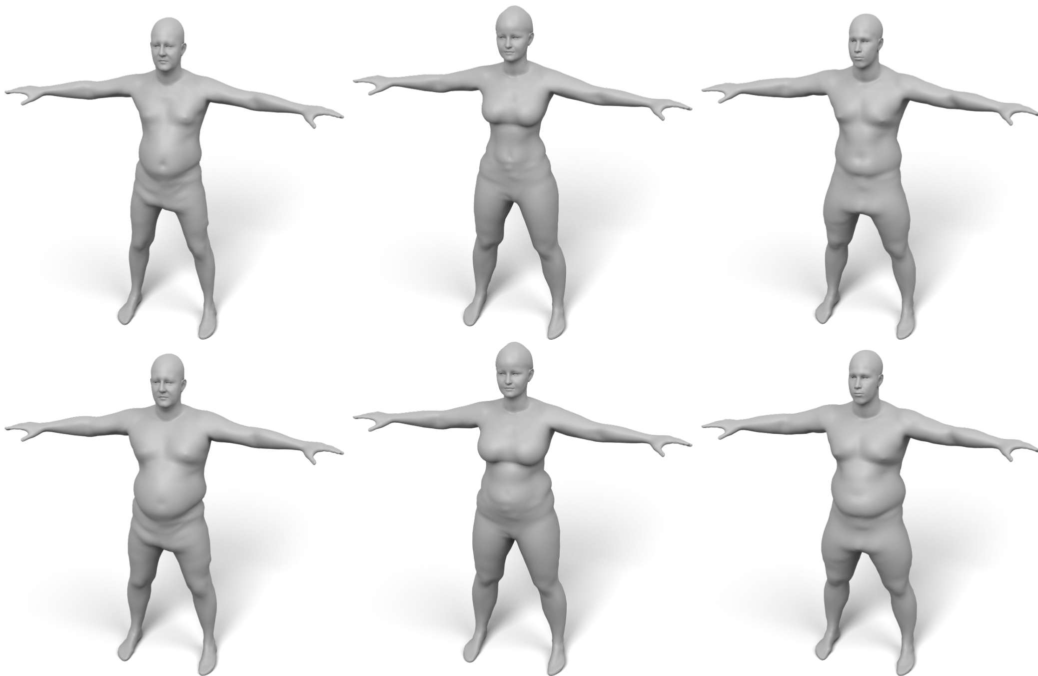


Figure 4.20: Examples of our fat growth simulation, with input models shown in the top row and their weight-gained version in the bottom row.

the static version of the PD solver (Section 2.2) using volume, bending and anchor constraints (Section 2.2.1). Saito et al. [2015] argued that *anisotropically* scaling fat tetrahedra in *one* direction does not produce plausible results. However, *isotropically* scaling the volume leaves the minimization more freedom and yields convincing results. Figure 4.19 compares the pressure-based and volumetric fat growth simulations. Figure 4.20 shows some more examples produced by combining both methods. Note that all examples did not require a per-vertex scalar growth field, however, we divided the body into four parts (i.e., legs, belly, chest, arms) in order to choose a local growth-rate per region.

If we want to use our volume-based fat growth to grow fat on a very skinny person, the initial (negligible) fat distribution does not provide enough information on where to grow fat. Manually defining a per vertex-scalar growth field like Saito et al. [2015] solves this issue, however, we found a simpler solution: since we can easily fit the volumetric template to several subjects, we can "copy" the distribution of fat prism volumes from another person and "paste" it onto the skinny target, which simply replaces the target volumes in (4.18). This enables a fat transfer between different subjects, which is shown in Figure 4.21. Note that this approach transfers just the fat distribution, and the scaling factor, i.e., the amount of additional weight, can still be set manually.

All presented approaches are not able to simulate the growth of visceral fat since only the subcutaneous fat layer is scaled. To solve this issue, we can re-apply our muscle fitting (Section 4.2.4) to the result of the fat growth.

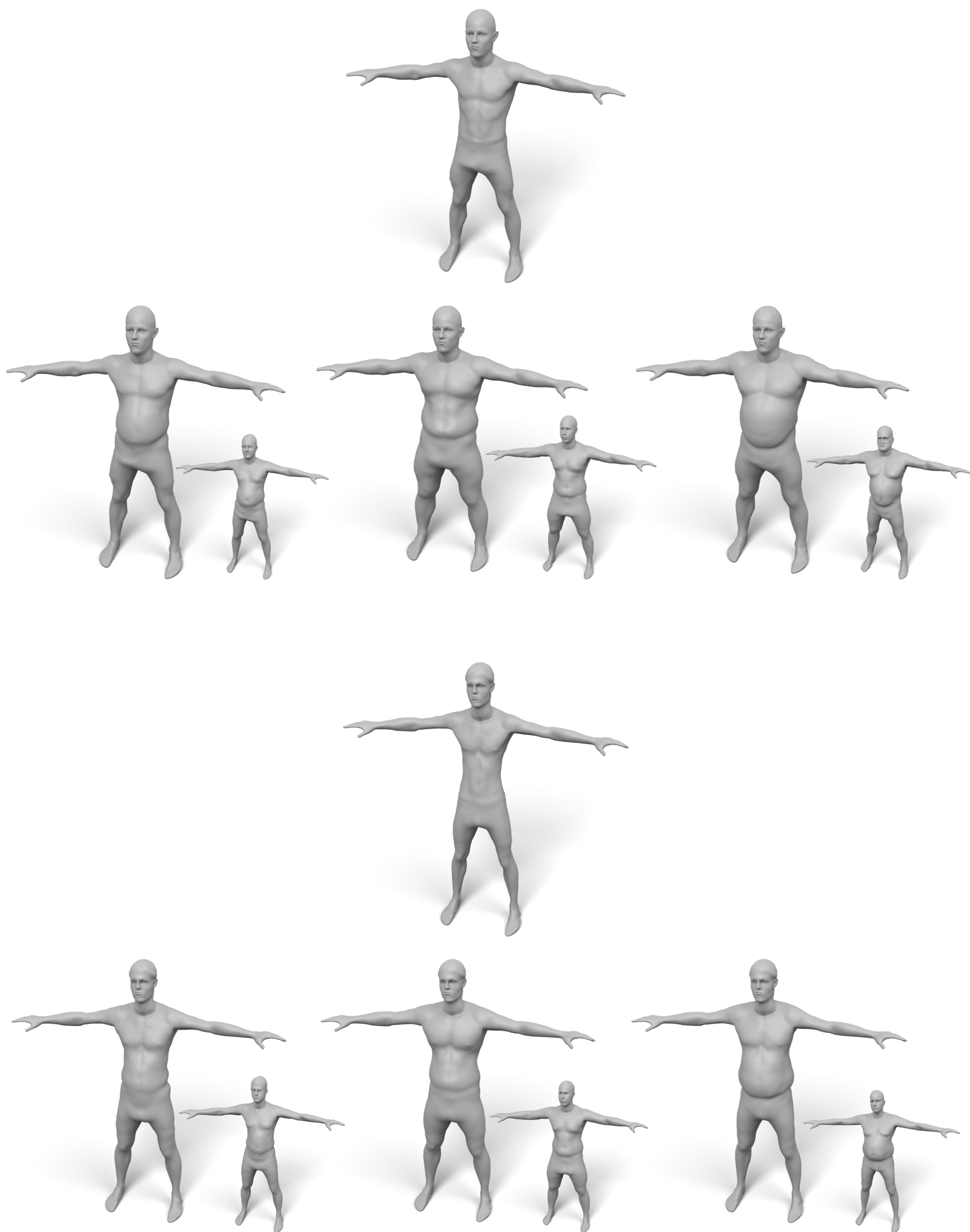


Figure 4.21: Examples of "fat transfer". The two subjects in the first and third row have a very low amount of body fat. Therefore, scaling their fat volumes is not suitable for fat growth. Instead, we copy the fat distributions of other subjects (shown as small insets). This allows us to simulate a similar fat growth behavior for the skinny targets.

