Vertex Block Descent

ANKA HE CHEN, University of Utah, USA
ZIHENG LIU, University of Utah, USA
YIN YANG, University of Utah, USA
CEM YUKSEL, University of Utah, USA and Roblox, USA

Fig. 1. *Example simulation results using our solver, both of those methods involve more than 100 million DoFs and 1 million active collisions.*

We introduce vertex block descent, a block coordinate descent solution for the variational form of implicit Euler through vertex-level Gauss-Seidel iterations. It operates with local vertex position updates that achieve reductions in global variational energy with maximized parallelism. This forms a physics solver that can achieve numerical convergence with unconditional stability and exceptional computation performance. It can also fit in a given computation budget by simply limiting the iteration count while maintaining its stability and superior convergence rate.

We present and evaluate our method in the context of elastic body dynamics, providing details of all essential components and showing that it outperforms alternative techniques. In addition, we discuss and show examples of how our method can be used for other simulation systems, including particle-based simulations and rigid bodies.

CCS Concepts: • **Computing methodologies** → **Physical simulation**; *Collision detection.*

Additional Key Words and Phrases: physics-based simulation, elastic body, rigid body, time integration

ACM Reference Format:
Anka He Chen, Ziheng Liu, Yin Yang, and Cem Yuksel. 2024. Vertex Block Descent. *ACM Trans. Graph.* 43, 4, Article 116 (July 2024), 16 pages. https://doi.org/10.1145/3658179

Authors' Contact Information: Anka He Chen, ankachan92@gmail.com, University of Utah, Salt Lake City, UT, USA; Ziheng Liu, NA, University of Utah, Salt Lake City, UT, USA; Yin Yang, NA, University of Utah, Salt Lake City, UT, USA; Cem Yuksel, cem@cemyuksel.com, University of Utah, Salt Lake City, UT, USA and Roblox, Salt Lake City, UT, USA.

1 INTRODUCTION

Physics-based simulation is the cornerstone of most graphics applications and the demands from simulation systems to deliver improved stability, accelerated computational performance, and enhanced visual realism are ever-growing. Particularly in real-time graphics applications, the stability and performance requirements are so strict that realism can sometimes be begrudgingly considered of secondary importance.

Notwithstanding the substantial amount of research and groundbreaking discoveries made on physics solvers over the past decades, existing methods still leave some things to be desired. They either deliver high-quality results, but fail to meet the computational demands of many applications or fit in a given computation time by sacrificing quality. Stability, on the other hand, is always a challenge, particularly with strict computation budgets.

In this paper, we introduce *vertex block descent (VBD),* a physics solver that offers unconditional stability, superior computational performance than prior methods, and the ability to achieve numerical convergence to an implicit Euler integration. Though our method is a general solution that can be used for a variety of simulation problems, we present and evaluate it in the context of elastic body dynamics. Then, we briefly discuss how our method can be applied to some other simulation systems, including particle-based simulations and rigid bodies.

Our VBD method is based on block coordinate descent that performs vertex-based Gauss-Seidel iterations to solve the variational form of implicit Euler. For elastic body dynamics, each iteration runs a loop over the mesh vertices, adjusting the position of a single vertex at a time, temporarily fixing all others. This offers maximized parallelism when coupled with vertex-based mesh coloring, which can achieve an order of magnitude fewer colors (i.e. serialized

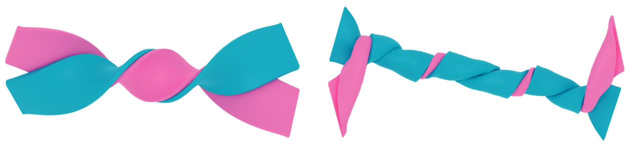


Fig. 2. *Twisting two beams together, totaling 97 thousand vertices and 266 thousand tetrahedra, demonstrating complex frictional contact and buckling.*

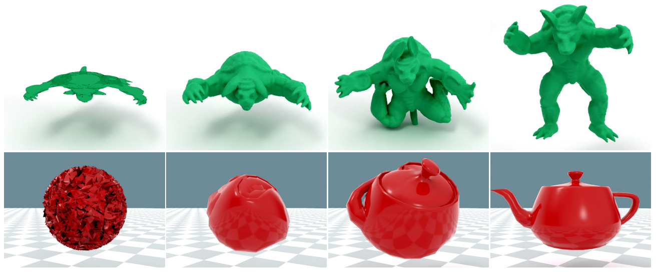


Fig. 3. *Stress tests that begin simulations under extreme deformations:* ***(top)*** *a perfectly flattened armadillo model with 15 thousand vertices and 50 thousand tetrahedra, and* ***(bottom)*** *a Utah teapot model with 2 thousand vertices and 8.5 thousand tetrahedra, deformed by randomly placing its vertices onto the surface of a sphere. Both models quickly recover to their original shape shortly after the simulation starts. Both simulations use accelerated iterations with* $\rho = 0.95$.

workloads) as compared to element-based parallelization. Our local position-based updates can ensure that the variational energy is always reduced. Therefore, our method maintains its stability even with a single iteration per time step and large time steps, operating with unconverged solutions containing a large amount of residual. With more iterations, it converges faster than its alternatives. Thus, it can more easily fit in a given computation budget, while maintaining stability with improved convergence.

We present all essential components of using VBD for elastic body dynamics, including formulations for damping, constraints, collisions, and friction. We also introduce a simple initialization scheme to warm-start the optimization and improve convergence. In addition, we discuss momentum-based acceleration techniques and parallelization in the presence of dynamic collisions. Our evaluation includes large simulations (Figure 1) and stress tests (e.g. Figure 2 and 3) that demonstrate VBD's performance, scalability, and stability.

2 RELATED WORK

There is a large body of work on physics-based simulation in computer graphics. Here we only discuss the recent and the most relevant work to our method.

Implicit time integrators are widely accepted as the primary methods for simulating elastic bodies in computer graphics due to their exceptional stability, especially when addressing stiff problems. Among these options, backward Euler [Baraff and Witkin 1998; Hirota et al. 2001; Martin et al. 2011; Volino and Magnenat-Thalmann 2001] is the most commonly utilized method, though other approaches like implicit Newmark [Bridson et al. 2002, 2005; Kane et al. 2000], BDF2 [Choi and Ko 2005; Hauth and Etzmuss 2001], and implicit-explicit [Eberhardt et al. 2000; Stern and Grinspun 2009] have also been explored. Backward Euler is often approximated as a single Newton step, solving a linear system of equations [Baraff and Witkin 1998]. Line search can be applied to improve robustness [Hirota et al. 2001]. Preconditioning [Ascher and Boxerman 2003] or positive-definite projection [Teran et al. 2005] can be used to improve convergence. To circumvent a full linear solve for every Newton step, Cholesky factorization [Hecht et al. 2012] emerges as a viable strategy. Techniques like multi-resolution [Capell et al. 2002; Grinspun et al. 2002] or multigrid [Bolz et al. 2003; Tamstorf et al. 2015; Wang et al. 2020; Xian et al. 2019; Zhu et al. 2010] solvers project finer details onto a coarser grid with fewer degrees of freedom, effectively reducing the computational cost of the linear system solver. Our solution linearizes the forces locally, avoiding the global linear system, and it converges to the same result as backward Euler with multiple Newton steps.

Additionally, stiffness warping [Müller et al. 2002] repurposes the stiffness matrix from the rest shape to handle significant rotational deformations. Using a quasi-Newton method [Liu et al. 2017] with an approximate Hessian can be far more cost-effective for computing its inverse with prefactoring than the actual Hessian. Example-based dynamics has also been explored [Chao et al. 2010; Martin et al. 2011; Müller et al. 2005], where the deformation energy is defined based on the nearest point in the example space. Projective Dynamics [Bouaziz et al. 2023] represents deformation energy via a series of constraints that can be solved independently, then synchronized either through a prefactorized global step to accelerate convergence.

The relation between dynamics, energy, and minimization has been leveraged in variational integrators [Kane et al. 1999, 2000; Kharevych et al. 2006; Lew et al. 2004; Simo et al. 1992]. Reformulating backward Euler to an optimization problem combined with optimization techniques to allows the usage of large steps [Gast et al. 2015; Martin et al. 2011]. Domain-decomposed optimization for solving the nonlinear problems of implicit numerical time integration can accelerate convergence Li et al. [2019]. Yet, the optimization formulation has its drawbacks, notably the varying problem formulation and initial positions in each step. Consequently, achieving consistent convergence within a fixed time budget becomes challenging. Real-time simulators compromise by accepting partially converged results, which visually resemble the final solution. Unfortunately, this compromise can lead to significant visual artifacts mainly due to retained residuals from earlier steps, which can accumulate across frames and can jeopardize the solver's stability. Our method, as illustrated in Figure 16, demonstrates exceptional stability even when retaining a substantial amount of residual with a limited number of iterations.

Position-based dynamics methods (PBD [Müller et al. 2007] and XPBD [Macklin et al. 2016]) convert forces into (soft) constraints and directly update the positions with Gauss-Seidel iterations operating on one constraint at a time. These position-based updates result in exceptional stability, which is often exploited by limiting the number of iterations to fix the computation cost, an effective strategy for real-time simulations. Akin to PBD, our method also works with position updates, but operates directly using the force formulations without converting them to constraints. Parallelization with XPBD is achieved by graph coloring the constraints (i.e. the dual graph) [Fratarcangeli and Pellacini 2015; Fratarcangeli et al. 2016; Ton-That et al. 2023]. However, this dual graph contains multiple times more connections (depending on the types of constraints) than the original graph of vertices, severely limiting the level of parallelism. In comparison, our method is parallelized by coloring the original graph, which leads to much fewer colors (i.e. computation groups that must be processed sequentially) and thereby better parallelism. More importantly, the approximations in XPBD's formulation introduce errors that make it diverge from the solution of implicit Euler and can degrade realism, particularly with large time steps and limited iteration count, which are common in practice. In addition, XPBD particularly struggles with high mass ratios. Our method, on the other hand, has none of these problems.

In recent years, a growing effort has been placed on accelerating simulations using GPUs [Huthwaite 2014; Lan et al. 2022; Li et al. 2020b, 2023; Macklin et al. 2020; Wang 2015; Wang and Yang 2016]. Among them, the first-order descent methods [Macklin et al. 2020; Wang and Yang 2016] have gained popularity due to their excellent parallelism. These methods employ a Jacobi-style preconditioned first-order descent on the backward Euler minimization formulation, enabling full vertex-level parallelization. However, Jacobi-style iterations typically converge substantially slower than Gauss-Seidel iterations. Also, such methods necessitate a line search to avoid overshooting and ensure stability.

