Towards Realtime: A Hybrid Physics-based Method for Hair Animation on GPU

LI HUANG, Digital Content Technology Center, Tencent Games
FAN YANG, Digital Content Technology Center, Tencent Games
CHENDI WEI, Digital Content Technology Center, Tencent Games
YU JU (EDWIN) CHEN, Digital Content Technology Center, Tencent Games
CHUN YUAN, Digital Content Technology Center, Tencent Games
MING GAO, Digital Content Technology Center, Tencent Games

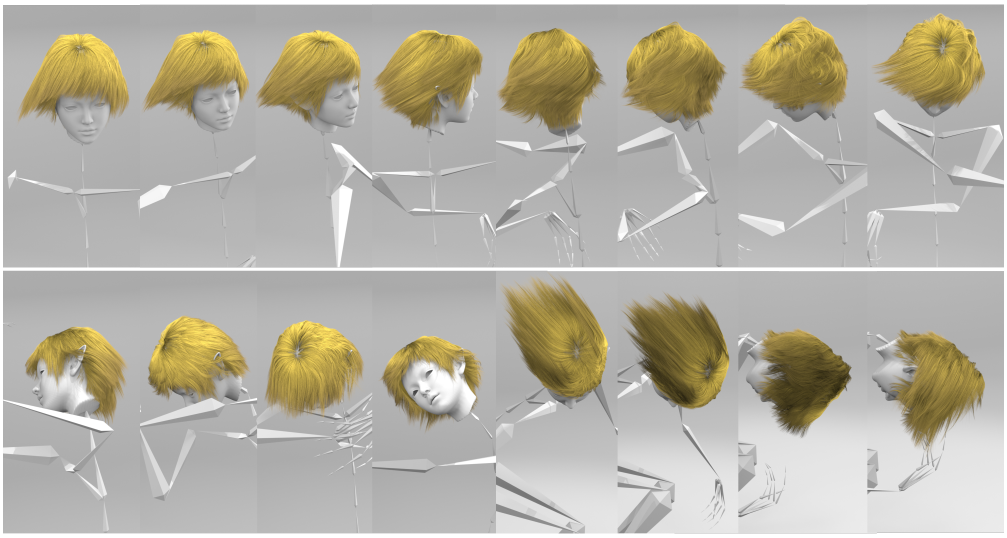

Fig. 1. **Dancing character.** Our method produces realistic hair simulation for an animated dancing character sequence. We simulate more than two thousand hair strands and achieve faster than real-time performance (260 frames per second).

This paper introduces a hair simulator optimized for real-time applications, including console and cloud gaming, avatar live-streaming, and metaverse environments. We view the collisions between strands as a mechanism to preserve the overall volume of the hair and adopt explicit **M**aterial **P**oint **M**ethod (**MPM**) to resolve the strand-strand collision. For simulating single-strand behavior, a semi-implicit **D**iscrete **E**lastic **R**ods (**DER**) model is used. We build upon a highly efficient GPU MPM framework recently presented by Fei et al. [2021b] and propose several schemes to largely improve the performance of building and solving the semi-implicit DER systems on GPU. We demonstrate the efficiency of our pipeline by a few practical scenes that achieve up to 260 frames-per-second (FPS) with more than two thousand simulated strands on Nvidia GeForce RTX 3080.

SCA'23, Aug 04–06, 2023, Los Angeles, CA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145.3606937

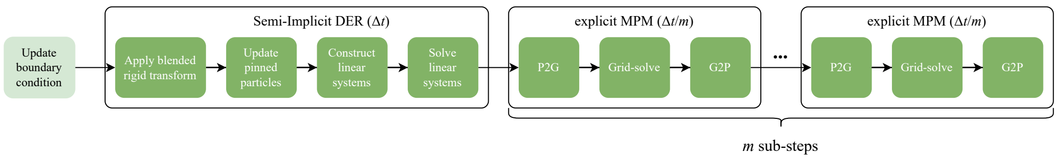

Fig. 2. **Method flowchart.** At each time step, we start with updating the status of collision objects. In the DER stage, we first apply blended rigid transformation if necessary, and then compute the velocities of the pinned particles before addressing the linear systems. In the MPM stage, multiple MPM cycles are conducted to adapt a smaller timestep $\frac{\Delta t}{m}$.

ACM Reference Format:
Li Huang, Fan Yang, Chendi Wei, Yu Ju (Edwin) Chen, Chun Yuan, and Ming Gao. 2023. Towards Realtime: A Hybrid Physics-based Method for Hair Animation on GPU. In . ACM, New York, NY, USA, Article 1, 16 pages. https://doi.org/10.1145.3606937

1 INTRODUCTION

The movie and animation industry has been iterating on techniques for creating realistic hair simulations for generations, resulting in increasingly sophisticated approaches. Unfortunately, these techniques are not easily transferable to real-time applications due to the significant computational burden, particularly related to collision detection and resolution.

We have observed that detailed collision resolution is not necessary when simulating hair dynamics in real-time. Instead, we consider collisions as a way of preserving volume, allowing us to use continuum methods for improved efficiency. Previous approaches to hair simulation, such as smoothed particle hydrodynamics (SPH) and fluid implicit particles (FLIP), have also used volume preservation techniques to model hair dynamics [Hadap and Magnenat-Thalmann 2001; McAdams et al. 2009]. However, we have chosen to adopt the Material Point Method (MPM), which provides the flexibility to switch between fluid and sand models. In addition to volume preservation, the sand model also enables us to reproduce friction between hair strands, which can be desirable in certain scenes.

Moreover, hair simulation involves a physically stiff system due to the material properties of the hair strands. To address this, we have implemented a separate collision handling stage, using the MPM method, which allows us to isolate the stiff system and solve it with a semi-implicit integration method for improved robustness. Instead of solving a large linearized system with collisions taken into consideration, we resolve each strand independently. This means that we can solve smaller systems in parallel, which is a perfect fit for modern GPU architecture.

In this work, we utilize the GPU framework presented in [Fei et al. 2021b] for the MPM stage, as it is specifically designed for real-time applications. Moreover, we propose GPU optimizations for constructing and solving the linear systems in the DER integrator. The resulting pipeline is highly efficient, achieving faster-than-real-time performance on modern GPUs. In addition to efficiency, stability is crucial in real-time scenarios. To ensure excessive stability and robustness, we devise a rigid motion blending strategy and a stability test. We have also developed a plug-in for Unreal Engine, demonstrating that our simulator can produce better visual results at interactive rates compared to currently available commercial tools.

2 RELATED WORK

Achieving a balance between physical accuracy and computational efficiency is a significant challenge when it comes to real-time hair simulation. In this section, we will examine various methods and their respective trade-offs.

Hair model. Mass-spring models are frequently employed in hair simulation [Jiang et al. 2020; Petrovic et al. 2005; Rosenblum et al. 1991; Selle et al. 2008]. However, these models often fail to accurately capture complex hair geometries due to the absence of angular states. While additional springs can be added to simulate bending and torsional behavior, doing so complicates the system. The Articulated-Body Method has also been utilized in hair simulation [Chang et al. 2002; Hadap 2006], but only for simple hair shapes [Ward et al. 2007]. In contrast, the use of elastic rods to simulate hair strands [Bergou et al. 2008; Derouet-Jourdan et al. 2013; Kaufman et al. 2014; Pai 2002] has become increasingly popular due to its ability to accurately capture a wide range of hair configurations and produce realistic simulation results.

Material point method. MPM is a hybrid Lagrangian/Eulerian method first developed by [Sulsky et al. 1994, 1995]. Although initially formulated for computational fluid dynamics by [Brackbill and Ruppel 1986], MPM has proven to be a versatile discretization choice for a range of materials, including snow [Stomakhin et al. 2013], sand [Klár et al. 2016], foam [Ram et al. 2015; Yue et al. 2015], cloth [Fei et al. 2018; Jiang et al. 2017], and solid-fluid mixtures [Pradhana et al. 2017]. The Drucker-Prager sand model is widely used to simulate friction between particles. However, traditional MPM can introduce artificial damping, making it difficult to separate particles. Fei et al. [2021a] overcame this issue by introducing novel integrators. Despite efforts by researchers to boost MPM performance by designing optimized GPU implementations [Fei et al. 2021b; Gao et al. 2018b; Hu et al. 2019; Wang et al. 2020], adopting MPM for real-time applications remains a challenge.