4.4 SUMMARY & LIMITATIONS

We created a simple layered volumetric template of the human anatomy and presented an approach for fitting it to surface scans of men and women of various body shapes and sizes. Our method generates plausible muscle and fat layers by estimating realistic muscle and fat masses from the surface scan alone. In addition to the method for fitting the layered template to the target, we also showed how to transfer internal anatomical structures, such as bones and muscles, using a high-quality space warp. Compared to previous work, our method is fully automatic and considerably faster, enabling the simple generation of personalized anatomical models from surface body scans. Besides educational visualization, we demonstrated the potential of our model for physics-based character animation and anatomically plausible fat growth simulation.

Our approach has some limitations: First, we do not generate individual layers for head, hands and toes, where in particular the head would require special treatment. Combining our layered body model with the multi-linear head model of Achenbach et al. [2018] is therefore a promising direction for future work. Our method assumes that target persons have four limbs, hence it cannot model humans who lost an arm or a leg. The regressors for fat and muscle mass could be further optimized by training on more body scans with known body composition. Given more accurate training data, as for instance provided by DXA scans, we could extend the fat/muscle estimations to individual body parts. Moreover, we focused on muscles, fat and bones and ignored other structures like blood vessels, organs, tendons and cartilages. These can be transferred from template to target by applying our harmonic space warp but our model is currently not able to simulate their functionality. Another limiting factor is that all models produced by our method share the basic shape of their muscles. In most cases this is a reasonable approximation but the abdominal muscles are an important exception. The 'six-pack' becomes visible for very low amounts of body fat and varies widely in size and symmetry for different people. In the current state, our muscle layer cannot adapt to these variations. The fact that the three layers of our model share the same topology/connectivity can also be considered a limitation, since we cannot use different, adaptive mesh resolutions in different layers. We minimized misalignment of corresponding triangles when generating the different template layers, however, our simple approach for creating the volumetric elements in between two surfaces can still result in tetraheda of high aspect ratio. While we did not notice any artifacts in the simulations, we could increase the quality of the volumetric elements by employing a more sophisticated tetrahedralization approach [Si 2015] using our surfaces $\mathcal{B}$, $\mathcal{M}$, $\mathcal{S}$ as input.

The presented applications of our models for animation and fat growth both provide space for further improvements. We used a modification of Fast Projective Skinning (see Section 4.3.3) allowing for simple and efficient animations of our anatomical models, however, we thereby also inherited the limitations of FPS, like its restriction to a linear elasticity model. In addition, bones should also be treated as a nonrigid object (especially the rib cage) to allow for a more realistic behavior. The simplified FPS skeletons made from spheres and cylinders prevent connected bones from intersecting for the majority of anatomically plausible joint angles. However, this is no longer the case when using a more realistic skeleton, for which we must carefully choose the rotation centers to minimize inter-bone penetrations at joints. Setting sliding regions like explained in Section 4.3.3 helps to avoid artifacts in these cases. In future, we would like to extend this approach and also simulate the deformation of the skeleton surface $\mathcal{B}$ rather than rigidly attaching it to bones. In this way, we could handle collisions with the detailed bone meshes and more realistically simulate the skin-sliding behavior.

We also plan to develop more automatic fat growth simulations. While we do not require a detailed fat growth texture like Saito et al. [2015], our approach is still a manual task, where we choose a region (e.g., belly, chest, legs) and set a desired amount of growth per region. In future, we would like to learn the typical fat distributions and ranges from the layered models of the Caesar dataset [Robinette et al. 2002] and try to find some clusters of 'fat growth types'. The goal would be to determine the corresponding cluster for a given obese person (or choose one for a skinny person) and simulate fat growth based on that specific type and the targeted weight.

Moreover, we focused only on *weight gain* but simulating realistic *weight loss* is also an interesting topic for future work. We can theoretically set a reduced volume in our current fat growth approach but this only works for small reductions. When targeting significantly smaller volumes, we also need to shrink the skin surface $\mathcal{S}$ realistically. A promising approach could be to first simulate fat growth for many persons and using this as training data to learn the inverse operation. We think that the simple structure of our layered model can be beneficial to generate synthetic training data for statistical analysis and machine learning applications in general.

5 CONCLUSION

In the last decade, companies realized something that researchers have known for a long time: the data of each aspect of everyone's live is a valuable currency. It is required for applications ranging from personalized advertisement up to methods based on artificial intelligence. However, it also raises the need for data privacy and significantly empowers those who can afford the most data. The same trend can be observed in many recent approaches for character animation relying on increasing amounts of data, in this case 3D-scans of humans in different poses (also called *examples*). Even though these example-based methods are able to produce very convincing animations, collecting examples requires expensive equipment, which is prohibitive for small developer teams and circumvents the creation of their own models based on their demands. Furthermore, since real-world data is not available for fictional characters, the usage of those approaches is limited.

A (monetary) cheaper and more general solution is offered by physics-based approaches. Rather than just capturing the results, they try to understand and model the rules of physics underlying *every* plausible motion and deformation of matter. Physics-based simulations are often used in movie productions, especially to animate very dynamic motions (e.g., cloth, hair, fluids), but are also employed to produce very convincing character animations. Typically, these simulations have two main disadvantages: they require volumetric character models that are cumbersome to create and have very high computational costs (multiple seconds or even minutes per frame are quite common). This thesis tried to tackle and solve these two problems of physics-based animation.

For this purpose, we developed Fast Projective Skinning, a physics based skinning approach being capable of simulating various character models in real-time. An important self-imposed requirement was that the user must provide just a minimal amount of input data for animating their model. Moreover, we intended not to rely on tedious, manual fine-tuning steps like the typical rigging process of common character animation pipelines. In the end, the resulting freely available implementation of our FPS-approach takes just the skin mesh and the desired skeletal structure as input. We also provide an additional small application for defining the latter (see Figure 5.1).

The volumetric model, which is required for the simulation, is automatically created from skin and skeleton. The structure of the model is inspired by the anatomy of real bodies: on an abstract level, they are built from soft tissue surrounding solid, tube-shaped bones. Following this idea, we built a simple volumetric skeleton and shrink the skin until it wraps this skeleton like a rubber-tube. Our shrinking process has several advantages: First, it can be applied very efficiently on a wide range of input data (both skeletons and

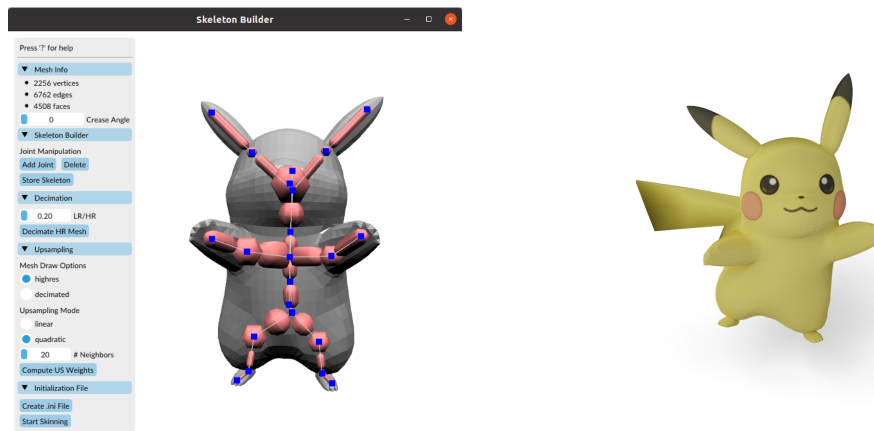


Figure 5.1: Our Fast Projective Skinning application includes a skeleton-builder that takes a skin mesh as input and allows a simple drag-and-drop construction of the skeleton graph (left). The user can also decimate the mesh and compute the upsampling weights with the help of this app. Thereby, our FPS approach provides a simple and intuitive way to animate the model (right).

skin meshes). Second, it provides a very simple and robust volumetric mesh generation that does not involve any additional software. Third, due to the tubular structure of our shrunken skin, the area in between two bones is not contained in the resulting tetrahedral mesh. These regions involve the most extreme deformations and can cause artifacts in physics-based simulations if not handled carefully. Our representation successfully avoids this problem and is even capable of simulating the stretching and compression of skin in these regions (skin sliding).