Our method can be categorized as a coordinate descent method for optimization [Wright 2015]. In graphics, this technique has been used for geometric processing [Naitsat et al. 2020] and simulation with a barrier function [Lan et al. 2023]. Recently, Lan et al. [2023] proposed a hybrid scheme where Gauss-Sediel and Jacobi iterations are combined at each parallel call. In comparison, our method uses blocks of coordinates based on vertices instead of blocks based on elements, which results in much better parallelism and smaller local linear systems to solve, leading to faster convergence. Concurrently, Y.Chen et al. [2024] present a similar approach to ours for simulating quasistatic hyberelasticity.

3 VERTEX BLOCK DESCENT FOR ELASTIC BODIES

Our vertex block descent method is essentially a solver for optimization problems. Therefore, it can be applied to various simulation problems, particularly if they can formulated as an optimization problem. In this section, we explain our method in the context of elastic body dynamics for objects represented by a set of vertices that carry mass and a set of force elements and constraints that act on them. Generalization of our method to other example simulation problems is discussed later in Section 6.

We begin with deriving our global optimization method that splits the simulated system into vertex-level local systems (Section 3.1). Then, we discuss the methods we use for solving the local systems (Section 3.2) and describe how we incorporate damping (Section 3.3) and constraints (Section 3.4). We present our collision formulation (Section 3.5) and friction formulation (Section 3.6). After explaining our methods for warm-starting our optimization to improve convergence (Section 3.7), we describe how to incorporate momentum-based acceleration to improve convergence (Section 3.8). Finally, we discuss the improved parallelization that our method provides along with methods for efficiently parallelizing dynamically-introduced force elements due to varying collision events (Section 3.9).

3.1 Global Optimization

Our vertex block descent method is formulated based on the optimization form of implicit Euler time integration [Gast et al. 2015; Martin et al. 2011]. Given a system with N vertices, we represent the state of our simulation at step t as $(\mathbf{x}^t, \mathbf{v}^t)$, where $\mathbf{x}^t \in \mathbb{R}^{3N}$ and $\mathbf{v}^t \in \mathbb{R}^{3N}$ are the concatenated position and velocity vectors, respectively. The resulting optimization problem can be written as

$$\mathbf{x}^{t+1} = \operatorname*{argmin}_{\mathbf{x}} G(\mathbf{x}) \,, \tag{1}$$

for evaluating the positions at the end of the time step $\mathbf{x}^{t+1}$ that minimizes the *variational energy* $G(\mathbf{x})$ in the form of

$$G(\mathbf{x}) = \frac{1}{2h^2}\|\mathbf{x}-\mathbf{y}\|_M^2 + E(\mathbf{x}) \,. \tag{2}$$

The first term here is the *inertia potential*, which contains the time step size h, the mass-weighted norm $\|\cdot\|_M$, and

$$\mathbf{y} = \mathbf{x}^t + h\mathbf{v}^t + h^2\mathbf{a}_{\text{ext}} \tag{3}$$

with a fixed external acceleration $\mathbf{a}_{\text{ext}}$, such as gravity. The second term $E(\mathbf{x})$ is the *total potential energy* evaluated at $\mathbf{x}$.

We propose an optimization technique that falls under the category of coordinate descent methods to efficiently minimize this energy G. If we only modify a single vertex at a time, fixing all other vertices, the part of the energy term $E(\mathbf{x})$ that is affected only includes the set of force elements $\mathcal{F}_i$ that are acting on (or using the position of) vertex i. Thus, we define the *local variational energy* G_i around vertex i as

$$G_i(\mathbf{x}) = \frac{m_i}{2h^2}\|\mathbf{x}_i - \mathbf{y}_i\|^2 + \sum_{j\in\mathcal{F}_i} E_j(\mathbf{x}) \,, \tag{4}$$

where m_i is the mass of the vertex and E_j is the energy of force element j in $\mathcal{F}_i$.

Note that G is not equal to the sum of local variational energies, i.e. $G(\mathbf{x}) \neq \sum_i G_i(\mathbf{x})$, simply because the force elements appear multiple times in this sum (once for each of its vertices). However, when we modify the position of a single vertex only, the reduction in G_i is equal to the resulting reduction in G.

Based on this observation, our method operates on a single vertex at a time and updates its position by minimizing the local energy

$$\mathbf{x}_i \leftarrow \operatorname*{argmin}_{\mathbf{x}_i} G_i(\mathbf{x}) \tag{5}$$

and solves the global system using Gauss-Seidel iterations. Each local minimization for a vertex effectively finds a descent step for G using the degrees of freedom (DoF) for the vertex as a block of coordinates, hence the name *vertex block descent* (VBD). After each iteration, the total reduction in G is equal to the sum of all reductions in G_i. In other words, the energy change of each vertex position adjustment is accumulated to the system energy. Consequently, if we can make sure that each local energy drops G_i when we are adjusting vertex i, we can guarantee that we are descending the system energy G.

Thus, our system directly operates on vertex positions and the resulting velocities are calculated following the implicit Euler formulation

$$\mathbf{v}^{t+1} = \frac{1}{h}\left(\mathbf{x}^{t+1} - \mathbf{x}^{t}\right) . \tag{6}$$

3.2 Local System Solver

The position updates per vertex (in Equation 5) involve solving a local system that only depends on the position change of a single vertex, represented by G_i. Note that Equation 4 only has 3 DoF, so the cost of evaluating and inverting its Hessian is much cheaper compared to the global problem in Equation 2. Therefore, we can fully utilize the second-order information and use Newton's method to minimize the localized energy G_i, which involves solving the 3D linear system

$$\mathbf{H}_i \, \Delta\mathbf{x}_i = \mathbf{f}_i \, , \tag{7}$$

where $\Delta\mathbf{x}_i$ is the change in position, $\mathbf{f}_i$ is the total force acting on the vertex, calculated using

$$\mathbf{f}_i = -\frac{\partial G_i(\mathbf{x})}{\partial \mathbf{x}_i} = -\frac{m_i}{h^2}\left(\mathbf{x}_i - \mathbf{y}_i\right) - \sum_{j\in\mathcal{F}_i} \frac{\partial E_j(\mathbf{x})}{\partial \mathbf{x}_i} \, , \tag{8}$$

and $\mathbf{H}_i \in \mathbb{R}^{3\times 3}$ is the Hessian of G_i with respect to the DoF of vertex i, such that

$$\mathbf{H}_i = \frac{\partial^2 G_i(\mathbf{x})}{\partial \mathbf{x}_i \partial \mathbf{x}_i} = \frac{m_i}{h^2}\mathbf{I} + \sum_{j\in\mathcal{F}_i} \frac{\partial^2 E_j(\mathbf{x})}{\partial \mathbf{x}_i \partial \mathbf{x}_i} \, . \tag{9}$$

Here, the first term is a diagonal matrix and the second one is the sum of Hessians of the force elements with respect to vertex i. Intuitively, the solution of this linear system is the extreme point of the quadratic approximation for the localized energy G_i. By reducing G_i with each iteration, we can guarantee a reduction in G, thereby iteratively solving the global system in Equation 2.

We can analytically solve this system using $\Delta\mathbf{x}_i = \mathbf{H}_i^{-1}\mathbf{f}_i$. For such a small system, the analytical solver is very efficient and stable. We found it to be faster than solvers based on Conjugate Gradient or LU/QR decomposition. Another advantage of the analytical solver is that it does not require the Hessian to be positive-definite. Of course, when the Hessian is not positive-definite, the direction given by Equation 7 may not be a descent direction. Nevertheless, we opt for this direction regardless, recognizing that even when $\mathbf{H}_i$ is not positive-definite, solving Equation 7 still brings us towards the extremum of the quadratic approximation for the localized energy G_i. This solution is close to where the gradient of the inertia and the potential terms balance out and it is usually a stable state of the system. Also, the motivation of the variational form of implicit Euler (Equation 2), is to find a point where $dG(\mathbf{x})/d\mathbf{x} = 0$. Therefore, any extreme point is a valid solution of implicit Euler, and it does not have to be a local minimum. In all our experiments, including those specifically designed stress tests (see Figure 2 and Figure 3), we have consistently observed that this scheme does not pose any issues concerning system stability or convergence.

An alternative solution to this is the PSD Hessian projection [Teran et al. 2005]. However, it is exceptionally rare for the Hessian to not be positive-definite, and the PSD projection process is notably costly due to multiple SVD decompositions. Engaging in this costly operation to prevent such rare events seems unjustified, especially considering that these occurrences do not jeopardize system stability or convergence significantly.

Another challenge with the analytical solver arises when encountering a (nearly) rank-deficient Hessian. To address this, we propose a simple solution: if $|\det(\mathbf{H}_i)| \le \epsilon$ for some small threshold ϵ, we opt to bypass adjusting this particular vertex for that iteration. Given that its neighboring vertices are likely to undergo adjustments before the next iteration, it is improbable that its Hessian will remain rank-deficient in subsequent iterations. With this simple solution, in the extreme scenario where all vertices possess a rank-deficient Hessian, the system could potentially become frozen. Yet, it is crucial to note that such a case is highly improbable, since the Hessian of the inertia potential is always full-rank. One potential remedy for this would be switching to the modified Conjugate Gradient solver [Lan et al. 2023] when such a case happens. However, doing this will add additional branching to the code and can slow down the solver. Thus, we have not included it in our implementation, but, depending on specific use cases, there is always the flexibility to opt for the Conjugate Gradient solver as a backup solution.

The linear system in Equation 7 corresponds to a single Newton step, so it does not necessarily provide the optimal solution for Equation 5. In fact, since it is just a second-order approximation of G_i, it does not even guarantee a reduction in G_i. To ensure the descent of energy with this single step, we can incorporate a backtracking line search along the descent direction. Note that, unlike global line searches in descent-based simulation methods (e.g. Wang and Yang [2016]), this line search operates locally. It specifically verifies the descent of G_i, which in turn guarantees a descent in G without the need for evaluating the global system.

Line search avoids over-shooting and, thereby, ensures stability. In practice, however, the additional computation cost of line search may not be justified. In our experiments, we found that line search costs an extra 40% computation time, while not providing any measurable benefits. This is because VBD can maintain stability even without line search. Therefore, the results we present in this paper do not include line search, though it is an option available.

3.3 Damping

Damping plays a crucial role in simulations. It prevents excessive oscillations and also enhances system stability. Despite the inherent numerical damping introduced by the implicit Euler method, providing users with manual control over damping is highly desirable. To address this, we have integrated a simplified Rayleigh damping model into our solver [Sifakis and Barbic 2012]. This process is both straightforward and efficient, as it also operates locally within the 3×3 system and utilizes the precomputed stiffness matrix.

