Fast Projective Skinning

Martin Komaritzan
Computer Graphics Group
Bielefeld University

Mario Botsch
Computer Graphics Group
Bielefeld University

Figure 1: Geometric skinning approaches, like Linear Blend Skinning (LBS) or Dual Quaternion Skinning (DQS), suffer from well-known collapsing or bulging artifacts in joint regions. The recent physics-based Projective Skinning (PS) avoids these problems and resolves *local* collisions near joints. Our new Fast Projective Skinning (FPS) is one order of magnitude faster than Projective Skinning, provides higher surface quality, and seamlessly handles *global* collisions.

ABSTRACT

We present a novel physics-based character skinning approach that improves the recent Projective Skinning in terms of animation quality and computational performance. Our method provides physically plausible animations, dynamic secondary motion effects, and global collision handling in a real-time skinning simulation. We achieve this through a custom-tailored GPU implementation of the underlying projective dynamics simulation and a high-quality upsampling from the simulation mesh to the high-resolution visualization mesh based on quadratic moving least squares.

CCS CONCEPTS

• **Computing methodologies** → **Physical simulation**; *Motion processing*; *Collision detection.*

KEYWORDS

character animation, projective dynamics, GPU computing

ACM Reference Format:
Martin Komaritzan and Mario Botsch. 2019. Fast Projective Skinning. In *Motion, Interaction and Games (MIG '19), October 28–30, 2019, Newcastle upon Tyne, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3359566.3360073

MIG '19, October 28–30, 2019, Newcastle upon Tyne, United Kingdom

ACM ISBN 978-1-4503-6994-7/19/10…$15.00
https://doi.org/10.1145/3359566.3360073

1 INTRODUCTION

Virtual characters are ubiquitous in a wide range of graphics applications from real-time computer games to (offline) special effects in movies. 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. In particular in this rapidly growing field of research, the steadily improving fidelity of character appearance increases the demand for more realistic character animation—while retaining interactive frame rates.

Geometric skinning approaches, like the well-known linear blend skinning [Magnenat-Thalmann et al. 1988] and dual quaternion skinning [Kavan et al. 2008], provide real-time performance, but suffer from collapsing or bulging artifacts near joints (Figure 1). Data-driven methods, like the SMPL model [Loper et al. 2015] and its variants, learn corrective blendshapes from a potentially large set of training examples. While they successfully avoid the artifacts of geometric skinning approaches, these methods still lack a proper handling of contact and (self-)collision. Sophisticated physics-based skinning approaches, such as [Kadleček et al. 2016; McAdams et al. 2011b], yield realistic dynamic tissue deformation including collision detection and response, but their computational performance is not sufficient for interactive applications.

Recent simulation frameworks, such as position-based dynamics [Bender et al. 2017] or projective dynamics [Bouaziz et al. 2014], make it possible to compute character animation through robust real-time physics simulation, as demonstrated in our recent projective skinning [Komaritzan and Botsch 2018]. This technique provides realistic deformation at interactive frame-rates and precomputed *local* collisions in joint regions, but lacks global collisions (Figure 1). Similarly, the combination of data-driven skinning with

a simulated physics layer of Kim et al. [2017] provides realistic secondary motions, but lacks global collision handling.

We present a method for real-time physics-based skinning animation that overcomes the above limitations. Building on and extending the projective skinning method [Komaritzan and Botsch 2018], our approach simulates plausible dynamic tissue deformations, provides secondary motion effects like jiggling and wobbling, and resolves arbitrary global self-collisions, while still being faster by an order of magnitude than the original method. We achieve this goal by deriving a GPU-implementation of projective skinning 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. Furthermore, we develop an upsampling technique for transferring the deformation from the lower-resolution simulation mesh to the high-resolution visualization mesh, which improves the upsampling of the original projective dynamics in both surface quality and computational performance.

Overall, the proposed *Fast Projective Skinning* is to our knowledge the first real-time skinning method to provide physics-based dynamic deformations and full global collision handling. To foster research in real-time physics-based character animation, we make our source code freely available for research purposes at https://github.com/mbotsch/FastProjectiveSkinning.

2 RELATED WORK

In this section we focus on work related to our contributions on collision-aware physics-based skinning and the underlying projective dynamics simulation. For details on general or physics-based collision-agnostic skinning we refer to the course notes of Jacobson et al. [2014] or the discussions in [Komaritzan and Botsch 2018].

Collision-Aware Skinning Methods

Collision response in character skinning is a challenging task since collision detection of deformable objects is expensive and there is also a high rate of contact collisions that do not separate once they are solved and hence require special treatment. Vaillant et al. [2014] introduced Implicit Skinning that represents different body parts as implicit functions, providing fast collision detection/response by an iso-surface projection. Though the resulting skin deformation looks convincing, the method is too slow for real-time applications. While the collision handling works for small local collisions, it is an unnatural approximation for global collisions. The Steklov-Poincare Skinning of Gao et al. [2014] can handle collisions, but is also too slow and does not provide dynamic secondary motion effects.

Physics simulations have also been used to simulate realisic skin behavior, but this comes at the cost of high computation times. For instance, Kavan and colleagues [Kadleček et al. 2016; Saito et al. 2015] simulate a biomechanical model of the human body. McAdams et al. [McAdams et al. 2011b] 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. [2005] achieved interactive frame-rates by using a very coarse simulation mesh. Kim et al. [2017] add a simulated FEM layer on top of a data-driven skinning animation, which allows for dynamic motion effects and physics-based interaction. Collision handling could be added to their method, but at the price of sacrifycing real-time performance. Casas and Otaduy [2018] learn nonlinear soft-tissue dynamics from training data, resulting in highly efficient animations, but do not handle collisions. Holden et al. [2019] show that collision handling can be learned by a neural network, but so far they do not handle self collisions. Position-Based Dynamics (PBD) solvers [Bender et al. 2017; Macklin et al. 2016; Müller et al. 2005] have been widely used to simulate all kinds of physical systems, since they are fast, easy to use, and unconditionally stable. Deul et al. [2013] uses PBD and shape matching to simulate a layered character model achieving convincing results and supporting collision handling. But they depend on high quality skinning weights to create their volumetric mesh and their simulation of a full character is too slow for real-time skinning.