Position-based solvers like Projective Dynamics are already capable of simulating our resulting volumetric models in real-time on the CPU for moderate resolutions. However, applications like video games typically involve multiple skinned characters in addition to numerous other effects that must be computed in each frame, thus, the remaining time budget per frame is extremely small in general. Therefore, we optimized our approach in multiple ways to meet those demands. First, we reduced the size of the simulated system by incorporating hard constraints in the Projective Dynamics algorithm. Additionally, we are able to perform our simulations on very coarse representations of the models by introducing an upsampling approach that efficiently maps to the full resolution. Our upsampling method outperforms similar previous techniques in terms of both memory consumption and computational efficiency. Furthermore, we have shown how to leverage the huge potential for parallelization by developing a highly optimized GPU implementation of Fast Projective Skinning. Overall, we were able to reduce the computation time per frame to less than 0.5 ms which is even comparable to common geometric approaches.

Regarding animation quality, our Fast Projective Skinning achieves very

convincing results and further supports dynamic jiggling of soft tissue. Moreover, we introduced two approaches for incorporating collision handling in FPS. We can pre-compute potential collision pairs for solving *local* collisions in proximity of joints, handled by special, light-weight local collision constraints. We recommend using these for CPU-based FPS for which a full collision detection would drastically impair the efficiency. However, GPU-based FPS is able to manage the additional overhead of a full global collision detection while maintaining real-time performance. As a result, Fast Projective Skinning is the first skinning approach capable of handling arbitrary global collisions for complete humanoid characters in real-time.

While FPS achieves convincing results without requiring any manual steps, it simultaneously offers some opportunities to modify the results based on artistic choices: First, the size and positioning of our volumetric joints influence the surrounding area. Moreover, all parameters can be defined on a vertex base. For instance, an artist could increase or decrease the amount of dynamic jiggling for a specific area by varying the mass. Furthermore, we have shown that FPS supports stretchable and twistable bones as well as virtual bones to simulate unconnected skeletons and soft limbs.

The flexibility of our approach is based on the simplifications we made to generate the volumetric models. While FPS overcomes the artifacts of common geometric approaches, the assumed simplifications can still impair the quality. Particularly for animations of humans, which can be observed on real persons every day, even small imperfections attract attention. A typical example is the poor approximation of the rib-cage in our models, or the missing information of fat and muscle distribution, both leading to unrealistic behavior, especially in dynamic simulations. In the second half of this thesis, we therefore introduced a second approach to generate anatomically plausible, personalized volumetric models of humans from 3D surface scans. We purposely did not built on volumetric scans (MRI, CT) of the subject since these are highly expensive and, in case of CT-scans, involve X-radiation.

Inspired by our FPS-models consisting of two topologically identical surfaces, we extended the idea of a layer-based volumetric representation. Starting from a detailed anatomical model, we generated a high-quality layered template model of an average male and female character. This time, we used three topologically identical surfaces: a skeleton surface wrapping the realistic skeleton, a muscle surface enclosing both muscles and skeleton and the skin.

We further built *personalized* three-layered models by employing a volumetric fitting process that subsequently modifies the template model until it perfectly fits the surface scan of the target person. By first registering each input mesh to the same animatable surface template, we can perform the volumetric fitting process in a uniform pose. This process further leverages the conformal layered structure of the template, and is therefore both much faster than previous approaches and produces more realistic results.

Dividing the soft tissue into plausible fat and muscle volumes had been one of the main challenges for our method. We built a regressor that is able to estimate the muscle and fat mass from the 3D-scan of a person and produces much more reliable results than comparable existing methods. The estimated muscle and fat mass values, as well as our layered structure, allow for a simple construction of the personalized muscle surface: after initializing the surface by constraining it between the subject's skin and skeleton surface, we can modify the thickness of both fat and muscle layer by moving the muscle surface to one of its two enclosing surfaces. This results in a plausible minimal and maximal muscle layer. The user can now either set a desired amount of muscle and fat mass, or we automatically detect these using the estimator. Our approach takes also visceral fat into account that produces a bulge of the abdominal muscles for corpulent persons.

Lastly, anatomical structures can be transferred from template to target with a triharmonic space warp. As a result, our approach creates personalized, anatomically plausible volumetric models for humans in a few seconds. These provide more realistic character animation than our simplistic two-layered models for both static poses and dynamic motions. We further presented a simple approach for fat growth and fat transfer.

Beyond physics simulations, we can think of various future applications for quickly generated, personalized anatomical models, particularly in medicine. For instance, a bone fracture could be explained to patients using their personalized anatomical model instead of a medical skeleton model. Here, our layered representation would provide a plausible guess for the majority of the body and the specific area could be augmented by volumetric (MRI, CT) scans. Our models can also be used for educational purpose: time-intensive memorization of human anatomy could be much more exciting for medical students if the reference were their own bodies or those of partners or celebrities, which could be studied in a VR-environment. Regarding cosmetic surgery, the ability to visualize potential outcomes of a surgical operation via physical simulations is very important for both patients and surgeons. Moreover, our model could be applied to enhance therapy for obese or anorexic people by confronting the patients with their personalized models while changing the amount of body fat with our fat growth approach. Furthermore, athletes could use our models to visualize their personal muscles and for simulating the effect of different amounts of body fat/muscles. All these fields of applications benefit from fast generation of personalized anatomical models.

We want to emphasize once more that our method cannot replace volumetric imaging techniques in a medical context, but instead provides a plausible guess for the interior structures of a person. Compared to the two-layered FPS models, it is less general since it focuses on human characters. This is compensated for by providing a more realistic human model.

Lastly, we would like to outline some promising directions of future work:

- If a patch of skin begins to stretch, the required force increases slowly for small deformations but grows more drastically for larger ones due to collagen fibers lining up with the stress direction [Gibson and Kenedi 1967]. Muscles are built from aligned fibers resulting in an anisotropic stretching behavior [Green et al. 2013]. To account for both effects, it would be interesting to experiment with solvers that support nonlinear and anisotropic elasticity models [Narain et al. 2017; Liu et al. 2017] in combination with our FPS approach.
- So far, the skin surface forms the outermost layer of our character models. But skin can also be covered by hair and cloth, which are currently "baked" into the skin surface (we have shown various examples of dressed characters with hair in Chapter 3). Modeling and simulating hair and cloth is a challenging task, especially in real-time, but would increase the realism of our character simulations by a huge amount.
- While the animation of the anatomical models is real-time capable, it is still considerably slower than the simulation of the simplistic two-layered models. Therefore, it would be interesting to see to which extent we could reproduce the results of the anatomical model by optimizing parameters of our simplified ones (local stiffness, masses, position and size of joints).
- For medical treatment, just a specific region of the body is typically scanned using volumetric imaging techniques. Like already mentioned before, enhancing our anatomical models with those partial volumetric scans could further extend the applicability of our method. Similarly, additional information about *local* fat and muscle distribution could enhance our process of fitting the muscle surface.
- The various three-layered models we generated could be used to build a statistical anatomical model for humans similar to former surface models [Hasler et al. 2009]. These represent the variations of human anatomy through a set of parameters and could also serve as a generator of volumetric human models. We already experimented with simple PCA models but found that they cannot prevent the surfaces from intersecting.
- We think that, due to their simple layered structure, our models could effectively be used in machine learning applications. For instance, a neuronal network could be trained to predict the skin deformation based on a given deformed skeleton surface using our simulations to generate large sets of training data. The fat growth simulations could also be learned and probably inverted to predict a leaner version of a person. We could also try to train a generator of our anatomical models based on different inputs like images, sketches or even descriptions of the person.