Since we are relying on implicit Euler, we can represent the velocity as position change, using $\mathbf{v}_i = (\mathbf{x}_i - \mathbf{x}_i^t)/h$. Then, we can add the damping term to the Hessian in Equation 9, resulting

$$\mathbf{H}_i = \frac{m_i}{h^2}\mathbf{I} + \sum_{j\in\mathcal{F}_i} \frac{\partial^2 E_j(\mathbf{x})}{\partial \mathbf{x}_i \partial \mathbf{x}_i} + \left(\sum_{j\in\mathcal{F}_i} \frac{k_d}{h} \frac{\partial^2 E_j(\mathbf{x})}{\partial \mathbf{x}_i \partial \mathbf{x}_i} \right) , \tag{10}$$

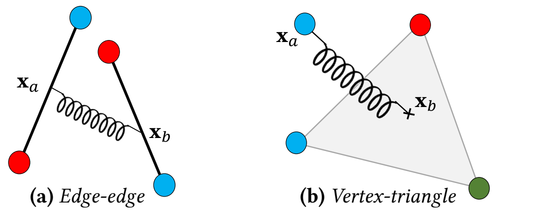


Fig. 4. *Two collision types: **(a)** edge-edge can have at most two pairs and **(b)** vertex-face can have at most one pair with the same color, since vertices on the same side of a collision must have different colors.*

where k_d is the damping coefficient. Finally, we add the damping force to Equation 8 using the same damping term, such that

$$\mathbf{f}_i = -\frac{m_i}{h^2}(\mathbf{x}_i - \mathbf{y}_i) - \sum_{j\in\mathcal{F}_i} \frac{\partial E_j(\mathbf{x})}{\partial \mathbf{x}_i} - \left(\sum_{j\in\mathcal{F}_i} \frac{k_d}{h}\frac{\partial^2 E_j(\mathbf{x})}{\partial \mathbf{x}_i \partial \mathbf{x}_i}\right)(\mathbf{x}_i - \mathbf{x}_i^t)\,. \tag{11}$$

3.4 Constraints

Since our method directly manipulates the position of each vertex, managing a constraint on a vertex becomes straightforward. Constraints generally fall into two categories: unilateral ($C(\mathbf{x}) \le 0$) or bilateral ($C(\mathbf{x}) = 0$). With bilateral constraints, if a vertex position is directly set to a specific value, we simply skip updating its position. Otherwise, it is constrained to a (usually linear) subspace. Our approach involves representing the constrained vertex position using the subspace basis. This transforms both the vertex position and gradient into an L-dimensional vector, where L is the subspace dimension. Consequently, handling local steps for constrained vertices involves solving an $L \times L$ system. Regarding unilateral constraints, we allow compromises and define potential energy to be solved alongside other potentials. This method handles world box constraints in our simulations.

3.5 Collisions

Collisions can be handled by simply introducing a quadratic collision energy per vertex, based on the penetration depth d, such that

$$E_c(\mathbf{x}) = \frac{1}{2} k_c\, d^2 \qquad \text{with} \qquad d = \max\left(0, (\mathbf{x}_b - \mathbf{x}_a)\cdot \hat{\mathbf{n}}\right), \tag{12}$$

where k_c is the collision stiffness parameter, $\mathbf{x}_a$ and $\mathbf{x}_b$ are the two *contact points* on either side of the collision, and $\hat{\mathbf{n}}$ is the contact normal.

There are two collision types for triangle meshes (Figure 4):

- Edge-edge collisions use continuous collision detection (CCD). $\mathbf{x}_a$ and $\mathbf{x}_b$ correspond to the intersection points on either edge and the contact normal is the direction between them, i.e. $\hat{\mathbf{n}} = \mathbf{n}/\|\mathbf{n}\|$, where $\mathbf{n} = \mathbf{x}_b - \mathbf{x}_a$.
- Vertex-triangle collisions are detected either by CCD or discrete collision detection (DCD). In this case, $\mathbf{x}_a$ is the colliding vertex and $\mathbf{x}_b$ is the corresponding point on either the collision point (for CCD) or the closest point (for DCD) on the triangle [Chen et al. 2023]. $\hat{\mathbf{n}}$ is the surface normal at $\mathbf{x}_b$.

In our implementation, we perform a DCD at the beginning of the time step using $\mathbf{x}^t$ to identify vertices that have already penetrated, and the rest of the collisions use CCD. We simplify the computation of the gradient and the Hessian of the collision energy by not differentiating through $\hat{\mathbf{n}}$, i.e. assuming that $\hat{\mathbf{n}}$ is constant.

Performing collision detection at every iteration using CCD can be expensive and can easily become the bottleneck. Therefore, in our implementation, we perform CCD once every n_{col} iterations. While this has the risk of missing some collision events, they are likely to be captured via DCD in the next time step. All detected collisions remain as force elements until the next collision detection. For vertex-triangle collisions detected with DCD, it is important to recompute the closest point ($\mathbf{x}_b$) before computing the gradient and Hessian of E_c.

3.6 Friction

To compute friction for collision c, we must consider the relative motion of the contact points defined as

$$\delta\mathbf{x}_c = (\mathbf{x}_a - \mathbf{x}_a^t) - (\mathbf{x}_b - \mathbf{x}_b^t)\,, \tag{13}$$

where $\mathbf{x}_a^t$ and $\mathbf{x}_b^t$ are the positions of $\mathbf{x}_a$ and $\mathbf{x}_b$ at the beginning of the time step.

With this $\delta\mathbf{x}_c$, we can use the friction model of *incremental potential contact* (IPC) [Li et al. 2020a]. First, we project $\delta\mathbf{x}_c$ to the 2D contact tangential space, using a transformation matrix $\mathbf{T}_c \in \mathbb{R}^{3\times 2}$, to evaluate the tangential relative translation $\mathbf{u}_c = \mathbf{T}_c^T \delta\mathbf{x}_c$, where $\mathbf{T}_c = [\hat{\mathbf{t}}\ \hat{\mathbf{b}}]$ is formed by a tangent $\hat{\mathbf{t}}$ and binormal $\hat{\mathbf{b}}$ vectors orthogonal to $\hat{\mathbf{n}}$. The signed magnitude of the collision force applied on vertex i is $\lambda_{c,i} = \frac{\partial E_c}{\partial \mathbf{x}_i}\cdot\hat{\mathbf{n}}$. Note that the sign of $\lambda_{c,i}$ is different for vertices on different sides of the collision. Let μ_c be the friction coefficient. We can then calculate the friction force using

$$\mathbf{f}_{c,i} = -\mu_c\,\lambda_{c,i}\,\frac{\partial \delta\mathbf{x}_c}{\partial \mathbf{x}_i}\,\mathbf{T}_c\, f_1(\|\mathbf{u}_c\|)\,\frac{\mathbf{u}_c}{\|\mathbf{u}_c\|}\,, \text{ where} \tag{14}$$

$$f_1(u) = \begin{cases} 2\left(\frac{u}{\epsilon_v h}\right) - \left(\frac{u}{\epsilon_v h}\right)^2, & u \in (0, h\epsilon_v) \\ 1, & u \ge h\epsilon_v \end{cases}. \tag{15}$$

Here, f_1 serves as a smooth transition function between static and dynamic friction. When the relative velocity exceeds a small threshold ϵ_v, dynamic friction is applied. Conversely, if the relative velocity is below this threshold, static friction is applied, scaling between the range of $[0, 1]$.

In our formulation, we need the Hessian of the friction term, which is the derivative of this function. We approximate the derivative by not differentiating through $\|\mathbf{u}_c\|$ for a more stable friction force formulation [Macklin et al. 2020], such that

$$\frac{\partial \mathbf{f}_{c,i}}{\partial \mathbf{x}_i} \approx -\mu_c\,\lambda_{c,i}\,\frac{\partial \delta\mathbf{x}_c}{\partial \mathbf{x}_i}\,\mathbf{T}_c\,\frac{f_1(\|\mathbf{u}_c\|)}{\|\mathbf{u}_c\|}\mathbf{T}_c^T\left(\frac{\partial \delta\mathbf{x}_c}{\partial \mathbf{x}_i}\right)^T. \tag{16}$$

Without this approximation, PSD projection and line search might be needed to ensure stability. Finally, all friction forces $\mathbf{f}_{c,i}$ and their derivatives $\partial\mathbf{f}_{c,i}/\partial\mathbf{x}_i$ are added to $\mathbf{f}_i$ and $\mathbf{H}_i$, respectively.

3.7 Initialization

Since our method is an iterative solver, we begin with an *initial guess* for $\mathbf{x}$. The closer this initial guess is to the resulting $\mathbf{x}^{t+1}$, the fewer number of iterations we would need to converge. Typically, if the simulation converges, different initializations should not sig-

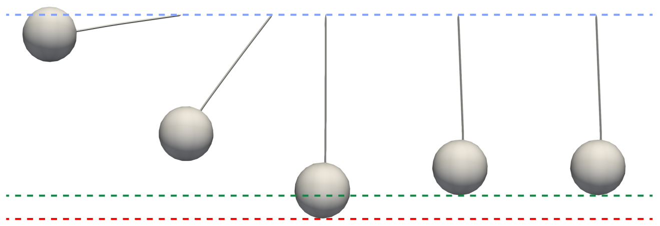


Fig. 5. *Different initialization options for a swinging elastic pendulum dropped from the same height (blue line) simulated with our method using only 20 iterations per frame, showing the same frame of the simulation. Notice that initializing using **(a)** previous position and **(b)** inertia fail to properly move under gravity, while **(c)** inertia and acceleration leads to accessive stretching (red line) when VBD does not run to convergence. **(d)** Our adaptive solution closely matches **(e)** the reference generated by fully converged Newton's method (green line).*

nificantly affect the final results, although they may influence the number of iterations required to achieve convergence.

Providing a good initial guess (one that is close to $\mathbf{x}^{t+1}$) is particularly important for applications with a limited computation budget, e.g. using a fixed number of iterations. In such applications, the initial guess can strongly impact the remaining residual at the end of the final iteration.

We begin with considering three simple options for initialization:

(a) **Previous position:** $\mathbf{x} = \mathbf{x}^t$

(b) **Inertia:** $\mathbf{x} = \mathbf{x}^t + h\mathbf{v}^t$

(c) **Inertia and acceleration:** $\mathbf{x} = \mathbf{x}^t + h\mathbf{v}^t + h^2\mathbf{a}_{\text{ext}} = \mathbf{y}$

Option (a) struggles with substantially stiff materials. As we adjust each vertex separately, assuming the others remain fixed, our method must lean on the inertia potential's gradient (i.e. $m_i(\mathbf{x}_i - \mathbf{y}_i)/2h^2$) to march toward the local minimum. With stiff materials, this method results in notably slower convergence rates due to the inertia potential being considerably less stiff. Consequently, it encounters challenges in simulating scenarios that resemble free fall, like a swinging elastic pendulum at its maximum height, as shown in Figure 5a.