There are, to our knowledge, just two approaches that support both collision handling and real-time performance while skinning an articulated virtual character. Pan et al. [2017] extends the PBD skinning of Rumman et al. [2015] by handling *local* collisions. They initialize the character by 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. Komaritzan and Botsch [2018] use Projective Dynamics (PD) [Bouaziz et al. 2014] for their Projective Skinning, which is independent of skinning weights and requires just a triangle mesh and a skeleton as input. While the skeleton-driven motion of a full character can be computed in real-time, collisions are restricted to pre-computed *local* collision pairs in joints regions. We extend this method to handle full *global* collisions.

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 iterations are 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. 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 Projective Skinning), where less iterations are required.

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. Their approach can be considered complementary to our acceleration and could be used to further speed-up Projective Skinning. Li et al. [2019] accelerate PD-simulations of 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.

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 mesh [2004] or hexahedral mesh [2004] for simulation. However, the *piecewise* (tri-)linear interpolation to transform the visualization vertices can lead to visual artifacts. Projective Skinning [Komaritzan and Botsch 2018] represent the visualization mesh as a normal displacement [Botsch and Sorkine 2008] of the simulation mesh, where the piecewise linear nature of the latter can again lead to artifacts, 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 PROJECTIVE SKINNING

This section provides a brief summary of Projective Skinning (PS) and defines the mathematical notation. For further details on projective skinning and the underlying projective dynamics we refer the reader to [Komaritzan and Botsch 2018] and [Bouaziz et al. 2014].

Projective Skinning gets as input a character surface mesh and an embedded skeleton. The latter is inflated to a volumetric representation consisting of spherical joints and cylindrical bones. The skin surface is then shrink-wrapped onto the volumetric skeleton, such that each skin vertex/triangle has a corresponding bone vertex/triangle. All pairs of corresponding skin and bone triangles define prismatic elements, which are split into three tetrahedra each, thereby defining the volumetric tissue mesh. This construction allows us to associate each tetrahedron with a surface triangle, which we exploit for collision handling (Section 6).

The character pose is changed by manipulating joint angles, from which a rigid transformation for each bone is determined through forward kinematics. Bone vertices should move according to their bone's rigid transformation, while the animated positions of skin vertices are determined by minimizing an elastic strain energy for the tissue tetrahedra. This elastic deformation is computed through Projective Dynamics [Bouaziz et al. 2014], using anchor constraint to fix bone vertices to their bones and tetrahedron strain constraints to penalize deformation of tetrahedra, i.e., to minimize their deviations from rigid motions.

These constraints define a matrix $\mathbf{Q} \in \mathbb{R}^{P \times N}$, where N denotes the number of simulated skin and bone vertices and P the overall number of constraints (one row per anchor constraint and four rows per tetrahedron strain constraint). The *local step* of Projective Dynamics computes a target position $\mathbf{p}_i$ for each vertex i for each constraint, corresponding to the closest projection that satisfies the constraint. These individual projections $\mathbf{p}_i$ are stacked into a matrix $\mathbf{p} \in \mathbb{R}^{P \times 3}$. The unknown vertex positions $\mathbf{x}_i$ are stacked into a matrix $\mathbf{x} \in \mathbb{R}^{N \times 3}$, and are solved for in the *global step* through a least squares fit to the projections:

$$\mathbf{Q}^\mathsf{T}\mathbf{Q}\mathbf{x} = \mathbf{Q}^\mathsf{T}\mathbf{p}. \tag{1}$$

Dynamic tissue effects ("wobbling") can be computed by adding a mass term to the system

$$\left(\frac{1}{h^2}\mathbf{M} + \mathbf{Q}^\mathsf{T}\mathbf{Q}\right)\mathbf{x} = \frac{1}{h^2}\mathbf{M}\mathbf{s} + \mathbf{Q}^\mathsf{T}\mathbf{p}, \tag{2}$$

where $\mathbf{M}$ is a diagonal matrix of per-vertex masses m_i, h is the simulation time step, $\mathbf{s} = \mathbf{x} + h\mathbf{v}$ an explicit Euler prediction, and $\mathbf{v} \in \mathbb{R}^{N \times 3}$ a matrix containing velocities. The mass term acts like anchor constraints trying to satisfy Newton's first law by constraining vertices to their velocity-based predictions. The velocity is updated at the end of each time-step through $\mathbf{v}_{t+1} = \lambda(\mathbf{x}_{t+1} - \mathbf{x}_t)/h$, where $\lambda \in [0, 1]$ can be used as damping parameter (0 corresponding to infinite damping and 1 to no damping). One time-step of Projective Skinning is shown in Algorithm 1.

Algorithm 1 Time-step of the Projective Dynamics solver

1: $\mathbf{s} \leftarrow \mathbf{x}_t + h\mathbf{v}_t$
2: $\mathbf{x} \leftarrow \mathbf{s}$
3: **for** n_{pd} iterations **do**
4: $\quad \mathbf{p} \leftarrow$ projectConstraints($\mathbf{x}$)
5: $\quad \mathbf{x} \leftarrow$ *solve* $\left(h^{-2}\mathbf{M} + \mathbf{Q}^\mathsf{T}\mathbf{Q}\right)\mathbf{x} = h^{-2}\mathbf{M}\mathbf{s} + \mathbf{Q}^\mathsf{T}\mathbf{p}$
6: $\mathbf{x}_{t+1} \leftarrow \mathbf{x}$
7: $\mathbf{v}_{t+1} \leftarrow \lambda\left(\mathbf{x}_{t+1} - \mathbf{x}_t\right)/h$

4 GPU-BASED PROJECTIVE SKINNING

In this section we will derive an efficient parallel GPU implementation of Projective Skinning. Most of the simulation steps explained in Algorithm 1 are easy to parallelize on the GPU: The vector updates (lines 1, 2, 6, and 7) can be processed with one thread per vertex. The constraint projections (line 4) can be computed using one thread per projection. To find the best rotation for the tetrahedron strain constraints, we use the GPU implementation of Gao et al. [2018] of the SVD algorithm of McAdams et al. [2011a]. The only critical point left is solving the linear system and computing its right-hand side (line 5), which we discuss in the following.