All in all, both Fast Projective Skinning and the efficient generation of personalized anatomical models presented in this thesis can help making physics-based simulations of virtual characters more applicable and simplify the animation process for both novice users and experts.

There is a final factor that should not be underestimated besides all the serious applications presented in this thesis: it can be simply fun to watch the mesmerizing dynamic motions of the animated characters.

BIBLIOGRAPHY

Achenbach, Jascha, Robert Brylka, Thomas Gietzen, Katja Zum Hebel, Elmar Schömer, Ralf Schulze, Mario Botsch, and Ulrich Schwanecke (2018), "A multilinear model for bidirectional craniofacial reconstruction." In *Proc. of Eurographics Workshop on Visual Computing for Biology and Medicine*, 67–76.

Achenbach, Jascha, Thomas Waltemate, Marc Erich Latoschik, and Mario Botsch (2017), "Fast generation of realistic virtual humans." In *Proc. of ACM Symposium on Virtual Reality Software and Technology*, 12:1–12:10.

Ackerman, Michael J (1998), "The visible human project." *Proceedings of the IEEE*, 86, 504–511.

Agisoft (2017), "Photoscan pro." http://www.agisoft.com/.

Anguelov, Dragomir, Praveen Srinivasan, Daphne Koller, Sebastian Thrun, Jim Rodgers, and James Davis (2005a), "Scape: Shape completion and animation of people." *ACM Transactions on Graphics*, 24, 408–416.

Anguelov, Dragomir, Praveen Srinivasan, Daphne Koller, Sebastian Thrun, Jim Rodgers, and James Davis (2005b), "SCAPE: Shape completion and animation of people." *ACM Transactions on Graphics*, 24, 408–416.

Anzt, Hartwig, William Sawyer, Stanimire Tomov, Piotr Luszczek, Ichitaro Yamazaki, and Jack Dongarra (2014), "Optimizing krylov subspace solvers on graphics processing units." In *Proc. of IEEE International Parallel Distributed Processing Symposium Workshops*.

Autodesk (2014), "Character generator." https://charactergenerator.autodesk.com/.

Aydin Kabakci, AD, M Buyukmumcu, MT Yilmaz, AE Cicekcibasi, D Akın, and E Cihan (2017), "An osteometric study on humerus." *International Journal of Morphology*, 35, 219–226.

Baran, Ilya and Jovan Popović (2007), "Automatic rigging and animation of 3d characters." *ACM Transactions on Graphics*, 26.

Bell, Nathan and Michael Garland (2008), "Efficient sparse matrix-vector multiplication on cuda." Technical Report NVR-2008-004, NVIDIA Corporation.

Bender, Jan, Matthias Müller, and Miles Macklin (2017), "A survey on position based dynamics." In *Eurographics Tuturials*.

Bogo, Federica, Javier Romero, Gerard Pons-Moll, and Michael J. Black (2017), "Dynamic FAUST: Registering human bodies in motion." In *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Bolz, Jeff, Ian Farmer, Eitan Grinspun, and Peter Schröder (2003), "Sparse matrix solvers on the gpu: Conjugate gradients and multigrid." *ACM Transactions on Graphics*, 22.

Botsch, Mario and Leif Kobbelt (2005), "Real-time shape editing using radial basis functions." *Computer Graphics Forum*, 24, 611–621.

Botsch, Mario, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy (2010), *Polygon Mesh Processing*. AK Peters, CRC press.

Botsch, Mario and Olga Sorkine (2008), "On linear variational surface deformation methods." *IEEE Transaction on Visualization and Computer Graphics*, 14.

Bouaziz, Sofien, Mario Deuss, Yuliy Schwartzburg, Thibaut Weise, and Mark Pauly (2012), "Shape-up: Shaping discrete geometry with projections." *Comput. Graph. Forum*, 31, 1657–1667.

Bouaziz, Sofien, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly (2014a), "Projective dynamics: Fusing constraint projections for fast simulation." *ACM Transactions on Graphics*, 33.

Bouaziz, Sofien, Andrea Tagliasacchi, and Mark Pauly (2014b), "Dynamic 2D/3D registration." In *Eurographics Tutorials*, 1–17.

Brandt, Christopher, Elmar Eisemann, and Klaus Hildebrandt (2018), "Hyper-reduced projective dynamics." *ACM Transactions on Graphics*, 37.

Brochu, Tyson, Essex Edwards, and Robert Bridson (2012), "Efficient geometrically exact continuous collision detection." *ACM Transactions on Graphics*, 31, 96:1–96:7.

Brožek, Josef, Francisco Grande, Joseph T. Anderson, and Ancel Keys (1963), "Densitometric analysis of body composition: Revision of some quantitative assumptions." *Annals of the New York Academy of Sciences*, 110, 113–140.

Brunel, Camille, Pierre Bénard, and Gaël Guennebaud (2021), "A time-independent deformer for elastic contacts." *ACM Transactions on Graphics*, 40.

Buatois, Luc, Guillaume Caumon, and Bruno Levy (2009), "Concurrent number cruncher: a gpu implementation of a general sparse linear solver." *International Journal of Parallel, Emergent and Distributed Systems*, 24.

Capell, Steve, Matthew Burkhart, Brian Curless, Tom Duchamp, and Zoran Popović (2005), "Physically based rigging for deformable characters." In *Proc. of ACM SIGGRAPH/Eurographics Symposium on Computer Animation*.

Capell, Steve, Seth Green, Brian Curless, Tom Duchamp, and Zoran Popović (2002), "Interactive skeleton-driven dynamic deformations." *ACM Transactions on Graphics*, 21, 586–593.

Casas, Dan and Miguel A. Otaduy (2018), "Learning nonlinear soft-tissue dynamics for interactive avatars." *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1.

Chao, Isaac, Ulrich Pinkall, Patrick Sanan, and Peter Schröder (2010), "A simple geometric model for elastic deformations." *ACM Transactions on Graphics*, 29, 38:1–38:6.

Christ, Andreas, Wolfgang Kainz, Eckhart G Hahn, Katharina Honegger, Marcel Zefferer, Esra Neufeld, Wolfgang Rascher, Rolf Janka, Werner Bautz, Ji Chen, et al. (2009), "The virtual family—development of surface-based anatomical models of two adults and two children for dosimetric simulations." *Physics in Medicine & Biology*, 55, N23–N38.

Dayal, Manisha R, Maryna Steyn, and Kevin L Kuykendall (2008), "Stature estimation from bones of south african whites." *South African Journal of Science*, 104, 124–128.

Deul, Crispin and Jan Bender (2013), "Physically-based character skinning." In *Proc. of Virtual Reality Interactions and Physical Simulations*.

Deuss, Mario, Anders Holden Deleuran, Sofien Bouaziz, Bailin Deng, Daniel Piker, and Mark Pauly (2015), "Shapeop – a robust and extensible geometric modelling paradigm." In *Proc. of Design Modelling Symposium*.

Dicko, Ali-Hamadi, Tiantian Liu, Benjamin Gilles, Ladislav Kavan, François Faure, Olivier Palombi, and Marie-Paule Cani (2013), "Anatomy transfer." *ACM Transactions on Graphics*, 32, 188:1–188:8.