Option (b) allows the system to start with the inertia of the previous step, which helps, but terminating the iterations prior to convergence can again result in local material stiffness overpowering external acceleration, as shown in Figure 5b.

Option (c) is similar to the initialization of position-based dynamics, and performs notably better as it effectively preserves inertia and properly reacts to external acceleration. However, with a limited number of iterations, materials behave softer than they should, often resulting in overstretching or collapsing under gravity. An example of this can be seen in Figure 5c, where the pendulum extends more than it should. Most notably, it struggles with steady objects at rest in contact, consistently initializing them into a penetrating state, as if they are in free fall. In such cases, the contact forces must entirely undo the position change of initialization during the iterations. This not only creates unnecessary computation, but also places considerable strain on the accuracy of the collision detection and handling methods (including friction). Therefore, properly simulating objects that are stacked on top of each other becomes a major challenge with this initialization option.

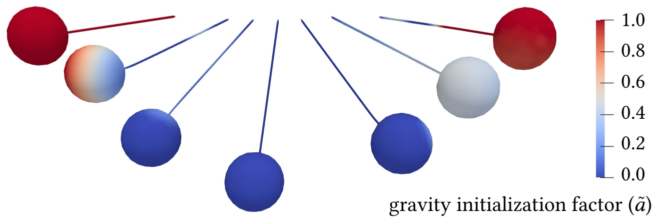


Fig. 6. *The ratio of gravity $\tilde{a}$ used with adaptive initialization during the swinging of an elastic pendulum. The model is a single-piece tetrahedral mesh. It automatically distinguishes vertices in approximate free-fall (red) and those where elasticity counteracts gravity (blue).*

We propose an adaptive initialization scheme that combines options (b) and (c), taking advantage of the freedom that VBD provides in the choice of initialization. This scheme uses

(d) **Adaptive:** $\mathbf{x} = \mathbf{x}^t + h\mathbf{v}^t + h^2\tilde{\mathbf{a}}$

replacing the external acceleration $\mathbf{a}_{\text{ext}}$ in (c) with an estimated acceleration term $\tilde{\mathbf{a}}$, determined by exploiting the typical similarity between two consecutive time steps. We begin with the acceleration of the previous frame $\mathbf{a}^t = (\mathbf{v}^t - \mathbf{v}^{t-1})/h$ and compute its component a^t_{ext} along the external acceleration direction $\hat{\mathbf{a}}_{\text{ext}} = \mathbf{a}_{\text{ext}}/\|\mathbf{a}_{\text{ext}}\|$, such that $a^t_{\text{ext}} = \mathbf{a}^t \cdot \hat{\mathbf{a}}_{\text{ext}}$. Then, we simply make sure that the estimated acceleration does not exceed the external acceleration, using

$$\tilde{\mathbf{a}} = \tilde{a}\,\mathbf{a}_{\text{ext}}\,, \quad \text{where} \quad \tilde{a} = \begin{cases} 1\,, & \text{if } a^t_{\text{ext}} > \|\mathbf{a}_{\text{ext}}\| \\ 0\,, & \text{if } a^t_{\text{ext}} < 0 \\ a^t_{\text{ext}}/\|\mathbf{a}_{\text{ext}}\|\,, & \text{otherwise.} \end{cases} \tag{17}$$

This adaptive approach includes external acceleration in the initialization when the motion of a vertex resembles free fall. When an object is stationary, however, as in rest-in-contact, it maintains the previous position in the initialization, preventing undesired penetrations before the first iteration. It also successfully avoids excessive stretching, as can be seen in Figure 5d. Visualizations of different $\tilde{a}$ values in this simulation are shown in Figure 6. In short, our adaptive initialization is a simple but effective strategy and it is possible because VBD does not dictate a particular initialization (unlike XPBD, for example).

3.8 Accelerated Iterations

We use the Chebyshev semi-iterative approach [Wang 2015] to improve the convergence of our method, though other momentum-based acceleration techniques, such as the Nesterov's method [Golub and Van Loan 2013] can be applied as well. The Chebyshev method iteratively computes an acceleration ratio based on the approximation of the system's spectral radius. Instead of directly using the output vertex positions of Gauss-Seidel $\bar{\mathbf{x}}^{(n)}$ after iteration n, it recomputes the positions at the end of the iteration using

$$\mathbf{x}^{(n)} = \omega_n(\bar{\mathbf{x}}^{(n)} - \mathbf{x}^{(n-2)}) + \mathbf{x}^{(n-2)}\,, \tag{18}$$

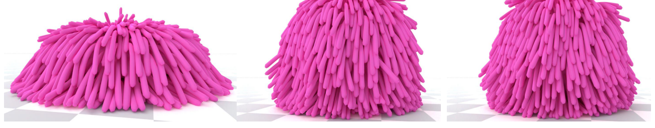

(a) *Not accelerated* **(b)** *Accelerated* **(c)** *Reference*

Fig. 7. *Demonstrating the accelerator's impact in a collision-intensive scene: a squishy ball (230K vertices, 700K tetrahedra) dropping and bouncing. Both* ***(a)*** *and* ***(b)*** *use $h = 1/120$ seconds with a constant number of* 120 *iterations per time step, taking* 0.11 *seconds of average computation time per frame.* ***(a)*** *Without acceleration 120 iterations appear to be insufficient.* ***(b)*** *Our acceleration scheme ($\rho = 0.95$), skipping colliding vertices, manages to resolve complex collisions, notably enhancing elasticity convergence for much stiffer outcomes, closely matching* ***(c)*** *the reference computed using 2000 iterations.*

where ω_n is the acceleration ratio that changes at each iteration as

$$\omega_n = \frac{4}{4 - \rho^2 \omega_{n-1}} \quad \text{with} \quad \omega_1 = 1 \quad \text{and} \quad \omega_2 = \frac{2}{2 - \rho^2}, \tag{19}$$

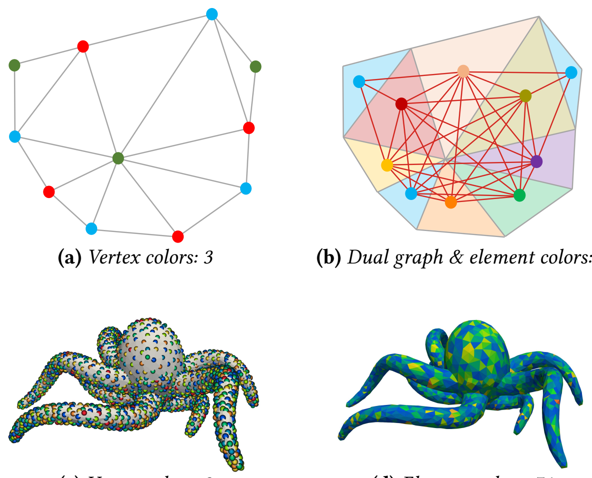
where $\rho \in (0, 1)$ is the estimated spectral radius, which can be set manually, or automatically tuned using the technique introduced in [Wang and Yang 2016]. Note that this position recomputation procedure is performed globally after each Gauss-Seidel iteration is completed, not after each local solver step.

This accelerator was originally developed for solving linear systems, assuming the energy to be smooth and (nearly) quadratic. Elastic energy generally fulfills these criteria. However, collision energy tends to be discontinuous and highly stiff, making the use of an accelerator in collision-intensive scenes prone to overshoot and compromise the system's stability. To address this, we propose a simple yet highly effective solution for accelerating scenes with collisions: skipping the accelerations for actively colliding vertices. Note that the acceleration must be a continuous process. If a vertex is detected colliding at a certain iteration, we will skip the acceleration for it in all the following iterations in the same step, regardless of whether the collision has been resolved. This approach has minimal impact on the convergence of elasticity, since typically only a small fraction of vertices are in collision. Also, for those colliding vertices, the elasticity is usually overpowered by the collision forces. Thus, this solution maintains the stability of the system, while effectively accelerating the convergence of elasticity, as shown in Figure 7.

3.9 Parallelization

Gauss-Seidel-type iterative methods are often parallelized using graph coloring by determining groups (i.e. colors) that can be handled in parallel without impacting the sequential nature of the Gauss-Seidel loop. Obviously, the same can be applied to VBD by simply coloring vertices such that no force element uses multiple vertices of the same color.

The advantage of VBD here is that, because it colors vertices, it typically results in much fewer colors as compared to techniques that color constraints/force elements, such as PBD. This is because coloring these elements is equivalent to coloring the nodes of the


(a) *Vertex colors: 3* **(b)** *Dual graph & element colors: 8*

(c) *Vertex colors: 8* **(d)** *Element colors: 76*

Fig. 8. *Coloring vertices vs. elements:* ***(a)*** *vertex coloring needs 3 colors for 10 vertices while* ***(b)*** *element coloring (i.e. coloring the vertices of the dual graph) needs 8 colors for 10 triangles in this example. The difference is more pronounced for tet-meshes:* ***(c)*** *our vertex coloring uses only 8 colors for 3,891 vertices while* ***(d)*** *our element coloring implementation needs 76 colors for 14,802 tetrahedra in this example.*

dual graph, which not only has more nodes, but, more importantly, also has a lot more connections per vertex in general. Examples of this are shown in Figure 8. Since different colors must be handled sequentially, fewer colors means better parallelism.

When all force elements are known ahead of the simulation, graph coloring can be performed as a preprocess. However, dynamically generated constraints/force elements, such as ones due to collisions, cannot be known ahead of time, requiring dynamic recoloring.

In our implementation for elastic body dynamics, we avoid the cost of recoloring by precomputing coloring only for material forces, ignoring dynamically generated force elements due to collisions. This means that these collision forces may use multiple vertices of the same color. Therefore, we cannot simply run a parallel loop over all vertices of the same color and update them, because handling a vertex with a dynamically generated force element may run into race conditions (with partially updated vertex positions) when accessing other vertices of the same color.

We resolve this by having an auxiliary vertex position buffer ($\mathbf{x}^{\text{new}}$) that stores the updated vertex position. When executing each local VBD position update, we write the updated vertex positions to the auxiliary buffer, instead of directly overwriting the original vertex position buffer. Then, we copy the updated positions from the auxiliary buffer to the original vertex position buffer after each color pass. This prevents the race conditions that arise from simultaneous read and write operations on vertex positions.