Contact. Simulating frictional contact is crucial for achieving realism in hair simulation. However, due to the large number of contact points involved, many contact algorithms become too computationally expensive for real-time applications [Daviet 2020; Ly et al. 2020]. Using continuum approaches with background grids, such as hybrid Lagrangian/Eulerian methods [Fei et al. 2019; Han et al. 2019; Jiang et al. 2017; McAdams et al. 2009], can be an attractive option since contact resolution is built-in. The use of sand model also allows us to simulate friction between strands without introducing expensive computation. In particular, [Han et al. 2019] work seems similar to ours; however, they use the MPM stage as a preprocessing step of the traditional Lagrangian collision resolution step. MPM helps prevent most of the penetrations. Our algorithm differs from the hair algorithm in [Han et al. 2019], in the sense that we adopt various strategies to accelerate the GPU implementation of DER and employ SFLIP to achieve better particle separation property in our MPM stage.

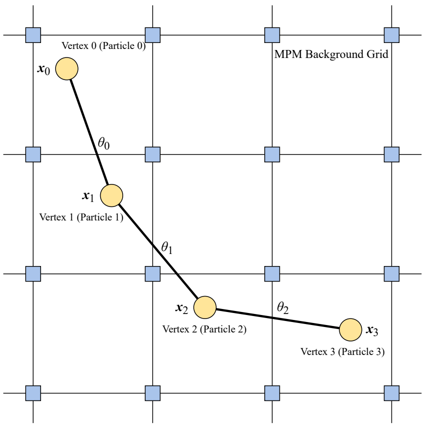

Fig. 3. **Hair Model**. We adopted isotropic, naturally straight rods as the hair model, and each strand has $4n - 1$ degrees of freedom (n is the number of vertices on the strand). It is worth noting that the DER vertices are identical with the MPM Particles.

3 METHOD

Hair Configuration. Our hair simulator combines two different materials: elastic strands and sand/fluid particles. It can be viewed as a prediction-correction model, with the DER stage predicting the movements of each strand independently, and the MPM stage correcting the behavior by taking collisions into account. The vertices of the DER strands and the MPM particles share the same set of three-dimensional points, which allows for seamless coupling between the materials. In this work, we will use the terms *vertex* and *particle* interchangeably, even though *vertex* is more commonly used in the DER stage, while *particle* is predominantly used in the MPM stage. The simulation flow is illustrated in Fig. 2.

We model our hair using isotropic poly-line rods, as shown in Fig. 3. Each strand has a rest shape, and each vertex has three degrees of freedom for translation (position $\boldsymbol{x}$), and each segment has one degree of freedom for rotation (rotation angle θ). The model is based on three types of energies: stretching energy, defined at segments, and bending and twisting energies, defined at vertices. The formulas for these energies are summarized as follows:

$$\begin{aligned} E(\Gamma) &= E_s(\Gamma) + E_b(\Gamma) + E_t(\Gamma) \\ &= \sum_{i=1}^{n-1} \epsilon_s^i + \sum_{i=1}^{n} \epsilon_{b,i} + \sum_{i=1}^{n} \epsilon_{t,i}, \\ \epsilon_s^i &= \frac{\pi r^2 Y}{2\bar{l}^i}(l^i - \bar{l}^i)^2, \\ \epsilon_{b,i} &= \frac{\pi r^4 Y}{16\bar{l}^i}\|\boldsymbol{\kappa}_i - \bar{\boldsymbol{\kappa}}_i\|^2, \\ \epsilon_{t,i} &= \frac{\pi r^4 G}{8\bar{l}^i}(\mu_i - \bar{\mu}_i)^2. \end{aligned} \tag{1}$$

Symbol	Physical quantity	Dimension
ϵ_s^i	Stretching energy at edge i	1
$\epsilon_{b,i}$	Bending energy at vertex i	1
$\epsilon_{t,i}$	Twisting energy at vertex i	1
r	Radius of the strand	1
Y	Young's modulus	1
G	Shear modulus	1
l^i	Length of edge i	1
$\bar{l}^i$	Initial length of edge i	1
$\boldsymbol{\kappa}_i$	Material curvature of vertex i	4×1
$\bar{\boldsymbol{\kappa}}_i$	Initial material curvature of vertex i	4×1
μ_i	twist of vertex i	1
$\bar{\mu}_i$	Initial twist of vertex i	1

For further details, please refer to [Bergou et al. 2010, 2008].

3.1 Implementation of discrete elastic rods on GPU

We use the DER model developed by [Bergou et al. 2010, 2008] and a semi-implicit backward Euler integrator [Baraff and Witkin 1998] to simulate individual hair strands. At each time step, we solve the linear system

$$(\boldsymbol{M} + h^2\boldsymbol{H})\boldsymbol{v} = \boldsymbol{M}\boldsymbol{v}_0 + \Delta t \boldsymbol{f} \tag{2}$$

where $\boldsymbol{v}_0$ is the velocity state vector at the current time step, $\boldsymbol{M}$ is the mass matrix, $\boldsymbol{f}$ and $\boldsymbol{H}$ are the force vector and the energy Hessian derived from Eq. 1, Δt is the time step size. This is a linear system of the form $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ where we solve for the unknown velocity state $\boldsymbol{v}$ at the next time step. The semi-implicit backward Euler method can be expressed as an optimization problem [Martin et al. 2011] that involves minimizing a sum of squares [Bergou et al. 2010]. Computing the full energy Hessian is rather expensive due to the second derivatives of the material curvature $\boldsymbol{\kappa}_i$. To avoid the cost of computing the full Hessian, we only used the first derivatives. This approach ensures that the system matrix $\boldsymbol{A}$ is symmetric positive definite and that the search direction is a decent direction. For detailed derivation of the derivatives, please refer to the supplemental document in [Fei et al. 2019].

Traditional solvers for the DER model require explicit handling of collisions between different strands as well as between segments within the same strand, as noted by [Kaufman et al. 2014]. However, in our framework, all such interactions are deferred to the MPM stage, and each strand can be independently handled. Instead of constructing a large linear system covering all strands, we solve many small linear systems in parallel. This approach allows for independent and efficient simulation of each strand, which is useful when simulating a large number of strands.

3.1.1 Construction of the linear system. For each strand, we construct the sparse matrix $\boldsymbol{A}$ and the right hand side $\boldsymbol{b}$, which are then stored in global memory. In our implementation, each CUDA thread calculates the quantities related to either one vertex (for bending and twisting) or one segment (for stretching). Since hair strands are modeled as naturally straight rods, the forces on vertices and the torques on edges are determined by up to three contributions[Bergou et al. 2008].

In other words, each vertex only interacts with its closest and second-closest neighboring vertices as shown in Fig. 4. The energy Hessian matrix of this system is a sparse banded diagonal matrix with a bandwidth of 10. Since self-collision in the same strand is temporarily ignored, there are no other non-zero entries outside the band in these small linear systems.

The system matrix $\boldsymbol{A}$ is split into two parts: $\boldsymbol{A}_d$, the diagonal part, and $\boldsymbol{A}_l$, the lower triangular part. As shown in Fig. 5(a), $\boldsymbol{A}_l$ is stored in the Array-of-Structure-of-Array (AoSoA) [Fei et al. 2021b; Wang et al. 2020] storage format with a column-major order, we will explain the reason in Sec. 3.1.2.

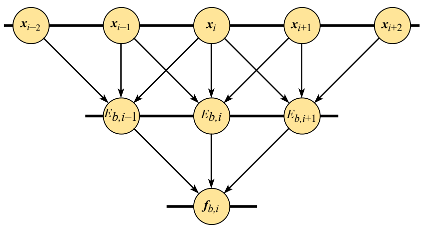

Fig. 4. **Computation of Bending Force.** To compute the bending force for vertex i, we need the positions of vertices from $i-2$ to $i+2$. This access pattern makes the shuffle intrinsic operations preferable.