Feng, Andrew, Dan Casas, and Ari Shapiro (2015), "Avatar reshaping and automatic rigging using a deformable model." In *Proc. of ACM SIGGRAPH Conference on Motion in Games*, 57–64.

Feng, Xiaowen, Hai Jin, Ran Zheng, Kan Hu, Jingxiang Zeng, and Zhiyuan Shao (2011), "Optimization of sparse matrix-vector multiplication with variant csr on gpus." In *Proc. of IEEE International Conference on Parallel and Distributed Systems*.

Fields, David A, Michael I Goran, and Megan A McCrory (2002), "Body-composition assessment via air-displacement plethysmography in adults and children: a review." *The American Journal of Clinical Nutrition*, 75, 453–467.

Fit3D (2021), "Fit3d scanner systems." https://fit3d.com/.

Fratarcangeli, Marco, Valentina Tibaldo, and Fabio Pellacini (2016), "Vivace: a practical gauss-seidel method for stable soft body dynamics." *ACM Transactions on Graphics*, 35.

Fries, T.P. and H.G. Matthies (2004), "Classification and overview of meshfree methods." Informatikbericht 2003-03, revised 2004, Institute of Scientific Computing, Technical University Braunschweig.

Gao, Ming, Nathan Mitchell, and Eftychios Sifakis (2014), "Steklov-poincaré skinning." In *Proc. of ACM SIGGRAPH/Eurographics Symposium on Computer Animation*.

Gao, Ming, Xinlei Wang, Kui Wu, Andre Pradhana, Eftychios Sifakis, Cem Yuksel, and Chenfanfu Jiang (2018), "Gpu optimization of material point methods." *ACM Transactions on Graphics*, 37.

Garland, Michael and Paul S. Heckbert (1997), "Surface simplification using quadric error metrics." In *Proc. of Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH, 209–216.

Gibson, T and RM Kenedi (1967), "Biomechanical properties of skin." *Surgical Clinics of North America*, 47.

Gietzen, Thomas, Robert Brylka, Jascha Achenbach, Katja Zum Hebel, Elmar Schömer, Mario Botsch, Ulrich Schwanecke, and Ralf Schulze (2019), "A method for automatic forensic facial reconstruction based on dense statistics of soft tissue thickness." *PloS one*, 14, e0210257.

Golub, Gene H. and Charles F. Van Loan (2012), *Matrix Computation*, 4th edition. John Hopkins University Press.

Green, MA, G Geng, E Qin, R Sinkus, SC Gandevia, and LE Bilston (2013), "Measuring anisotropic muscle stiffness properties using elastography." *NMR in Biomedicine*, 26.

Guennebaud, Gaël, Benoît Jacob, et al. (2018), "Eigen v3." URL http://eigen.tuxfamily.org.

Guo, Dahai, William Gropp, and Luke N Olson (2016), "A hybrid format for better performance of sparse matrix-vector multiplication on a gpu." *International Journal of High Performance Computing Applications*, 30.

Hasler, N., C. Stoll, M. Sunkel, B. Rosenhahn, and H.-P. Seidel (2009), "A statistical model of human pose and body shape." *Computer Graphics Forum*, 28, 337–346.

Heymsfield, Steven B, Moonseong Heo, Diana Thomas, and Angelo Pietrobelli (2011), "Scaling of body composition to height: relevance to height-normalized indexes." *The American journal of clinical nutrition*, 93, 736–740.

Higham, Nicholas J (1986), "Computing the polar decomposition with applications." *SIAM J. Sci. Stat. Comput.*, 7, 1160–1174.

Holden, Daniel, Bang Chi Duong, Sayantan Datta, and Derek Nowrouzezahrai (2019), "Subspace neural physics: fast data-driven interactive simulation." In *Proc. of ACM SIGGRAPH/Eurographics Symposium on Computer Animation*.

Horn, Berthold KP, Hugh M Hilden, and Shahriar Negahdaripour (1988), "Closed-form solution of absolute orientation using orthonormal matrices." *Journal of the Optical Society of America A*, 5, 1127–1135.

Hu, Yixin, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo (2018), "Tetrahedral meshing in the wild." *ACM Transactions on Graphics*, 37.

Ichim, Alexandru-Eugen, Petr Kadleček, Ladislav Kavan, and Mark Pauly (2017), "Phace: Physics-based face modeling and animation." *ACM Transactions on Graphics*, 36, 153:1–153:14.

Ichim, Alexandru-Eugen, Ladislav Kavan, Merlin Nimier-David, and Mark Pauly (2016), "Building and animating user-specific volumetric face rigs." In *Symposium on Computer Animation*, 107–117.

Jackson, Andrew S. and Michael L. Pollock (1985), "Practical assessment of body composition." *The Physician and Sportsmedicine*, 13, 76–90.

Jacobson, Alec, Ilya Baran, Ladislav Kavan, Jovan Popović, and Olga Sorkine (2012), "Fast automatic skinning transformations." *ACM Transactions on Graphics*, 31, 77:1–77:10.

Jacobson, Alec, Ilya Baran, Jovan Popović, and Olga Sorkine (2011), "Bounded biharmonic weights for real-time deformation." *ACM Transactions on Graphics*, 30, 78:1–78:8.

Jacobson, Alec, Zhigang Deng, Ladislav Kavan, and J.P. Lewis (2014), "Skinning: Real-time shape deformation." In *ACM SIGGRAPH Courses*.

Jacobson, Alec and Olga Sorkine (2011), "Stretchable and twistable bones for skeletal shape deformation." *ACM Transactions on Graphics*, 30, 165:1–165:8.

Kadleček, Petr, Alexandru-Eugen Ichim, Tiantian Liu, Jaroslav Křivánek, and Ladislav Kavan (2016), "Reconstructing personalized anatomical models for physics-based body animation." *ACM Transactions on Graphics*, 35.

Kavan, Ladislav, Steven Collins, Jiří Žára, and Carol O'Sullivan (2008), "Geometric skinning with approximate dual quaternion blending." *ACM Transactions on Graphics*, 27.

Kavan, Ladislav and Olga Sorkine (2012), "Elasticity-inspired deformers for character articulation." *ACM Transactions on Graphics*, 31, 196:1–196:8.

Kim, Meekyoung, Gerard Pons-Moll, Sergi Pujades, Seungbae Bang, Jinwook Kim, Michael J. Black, and Sung-Hee Lee (2017), "Data-driven physics for human soft tissue animation." *ACM Transactions on Graphics*, 36.

Kimmel, R. and J. A. Sethian (1998), "Computing geodesic paths on manifolds." *Proceedings of the National Academy of Sciences*, 95.

Kugelstadt, Tassilo, Dan Koschier, and Jan Bender (2018), "Fast corotated fem using operator splitting." *Computer Graphics Forum*, 37.

Kyle, Ursula G, Laurence Genton, Daniel O Slosman, and Claude Pichard (2001), "Fat-free and fat mass percentiles in 5225 healthy subjects aged 15 to 98 years." *Nutrition*, 17, 534–541.

Lan, Lei, Ran Luo, Marco Fratarcangeli, Weiwei Xu, Huamin Wang, Xiaohu Guo, Junfeng Yao, and Yin Yang (2020), "Medial elastics: Efficient and collision-ready deformation via medial axis transform." *ACM Transactions on Graphics*, 39.

Le, Binh Huy and Jessica K. Hodgins (2016), "Real-time skeletal skinning with optimized centers of rotation." *ACM Transactions on Graphics*, 35, 37:1–37:10.

Le, Binh Huy and JP Lewis (2019), "Direct delta mush skinning and variants." *ACM Transactions on Graphics*, 38.