With this process, dynamically generated force elements using multiple vertices of the same color result in (partially) Jacobi-style iterations for those vertices, because they rely on the positions from the previous iteration of those vertices. For vertices with different colors, it is equivalent to Gauss-Seidel iterations, considering the updated positions of the vertices with different colors. Note

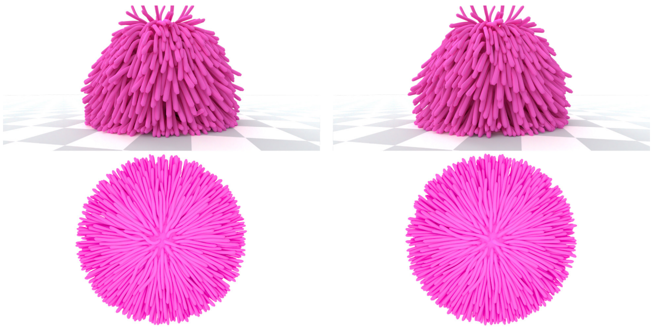


Fig. 9. *Handling collisions using **(a)** our scheme without recoloring and **(b)** recoloring to achieve perfect Gauss-Seidel iterations, both simulated using friction forces and accelerated iterations with* $\rho = 0.95$. *Notice that the results are highly similar, though not identical.*

that our algorithm does not explicitly switch between Jacobi and Gauss-Seidel iterations, but the resulting iteration we describe above corresponds to either (partially) Jacobi or Gauss-seidel, depending on the colors of the colliding vertices.

One might expect this solution to negatively impact the convergence of VBD. Fortunately, however, such Jacobi-style information exchanges are relatively rare. This is because, as shown in Figure 4, with vertex-face collisions at most one pair and with edge-edge collisions at most two pairs of vertices can have the same color. Also, vertices with the same colors must be located on different sides of the collision; therefore, their elastic energies are usually decoupled. This ensures that the majority of the information exchange follows the Gauss-Seidel order and thereby the impact of our solution on convergence is minimal. Figure 9 presents an example with a large number of collisions, comparing our solution of skipping recoloring to proper Gauss-Seidel iterations with recoloring, showing that the differences are minor.

Our solution also works with other types of topological changes, such as tearing and fracturing. Deleting force elements does not require any changes to vertex coloring. When an object is split by duplicating vertices, as in the case of tearing a piece of cloth along some edges (see Figure 10), duplicated vertices can safely inherit the colors of their original vertices.

4 GPU IMPLEMENTATION

In this section, we describe a GPU implementation specifically designed to leverage the inherent parallelization mechanism of modern GPUs, which consists of two hierarchical levels: block-level and thread-level parallelism. Block-level parallelism facilitates large-scale parallel operations, assuming that each block operates independently. On the other hand, thread-level parallelism provides finer, single-instruction-multiple-thread (SIMT) style parallelism, allowing for inter-thread communication and synchronization within the same block.

Reflecting on VBD, we observed that it naturally aligns with this hierarchical architecture. We have thousands of vertices within a single color category that operate independently, and each vertex is associated with multiple force elements, which can be processed

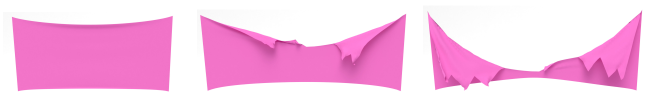


Fig. 10. *Tearing a piece of cloth with 2500 vertices and 4800 triangles.*

concurrently. Algorithm 1 shows the pseudocode of our implementation. It uses block-level parallelism for processing each vertex. The threads within each block are used for computing the forces and Hessians, storing them in local shared memory, and computing the sums via reduction. We use a fixed number of threads for each block. When the total number of adjacent force elements exceeds the number of threads for each block, individual threads will loop over their assigned elements. During this process, they calculate the forces and Hessians for these force elements and then sum them to their assigned shared memory.

In our experiments, we observed nearly *an order of magnitude performance improvement*, as compared to processing each vertex with a single thread. The primary advantage lies in the optimization of the memory access pattern, a common bottleneck in GPU programs. This implementation reduces memory divergence within blocks. Because the neighboring force elements of a vertex often share multiple vertices, the threads within the same block can share a global memory access to those shared vertices. Furthermore, this strategy improved the parallelism of the algorithm by allowing parallel evaluation of the force and Hessian of adjacent force elements, which are then written to the significantly faster shared memory. This bypasses the slower global memory and enables the parallel aggregation of force and Hessian values, enhancing the overall efficiency of the process.

5 RESULTS

We evaluate our method with elastic body dynamics qualitatively with various tests and quantitatively with comparisons to alternative methods. We use Neo-Hookean [Smith et al. 2018] materials (without the logarithmic term) for our volumetric objects, StVK [Volino et al. 2009] for clothes, and linear springs for elastic rods.

We use a fixed frame time of $1/60$ seconds and a fixed iteration count $n_{\max}$, instead of estimating convergence after every iteration. Each frame is computed using S substeps, such that $h = 1/(60S)$ seconds. Using smaller time steps increases accuracy and reduces numerical damping with any implicit Euler method. With VBD, a smaller time step only requires fewer iterations per step for similar visual quality, but it can also achieve a smaller residual with the same number of total iterations per frame (i.e., $Sn_{\max}$). The number of threads per block is set to 16 for all the experiments.

We use no line search in our tests, as none of our tests required it for stability, and running line search can result in a noticeable drop in performance. In our experiments, we apply CCD only in the first iteration (i.e. $n_{\text{col}} = n_{\max}$), unless otherwise specified.

In our implementation, we handle collisions on the CPU using Intel's Embree library [Wald et al. 2014]. The two phases with parallel loops are implemented on the GPU using CUDA. All timing results are generated on a computer with an AMD Ryzen 5950X CPU, 64GB

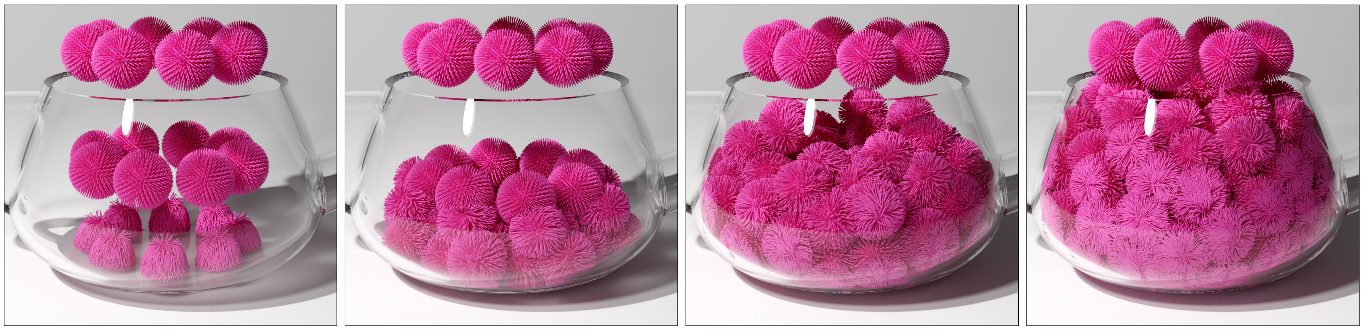


Fig. 11. *Simulation of 216 squishy balls with tentacles, a total of 48 million vertices and 151 million tetrahedra, dropped into a Utah teapot, forming a stable pile with active frictional contacts. The average and maximum computation time per time step is 3.6 and 3.9 seconds, respectively, using* $S = 4$ *substeps per frame and* $n_{\max} = 40$ *iterations per step. The final frame of this simulation is shown in Figure 1.*

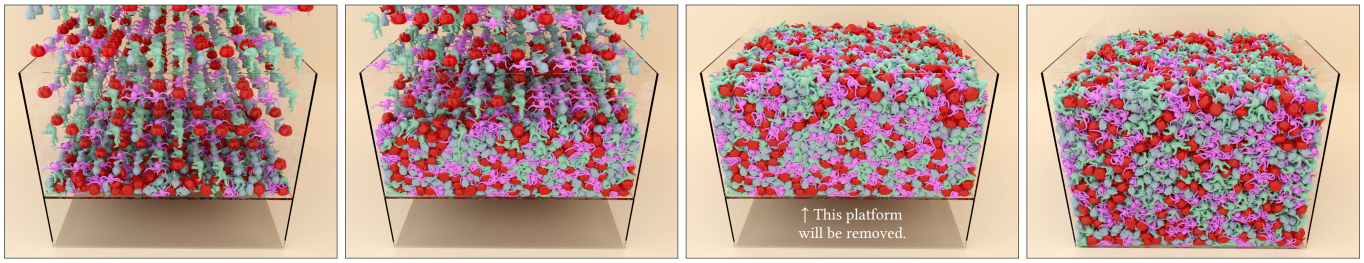


Fig. 12. *Simulation of 10,368 deformable objects, totaling over 36 million vertices and 124 million tetrahedra, dropped onto a platform inside a box container. Then, the platform is suddenly removed and the objects collectively fall onto the ground, forming stable piles both before and after the platform is removed. The average and maximum computation times per time step are 4.2 and 4.7 seconds, respectively, using* $S = 2$ *substeps and* $n_{\max} = 60$ *iterations per step. The final frame of this simulation is shown in Figure 1.*

DDR3 RAM, and an NVIDIA RTX 4090 GPU. The runtime statistics and parameters of all our experiments can be found in Table 1.

5.1 Large-Scale Tests

In Figure 1 we present two large-scale test, showcasing our method's performance, scalability, and stability in scenarios involving a large number of complex collisions, including stacking and rest in contact.

The first one has 216 squishy balls with tentacles, totaling 48 million vertices and 151 million tetrahedra acting as force elements, dropped into a Utah teapot. Intermediate frames of this simulation are shown in Figure 11.

The second one includes more than 10 thousand deformable objects, totaling over 36 million vertices and 124 million tetrahedra, dropped into a box and piled on a platform, which is then suddenly removed, making the pile collectively fall onto the ground. The intermediate frames are shown in Figure 12.

As can be seen in our supplemental video, both of these simulations exhibit stable motion, quickly forming static piles, while maintaining rest-in-contact behavior with over 1 million active collisions. VBD's parallelism and fast convergence resulted in an average computation time of 40 and 25 seconds per frame in these simulations, respectively.