4.1 GPU Conjugate Gradients

The system matrix $\mathbf{A} = h^{-2}\mathbf{M} + \mathbf{Q}^\mathsf{T}\mathbf{Q}$ in Algorithm 1 is sparse, symmetric, positive definite, and *constant*, such that its Cholesky factorization can be pre-computed. Since the three spatial coordinates can be solved for in parallel, a pre-factorized sparse Cholesky solver is the most efficient option 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 GPU parallelization ideas [Liu et al. 2016; Naumov 2011], 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, in our context the solution of the previous time-step can be used as an initial guess in the iterative solver, thus reducing the number of required iterations. We therefore 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 the CPU-based 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], using 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 (Section 4.2). 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 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 not supported by CUDA.

Algorithm 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: $\delta \leftarrow \mathbf{d}^\mathsf{T}\mathbf{r}$ } *Init-Kernel*
4: **for** n_{pcg} iterations **do**
5: $\gamma \leftarrow \delta$
6: $\mathbf{q} \leftarrow \mathbf{Ad}$
7: $\alpha \leftarrow \mathbf{d}^\mathsf{T}\mathbf{q}$ } *Kernel 1*
8: $\alpha \leftarrow \gamma / \alpha$
9: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{d}$
10: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{q}$
11: $\delta \leftarrow \mathbf{r}^\mathsf{T}\mathbf{Jr}$ } *Kernel 2*
12: $\beta \leftarrow \delta / \gamma$
13: $\mathbf{d} \leftarrow \mathbf{r} + \beta\mathbf{d}$ } *Kernel 3*

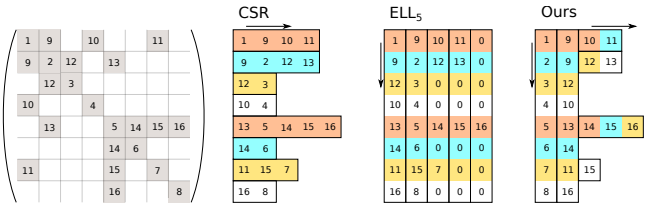

Figure 2: In CRS sparse matrix format, data is stored row-wise. If we use one thread per matrix row, the access pattern will be uncoalesced (first thread access in red, second blue, third yellow, fourth white; four threads per block used). The ELL format uses zero padding, and column-wise storage to support coalesced access. Our format is a combination of both formats to reduce zero padding while still guaranteeing coalesced access. We store diagonal elements first since they are needed for our preconditioner.

4.2 Matrix and Vector Storage

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, meaning that data is stored in structures of arrays rather than in an array of structures, since it can be cached this way so that consecutive threads will read and store consecutive data simultaneously. The most expensive step in Algorithm 2 is the sparse matrix-vector multiplication, where sparse matrix formats can reduce memory consumption, memory accesses, and computing operations.

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 an array of non-zeros per row. Unfortunately, each per-row thread accesses a different amount of elements in an uncoalesced way. The ELLPACK (ELL) matrix format therefore uses a per-row padding with zero elements to achieve a constant number of elements per row and a column-wise data storage (see Figure 2). This results in a coalesced access. Here it is important to know how many non-zero elements per row we typically have in a projective skinning 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 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.

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 amount of non-zeros per row for each thread block instead of for the complete matrix. There are a lot of other proposed formats that optimize the

Table 1: Timings (in ns) for sparse matrix-vector multiplication Ax for different matrix formats, for a high resolution character mesh of 30k simulated vertices. The last four cases use texture memory to speed up the random access into x. The CPU version is implemented with EIGEN and parallelized using OpenMP. In the 3D case, we process all three spatial xyz-coordinates of x by one single thread.

	1D		3D	
Method	timing	speedup	timing	speedup
CPU	393	1	510	1
cuSPARSE	14.4	27.3	25.8	19.7
CRS	23.3	16.9	35.1	14.5
ELL	16.9	23.3	22.0	23.2
CRS + ELL	17.7	22.2	26.1	19.5
Ours	8.3	47.3	18.5	27.5

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 use 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 that can be used 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 that is set to the minimum amount of non-zeros per row. The remaining row elements are stored in CRS format that are read coalesced, multiplied to the vector element, and stored in shared memory. The usage of shared memory enables us to read the data of all rows corresponding to the thread block by the complete thread block instead of using just one thread per row (see Figure 2 right). The amount of rows processed by the thread block is equal to the amount of threads per block. While the amount 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.

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 use texture memory here, 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.

A performance comparison for the PCG matrix-vector multiplication is shown in Table 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 53.7 ns. Handling all three coordinates in one thread then improves performance to 28.6 ns. Using texture memory to store the right-hand side and using shared memory to distribute/coalesce the operations on the CRS part finally results in the 18.5 ns reported in Table 1.

A drawback of our format is that we have to allocate an amount of shared memory that is equal to the number of CRS-values in the thread block. Since we cannot specify a different amount of shared memory for each block, we have to allocate the maximum amount of all blocks. Shared memory on our device (GTX 1080 TI, compute capability 5.2) can use up to 48 kB of shared memory per block and 96 kB per multiprocessor. Each multiprocessor can run up to 32 blocks with a maximum total of 2048 threads in parallel. Hence, even if we reduce the block-size, there is a limitation for the size of the CRS part of 12 non-zeros ($96\,\text{kB}/(4\,\text{B} \cdot 2048)$) 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 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 of the different formats becomes very small. The usage of just 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. We process each multiplication by a different thread and store the result in shared memory. After a thread block 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.