Access neighboring vertices/edges. A straightforward way to read quantities on neighboring vertices or edges is to use shared memory as a scratchpad, which demands three operations in total – one writing operation, one CUDA block level synchronization and finally one reading operation. However, we observe that in DER model one vertex would only need information from either its preceding or its succeeding vertices (e.g., for computing the energies defined at itself). For instance, the formula for calculating bending force for vertex i is presented in Eq. 3 ($\mathbf{C}_i^k$ stands for the force exerted on vertex i by vertex k), where only the neighboring vertices $i-1$, i, and $i+1$ are involved in the computation. As a result, the data exchange would only happen between neighboring threads. Thus we propose to use shuffle intrinsic operations for exchanging data as such operations can directly access other threads' registers in the same warp and achieve better efficiency.

$$f_{b,i} = -\frac{\partial E_b(\Gamma)}{\partial \boldsymbol{x}_i} = -\sum_{k=i-1,i,i+1} \frac{\partial \epsilon_{b,k}}{\partial \boldsymbol{x}_i} = -\sum_{k=i-1,i,i+1} \mathbf{C}_i^k \tag{3}$$

Increase GPU occupancy. Compared with offline scenarios, we simulate a fewer number of hair strands (e.g., 500 - 2000 strands) for real-time applications. Consequently, we need a strategy to better fill up the GPU streaming multiprocessors to achieve higher occupancy and reduce tail effect. Instead of feeding every single strand to one CUDA block, we divide strands into groups of 32 vertices and assign each group to a CUDA block of 32 threads (or a warp). For example, a strand with 50 vertices will be handled by two CUDA blocks (or two warps) in parallel.

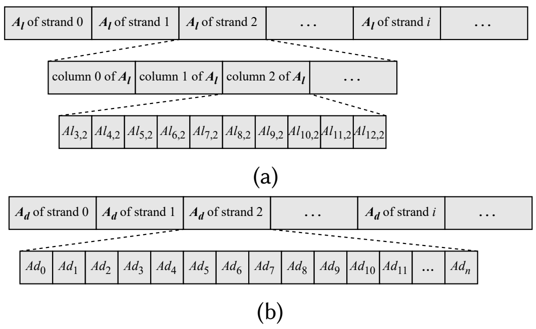

Fig. 5. (a) **Storage for $\boldsymbol{A}_l$ and $\boldsymbol{L}$.** Banded matrices $\boldsymbol{A}_l$ and $\boldsymbol{L}$ are stored in a column-major AOSOA format, facilitating column-major global memory access. (b) **Storage for $\boldsymbol{A}_d$ and $\boldsymbol{b}$.** $\boldsymbol{A}_d$ and $\boldsymbol{b}$ are stored in a regular SOA format.

Avoid thread divergence. As shown in Fig. 6, a straightforward way to divide vertices of a strand is to group them in a disjoint fashion. Due to the dependence between neighboring vertices, however, this introduces heavy thread divergence. In order to calculate quantities like material curvature or energy, boundary vertices (the first and last vertices) of each group need to access the vertices of neighboring groups (e.g., in Fig. 6, vertex $32j+31$ needs to read information from and write contributions to vertex $32j+32$), which demands a large amount of branching.

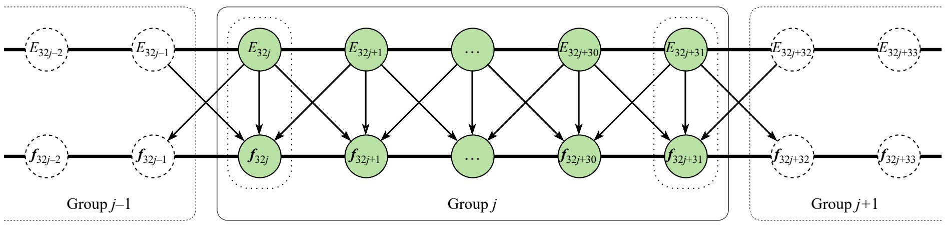

Fig. 6. **Disjoint groups.** Grouping the vertices without overlap seems plausible. However, additional operations on boundary vertices are required, introducing heavy thread divergence.

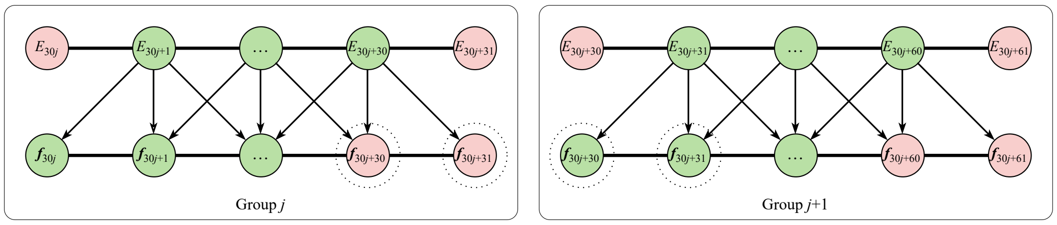

Fig. 7. **Overlapping groups.** The two overlapped vertices, namely vertex $30j + 30$ and vertex $30j + 31$, are shared by two adjacent groups and processed by both of them. The incomplete forces of these vertices, calculated by the neighboring groups, complement each other and are added through atomic operations.

We propose *overlapped grouping* to ameliorate this issue as shown in Fig. 7. Under this division scheme, the first and second vertices of a group are the same as the last and the second last vertices of the previous group. When calculating the bending or twisting energies for boundary vertices (the first and the last vertices) of a group, we don't read information from the vertices of neighboring groups to get the correct energies. Instead, we do nothing but just leave the energies wrong. Then when calculating forces (or components of Hessian) for overlapped vertices (boundary vertices plus the second and the second last vertices), we set contributions from boundary vertices to zero (line 10, Alg. 1). Though each overlapped vertex gets incomplete contributions during the process of a group, it gets complete contributions after being processed by both groups, which leads to a correct final result. That is, the force contributions from neighboring groups are complementary. Potential writing conflicts are resolved with atomic operations. Eq. 4 gives the bending force breakdown for vertices in group j. This way, additional read and write operations on boundary threads are eliminated, and setting zero is trivial for boundary threads and won't introduce heavy thread divergence. We follow a similar way to process stretching force and Hessian.

$$f_{b,i} = \begin{cases} -\sum\limits_{k=i+1} \mathbf{C}_i^k & i = 30j \\ -\sum\limits_{k=i,i+1} \mathbf{C}_i^k & i = 30j+1 \\ -\sum\limits_{k=i-1,i} \mathbf{C}_i^k & i = 30j+30 \\ -\sum\limits_{k=i-1} \mathbf{C}_i^k & i = 30j+31 \\ -\sum\limits_{k=i-1,i,i+1} \mathbf{C}_i^k & \text{otherwise} \end{cases} \tag{4}$$

Coalesced writing. After each thread has finished its computing, they don't write the results immediately to global memory as these quantities are not consecutive in memory (see Fig. 5 and line 14, Alg. 1). Instead, we allocate a chunk of shared memory as a scratchpad to store these quantities temporarily. Then all threads in a CUDA warp will be used to write adjacent values of these quantities to global memory to guarantee the final writing is coalesced. Notice that DER is a computationally expensive model and each thread consumes many resources such as registers and shared memory, which is another reason that we use CUDA blocks consisting of only one CUDA warp.

Alg. 1 outlines a pseudo-code demonstrating the construction of bending force. Although the construction of other parts of a linear system may vary slightly with the mathematical definitions, the key challenges can be addressed in similar ways. Notably, we consolidated the construction of all components of a linear system into a single CUDA kernel rather than several small kernels, to reduce the overhead of kernel launch.