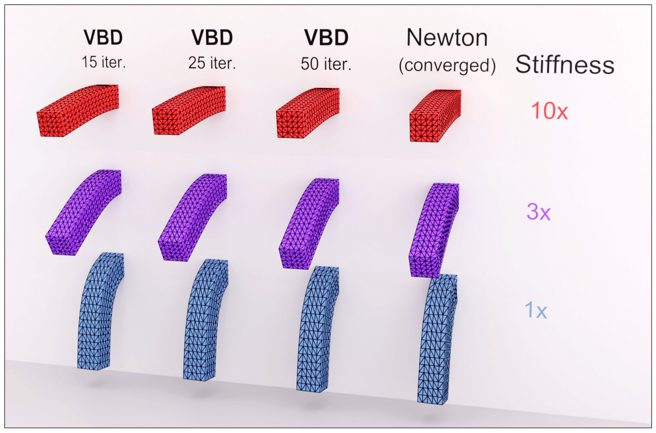


Fig. 13. *Visual convergence with different numbers of iterations per frame for different material stiffness (with accelerated iterations using* $\rho = 0.75, 0.86, 0.93$ *top to bottom), simulating a beam with 463 vertices and 1.5 thousand tetrahedra.*

5.2 Unit Tests

The convergence rate of VBD depends on the stiffness of the simulated system. This is demonstrated in Figure 13, comparing VBD with different numbers of iterations per frame to the converged solution computed using Newton's method. As expected, VBD con-

Algorithm 1: VBD simulation for one time step.

```
Input: x^t: the positions of the previous step; v^t: the
       velocities of the previous step; a_ext: the external
       acceleration
Output: This step's position x^{t+1} and velocity v^{t+1}.
1  $\mathbf{y} \leftarrow \mathbf{x}^t + h\mathbf{v}^t + h^2\mathbf{a}_{\text{ext}}$
2  Initial DCD using $\mathbf{x}^t$
3  $\mathbf{x} \leftarrow$ initial guess with adaptive initialization
4  for each iteration $n \le n_{\max}$ do
5      if $n \bmod n_{\text{col}} = 1$ then CCD using $\mathbf{x}$
6      for each color $c$ do
           // Block-level parallelization
7          parallel for each vertex $i$ in color $c$ do
               // Thread-level parallelization
8              parallel for each $j \in \mathcal{F}_i$ do
                   // Variables in shared memory
9                  $\mathbf{f}_{i,j} = -\frac{\partial E_j}{\partial \mathbf{x}_i}$
10                 $\mathbf{H}_{i,j} = \frac{\partial^2 E_j}{\partial \mathbf{x}_i \partial \mathbf{x}_i}$
11             end
               // Local reduction sums
12             $\mathbf{f}_i = \sum_{j \in \mathcal{F}_i} \mathbf{f}_{i,j}$
13             $\mathbf{H}_i = \sum_{j \in \mathcal{F}_i} \mathbf{H}_{i,j}$
14             $\Delta\mathbf{x}_i \leftarrow \mathbf{H}_i^{-1}\mathbf{f}_i$
15             $\Delta\mathbf{x}_i \leftarrow$ optional line search from $\mathbf{x}_i + \Delta\mathbf{x}_i$ to $\mathbf{x}_i$
16             $\mathbf{x}_i^{\text{new}} \leftarrow \mathbf{x}_i + \Delta\mathbf{x}_i$
17         end
           // Copy updated positions back to the vertex buffer
18         parallel for each vertex $i$ in color $c$ do
19             $\mathbf{x}_i = \mathbf{x}_i^{\text{new}}$
20         end
21     end
       // Optional: accelerated iteration
22     parallel for each vertex $i$ do
23         Update $\mathbf{x}_i$ using Equation 18.
24     end
25 end
26 $\mathbf{v} = (\mathbf{x} - \mathbf{x}^t)/h$
27 return $\mathbf{x}, \mathbf{v}$
```

verges slower for stiffer materials, which is common for descent-based solvers. In this example, 15 iterations are more than sufficient for the softest material, while stiffer ones require more. As can be seen in our supplemental video, even though VBD can qualitatively imitate the behavior of stiff materials with a relatively small number of iterations, the motion can quickly diverge from the converged solution, due to the remaining residual, unless a sufficient number of iterations are used.

We present our tests with different friction coefficients μ_c for friction forces in Figure 14. Notice that, μ_c impacts the motion, as expected, and with sufficiently high μ_c, we can properly preserve the position on an incline and form taller piles.

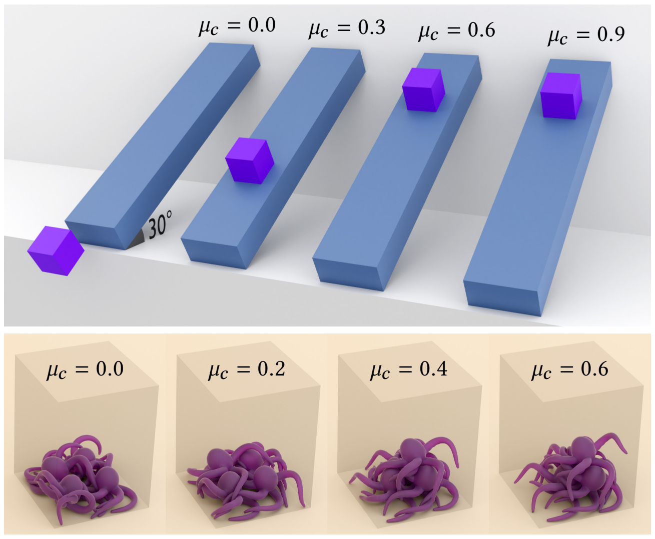


Fig. 14. *Testing different friction coefficients μ_c for **(top)** an elastic cube with 400 vertices and 1.45 thousand tetrahedra, initially resting on an incline, and **(bottom)** 4 elastic octopus models, totaling 15.6 thousand vertices and 60 thousand tetrahedra, dropped into a box.*

In our supplemental video, we also include a comparison of different damping stiffness k_d, showing that, despite numerical damping of implicit Euler, without damping we can preserve kinetic energy for a long time. As we increase k_d, the motion subsides quicker, as expected.

5.3 Stress Tests

We present a challenging frictional contact case in Figure 2, twisting two thin beams together. This example includes extreme deformations, generating strong material forces that compete with self-collisions and collisions between the two beams. It is simulated with collision detection occurring once every 5 iterations (i.e. $n_{\text{col}} = 5$). Notice that VBD can properly handle such strong deformations with frictional contact.

We show two simulations in Figure 3 for stress-testing the stability of VBD under extreme deformations. The first one shows an armadillo model that is perfectly flattened and the second one is a Utah teapot model with all vertices randomly scrambled and placed on the surface of a sphere. Even though the simulations begin with these extremely unstable energy configurations, VBD quickly recovers these models without performing a line search and using 100 iterations per frame.

Another stress test is shown in Figure 15. In this case, 10 vertices of a Stanford bunny model are first slowly pulled away, generating a state with considerable potential energy, and then suddenly released (right after Figure 15b), causing severe deformations. Once again, VBD successfully handles this challenging simulation case, involving self-collisions with high-velocity impacts, using only $n_{\max} = 10$ iterations per step and $S = 5$ per frame. Figure 16 presents a stability test under large residuals by using only a single iteration per frame (i.e. $S = 1$ and $n_{\max} = 1$). Notice that our method produces stable deformations with extreme stretching, even though the simulation

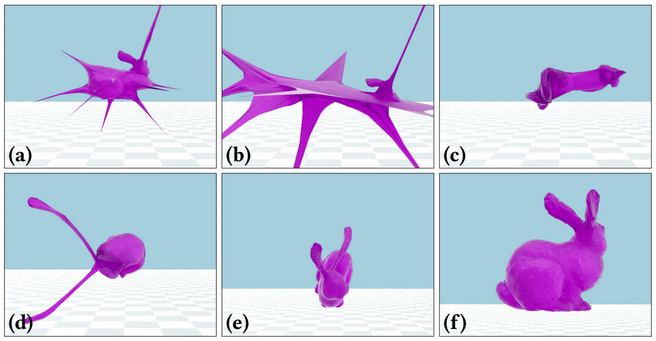


Fig. 15. *A stress test with extreme stretching: a Stanford bunny model with 1.8 thousand vertices and 5.9 thousand tetrahedra is stretched by slowly pulling 10 vertices away, which are then suddenly released. The model recovers its shape after going through considerable deformations and high-velocity motion, simulated with self-collisions and using* $S = 5$ *substeps and* $n_{\max} = 10$ *iterations per step.*

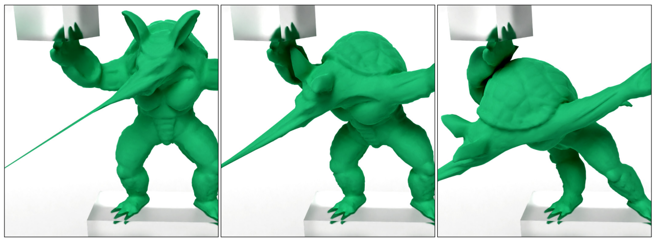


Fig. 16. *A stress test using only a single iteration per frame (i.e. a time step of* $h = 1/60$ *seconds and* $n_{\max} = 1$*). One vertex on the armadillo model's nose is pulled while the finger and toe vertices are fixed. The model has 15 thousand vertices and 50 thousand tetrahedra.*

lacks a sufficient iteration count to properly reduce the residual at each frame.

5.4 Convergence Rate

To evaluate the convergence rate of VBD, we present a simple test shown in Figure 17, where an armadillo model that was previously stretched is suddenly released. Here, we calculate the relative loss after each iteration and compare it to alternative solvers.

The first alternative is preconditioned gradient descent (GD) [Wang and Yang 2016] implemented on GPU within the same framework as ours. GD requires a form of line search for stability, which is implemented as testing the variational energy after every 8 iterations and, when needed, reducing the optimization step size and backtracking (following the implementation of Wang and Yang [2016]). We also include a version of GD that is accelerated using the Chebyshev semi-iterative approach [Wang 2015], as our method. The iterations of GD are about 30% faster than ours without line search. However, it necessitates line search for stability. This makes it about 10% slower than our VBD, which does not need line search. Furthermore, its convergence rate per iteration is considerably slower, as it corresponds to Jacobi iterations. In this example, when combined with acceleration, GD performs similar to our method without acceleration but lags significantly behind our method with acceleration.

We also compare to a version of our method that uses Jacobi iterations, called *Block Jacobi*, implemented by computing the position change for all vertices in parallel and then applying the position update simultaneously to all vertices at the end of each iteration. We incorporate the same line search scheme as GD for Block Jacobi, as it is necessary for stability. Block Jacobi outperforms GD, as it uses vertex blocks for computation, which corresponds to using a diagonal Hessian block as a precondition, as opposed to just a diagonal line that GD uses. Without line search, it achieves 20% faster iteration times than our VBD, due to its perfect vertex-level parallelization (without any coloring). However, combined with line search, its iterations are about 17% slower than VBD. More importantly, because it uses Jacobi iterations, its convergence is hindered, as compared to our Gauss-Seidel iterations.