4.3 Soft Constraints vs Hard Constraints

In the original Project Skinning [Komaritzan and Botsch 2018], bone vertices were attached to their corresponding bones through anchor constraints, and skin vertices were transformed through the tetrahedron strain constraints. In the context of Equations (1) and (2), anchor constraints act as *soft constraints* by minimizing the deviation of vertex positions from their target locations in a least-squares manner.

We improve on this in a simple but effective manner: By replacing soft anchor constraints by *hard Dirichlet constraints*, we remove the N_b bone vertices from the set of unknowns and thereby reduce the degrees of freedom from all $N = N_b + N_s$ vertices to only the

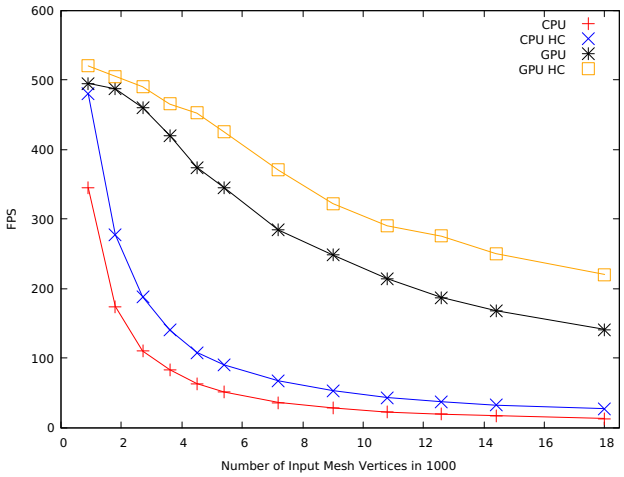

Figure 3: Performance of Projective Skinning (in frames-per-second) on CPU (Intel Xeon 6-core 3.6 GHz) or GPU (Nvidia GTX 1080 TI) and without or with hard constraints (HC).

N_s skin vertices. To this end, we partition the vectors and matrices involved in (2) according to bone and skin vertices:

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_s \\ \mathbf{x}_b \end{pmatrix}, \quad \mathbf{M} = \begin{pmatrix} \mathbf{M}_s & \mathbf{0} \\ \mathbf{0} & \mathbf{M}_b \end{pmatrix}, \quad \mathbf{Q} = \begin{pmatrix} \mathbf{Q}_s & \mathbf{Q}_b \end{pmatrix},$$

and analogously for $\mathbf{s}$. Note that $\mathbf{Q}$ and its sub-matrices (and the projections $\mathbf{p}$) now consist of (projections of) tetrahedron strain constraints only and do no longer include anchor constraints. This replaces the global system from (2) by the reduced version

$$\left(h^{-2}\mathbf{M}_s + \mathbf{Q}_s^\mathsf{T}\mathbf{Q}_s\right)\mathbf{x}_s = h^{-2}\mathbf{M}_s\mathbf{s}_s + \mathbf{Q}_s^\mathsf{T}\left(\mathbf{p} - \mathbf{Q}_b\mathbf{x}_b\right).$$

The above matrix is considerably smaller than the original one, since half of the vertices are removed from the system. The absence of soft constraints furthermore improves the matrix condition [Botsch and Sorkine 2008], which improves PCG convergence. Both effects lead to speed-up of 1.4 (for small meshes) up to 2.0 (for large meshes) on the CPU (Figure 3). On the GPU the performance gain is only about 5% for small meshes, because our 3584 CUDA cores can solve smaller systems completely in parallel, such that any further reduction just leaves some threads idle. For larger systems, however, the reduced formulation yields a speed-up of about 1.6 (see Figure 3), being up to 16× faster than the original CPU Projective Skinning.

5 UPSAMPLING

Like many physics-based simulations, Projective Skinning [Komaritzan and Botsch 2018] 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*. However, the simple normal displacements [Botsch and Sorkine 2008] used in [Komaritzan and Botsch 2018] for this purpose 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 4 and shown for the shoulder region in Figure 5d.

Figure 4: Upsampling the red mesh to the black one using precomputed normal displacements of the undeformed meshes (left) is not able to reproduce the deformation properly (middle) like our MLS upsampling (right).

Inspired by [Martin et al. 2010] we employ a moving-least-squares (MLS) approach to upsample deformations. To explain the idea, 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. We refer to [Fries and Matthies 2004] for more details on MLS interpolation.

Upsampling approaches operate on vertex displacements $\mathbf{u}(\mathbf{x}) = \mathbf{x}' - \mathbf{x}$, where $\mathbf{x}'$ denotes the vertex position in the animated skinned pose and $\mathbf{x}$ in the rest pose. Displacements of simulated vertices $\mathbf{u}_i = \mathbf{u}(\mathbf{x}_i)$ are known and are upsampled to $\mathbf{u}(\mathbf{x}_j)$ by fitting a local affine transformation through weighted least-squares minimization:

$$\min_{\mathbf{a}} \sum_i w\left(\left\|\mathbf{x}_j - \mathbf{x}_i\right\|\right) \left\|\mathbf{a}(\mathbf{x}_j)\,\boldsymbol{\pi}(\mathbf{x}_j) - \mathbf{u}_i\right\|^2, \tag{3}$$

where $\boldsymbol{\pi}(\mathbf{x}) = (1, x, y, z)^\mathsf{T}$ is a vector of linear monomials, $\mathbf{a}(\mathbf{x})$ a 3×4 matrix of coefficients, and $w(r)$ a smooth weighting function

$$w(r) = \begin{cases} (1 - r/\rho)^3 & r < \rho \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

with ρ defining the support radius of the MLS kernels. Minimizing (3) with respect to $\mathbf{a}$ results in the upsampled displacements

$$\mathbf{u}(\mathbf{x}_j) = \sum_i \mathbf{u}_i N_{ij} \quad \text{with} \tag{5}$$

$$N_{ij} = w\left(\left\|\mathbf{x}_j - \mathbf{x}_i\right\|\right) \boldsymbol{\pi}(\mathbf{x}_i)^\mathsf{T}\, \mathbf{G}_j^{-1}\, \boldsymbol{\pi}(\mathbf{x}_j), \tag{6}$$

$$\mathbf{G}_j = \sum_i w\left(\left\|\mathbf{x}_j - \mathbf{x}_i\right\|\right) \boldsymbol{\pi}(\mathbf{x}_i)\, \boldsymbol{\pi}(\mathbf{x}_i)^\mathsf{T}, \tag{7}$$

where all N_{ij} can be precomputed. This results in a simple and perfectly parallel update rule (5) for the visualization vertices $\mathbf{x}_j$.