ALGORITHM 1: Construction of bending force

```
   input  : pos, group_id, lane_id
   output : b
1  __shared__ float f[96]
2  $x_i$ = pos[group_id, lane_id]  // Index conversion is omitted
3  $x_{i+1}$ = __shfl_down($x_i$)      // $x_{i+1}$ is wrong for lane 31.
4  $x_{i-1}$ = __shfl_up($x_i$)        // $x_{i-1}$ is wrong for lane 0.
5  $E$ = ComputeBendingEnergy($x_{i-1}, x_i, x_{i+1}$)
6  $C_{i-1}$ = ComputeBendingContribution($E, x_{i-1}$)
7  $C_i$ = ComputeBendingContribution($E, x_i$)
8  $C_{i+1}$ = ComputeBendingContribution($E, x_{i+1}$)
9  if lane_id == 0 Or lane_id == 31 then
10     $C_{i-1} = C_i = C_{i+1} = 0$
11 end
12 $f_i$ = __shfl_up($C_{i+1}$) + $C_i$ + __shfl_down($C_{i-1}$)
13 for $k = 0$ to 2 do
14     f[lane_id * 3 + k] = $f_i[k]$
15 end
16 __syncthreads()
17 group_offset = ComputeOffset(group_id)
18 b[group_offset + lane_id] += f[lane_id]
```

3.1.2 Solution of the linear system. We first describe the standard way of solving $\boldsymbol{x}$ from $\boldsymbol{Ax} = \boldsymbol{b}$ and then show how we analyze and implement the solver on GPU to achieve better efficiency.

Standard LDL solver. The algorithm is outlined in Alg. 2. Given that the system matrix $\boldsymbol{A}$ is symmetric positive definite, we can apply an LDL decomposition: $\boldsymbol{A} = \boldsymbol{LDL}^T$, where $\boldsymbol{L}$ is a lower triangular matrix with all diagonal entries being 1, and $\boldsymbol{D}$ is a diagonal matrix. The original Cholesky decomposition is less preferred as it requires expensive square root operations. We first compute a temporary vector $\boldsymbol{y}$ using forward substitution ($\boldsymbol{LDy} = \boldsymbol{b}$) and then we solve for the final result $\boldsymbol{x}$ using back substitution ($\boldsymbol{L}^T\boldsymbol{x} = \boldsymbol{y}$).

ALGORITHM 2: LDL solver

```
   input  : A, b
   output : L, D, y, x
   /* Step 0: LDL decomposition                         */
1  for $j = 0$ to $n-1$ do
2      $D_j = A_{jj} - \sum_{k=0}^{j-1} L_{jk}^2 D_k$
3      for $i = j+1$ to $n-1$ do
4          $L_{ij} = \frac{1}{D_j}(A_{ij} - \sum_{k=0}^{j-1} L_{ik}L_{jk}D_k)$
5      end
6  end
   /* Step 1: forward substitution                      */
7  for $j = 0$ to $n-1$ do
8      $y_j = \frac{1}{D_j}(b_j - \sum_{k=0}^{j-1} y_k L_{jk} D_k)$
9  end
   /* Step 2: back substitution                         */
10 for $j = n-1$ to 0 do
11     $x_j = y_j - \sum_{k=j+1}^{n-1} L_{kj} x_k$
12 end
```

GPU implementation. We use one CUDA block to solve the linear system of each strand, and each CUDA block consists of one thread warp. Unlike the construction of the linear system, here we use all threads in the block to process one column of the system at one time. The pseudo-code of the CUDA kernel is presented in Alg. 3. The special structure of $\boldsymbol{A}$ enables certain optimization strategies, which we will elaborate on in this section.

- Though not stated explicitly in Alg. 3, all 2-D indices (e.g., $[j, k]$ on line 5) will be converted into a proper 1-D index.
- Similar to the construction of the linear system, shuffle instrinsics can also be utilized here to perform warp-level reductions, thereby enhancing the efficiency of summations on line 2, line 8, and line 11.
- In Alg. 2, we observe that D_j on line 8 is computed on line 2 and y_k, L_{jk}, D_k are computed from previous for-loops. Thus we can merge the for-loops on line 1 and line 7 into one for-loop. As a result, D_j doesn't need to be stored in global memory. Note that in the first loop of Alg. 3, 10 threads are used to compute D_j and y_j, only the first thread will be responsible for writing the results. Then each thread will be responsible for computing and writing L_{ij} on its own, so here we cannot use warp-level reduction to handle the summation on line 4 of Alg. 2.

- As discussed in Sec. 3.1.1, $\boldsymbol{A}_l$ is stored in a column-major AOSOA format (see Fig. 5(a)), so the reading of A_{ij} (line 4, Alg. 2; line 12, Alg. 3) can be coalesced. Note that we reuse $\boldsymbol{A}_l$ to store the matrix $\boldsymbol{L}$, so similiarly, the writing of L_{ij} (line 4, Alg. 2; line 13, Alg. 3) and the reading of L_{kj} (line 11, Alg. 2; line 22, Alg. 3) are also coalesced.
- A side-effect caused by column-major storage is that the reading of L_{ik} and L_{jk} (line 4 and line 8, Alg. 2) becomes uncoalesced. To ameliorate this problem, we use three circular buffers on shared memory to record $L_{jk}D_k$, D_k, and y_k for the recently processed 10 columns. This is because, for any column, only the closest 10 columns to its left are used to compute the elements on that column. Every time we start processing one column, we first read cached values from the circular buffers, and update the circular buffers after the computing of the column is finished. As a result, with the use of circular buffers, the only uncoalesced global memory access left is the reading of L_{ik} (line 12, Alg. 3).
- Finally, as $\boldsymbol{A}$ is a banded matrix with a bandwidth of 10, only 10 threads are needed to solve one hair strand, making it possible to process 3 hair strands in parallel with a CUDA warp. While the details are omitted in Alg. 3 for the sake of conciseness, this can be done by slightly adjusting the memory allocation and index computation involved in Alg. 3 and can further reduce the overall execution time.

In conclusion, we have implemented several optimization strategies to reduce the memory access latency and improve the parallelism of our algorithm. A benchmark for our LDL solver is presented in Sec. 5. With this optimized GPU solver, the performance of solving the linear system can be substantially boosted and becomes comparable to the MPM particle-to-grid operation. In Sec. 3.3, we propose a substepping scheme to further reduce the cost.

ALGORITHM 3: GPU Implementation of LDL solver

```
input :$\boldsymbol{A}_l, \boldsymbol{A}_d, \boldsymbol{b}, lane\_id$
output:$\boldsymbol{b}$
/* Circular buffers for $L_{jk}D_k$, $D_k$, and $y_k$ */
1  __shared__ float cb_LD[100], cb_D[10], cb_y[10]
/* Step 0: LDL decomposition and forward substitution */
2  for $j = 0$ to $n - 1$ do
3      if $lane\_id < 10$ then
           /* Compute $D_j$ and $y_j$ */
4          $k = j - lane\_id - 1$
5          $L_{jk}D_k = \text{cb\_LD}[j, k]$   // Index conversion is omitted.
6          $D_k = \text{cb\_D}[j, k]$
7          $y_k = \text{cb\_y}[j, k]$
8          $D_j = \boldsymbol{A}_d[j] - \textbf{WarpReduction}((L_{jk}D_k)^2/D_k)$
9          $y_j = \frac{1}{D_j}(\boldsymbol{b}[j] - \textbf{WarpReduction}(y_k L_{jk} D_k))$
10         $\boldsymbol{b}[j] = y_j$   // $\boldsymbol{b}$ is reused to store $\boldsymbol{y}$.
           /* Compute $L_{ij}$ */
11         $i = j + lane\_id + 1$
12         $L_{ij} = \frac{1}{D_j}(A_l[i, j] - \sum_{k=j-10}^{j-1} A_l[i, k] L_{jk} D_k)$
13         $A_l[i, j] = L_{ij}$   // $\boldsymbol{A}_l$ is reused to store $\boldsymbol{L}$.
           /* Update circular buffers */
14         $\text{cb\_LD}[i, j] = L_{ij} \cdot D_j$
15         $\text{cb\_D}[j, j] = D_j$
16         $\text{cb\_y}[j, j] = y_j$
17     end
18 end
/* Step 1: back substitution */
19 for $j = n - 1$ to 0 do
20     if $lane\_id < 10$ then
21         $k = j + lane\_id + 1$
22         $L_{kj} = \boldsymbol{A}_l[k, j]$
23         $x_k = \boldsymbol{b}[k]$
24     end
25     $y_j = \boldsymbol{b}[j]$
26     $x_j = y_j - \textbf{WarpReduction}(L_{kj} x_k)$
27     $\boldsymbol{b}[j] = x_j$   // $\boldsymbol{b}$ is reused to store $\boldsymbol{x}$.
28 end
```

3.1.3 Boundary condition in DER.

Analytic capsule SDF. Given our focus on real-time applications, we use analytic capsule-shaped SDFs (signed distance fields) to represent the head and body of a character, which serve as boundary conditions in both the DER and MPM stages. While more accurate alternatives such as meshes or level-sets exist, we prioritize the efficiency of the simulation to achieve real-time performance.