Lewis, J.P., Matt Cordner, and Nickson Fong (2000), "Pose space deformations: A unified approach to shape interpolation and skeleton-driven deformation." In *Proc. of SIGGRAPH*, 165–172.

Li, Jing, Tiantian Liu, and Ladislav Kavan (2019), "Fast simulation of deformable characters with articulated skeletons in projective dynamics." In *Proc. of ACM SIGGRAPH/Eurographics Symposium on Computer Animation*.

Li, Tianxing, Rui Shi, and Takashi Kanai (2021), "Multiresgnet: Approximating nonlinear deformation via multi-resolution graphs." *Computer Graphics Forum*, 40.

Liu, Tiantian, Sofien Bouaziz, and Ladislav Kavan (2017), "Quasi-newton methods for real-time simulation of hyperelastic materials." *ACM Transactions on Graphics*, 36, 23:1–23:16.

Liu, Weifeng, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter (2016), "A synchroniza-tion-free algorithm for parallel sparse triangular solves." In *Proc. of European Conference on Parallel Processing*.

Loper, Matthew, Naureen Mahmood, Javier Romero, Gerard Pons-Moll, and Michael J. Black (2015), "Smpl: A skinned multi-person linear model." *ACM Transactions on Graphics*, 34.

Maalin, Nadia, Sophie Mohamed, Robin SS Kramer, Piers L Cornelissen, Daniel Martin, and Martin J Tovée (2020), "Beyond BMI for self-estimates of body size and shape: A new method for developing stimuli correctly calibrated for body composition." *Behavior Research Methods*.

Macklin, Miles and Matthias Müller (2021), "A constraint-based formulation of stable neo-hookean materials." In *Motion, Interaction and Games*, 1–7.

Macklin, Miles, Matthias Müller, and Nuttapong Chentanez (2016), "Xpbd: Position-based simulation of compliant constrained dynamics." In *Proceedings of the 9th International Conference on Motion in Games*, 49–54.

Magnenat-Thalmann, Nadia, Richard Laperrière, and Daniel Thalmann (1988), "Joint-dependent local deformations for hand animation and object grasping." In *Proc. of Graphics Interface*.

Mancewicz, Joe, Matt L Derksen, Hans Rijpkema, and Cyrus A Wilson (2014), "Delta mush: Smoothing deformations while preserving detail." In *Proceedings of the Fourth Symposium on Digital Production*, 7–11.

Martin, Sebastian, Peter Kaufmann, Mario Botsch, Eitan Grinspun, and Markus Gross (2010), "Unified simulation of elastic rods, shells, and solids." *ACM Transactions on Graphics*, 29.

Martin, Sebastian, Bernhard Thomaszewski, Eitan Grinspun, and Markus Gross (2011), "Example-based elastic materials." In *ACM SIGGRAPH 2011 Papers*, 1–8.

McAdams, Aleka, Yongning Zhu, Andrew Selle, Mark Empey, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis (2011), "Efficient elasticity for character skinning with contact and collisions." *ACM Transactions on Graphics*, 30.

Monakov, Alexander, Anton Lokhmotov, and Arutyun Avetisyan (2010), "Automatically tuning sparse matrix-vector multiplication for gpu architectures."

In *Proc. of International Conference on High-Performance Embedded Architectures and Compilers*.

Müller, Matthias and Markus Gross (2004a), “Interactive virtual materials.” In *Proc. of Graphics Interface*.

Müller, Matthias, Bruno Heidelberger, Marcus Hennix, and John Ratcliff (2007), “Position based dynamics.” *J. Vis. Comun. Image Represent.*, 18, 109–118.

Müller, Matthias, Bruno Heidelberger, Matthias Teschner, and Markus Gross (2005), “Meshless deformations based on shape matching.” *ACM Transactions on Graphics*, 24.

Müller, Matthias, Matthias Teschner, and Markus Gross (2004b), “Physically based simulation of objects represented by surface meshes.” In *Proc. of Computer Graphics International*.

Myronenko, Andriy and Xubo B. Song (2009), “On the closed-form solution of the rotation matrix arising in computer vision problems.” *CoRR*, abs/0904.1613, URL http://arxiv.org/abs/0904.1613.

Narain, Rahul, Matthew Overby, and George E. Brown (2017), “ADMM $\supseteq$ projective dynamics: Fast simulation of hyperelastic models with dynamic constraints.” *IEEE Transaction on Visualization and Computer Graphics*, 23, 2222–2234.

Naumov, Maxim (2011), “Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu.” Technical Report NVR-2011-001, NVIDIA Corporation.

Ng, B. K., B. J. Hinton, B. Fan, A. M. Kanaya, and J. A. Shepherd (2016), “Clinical anthropometrics and body composition from 3D whole-body surface scans.” *European Journal of Clinical Nutrition*, 70, 1265–1270.

NVIDIA, C (2021), “Cuda c++ programming guide.” https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 2021-09-03.

Pan, Junjun, Lijuan Chen, Yuhan Yang, and Hong Qin (2017), “Automatic skinning and weight retargeting of articulated characters using extended position-based dynamics.” *The Visual Computer*, 34.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay (2011), “Scikit-learn: Machine learning in Python.” *Journal of Machine Learning Research*, 12, 2825–2830.

Peng, Yue, Bailin Deng, Juyong Zhang, Fanyu Geng, Wenjie Qin, and Ligang Liu (2018), "Anderson acceleration for geometry optimization and physics simulation." *ACM Transactions on Graphics*, 37.

Piryankova, I., J. Stefanucci, J. Romero, S. de la Rosa, M. Black, and B. Mohler (2014), "Can I recognize my body's weight? The influence of shape and texture on the perception of self." *ACM Transactions on Applied Perception*, 11, 13:1–13:18.

Riviere, Jérémy, Paulo Gotardo, Derek Bradley, Abhijeet Ghosh, and Thabo Beeler (2020), "Single-shot high-quality facial geometry and skin appearance capture." *ACM Transactions on Graphics*, 39, 81:1–81:12.

Robinette, Kathleen M, Sherri Blackwell, Hein Daanen, Mark Boehmer, and Scott Fleming (2002), "Civilian american and european surface anthropometry resource (ceasar), final report. volume 1: Summary." Technical report, Sytronics Inc.

Rohmer, Damien, Marco Tarini, Niranjan Kalyanasundaram, Faezeh Moshfeghifar, Marie-Paule Cani, and Victor Zordan (2021), "Velocity skinning for real-time stylized skeletal animation." *Computer Graphics Forum*, 40.

Romero, Cristian, Miguel A Otaduy, Dan Casas, and Jesus Perez (2020), "Modeling and estimation of nonlinear skin mechanics for animated avatars." *Computer Graphics Forum*, 39, 77–88.

Roussellet, Valentin, Nadine Abu Rumman, Florian Canezin, Nicolas Mellado, Ladislav Kavan, and Loïc Barthe (2018), "Dynamic implicit muscles for character skinning." *Computers & Graphics*, 77, 227–239.

Rumman, Nadine Abu and Marco Fratarcangeli (2014), "Position based skinning of skeleton-driven deformable characters." In *Proc. of Spring Conference on Computer Graphics*, 83–90.

Rumman, Nadine Abu and Marco Fratarcangeli (2015), "Position-based skinning for soft articulated characters." *Computer Graphics Forum*, 34.

Saito, Shunsuke, Zi-Ye Zhou, and Ladislav Kavan (2015), "Computational bodybuilding: Anatomically-based modeling of human bodies." *ACM Transactions on Graphics*, 34.

Santesteban, Igor, Elena Garces, Miguel A Otaduy, and Dan Casas (2020), "Softsmpl: Data-driven modeling of nonlinear soft-tissue dynamics for parametric humans." *Computer Graphics Forum*, 39.