Furthermore, we compare the convergence of our method to two implementations of Newton's method: first using a direct Cholesky (LDLT) factorization solver provided by Intel's MKL library [Wang et al. 2014] running on the CPU, and the second using a GPU-based conjugate gradient (CG) method. To ensure convergence, we do a PSD projection for each tet's Hessian of elasticity. Newton's method uses a line search for each iteration. Since its iterations are slow, the computational overhead of line search is negligible. Though the convergence of Newton's method per iteration is far superior to all others, because of its expensive computation time per iteration, in these examples it lags far behind. Nonetheless, for relatively tame experiments with less stretching and motion, and especially for highly stiff and high-resolution simulations that are much more expensive to simulate, we would expect Newton's method to eventually overtake all alternatives beyond a certain level of convergence.

Finally, we compare our method to a quasi-Newton approach using Laplacian preconditioning [Liu et al. 2017]. We utilize a GPU-based conjugate gradient solver, similar to the one employed in our GPU-CG Newton's method's implementation. Laplacian preconditioning accelerates each linear solve, as it eliminates the need for PSD projection and involves solving a smaller system. Practically, this method demonstrates faster convergence than Newton's method, particularly in the initial stages of the optimization process. Nevertheless, it still lags behind both the accelerated and non-accelerated versions of our VBD. A part of the performance advantages of our VBD method presented above is related to its efficiency in parallel execution on the GPU. To demonstrate its convergence in the absence of parallel computation, we include a comparison using single-threaded CPU implementations of our VBD and Newton's method with Cholesky factorization and CG. Figure 18 shows the convergence results for the same experiment in the bottom row of Figure 17. In these tests, our method initially demonstrates faster convergence than both versions of Newton's method. Over time, the CG-based Newton's method catches up to our VBD without Chebyshev acceleration. However, VBD with Chebyshev acceleration maintains a significant performance lead over the others. This experiment shows that the performance advantages of our method are not only due to its GPU parallelism.

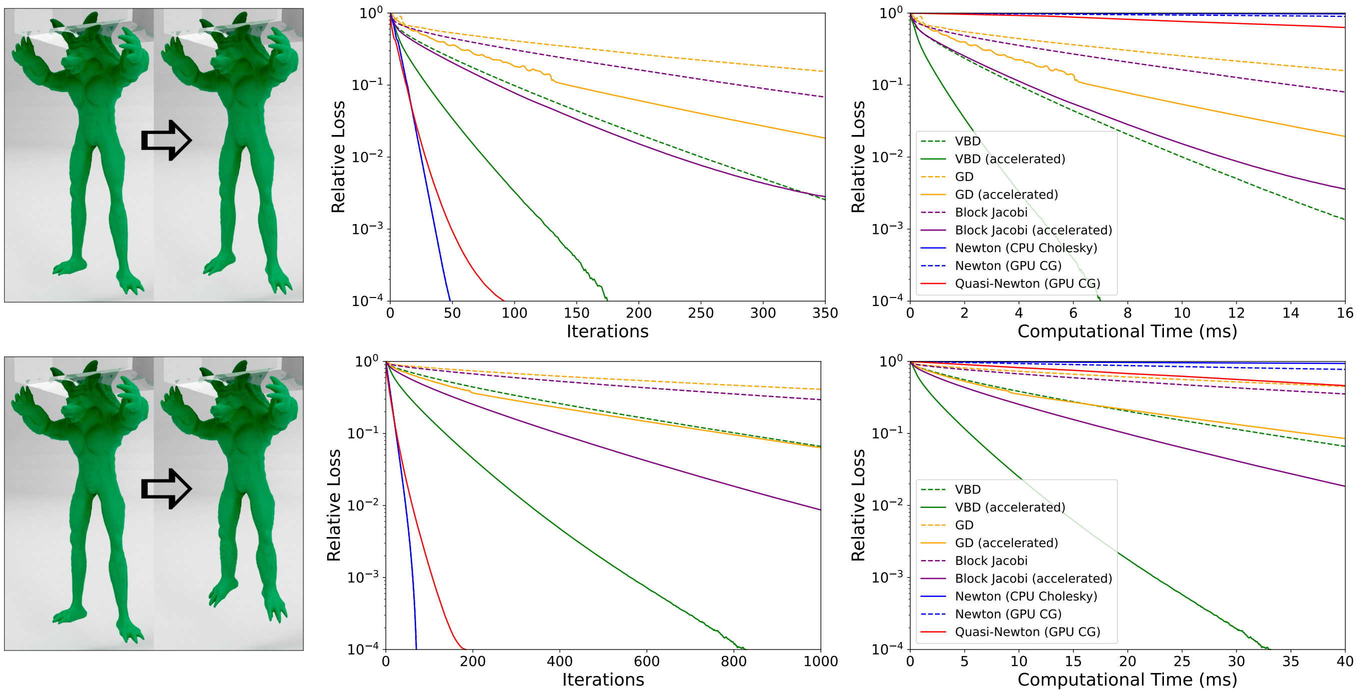


Fig. 17. *Convergence of different descent methods for simulating an armadillo model with 15 thousand vertices and 50 thousand tetrahedra with **(top)** a relatively soft material and **(bottom)** a 10× stiffer material. Vertices near the top inside the glass block are fixed and the models are initially stretched, as shown on the left, by pulling down foot vertices. Then, the position constraints on foot vertices are suddenly removed, allowing the model to deform for 33 ms. The deformation is computed using a single time step of $h = 33$ ms. The graphs show relative loss over iterations and computation time. All methods are implemented on the GPU using the same framework with single precision (32-bit) floating-point numbers, except for Newton's method with Cholesky factorization, which runs on the CPU using double precision (64-bit). Accelerated versions use $\rho = 0.95$.*

Table 1. *Performance results and simulation parameters.*

Experiment Name	Number of			Material			Contact &Friction		Simulation Parameters		Time per step
	Vert.	Tet.	Color	Type	Stiffness	Damping	k_c	μ_c, ϵ_v	h(sec.)	Iterations	avg./max
Twisting Thin Beams (Figure 2)	97K	266K	8	NeoHookean	$\mu = 5e4, \lambda = 1e6$	1e-6	1e6	0.1, 1e-2	1/300	100	60/78ms
Flattening initialization (Figure 3)	15K	50K	8	NeoHookean	$\mu = 2e6, \lambda = 1e7$	1e-6	NA	NA	1/60	100	3.3/3.8ms
Random initialization (Figure 3)	2K	8.5K	8	NeoHookean	$\mu = 2e6, \lambda = 1e7$	1e-6	NA	NA	1/60	100	2.6/2.8ms
Tetmesh Pendulum (Figure 6)	304	755	6	NeoHookean	$\mu = 1e7, \lambda = 1e8$	0	NA	NA	1/300	20	<0.1ms
Squishy Ball Drops (Figure.7,9,19)	230K	700K	8	NeoHookean	$\mu = 2e6, \lambda = 2e7$	1e-7	1e7	0.1, 1e-2	1/120	120	15/17ms
Tearing Cloth (Figure 10)	2500	4800	3	StVK	$\mu = 1e4, \lambda = 1e4$	1e-5	NA	NA	1/300	20	11.2/11.5 ms(cpu)
Dropping 216 Squshy Balls (Figure 11)	48M	151M	9	NeoHookean	$\mu = 2e6, \lambda = 2e7$	1e-7	1e7	0.1, 1e-2	1/240	40	3.6/3.9s
Dropping 10368 Models (Figure 12)	36M	124M	8	NeoHookean	$\mu = 1e6, \lambda = 1e7$	1e-7	1e7	0.1, 1e-2	1/120	60	4.2/4.7s
Beam Sagging (Figure 13)	463	1.5K	6	NeoHookean	$\mu = 1e6/3e6/1e7$	1e-6	NA	NA	1/300	3/5/10	avg.: 0.08/0.12/0.24ms
					$\lambda = 1e7/3e7/1e8$			NA			max: 0.08/0.16/0.31ms
Cude Sliding (Figure 14)	800	2.9K	6	NeoHookean	$\mu = 1e6, \lambda = 1e7$	1e-6	1e7	0/0.3/0.6/0.9, 1e-2	1/300	10	0.10/0.17ms
Octopi Stacking (Figure 14)	15.6K	60K	8	NeoHookean	$\mu = 1e6, \lambda = 1e7$	1e-6	1e7	0/0.1/0.4/0.6, 1e-2	1/300	10	1.1/1.3ms
Extreme Stretch (Figure 16)	1.8K	5.9K	8	NeoHookean	$\mu = 2e6, \lambda = 1e7$	1e-6	1e7	0.2, 1e-2	1/300	10	0.36/1.02ms
2 Cube colliding (Figure 20)	800	2.9K	6	NeoHookean	$\mu = 1e6, \lambda = 1e7$	1e-6	1e7	0.3, 1e-2	1/300	10	0.16/0/20ms

5.5 Comparisons to XPBD

Our method has an entirely different formulation than XPBD, but there are some strong similarities, as both methods operate with position updates using Gauss-Seidel iterations. Here we provide two direct comparisons to highlight some important differences.