Pinned particles. To account for pinned particles at the root of each strand, we must modify their velocities based on the movement of the head capsule as they are constraints rather than degrees of freedom. To accomplish this, we use the head capsule's translation and rotation as the target position for the pinned particles and calculate their corresponding target velocities. We then blend the pinned particles' original velocities from the previous time step with the target velocities to obtain the final velocities, which are used to compute the right-hand side of the linear system. It is important to note that, like in MPM [Klár et al. 2016], we do not alter the positions of the pinned particles.

Bangs preservation. To preserve the initial style of certain strands and prevent tangling in specific areas, such as the shorter strands in front of the forehead which are carefully designed by artists, we propose selectively increasing the number of pinned particles. This means we can choose to pin more particles for strands on the forehead compared to other strands, which better maintains the hairstyle.

3.2 Material point method

One efficient way to handle collisions in hair simulation is by using continuum methods. McAdams et al. [2009] proposed the use of an incompressible fluid solver called FLIP to preserve hair volume. Another approach is to use Material Point Method (MPM), which has been successfully used in recent works such as [Fei et al. 2021a; Han et al. 2019; Jiang et al. 2017] for collision resolution in hair simulation.

Collisions are handled through two aspects. Firstly, particles that are about to collide will have velocities with opposite directions. During the particle-to-grid (P2G) process, these velocities are averaged at grid nodes, reducing the tendency for collision. Secondly, the reduced kinematic energy is converted to potential energy stored within each particle, typically described using a constitutive model. In the subsequent time step, this potential energy generates forces that further prevent collisions.

3.2.1 Consititutive models. To ensure volume preservation, we can use the weakly compressible model of MPM proposed by Pradhana et al. [2017], which simulates fluid similar to FLIP. However, FLIP methods lack the ability to model friction between hair strands, a key feature in hair simulation that can have a significant impact on strand-strand interactions as discussed in [Daviet et al. 2011]. On the other hand, MPM can easily incorporate friction using the Drucker-Prager sand model [Klár et al. 2016; Yue et al. 2018]. Figure 11 shows a comparison of the visual differences between hair simulated using the fluid model and hair simulated using the sand model.

3.2.2 Integrator. We have chosen to use the newly introduced explicit MPM pipeline on a single GPU, as presented in [Fei et al. 2021b]. This optimized approach outperforms earlier methods [Gao et al. 2018b; Hu et al. 2019; Wang et al. 2020] and is capable of providing superior performance. To further enhance our simulation, we utilize the Separable FLIP integrator (SFLIP) method [Fei et al. 2021a], which efficiently reduces numerical viscosity in particle-grid hybrid techniques.

3.2.3 Boundary condition in MPM. We adopt a traditional approach for collision handling in MPM, as described in [Jiang et al. 2016]. However, in our case, the boundary conditions are only enforced at grid nodes. Specifically, we first check whether the new position of each grid node intersects with the SDFs. If there is an intersection, we use an impulse method to resolve the collision. For the pinned particles, no special treatment is required during the particle-to-grid transfers. However, during the grid-to-particle step, we compute the new positions from the velocities prescribed in Sec. 3.1.3. The pinned particles' velocities derived in the MPM stage can be updated in the standard MPM way since they will be discarded and modified in the DER stage of the next time step. It should be noted that the gravity force must not be applied to pinned particles.

3.3 Time step

Fixed Time Step. We have chosen to use fixed time steps for our simulations. Specifically, all of our simulations run at 60 frames-per-second, with each frame consisting of 6 steps. This gives us a fixed time step of $\frac{1}{360}$ seconds for each step.

DER. Due to the semi-implicit integration used in DER, relatively large time steps can be taken. Therefore, the fixed time step used in our simulations is usually sufficient.

MPM. While MPM typically requires small time steps due to its use of explicit time integration, we can use larger time steps in our simulation because we are not modeling real sand or fluid dynamics. Specifically, we can use relatively small values for the Young's modulus in both models, as indicated in Table 2. To ensure stability, we use a smaller FLIP ratio in SFLIP than what is typically used in MPM simulations, and we do not include the affine augmentation.

Sub-stepping. While our optimizations on the DER part can substantially reduce the cost of solving the linear system, we propose a simpler version of sub-stepping scheme from [Gao et al. 2018a] for better efficiency. Since DER uses semi-implicit integration, it can remain stable at a larger time step compared to the fixed time step MPM mentioned earlier, which is determined by the explicit MPM. In other words, DER can execute once per N fixed time steps where N ranges from 1 to 3 in our demos. We record the momentum changes of all particles in the DER stage. Then, in each MPM sub-step, the particles can get $\frac{1}{N}$ of the stored momentum changes. The boundary conditions need to be carefully adjusted in sub-stepping. Specifically, semi-implicit DER requires the boundary condition at time $t + N\Delta t$ while multiple explicit MPM steps take the boundary conditions at time $t, t + \Delta t, ..., t + (N - 1)\Delta t$ correspondingly.

4 STABILITY CONSIDERATIONS

Ensuring stability is a critical consideration when adopting new algorithms in real-time applications, as we don't want to see the character's hair or clothes blowing up during gameplay or live-streaming. To deliver a robust and efficient framework, we use several schemes to ensure stability.

4.1 Clamping

In certain extreme scenarios, it becomes crucial to pay close attention to intermediate quantities during the simulation process. For instance, at a vertex, the curvature binormal is defined as $(\kappa \boldsymbol{b})_i = \frac{2\boldsymbol{t}^{i-1} \times \boldsymbol{t}^i}{1+\boldsymbol{t}^{i-1} \cdot \boldsymbol{t}^i}$ and can approach nearly infinite values, as depicted in Fig. 8. To prevent the simulation from blowing up, we opt to clamp these intermediate quantities to a safe value when such cases arise.

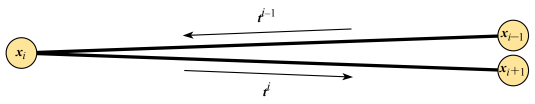

Fig. 8. **Overlapped neighboring segments.** Numerical problems arise when two adjacent segments become too close that they almost overlap and form a small acute angle.

4.2 Blend with rigid body motion

Physics-based simulations can become unstable when the time step is too large to capture the scale of the simulated dynamics. For instance, in a fixed time step simulation, if the head capsule moves too quickly, the simulation can easily become unstable. However, in real-time applications, the budget for physics is typically limited, making it infeasible to dynamically adjust the time step size. On the other hand, rigid body motion is unconditionally stable. To enhance robustness and stability, we propose blending the simulated results with rigid body motion. To achieve this, we set maximum limits for the head capsule's displacement and rotation in one fixed time step. When the capsule's transformation exceeds these limits, we apply a rigid body motion to all particles to ensure that the new transformation lies within the prescribed range. Then, we can proceed with the simulation as usual.

4.3 Stability test

We have implemented a stability test to evaluate the effectiveness of the schemes proposed in the previous subsections. To conduct this test, we randomly set the translation and rotation of the head capsule for each time step, which can be significantly different from their previous state, and run the simulation for several days or millions of frames. We conduct this test on the hair models used in Table 1 before using them to generate demos. The results of this test demonstrate that our method is highly robust for real-time applications. In Fig. 9, we present a sample trajectory of the head capsule during the stability test.