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

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

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

This allows us to locally reproduce linear and quadratic transformation, which yields more faithful reconstructions in regions of strong bending (see Figure 5).

Third, in degenerate situations (Figure 5c) where $\mathbf{G}$ in (6) is not invertible (when the points $\mathbf{x}_i$ 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 $\mathbf{G}^{-1}$ by the Moore-Penrose pseudo-inverse $\mathbf{G}^{+}$. We further enhance the numerical robustness by mean-centering and scaling the points $\mathbf{x}_i$ within the support radius ρ (as well as

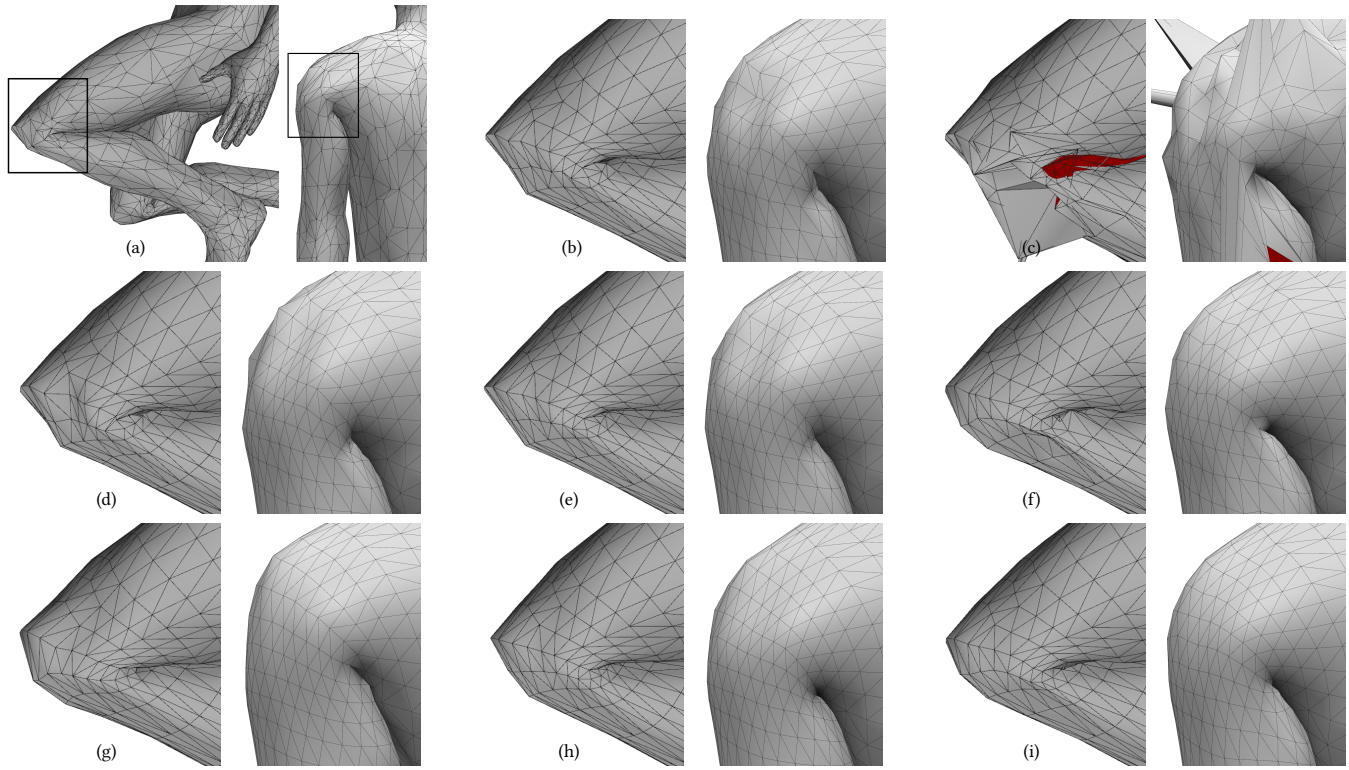

Figure 5: Different examples of our upsampling. Linear (2nd column) and quadratic (3rd column) MLS upsampling using $k = 8$ (b),(c), 16 (e),(f) and 32 (h),(i) vertices of the simulated low resolution mesh (a). For small k-values, matrix G in (7) can become ill conditioned (see (c)). Using a quadratic approximation leads to smoother result and less distortion of the simulated vertices. The old approach (d) was 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 *character low* in Table 2.

the point of evaluation $\mathbf{x}_j$) to the unit sphere. The second case is very rare 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 $\boldsymbol{\pi}(\mathbf{x})$ to the price of sacrificing quadratic or linear precision. Since our upsampling is very efficient, increasing the size of ρ can also be a solution in these cases.