Shoemake, Ken and Tom Duff (1992), "Matrix animation and polar decomposition." In *Proceedings of the Conference on Graphics Interface*, 258—264.

Si, Hang (2015), "Tetgen, a delaunay-based quality tetrahedral mesh generator." *ACM Trans. Math. Softw.*, 41, 11:1–11:36.

Sieger, Daniel, Stefan Menzel, and Mario Botsch (2013), "High quality mesh morphing using triharmonic radial basis functions." In *Proceedings of the 21st International Meshing Roundtable*, 1–15.

Siri, William E. (1956), "Body composition from fluid spaces and density: analysis of methods." Technical Report UCRL-3349, Lawrence Berkeley National Laboratory.

Sorkine, Olga and Marc Alexa (2007), "As-rigid-as-possible surface modeling." In *Proc. of Eurographics Symposium on Geometry Processing*, 109–116.

Su, Ching-Lung, Po-Yu Chen, Chun-Chieh Lan, Long-Sheng Huang, and Kuo-Hsuan Wu (2012), "Overview and comparison of opencl and cuda technology for gpgpu." In *IEEE Asia Pacific Conference on Circuits and Systems*, 448–451.

Tapia, Javier, Cristian Romero, Jesús Pérez, and Miguel A Otaduy (2021), "Parametric skeletons with reduced soft-tissue deformations." *Computer Graphics Forum*.

Teschner, Matthias, Bruno Heidelberger, Matthias Müller, Danat Pomerantes, and Markus Gross (2003), "Optimized spatial hashing for collision detection of deformable objects." In *Proc. of Vision, Modeling and Visualization*.

Tomlinson, DJ, RM Erskine, CI Morse, K Winwood, and Gladys Onambélé Pearson (2016), "The impact of obesity on skeletal muscle strength and structure through adolescence to old age." *Biogerontology*, 17, 467–483.

Vaillant, Rodolphe, Loïc Barthe, Gaël Guennebaud, Marie-Paule Cani, Damien Rohmer, Brian Wyvill, Olivier Gourmel, and Mathias Paulin (2013), "Implicit skinning: Real-time skin deformation with contact modeling." *ACM Transactions on Graphics*, 32, 125:1–125:12.

Vaillant, Rodolphe, Gäel Guennebaud, Loïc Barthe, Brian Wyvill, and Marie-Paule Cani (2014), "Robust iso-surface tracking for interactive character skinning." *ACM Transactions on Graphics*, 33.

Vázquez, Francisco Bonilla, Ester M. Garzón, J. A. Martínez, and Jonathan Carrero Fernández (2009), "The sparse matrix vector product on gpus." In *Proc. of International Conference on Computational and Mathematical Methods in Science and Engineering*.

Wang, Huamin (2015), "A chebyshev semi-iterative approach for accelerating projective and position-based dynamics." *ACM Transactions on Graphics*, 34.

Wang, Xiaohuan Corina and Cary Phillips (2002), "Multi-weight enveloping: Least-squares approximation techniques for skin animation." In *Proc. of ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 129–138.

Weber, Daniel, Jan Bender, Markus Schnoes, André Stork, and Dieter Fellner (2013), "Efficient gpu data structures and methods to solve sparse linear systems in dynamics applications." *Computer Graphics Forum*, 32.

Weber, Ofir, Olga Sorkine, Yaron Lipman, and Craig Gotsman (2007), "Context-aware skeletal shape deformation." *Computer Graphics Forum*, 26, 265–274.

Weng, Chung-Yi, Brian Curless, and Ira Kemelmacher-Shlizerman (2019), "Photo Wake-Up: 3D character animation from a single photo." In *Proc. of IEEE Conference on Computer Vision and Pattern Recognition*, 5908–5917.

Wenninger, Stephan, Jascha Achenbach, Andrea Bartl, Marc Erich Latoschik, and Mario Botsch (2020), "Realistic virtual humans from smartphone videos." In *Proc. of ACM Symposium on Virtual Reality Software and Technology*, 29:1–29:11.

Xian, Zangyueyang, Xin Tong, and Tiantian Liu (2019), "A scalable galerkin multigrid method for real-time simulation of deformable objects." *ACM Transactions on Graphics*, 38.

Xu, Zhan, Yang Zhou, Evangelos Kalogerakis, Chris Landreth, and Karan Singh (2020), "Rignet: Neural rigging for articulated characters." *ACM Transactions on Graphics*, 39.

Yoshizawa, Hiroki and Daisuke Takahashi (2012), "Automatic tuning of sparse matrix-vector multiplication for crs format on gpus." In *Proc. of IEEE International Conference on Computational Science and Engineering*.

Zhang, Jiayi Eris, Seungbae Bang, David I. W. Levin, and Alec Jacobson (2020), "Complementary dynamics." *ACM Transactions on Graphics*, 39.

Zhu, Lifeng, Xiaoyan Hu, and Ladislav Kavan (2015), "Adaptable anatomical models for realistic bone motion reconstruction." *Computer Graphics Forum*, 34, 459–471.

Ziylan, Taner and Khalil Awadh Murshid (2002), "An analysis of anatolian human femur anthropometry." *Turkish Journal of Medical Sciences*, 32, 231–235.

Zygote (2020). https://www.zygote.com.

APPENDIX A

Intersections between Lines and the FPS Skeleton

In order to shrink the skin onto the simplified skeletons that we use in our Fast Projective Skinning (see Section 3.2.1), we need to compute the intersection between a line (from skin to skeleton's bone-line) and a cylinder or a sphere. Each point $\mathbf{x}_l$ on a line between two points $\mathbf{l}_0$ and $\mathbf{l}_1$ can be defined via

$$\mathbf{x}_l(\lambda) = \mathbf{l}_0 + \lambda \mathbf{l}, \tag{A.1}$$

with the line's direction $\mathbf{l} = (\mathbf{l}_1 - \mathbf{l}_0) / \|\mathbf{l}_1 - \mathbf{l}_0\|$ and $\lambda_l \in [0, 1]$. Each point $\mathbf{x}$ on a sphere with center $\mathbf{c}_0$ and radius r_0 satisfies

$$\|\mathbf{x} - \mathbf{c}_0\| = r_0 \tag{A.2}$$

To define a cylinder of the same radius between $\mathbf{c}_0$ and another point $\mathbf{c}_1$ (length $C = \|\mathbf{c}_1 - \mathbf{c}_0\|$, direction $\mathbf{c} = (\mathbf{c}_1 - \mathbf{c}_0)/C$), the left term in (A.2) can be altered to the shortest distance from $\mathbf{x}$ to the mid-line

$$\left\| (\mathbf{x} - \mathbf{c}_0) - \left((\mathbf{x} - \mathbf{c}_0)^\mathsf{T} \mathbf{c} \right) \mathbf{c} \right\| = r_0. \tag{A.3}$$

In the region between a bone's cylinder and a joint's sphere of an FPS-skeleton, we test for intersections with a conical frustum (see Figure A.1). Starting from (A.3), we can create a conical frustum by linearly interpolating the radius from