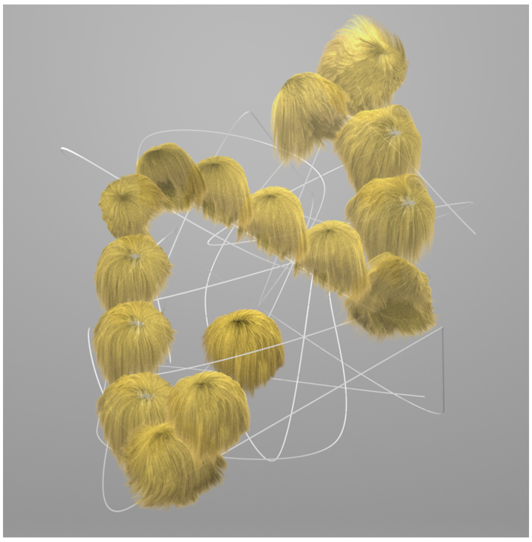
Fig. 9. Stability test. The hair moves along a randomly generated trajectory to verify the robustness. We show that our method remains "unconditionally" stable even with sudden moves and sharp turnarounds.

5 IMPLEMENTATION AND RESULTS

The hair simulation only explicitly simulates around 3% of the total number of hair strands, with the remaining strands being interpolated. To interpolate these strands, we record the closest three simulated strands in the reference space for each interpolated strand. During the simulation, we can compute two sets of results: one using a single strand interpolation scheme and the other using a multi-strand interpolation scheme. We then blend between these two results to achieve a realistic final hair simulation. Interested readers can find more information about this interpolation technique in the work by Yuksel and Tariq [2010].

We have fixed the cell size of the MPM background grid at 1 cm, which is appropriate for the hair models we use. The segment length of these models varies from 0.6 cm to 1 cm. The strand length of our short hair model ranges from 3 cm to 25 cm, while the strand length of our long hair model ranges from 15 cm to 50 cm.

We adopt identical physical parameters of DER and MPM as recorded in Table 2 for the demos of Fig. 1, Fig. 9, Fig. 13, Fig. 14, and Fig. 15. And detailed statistics of the demos are provided in Table 1.

Volume preservation. In this example, we use the Drucker-Prager sand model [Klár et al. 2016; Yue et al. 2018] in the MPM stage to demonstrate how the Young's modulus affects the ability to preserve volume.

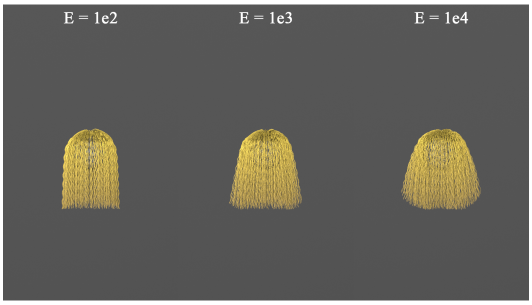

Fig. 10. Volume preservation. By changing the Young's modulus in the MPM stage, we can adjust the overall volume of the hair.

Comparing sand and liquid model. We compared the sand model with the weakly compressible liquid model [Pradhana et al. 2017] in the MPM stage. To observe how these models affect the movement of hair strands, we dropped a group of hair strands onto the ground. As depicted in Fig. 11, the liquid model can preserve volume, but it lacks inter-strand friction, which allows hair strands to spread out on the ground. The sand model, on the other hand, is capable of keeping the strands clustered together, and we can easily adjust the effect by changing the friction angles.

Dancing character. We use the hair model from the stability test (Sec. 4.3) to simulate the hair dynamics of a dancing character (Fig. 1). The time breakdown of this demo is shown in Table 3. Note that with 1 sub-step, solving the linear system remains to be the bottleneck of our pipeline; while with 3 sub-steps, P2G of MPM becomes the bottleneck.

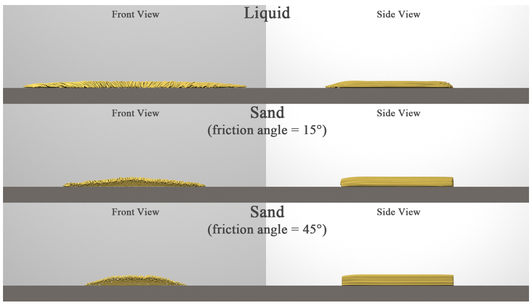

Fig. 11. Comparing two MPM models. Compared with a fluid model, the sand model provides better flexibility as it reproduces the inter-strand friction which can be adjusted by changing the friction angle.

Sub-stepping scheme. We compared the simulated results produced with different numbers of sub-steps. Although the visual differences were negligible, we observed that a larger number of sub-steps improved the performance of the simulation. The comparison results are shown in Fig. 13.

Unreal Engine interactive plug-in. Our method has been integrated as a plug-in in Unreal Engine. Fig. 14 shows that our plug-in can simulate long hair dynamics, and it produces realistic results while maintaining stability even when the play rate of the underlying skeleton (i.e., the head capsule) is increased from normal speed to super-high speed.

We also compare our method with Groom, the official hair simulation tool of Unreal Engine, in the same scenario (Fig. 15). The animated result of our plug-in is much more dynamic and realistic than the result of Groom, which seems to suffer from a considerable amount of damping and viscosity.

LDL solver benchmark. We compare our optimized GPU LDL solver discussed in Alg. 3 against a naive GPU LDL solver (i.e., a simple implementation of Alg. 2) and a CPU LDL solver from Eigen[Guennebaud et al. 2010] and plot the results in Fig. 12. The performance of each solver was evaluated on a modern desktop with an i9-9900K CPU and an Nvidia RTX 2080Ti GPU. Our optimized GPU solver performs significantly better than the CPU solver from Eigen.

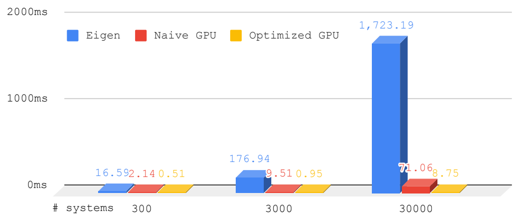

Fig. 12. **LDL solver benchmark.** The time cost of each solver is represented on the vertical axis, while the horizontal axis shows the number of linear systems ranging from 300 to 30000. It is worth noting that the dimension of each system is fixed at 400.

For 3000 linear systems (i.e., hair strands in the context of this paper), we could achieve up to 186× speed-up compared with the CPU solver and 10× speed-up compared with the naive GPU version. Note that our optimized implementation scales well with the number of systems, allowing for even greater speed-up (200×) for 30000 systems.

6 DISCUSSION

We introduce a highly efficient hybrid methodology for simulating character hair, which combines the DER model for single-strand dynamics with an MPM stage for inter-strand collisions. In the

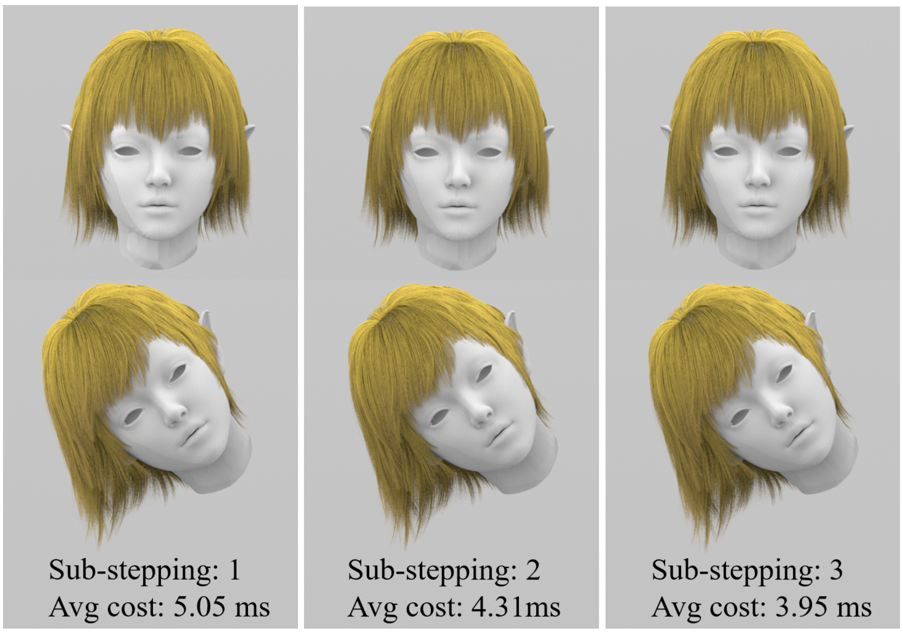

Fig. 13. **Sub-stepping scheme.** Results with different numbers of sub-step are almost identical.