Due to the partition of unity and linear reproduction of MLS shape functions [Fries and Matthies 2004], the MLS upsampling weights N_{ij} reproduce the initial mesh $\mathbf{x}_j = \sum_i \mathbf{x}_i N_{ij}$. 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

$$\mathbf{x}'_j = \mathbf{x}_j + \sum_i \underbrace{(\mathbf{x}'_i - \mathbf{x}_i)}_{=\mathbf{u}_i} N_{ij} = \sum_i \mathbf{x}'_i N_{ij} + \underbrace{\left(\mathbf{x}_j - \sum_i \mathbf{x}_i N_{ij}\right)}_{=\mathbf{0}} \quad (8)$$

The implementation of this upsampling approach in CUDA is simple. Working with positions (8) instead of displacements (5) is a massive reduction of memory accesses. To simplify the kernel even more, we use a fixed amount k of nearest neighbors $\mathbf{x}_i$ for each $\mathbf{x}_j$ (we found $k = 20$ to be sufficient). The radius ρ in (4) is set to the distance of the $(k+1)$st nearest neighbor for each vertex individually. Figure 5 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. Quadratic MLS can better reproduce the deformation. 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. Those can be accessed in a coalesced way as well as the N_{ij}. The read access of low resolution is highly unordered and can thus be accelerated by using texture memory as already mentioned in Section 4.2.

Updating vertex normals for the deformed visualization mesh, typically as a weighted average of incident triangles' normals, is another computational bottleneck. Our approach allows to use MLS upsampling for normal vectors, too, simply by replacing $\boldsymbol{\pi}(\mathbf{x})$ by $\boldsymbol{\pi}(\mathbf{n})$ in (7) and (6). 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 2× faster than the normal displacements in the original Projective Skinning [Komaritzan and Botsch 2018]. The GPU implementation of MLS upsampling is another 10× faster than its CPU version.

6 GLOBAL COLLISION HANDLING

In Projective Dynamics collision response is handled through *unilateral constraints*: In case of a collision, the colliding vertex is projected to the closest collision-free target position. If no collision occurs, the constraint does not do anything. Since in Projective Dynamics each constraint has to provide a target position, unilateral collision constraints are *added on demand* whenever a collision is detected, and removed as soon as the collision is resolved. This approach, however, adds/removes rows to/from the matrix $\mathbf{Q}$ in (1), which changes the global system matrix $\mathbf{A}$ in (2). When using a Cholesky solver, each change in collision constraints triggers a re-factorization, which drastically slows down the simulation.

Komaritzan and Botsch [2018] show that the naive approach to simply project non-colliding vertices to their current position $\mathbf{x}_t$ or to their velocity-update $\mathbf{s}$ leads to unnatural dynamic behavior, because it artificially increases the weight of mass-inertia constraints relative to the tetrahedron strain constraints. They therefore precompute potential *local* collisions and use two different system matrices, one with and one without collision constraints. In each time step (see Algorithm 1), the first $n_{\text{pd}}/2$ iterations use the matrix without collision constraints to get an initial guess for all vertices. The second half of iterations uses the matrix with collision constraints, projecting vertices to either their collision-resolved state or to their initial guess from the first iterations. While this methods works well for *local* collisions, extending it to *global* collisions requires a collision constraint for *each* vertex-triangle pair in the second half of iterations, thereby slowing down the simulation.

Matrix changes are not a performance problem for our iterative PCG solver, since no factorization has to be updated and recomputing the diagonal Jacobi preconditioner $\mathbf{J}$ is trivial. However, special care has to be taken due to the optimized matrix format (see Section 4.2), since completely re-building this matrix format would be prohibitive. We therefore do not update $\mathbf{A}$, but instead represent the system matrix as the sum $\mathbf{A} + \mathbf{A}_{\text{col}}$. 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 4.2. The computation of $\mathbf{Q}^{\mathsf{T}}\mathbf{p}$ in (2) is handled in a similar way.

Collision detection is done on the CPU using point-in-tetrahedron tests accelerated by spatial hashing [Teschner et al. 2003]. The nearest surface point for a colliding vertex can be efficiently determined due to our volumetric tissue mesh construction: Each tetrahedron can be 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 6 and the supplementary video).

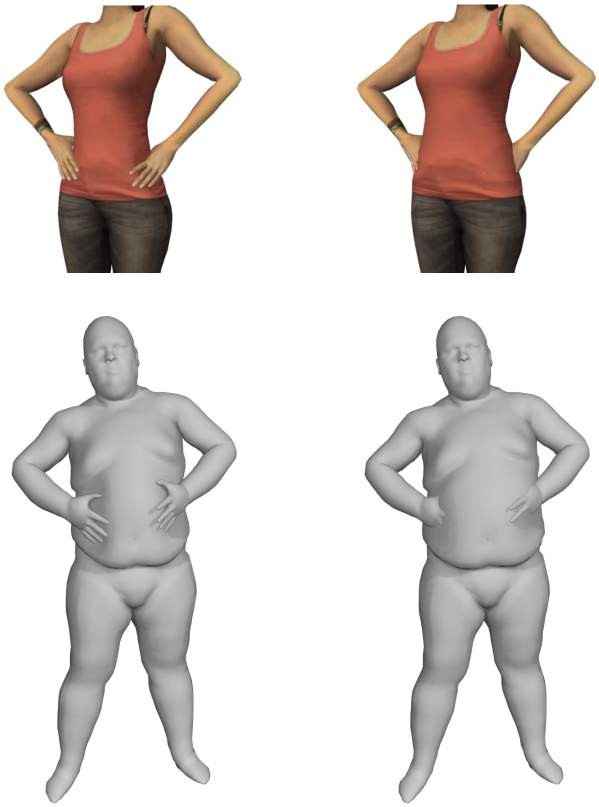

Figure 6: Global collisions (e.g., hands and body) lead to more realistic character animations (left). The original Projective Skinning (right) only supports *local* collisions in joint regions but not *global* collisions. The male model is part of the MPI Dynamic FAUST dataset [Bogo et al. 2017].

For skinning animations we have a lot of resting contacts. If we removed a resting constraint as soon as the colliding vertex has left the tetrahedron, the collision constraint would no longer be active and the strain constraints would push the vertex back into the colliding state in the next iteration. This would cause 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.