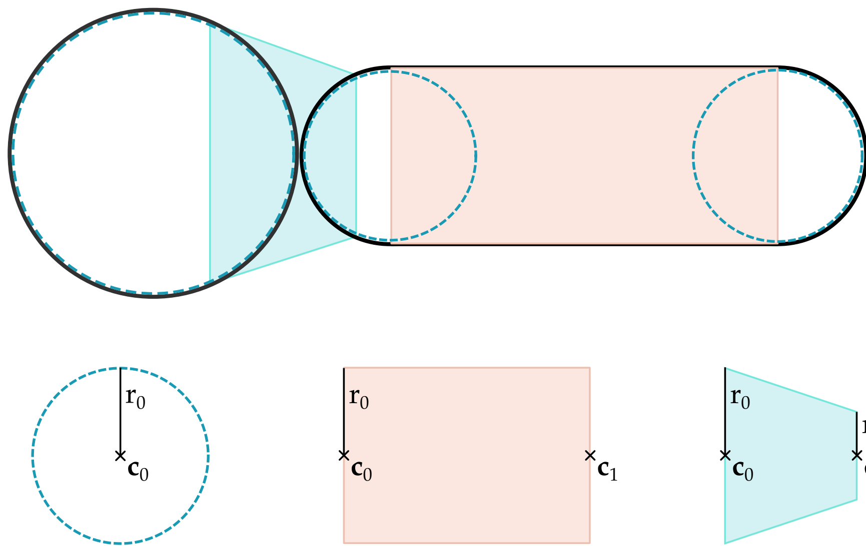


Figure A.1: In order to get a tube-like wrap around our volumetric skeleton (black) we check for intersections with three different shapes: spheres (dashed blue), cylinders (filled red) and conical frustums (filled blue).

r_0 (at $\mathbf{c}_0$) to a second radius r_1 (at $\mathbf{c}_1$)

$$\left\| (\mathbf{x} - \mathbf{c}_0) - \left((\mathbf{x} - \mathbf{c}_0)^\top \mathbf{c} \right) \mathbf{c} \right\| = r_0 + \left((\mathbf{x} - \mathbf{c}_0)^\top \mathbf{c} \right) \frac{r_1 - r_0}{C}. \tag{A.4}$$

(A.4) can be re-transformed to (A.3) by setting $r_1 = r_0$, and similarly (A.3) to (A.2) by setting $\mathbf{c} = \mathbf{0}$. Therefore, we first solve the general case of intersecting a line with a conic frustum by inserting (A.1) into (A.4). The point of intersection is $\mathbf{x}_S = \mathbf{x}_l(\lambda_S)$ with

$$\begin{aligned}
\lambda_S &= (-b - \sqrt{d})/a \\
a &= 1 - (\mathbf{l}^\top \mathbf{c})^2 (1 + r_d^2) \\
b &= \mathbf{l}^\top (\mathbf{l}_0 - \mathbf{c}_0) - (\mathbf{l}^\top \mathbf{c}) \left((\mathbf{l}_0 - \mathbf{c}_0)^\top \mathbf{c} \right) (1 + r_d^2) - (\mathbf{l}^\top \mathbf{c}) r_d r_0 \\
c &= (\mathbf{l}_0 - \mathbf{c}_0)^2 - r_0^2 - \left((\mathbf{l}_0 - \mathbf{c}_0)^\top \mathbf{c} \right)^2 (1 + r_d^2) - 2 r_d r_0 (\mathbf{l}_0 - \mathbf{c}_0)^\top \mathbf{c} \\
d &= b^2 - ac \\
r_d &= (r_1 - r_0)/C.
\end{aligned}$$

In case of a line-cylinder intersection ($r_d = 0$), variables a, b, c simplify to

$$\begin{aligned}
a &= 1 - (\mathbf{l}^\top \mathbf{c})^2 \\
b &= \mathbf{l}^\top (\mathbf{l}_0 - \mathbf{c}_0) - (\mathbf{l}^\top \mathbf{c}) \left((\mathbf{l}_0 - \mathbf{c}_0)^\top \mathbf{c} \right) \\
c &= (\mathbf{l}_0 - \mathbf{c}_0)^2 - r_0^2 - \left((\mathbf{l}_0 - \mathbf{c}_0)^\top \mathbf{c} \right)^2,
\end{aligned}$$

and in case of a line-sphere intersection ($\mathbf{c} = \mathbf{0}$) these further simplify to

$$\begin{aligned}
a &= 1 \\
b &= \mathbf{l}^\top (\mathbf{l}_0 - \mathbf{c}_0) \\
c &= (\mathbf{l}_0 - \mathbf{c}_0)^2 - r_0^2
\end{aligned}$$

There is no intersection if $d < 0$. Additionally, both $(\mathbf{x}_S - \mathbf{c}_0)^\top \mathbf{c}$ and $(\mathbf{c}_1 - \mathbf{x}_S)^\top \mathbf{c}$ must be positive in case of a conic or cylinder. Otherwise, the resulting intersection is not located in between the two confining points ($\mathbf{c}_0$ and $\mathbf{c}_1$). To take the the spherical caps of the volumetric bones into account, we apply two additional line-sphere intersections per bone. The two conic radii are set to 80% of the connected bone radius and joint radius, respectively. We check for intersections with all volumetric elements of the skeleton and choose the one with the smallest positive value of λ_S, which yields the final point of the skeleton surface.

Iterative Polar Decomposition Algorithm

Computing the polar decomposition of a 3×3 matrix $\mathbf{F} = \mathbf{RS}$ via singular value decomposition is computationally expensive. In case of $\det(\mathbf{F}) > 10^{-5}$, the closest rotation $\mathbf{R}$ can also be found using the faster, iterative approach of Higham [1986]. The algorithm initializes $\mathbf{R}_0 = \mathbf{F}$ and iterates

$$\mathbf{R}_{k+1} = \frac{1}{2}(\mathbf{R}_k + \mathbf{R}_k^{-\mathsf{T}}), \tag{A.5}$$

which converges to the rotation $\mathbf{R}$ as explained in the following.

Inserting the singular value decomposition $\mathbf{F} = \mathbf{U\Sigma V}^\mathsf{T}$ into the first iteration of (A.5) yields

$$\mathbf{R}_1 = \frac{1}{2}(\mathbf{F} + \mathbf{F}^{-\mathsf{T}}) = \frac{1}{2}(\mathbf{U\Sigma V}^\mathsf{T} + \mathbf{U\Sigma}^{-1}\mathbf{V}^\mathsf{T}) = \mathbf{U}\left(\frac{1}{2}(\mathbf{\Sigma} + \mathbf{\Sigma}^{-1})\right)\mathbf{V}^\mathsf{T}.$$

Each update can therefore be formulated as

$$\begin{aligned} \mathbf{R}_{k+1} &= \mathbf{U\Sigma}_{k+1}\mathbf{V}^\mathsf{T} \\ \mathbf{\Sigma}_{k+1} &= \frac{1}{2}(\mathbf{\Sigma}_k + \mathbf{\Sigma}_k^{-1}), \end{aligned}$$

with $\mathbf{\Sigma}_0 = \mathbf{\Sigma}$. Since $\mathbf{\Sigma}$ is diagonal and holds the singular values of $\mathbf{F}$, each diagonal value is updated via $\sigma_{k+1} = (\sigma_k + 1/\sigma_k)/2$, which can be identified as Heron's method $x_{k+1} = (x_k + a/x_k)/2$ for iteratively computing the square root of $a = 1$, or equivalently, Newton's method for finding the root of $f(x) = x^2 - 1$. Hence, $\mathbf{\Sigma}_k$ converges to identity and thereby $\mathbf{R}_k \to \mathbf{UV}^\mathsf{T}$, which is equivalent to the rotational part of $\mathbf{F}$ as long as $\det(\mathbf{UV}^\mathsf{T})$ is positive. The latter is guaranteed by our initial condition $\det(F) > 10^{-5}$. Higham [1986] also shows that the algorithm converges quadratically and can be further accelerated with the help of an additional parameter. For our small 3×3 matrices, this variation showed no noticeable speed-up, hence, we simply use Equation (A.5) in our implementation of Fast Projective Skinning. We stop iterating if $\|\mathbf{R}_{k+1} - \mathbf{R}_k\|^2 < 10^{-5}$ which in general takes less than 4 iterations.