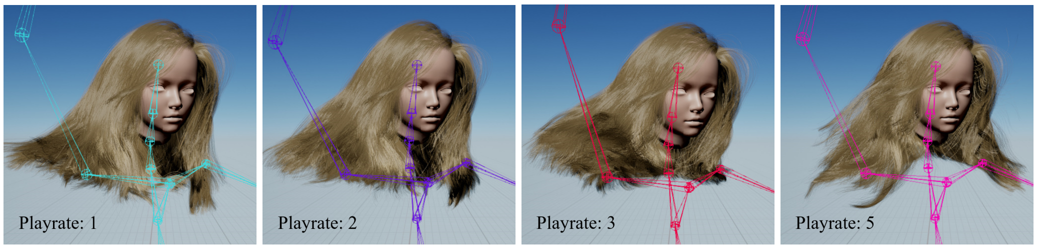

Fig. 14. **Different play-rates.** As the playback rate increases, our method remains stable and the simulated results are realistic.

MPM stage, we use SFLIP, a superior integrator, allowing us to produce flowing and waving hair without Lagrangian collision detection. Our approach includes optimization schemes for efficiently building and solving the linear system of DER on GPUs, resulting in hair animations of high fidelity within a limited time budget. Our pipeline enables realistic character hair simulation in real-time applications. While our demos typically run with 5 milliseconds per frame, further increasing the FPS is possible by decreasing the number of simulated strands. For example, reducing the number of strands from 2000 to 500 can improve the FPS without sacrificing quality. We have also proposed several stability-enhancing schemes for our pipeline.

Limit. While the DER model enables us to simulate complex hair dynamics such as twisting, it may still be too computationally expensive for many scenarios. Alternatively, constraint-based methods like Position-Based Dynamics [Müller et al. 2007] and Projective Dynamics [Bouaziz et al. 2014] can offer better control for artists. Further optimization could be done for increasing efficiency, but this could also make the pipeline even more complicated and uneasy to adapt to other scenarios.

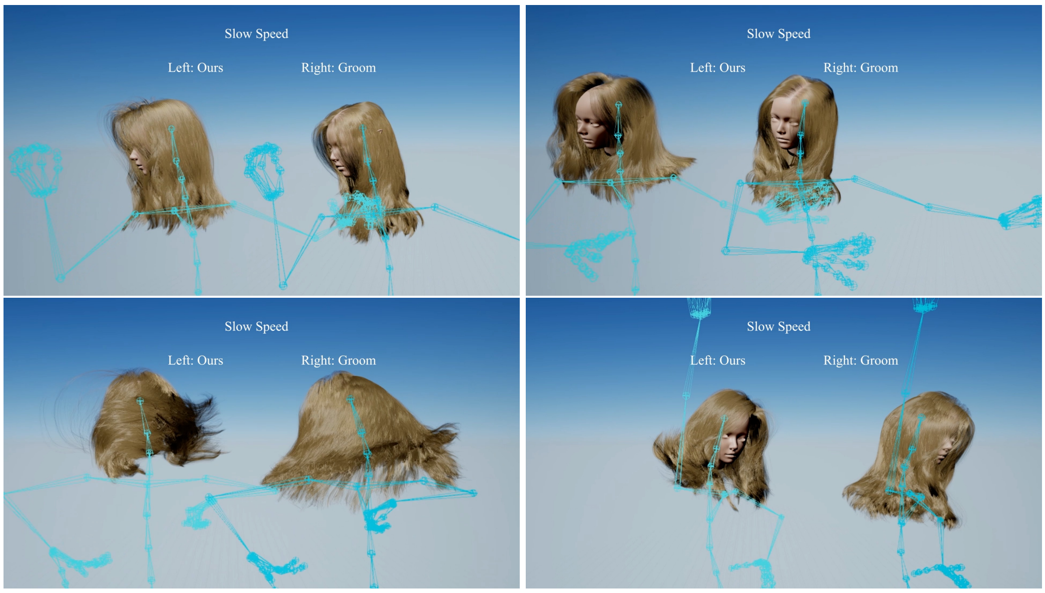

Fig. 15. **Comparison with Groom.** In this example, our method produces visually better dynamics compared to Groom. With our method, a bunch of strands settle down slowly when the head capsule suddenly stops rotating, while with Groom, the deformation of the hair strands appears more rigid.

Future work. Our current method is limited to handling collisions with characters represented by capsules, and does not resolve interactions between hair and garments. This can result in undesired hair penetration in scenarios where complex shapes are used for characters or when long hair is in contact with clothing. To address this issue, we plan to adopt level-set or more accurate boundary condition modeling methods in future work.

One potential direction is to extend our hybrid method to cloth simulations. However, unlike strands that can be regarded as individual small systems, cloth is typically treated as a whole. Thus, a large linear system must be handled, which can be challenging for real-time applications.

Another direction for future work is to adapt our method to other parallel computing platforms besides CUDA. While we have already transplanted the pipeline to DirectX11 and DirectX12, utilizing Compute Shader as an alternative to CUDA, future adaptations to OpenGL or Metal may be necessary to enable hair simulation on mobile devices. Providing support for multiple platforms will increase the accessibility and versatility of our method.

REFERENCES

D. Baraff and A. Witkin. 1998. Large Steps in Cloth Simulation. In *Proc ACM SIGGRAPH.* 43–54.

Miklós Bergou, Basile Audoly, Etienne Vouga, Max Wardetzky, and Eitan Grinspun. 2010. Discrete Viscous Threads. In *ACM SIGGRAPH 2010 Papers* (Los Angeles, California) *(SIGGRAPH '10).* Association for Computing Machinery, New York, NY, USA, Article 116, 10 pages. https://doi.org/10.1145/1833349.1778853

Miklós Bergou, Max Wardetzky, Stephen Robinson, Basile Audoly, and Eitan Grinspun. 2008. Discrete elastic rods. In *ACM SIGGRAPH 2008 papers.* 1–12.

S. Bouaziz, S. Martin, T. Liu, L. Kavan, and M. Pauly. 2014. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Trans. Graph.* 33, 4 (2014), 154:1–154:11.

J. Brackbill and H. Ruppel. 1986. FLIP: A method for adaptively zoned, Particle-In-Cell calculations of fluid flows in two dimensions. *J Comp Phys* 65 (1986), 314–343.

J. Chang, J. Jin, and Y. Yu. 2002. A Practical Model for Hair Mutual Interactions. In *Symp. Comp. Anim.* 73–80.

Table 1. Demo parameters and performance.

Parameters and statistics	Dance (Fig. 1)	Random trajectory (Fig. 9)	Sub-stepping comparison (Fig. 13)	Long hair (Fig. 14, Playrate: 1)
# of particles	27,114	27,114	27,114	43,262
# of strands	2,076	2,076	2,076	1,053
Ratio of simulated strands	0.03	0.03	0.03	0.03
Max frame cost	6.38 ms	6.37 ms	6.02 ms	7.07 ms
Avg frame cost	3.84 ms	4.11 ms	3.95 ms	4.58 ms
Frame per second	260	243	253	218
Sub-step number	3	2	3	3
DER step size	1/120 s	1/180 s	1/120 s	1/120 s
MPM step size	1/360 s	1/360 s	1/360 s	1/360 s

Table 2. Simulation parameters.

Stage	Property	Value
DER	Stretching Young's modulus (Pa)	1.6e7
	Bending Young's modulus (Pa)	1.5e10
	Shear modulus (Pa)	7.7e9
MPM	Grid size (cm)	1
	Flip ratio	0.3
	Young's modulus (Pa)	1.2e3
	Poisson ratio	0.3
	Mass density (g/cm^3)	1.3
	Friction angle (°)	20
	Trap breaking ratio (min)	0.05
	Trap breaking ratio (max)	0.2

Table 3. Time breakdown for demo Dance.