Resting contacts cause a second problem: As discussed by Komaritzan and Botsch [2018], using target positions in collision constraints can lead to unnatural dynamic effects. For example, if the character has colliding skin parts between upper arm and forearm while jumping up and down, the global translation of the jumping skeleton will be transferred through the strain constraints, but for colliding vertices their target position would still be the untranslated one, slowing these vertices down. To solve this problem, 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.

Table 2: Benchmark results for several models with N_s simulated vertices, T simulated tetrahedra, and V visualization vertices, comparing the original Projective Skinning (PS) with the CPU and GPU version of our approach. We list the simulation time t_{sim} of one time-step (10 local-global iterations, each with 10 PCG iterations); the time t_{us} for upsampling to the visualization mesh; performance in frames per second for skinning, upsampling, and visualization (fps); the time t_{coll} for collision detection and matrix update; performance in FPS for skinning, upsampling, collision handling, and visualization (fps_{coll}).

				PS			CPU-FPS					GPU-FPS				
Model	N_s	T	V	t_{sim}	t_{us}	fps	t_{sim}	t_{us}	fps	t_{col}	fps_{col}	t_{sim}	t_{us}	fps	t_{col}	fps_{col}
Cylinder	800	4.8k	800	2.7	—	333	1.8	—	400	5.0	130	1.7	—	520	0.6	335
Character low	3801	20.3k	18k	12	0.75	70	6.6	0.47	120	22	33	1.8	0.04	460	2.0	220
Character mid	9605	50.2k	18k	35	0.74	26	17.2	0.48	52	57	12	2.9	0.04	315	5.4	108
Character high	18k	91k	18k	67	—	14	33.8	—	27	106	6	4	—	227	14.1	54
Armadillo	5189	31.1k	173k	21	11	25	11.1	5.3	43	37	14	2.2	0.54	307	3.8	145

7 RESULTS

Similar to the original Projective Skinning (PS), our new method, which we call *Fast Projective Skinning* (FPS), avoids the well-known artifacts of purely geometric methods, as shown in Figure 1. In comparison to the local collisions of PS, our FPS provides full global collision handling, leading to significantly improved results. This is demonstrated in Figure 1 and Figure 6, but better visible in the dynamic animations shown in the supplementary video. Furthermore, our new upsampling technique provides better reconstructions of the high-resolution visualization mesh (Figure 5).

We compare the original PS to our new FPS on a range of different character models. All timings were measured on a standard workstation equipped with Intel Xeon CPU (6 cores, 12 threads × 3.6 GHz) and an Nvidia GTX 1080 TI (3584 CUDA cores, compute capability 5.2). For parallelization we used OpenMP on the CPU and CUDA 9.2 on the GPU. Note that while the GPU version of FPS exploits the iterative PCG solver discussed above, the CPU version employs EIGEN's sparse Cholesky solver, since this is faster than the CPU-based PCG solver. Timing results for the different methods and their algorithmic components are provided in Table 2.

Without collisions, our CPU-FPS is about twice as fast as the original PS, thanks to our hard-constraints formulation (Section 4.3) and the MLS-upsampling (Section 5). Comparing PS to GPU-FPS, the latter is faster by an additional factor of 4–8, depending on model size. For large meshes, GPU-FPS is up to 16× faster than PS.

Analyzing FPS *with global collisions*, the CPU version has to recompute the Cholesky factorization, which the PCG solver of the GPU version avoids. Comparing the two, GPU-FPS is about 6×–10× faster than CPU-FPS. Even with full collision handling, GPU-FPS is faster than the original PS without collisions. Since both FPS versions perform collision detection on the CPU, the difference in their t_{col} times is due to matrix re-factorization in the CPU case.

Comparing the low-, mid-, and high-resolution character results, one can observe that simulation time scales linearly with the number of simulation vertices N_s in the CPU. On the GPU, we observe a sub-linear scaling due to overheads of kernel calls or a low GPU occupancy for smaller meshes. When analyzing frames per second we include the rendering of the visualization mesh of V vertices.

All GPU-FPS timings reported in Table 2 have been computed with $n_{\text{pd}} = 10$ PD iterations in Algorithm 1, each of which uses $n_{\text{pcg}} = 10$ PCG iterations in Algorithm 2. While this performance was fully sufficient in our experiments, the number of PD iterations can be further reduced to $n_{\text{pd}} = 2$ without noticing major visual differences, as shown in the supplementary video (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 2000 frames per second when using the low-resolution simulation mesh with MLS upsampling and without collision handling, which is not much slower than skinning a character with standard linear blend skinning (yielding a considerably lower quality).

8 CONCLUSION

We presented Fast Projective Skinning, an extension of Projective Skinning that improves upon it in terms of both computational performance and animation quality. 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 also yields better results as well as better computational performance.

Like for Projective Skinning, the physical/anatomical plausibility of our approach is limited by the simple one-layer tissue mesh spanned between skin and bones, which can partly be observed in the accompanying video. The computational performance of FPS will allow us to use more sophisticated tissue meshes in the future, for instance featuring rib cage and hips, as well as several tissue layers of varying tissue stiffness. Thickness and stiffness of those layers can maybe be learned from data to support an even more realistic skinning while retaining the simplicity of our FPS approach. Improving the anatomical correctness might eventually make our character animation useful beyond computer games, such as in real-time applications in a medical context.

ACKNOWLEDGMENTS

We would like to thank Bogo et al. [2017], whose models provide mesmerizing collision potentials. We are also very grateful to Stephan Wenninger for his amazing dance moves shaking each cell of the human body and to Wolf-Matthias Vogelsang for his work on the upsampling implementation.

REFERENCES

Hartwig Anzt, 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*.

Nathan Bell and Michael Garland. 2008. *Efficient sparse matrix-vector multiplication on CUDA*. Technical Report NVR-2008-004. NVIDIA Corporation.

Jan Bender, Matthias Müller, and Miles Macklin. 2017. A Survey on Position Based Dynamics. In *Eurographics Tuturials*.

Federica Bogo, 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)*.

Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. 2003. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Computer Graphics* 22, 3 (2003).

Mario Botsch and Olga Sorkine. 2008. On Linear Variational Surface Deformation Methods. *IEEE Transaction on Visualization and Computer Graphics* 14, 1 (2008).

Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Transactions on Computer Graphics* 33, 4 (2014).

Christopher Brandt, Elmar Eisemann, and Klaus Hildebrandt. 2018. Hyper-reduced projective dynamics. *ACM Transactions on Computer Graphics* 37, 4 (2018).

Luc Buatois, 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, 3 (2009).

Steve Capell, 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*.

Dan Casas and Miguel A. Otaduy. 2018. Learning Nonlinear Soft-Tissue Dynamics for Interactive Avatars. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 1 (2018).

Crispin Deul and Jan Bender. 2013. Physically-Based Character Skinning. In *Proc. of Virtual Reality Interactions and Physical Simulations*.

Xiaowen Feng, 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*.

Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. 2016. Vivace: a practical Gauss-Seidel method for stable soft body dynamics. *ACM Transactions on Computer Graphics* 35, 6 (2016).

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

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

Ming Gao, Xinlei Wang, Kui Wu, Andre Pradhana, Eftychios Sifakis, Cem Yuksel, and Chenfanfu Jiang. 2018. GPU Optimization of Material Point Methods. *ACM Transactions on Computer Graphics* 37, 6 (2018).

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

Dahai Guo, 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, 1 (2016).

Daniel Holden, 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*.

Alec Jacobson, Zhigang Deng, Ladislav Kavan, and J.P. Lewis. 2014. Skinning: Real-time Shape Deformation. In *ACM SIGGRAPH Courses*.

Petr Kadleček, Alexandru-Eugen Ichim, Tiantian Liu, Jaroslav Křivánek, and Ladislav Kavan. 2016. Reconstructing Personalized Anatomical Models for Physics-based Body Animation. *ACM Transactions on Computer Graphics* 35, 6 (2016).

Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. 2008. Geometric Skinning with Approximate Dual Quaternion Blending. *ACM Transactions on Computer Graphics* 27, 4 (2008).

Meekyoung Kim, 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 Computer Graphics* 36, 4 (2017).

R. Kimmel and J. A. Sethian. 1998. Computing geodesic paths on manifolds. *Proceedings of the National Academy of Sciences* 95, 15 (1998).

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

Jing Li, 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*.

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

Matthew Loper, Naureen Mahmood, Javier Romero, Gerard Pons-Moll, and Michael J. Black. 2015. SMPL: A Skinned Multi-person Linear Model. *ACM Transactions on Computer Graphics* 34, 6 (2015).

Miles Macklin, Matthias Müller, and Nuttapong Chentanez. 2016. XPBD: Position-based Simulation of Compliant Constrained Dynamics. In *Proc. of ACM International Conference on Motion in Games*.

Nadia Magnenat-Thalmann, Richard Laperrière, and Daniel Thalmann. 1988. Joint-dependent Local Deformations for Hand Animation and Object Grasping. In *Proc. of Graphics Interface*.

Sebastian Martin, Peter Kaufmann, Mario Botsch, Eitan Grinspun, and Markus Gross. 2010. Unified Simulation of Elastic Rods, Shells, and Solids. *ACM Transactions on Computer Graphics* 29, 4 (2010).

Aleka McAdams, Andrew Selle, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. 2011a. *Computing the singular value decomposition of* 3×3 *matrices with minimal branching and elementary floating point operations*. Technical Report 1690. University of Wisconsin-Madison.

Aleka McAdams, Yongning Zhu, Andrew Selle, Mark Empey, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. 2011b. Efficient Elasticity for Character Skinning with Contact and Collisions. *ACM Transactions on Computer Graphics* 30, 4 (2011).

Alexander Monakov, 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*.

Matthias Müller and Markus Gross. 2004. Interactive Virtual Materials. In *Proc. of Graphics Interface*.

Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. 2005. Meshless Deformations Based on Shape Matching. *ACM Transactions on Computer Graphics* 24, 3 (2005).

Matthias Müller, Matthias Teschner, and Markus Gross. 2004. Physically Based Simulation of Objects Represented by Surface Meshes. In *Proc. of Computer Graphics International*.

Maxim Naumov. 2011. *Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU*. Technical Report NVR-2011-001.

Junjun Pan, 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, 10 (2017).

Yue Peng, Bailin Deng, Juyong Zhang, Fanyu Geng, Wenjie Qin, and Ligang Liu. 2018. Anderson acceleration for geometry optimization and physics simulation. *ACM Transactions on Computer Graphics* 37, 4 (2018).

Nadine Abu Rumman and Marco Fratarcangeli. 2015. Position-Based Skinning for Soft Articulated Characters. *Computer Graphics Forum* 34, 6 (2015).

Shunsuke Saito, Zi-Ye Zhou, and Ladislav Kavan. 2015. Computational Bodybuilding: Anatomically-based Modeling of Human Bodies. *ACM Transactions on Computer Graphics* 34, 4 (2015).

Matthias Teschner, 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*.

Rodolphe Vaillant, Gäel Guennebaud, Loïc Barthe, Brian Wyvill, and Marie-Paule Cani. 2014. Robust Iso-surface Tracking for Interactive Character Skinning. *ACM Transactions on Computer Graphics* 33, 6 (2014).

Francisco Bonilla Vázquez, 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*.

Huamin Wang. 2015. A Chebyshev semi-iterative approach for accelerating projective and position-based dynamics. *ACM Transactions on Computer Graphics* 34, 6 (2015).

Daniel Weber, Jan Bender, Markus Schnoes, André Stork, and Dieter Fellner. 2013. Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications. 32, 1 (2013).

Hiroki Yoshizawa 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*.