Component	Percentage	
Number of sub-steps	1	3
Particles to grid	24.6%	38.0%
Solve DER system	49.0%	25.6%
Strands interpolation	5.9%	9.2%
Grid to particles	2.5%	3.9%
Build DER system	7.1%	3.7%
Others	10.9%	19.6%

Gilles Daviet. 2020. Simple and Scalable Frictional Contacts for Thin Nodal Objects. *ACM Trans. Graph.* 39, 4, Article 61 (jul 2020), 16 pages. https://doi.org/10.1145/3386569.3392439

G. Daviet, F. Bertails-Descoubes, and L. Boissieux. 2011. A Hybrid Iterative Solver for Robustly Capturing Coulomb Friction in Hair Dynamics. *ACM Trans. Graph.* 30, 6 (2011), 139:1–139:12.

A. Derouet-Jourdan, F. Bertails-Descoubes, G. Daviet, and J. Thollot. 2013. Inverse Dynamic Hair Modeling with Frictional Contact. *ACM Trans. Graph.* 32, 6 (2013), 159:1–159:10.

Yun Fei, Qi Guo, Rundong Wu, Li Huang, and Ming Gao. 2021a. Revisiting integration in the material point method: a scheme for easier separation and less dissipation. *ACM Transactions on Graphics (TOG)* 40, 4 (2021), 1–16.

Y. (R.) Fei, C. Batty, E. Grinspun, and C. Zheng. 2018. A Multi-scale Model for Simulating Liquid-fabric Interactions. *ACM Trans. Graph.* 37, 4 (2018), 51:1–51:16.

Yun (Raymond) Fei, Christopher Batty, Eitan Grinspun, and Changxi Zheng. 2019. A Multi-Scale Model for Coupling Strands with Shear-Dependent Liquid. *ACM Trans. Graph.* 38, 6, Article 190 (nov 2019), 20 pages. https://doi.org/10.1145/3355089.3356532

Yun (Raymond) Fei, Yuhan Huang, and Ming Gao. 2021b. Principles towards Real-Time Simulation of Material Point Method on Modern GPUs. *arXiv preprint* abs/2111.00699 (2021). arXiv:2111.00699 https://arxiv.org/abs/2111.00699

M. Gao, A. Pradhana, X. Han, Q. Guo, G. Kot, E. Sifakis, and C. Jiang. 2018a. Animating fluid sediment mixture in particle-laden flows. *ACM Trans. Graph.* 37, 4 (2018), 149.

M. Gao, X. Wang, K. Wu, A. Pradhana, E. Sifakis, C. Yuksel, and C. Jiang. 2018b. GPU Optimization of Material Point Methods. *ACM Trans. Graph.* 37, 6, Article 254 (2018), 12 pages.

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

Sunil Hadap. 2006. Oriented strands: dynamics of stiff multi-body system. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation.* 91–100.

S. Hadap and N. Magnenat-Thalmann. 2001. Modeling dynamic hair as a continuum. In *Comp Graph Forum*, Vol. 20. 329–338.

Xuchen Han, Theodore F Gast, Qi Guo, Stephanie Wang, Chenfanfu Jiang, and Joseph Teran. 2019. A hybrid material point method for frictional contact with diverse materials. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 2, 2 (2019), 1–24.

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–16.

C. Jiang, T. Gast, and J. Teran. 2017. Anisotropic elastoplasticity for cloth, knit and hair frictional contact. *ACM Trans. Graph.* 36, 4 (2017).

C. Jiang, C. Schroeder, J. Teran, A. Stomakhin, and A. Selle. 2016. The material point method for simulating continuum materials. In *SIGGRAPH Course.* 24:1–24:52.

Jianwei Jiang, Bin Sheng, Ping Li, Lizhuang Ma, Xin Tong, and Enhua Wu. 2020. Real-time hair simulation with heptadiagonal decomposition on mass spring system. *Graphical Models* 111 (2020), 101077.

D. Kaufman, R. Tamstorf, B. Smith, J. Aubry, and E. Grinspun. 2014. Adaptive Nonlinearity for Collisions in Complex Rod Assemblies. *ACM Trans. Graph.* 33, 4 (2014), 123:1–123:12.

G. Klár, T. Gast, A. Pradhana, C. Fu, C. Schroeder, C. Jiang, and J. Teran. 2016. Drucker-prager elastoplasticity for sand animation. *ACM Trans. Graph.* 35, 4 (2016), 103:1–103:12.

Mickaël Ly, Jean Jouve, Laurence Boissieux, and Florence Bertails-Descoubes. 2020. Projective Dynamics with Dry Frictional Contact. *ACM Trans. Graph.* 39, 4, Article 57 (jul 2020), 8 pages. https://doi.org/10.1145/3386569.3392396

Sebastian Martin, Bernhard Thomaszewski, Eitan Grinspun, and Markus Gross. 2011. Example-Based Elastic Materials. In *ACM SIGGRAPH 2011 Papers* (Vancouver, British Columbia, Canada) *(SIGGRAPH '11)*. Association for Computing Machinery, New York, NY, USA, Article 72, 8 pages. https://doi.org/10.1145/1964921.1964967

A. McAdams, A. Selle, K. Ward, E. Sifakis, and J. Teran. 2009. Detail Preserving Continuum Hair Simulation. *ACM Trans. Graph.* 28, 3 (2009), 62:1–62:6.

Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. 2007. Position based dynamics. *Journal of Visual Communication and Image Representation* 18, 2 (2007), 109–118.

Dinesh K Pai. 2002. Strands: Interactive simulation of thin solids using cosserat models. In *Computer graphics forum*, Vol. 21. Wiley Online Library, 347–352.

L. Petrovic, M. Henne, and J. Anderson. 2005. Volumetric methods for simulation and rendering of hair. *Tech Report* (2005).

A. Pradhana, T. Gast, G. Klár, C. Fu, J. Teran, C. Jiang, and K. Museth. 2017. Multi-species simulation of porous sand and water mixtures. *ACM Trans. Graph.* 36, 4 (2017).

D. Ram, T. Gast, C. Jiang, C. Schroeder, A. Stomakhin, J. Teran, and P. Kavehpour. 2015. A material point method for viscoelastic fluids, foams and sponges. In *Symp. Comp. Anim.* 157–163.

R. Rosenblum, W. Carlson, and E. Tripp. 1991. Simulating the structure and dynamics of human hair: modelling, rendering and animation. *J Vis Comp Anim* 2, 4 (1991), 141–148.

A. Selle, M. Lentine, and R. Fedkiw. 2008. A Mass Spring Model for Hair Simulation. *ACM Trans. Graph.* 27, 3 (2008), 64:1–64:11.

A. Stomakhin, C. Schroeder, L. Chai, J. Teran, and A. Selle. 2013. A material point method for snow simulation. *ACM Trans. Graph.* 32, 4 (2013), 102:1–102:10.

D. Sulsky, Z. Chen, and H. Schreyer. 1994. A particle method for history-dependent materials. *Comp Meth App Mech Eng* 118, 1 (1994), 179–196.

D. Sulsky, S. Zhou, and H. Schreyer. 1995. Application of a particle-in-cell method to solid mechanics. *Comp Phys Comm* 87, 1 (1995), 236–252.

Xinlei Wang, Yuxing Qiu, Stuart R Slattery, Yu Fang, Minchen Li, Song-Chun Zhu, Yixin Zhu, Min Tang, Dinesh Manocha, and Chenfanfu Jiang. 2020. A massively parallel and scalable multi-GPU material point method. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 30–1.

K. Ward, F. Bertails, T. Kim, S. Marschner, M. Cani, and M. Lin. 2007. A survey on hair modeling: Styling, simulation, and rendering. *IEEE Trans Vis Comp Graph* 13, 2 (2007), 213–234.

Y. Yue, B. Smith, C. Batty, C. Zheng, and E. Grinspun. 2015. Continuum foam: a material point method for shear-dependent flows. *ACM Trans. Graph.* 34, 5 (2015), 160:1–160:20.

Y. Yue, B. Smith, P. Y. Chen, M. Chantharayukhonthorn, K. Kamrin, and E. Grinspun. 2018. Hybrid Grains: Adaptive Coupling of Discrete and Continuum Simulations of Granular Media. *ACM Trans. Graph.* 37, 6, Article 283 (2018), 19 pages.

Cem Yuksel and Sarah Tariq. 2010. Advanced techniques in real-time hair rendering and simulation. In *ACM SIGGRAPH 2010 Courses.* 1–168.