XPBD replaces the Hessian matrix and uses only the Hessian of inertia potential. This omission is justified by using a small time step, because the significance of the inertia potential increases quadratically as the time step decreases. Nonetheless, with complex examples, the impact of this approximation can be severe, even with small time step. This is demonstrated in Figure 19 with a challenging collision-rich scenario involving a squishy ball with tentacles dropped to the ground. Comparing XPBD with 120 iterations per step (Figure 19a) to our method with the same number of iterations (Figure 19d), we can see that our method not only achieves a more stable animation, it also performs faster because of its improved parallelism, as compared to XPBD. Reducing the time step helps XPBD even when using a similar total number of iterations per frame (Figure 19b). However, simply reducing the time step is not

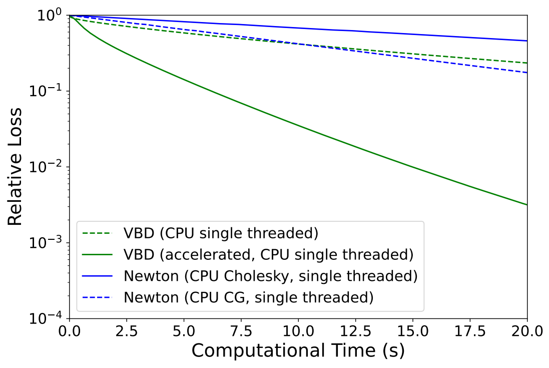
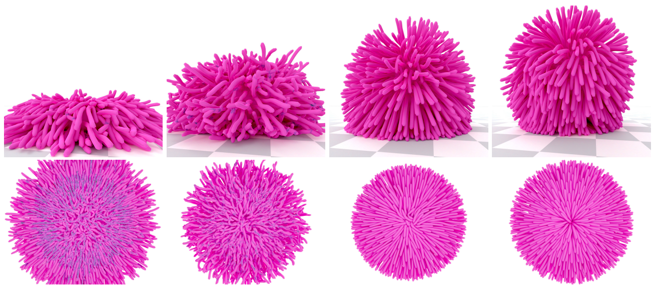


Fig. 18. *Comparing a single-threaded CPU implementation of our method with single-threaded Newton's method (using both CG and Cholesky). The scene is identical to the bottom row of Figure 17 .*

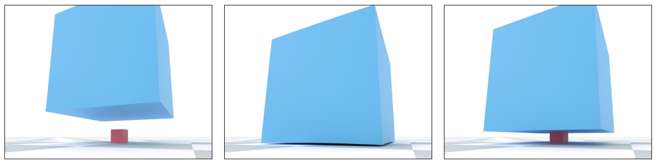


(a) *XPBD*	**(b)** *XPBD*	**(c)** *XPBD*	**(d)** ***VBD (ours)***
$h = 1/120$ *sec.*	$h = 1/3000$ *sec.*	$h = 1/3000$ *sec.*	$h = 1/120$ *sec.*
$n_{\max} = 120$	$n_{\max} = 5$	$n_{\max} = 5$	$n_{\max} = 120$
$n_{col} = 120$	$n_{col} = 125$	$n_{col} = 5$	$n_{col} = 120$
(0.32 sec./frame)	(0.35 sec./frame)	(3.1 sec./frame)	(0.031 sec./frame)

Fig. 19. *A squishy ball with tentacles, comprising 230 thousand vertices and 700 thousand tetrahedra, dropped on the ground, simulated using **(a)** XPBD with a large time step and 240 iterations per frame, **(b)** XPBD with a 25× smaller time step and 250 total iterations per frame, **(c)** XPBD with the same small time step and iteration count but with 25× more frequent collision detection, and **(d)** VBD with a large time step and 240 iterations per frame. Comparing **(a)** and **(d)**, VBD is faster than XPBD with the same settings. XPBD's solution approaches VBD as the time step decreases, but it also requires more frequent collision detection to achieve a visually similar result to VBD.*

sufficient in this case, as XPBD also needs more frequent collision detection (Figure 19c). Using collision detection with the same frequency as ours while taking small time step (Figure 19b) leads to collisions that are detected too late and cause stability issues in this case. This is not only because the collisions that are detected too late are deeper, but also because smaller time step lead to higher vertex velocities when resolving stiff collisions.

One of the fundamental challenges of XPBD is handling high mass ratios. This is demonstrated with a simple example in Figure 20, where a large and heavy elastic cube is dropped onto a smaller and much lighter cube, with a mass ratio of 1:2000. In this example, XPBD's collision constraints, even with infinite stiffness, cannot overcome the mass ratio and the smaller cube is entirely crushed upon contact. This is because of the dual formulation of XPBD [Macklin et al. 2020]. Our method, on the other hand, has no such difficulties with handling high mass ratios.



(a) *Initial State* **(b)** *XPBD* **(c)** ***VBD (ours)***

Fig. 20. *Dropping a large and heavy elastic cube onto a smaller and much lighter box with a mass ratio of 1:2000. Each cube has 400 vertices and 1.5 thousand tetrahedra.*

6 VBD FOR OTHER SIMULATION SYSTEMS

We have described our method in Section 3 in the context of elastic body dynamics. Yet, VBD is not limited to such simulations and can be used to solve various optimization problems. Here, we consider some other example simulation systems and briefly discuss how our method can be applied. This is not intended as an exhaustive list but merely as examples that could guide the reader to discern how their specific simulation problem could utilize VBD.

6.1 Particle-Based Simulations

Particle-based simulations can easily use VBD by simply replacing the vertices in our description above with particles. Since VBD needs the Hessian of the force element energies, implementations would require computing the derivatives of all forces acting on a particle wrt. its position.

Parallelizing particle-based simulations also involves additional considerations. Mass-spring type simulations, such as peridynamics [Levine et al. 2015], can use our parallelization approach with vertex coloring. However, simulations involving disjoint or loosely-joined particles, such as particle-based fluid simulation [Müller et al. 2003; Peer et al. 2015; Takahashi et al. 2015], would not only require recoloring at each time step but also using a conservative neighborhood definition (including position change within a time step) for coloring, since position updates can alter the set of particles that interact with each particle. Figure 21 shows a simple example where 20 particles, including one that is 1000× heavier, are connected with springs of two different stiffness, simulated using VBD.

6.2 Rigid Body Simulation

For handling rigid body simulations with VBD, we can replace each vertex in our formulation with an entire rigid body, using the variational formulation of rigid body dynamics [Ferguson et al. 2021]. Unlike a vertex that has only 3 DoF, a rigid body also has rotational DoF, resulting in 6 DoF. Therefore, in our local system, we must solve a larger problem, where $\mathbf{x}_i \in \mathbb{R}^6$ and $\mathbf{H}_i \in \mathbb{R}^{6\times6}$, including Hessians of all force elements wrt. all 6 DoF of $\mathbf{x}_i$. Note that, in this case, these force elements are not internal material forces, but external forces acting on the rigid body, due to collisions or other constraints.

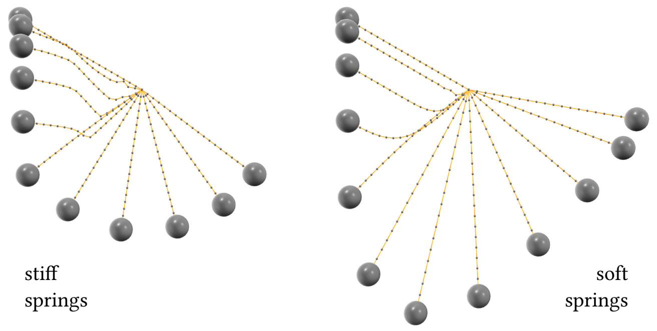


Fig. 21. *20 particles attached with springs, forming a swinging chain, simulated using VBD with a $S = 1$ substep and 100 iterations per step. The particle on one end of the chain is fixed and the particle on the other end has 1000× more mass than the others.* ***(Left)*** *using sufficiently stiff springs, they expand no more than 0.7% of their rest lengths, despite the substantial mass difference.* ***(Right)*** *using 100× less stiff springs, the chain undergoes a visible expansion as it swings.*

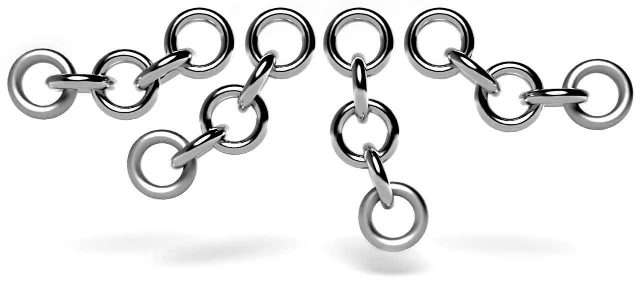


Fig. 22. *5 rigid bodies, each with 6 DoF, forming a chain through collisions, simulated using our VBD formulation for rigid body dynamics.*

Other than this additional complexity, we can follow the same procedure with VBD. Parallelization with coloring depends on the nature of the rigid body simulation. For example, pre-coloring, as we used in our examples for elastic bodies might work for problems like a rigid body chain. For disjoint rigid bodies interacting through collisions only, dynamic recoloring might be needed.

Articulated rigid bodies can be handled by defining joint constraints with an elastic potential. Infinitely stiff constraints are also possible, but VBD cannot guarantee that they will be satisfied using a fixed number of iterations. Another alternative is hard constraints can be introduced by reducing the total DoF in the system and replacing the vertices in our formulation with an articulated rigid body, having more than 6 DoF. Obviously, this would lead to an even larger local system, requiring modifications to the variational formulation.

Example rigid body simulations are shown in Figure 22 and Figure 23, simulated using VBD, as described above.

Note that since our collision formulation is based on penetration potential, it corresponds to penalty forces. We leave the exploration of handling impulse-based collisions [Mirtich 1996] with VBD to future work.

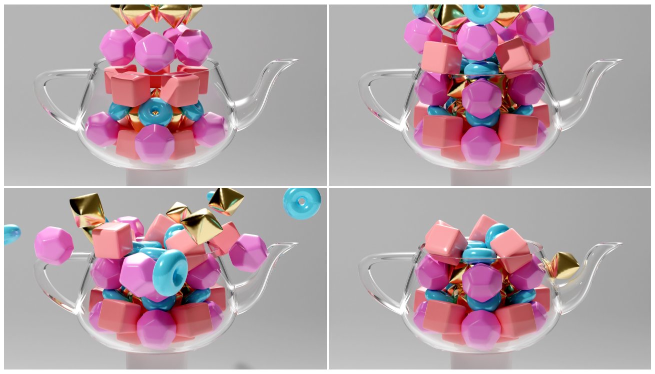


Fig. 23. *Dropping 60 rigid bodies into a Utah teapot, showcasing collisions and frictional contact. Remarkably, one rigid body stays on the spout due to friction.*

6.3 Unified Simulations

Unified simulation systems are useful for handling scenarios that involve different material types. Typical unified simulation systems use a fundamental building block, such as a particle, to represent all supported materials [Becker et al. 2009; Macklin et al. 2014; Martin et al. 2010; Müller et al. 2004; Solenthaler et al. 2007]. We can form a unified simulation system using VBD without representing all materials using the same building block. For any simulation system described above, we can combine it with another, provided that we can define the information exchange as an energy potential. For example, when the interactions take place as collisions, we can easily join rigid body simulations with elastic bodies or particles via the collision potential. Joint constraints with elastic potential would be another easy way to combine different simulation systems. The advantage here is that a large rigid body, for example, can be represented as a single object with just 6 DoF, as opposed to using multiple building blocks that are constrained to move as a rigid construction. This way, a heterogeneous collection of representations can be joined within the same integrator using VBD.

On the other hand, this form of defining a unified simulation system may be challenging for other types of information exchange, such as evaluating buoyancy. Exploring such problems would be another interesting direction for future research.

7 DISCUSSION

We derive our method as a block coordinate descent method for variational time integrators, which offers optimization techniques like PSD projection and line search. The fact that we do not require those techniques to guarantee stability actually makes our method a more general solver of nonlinear equations. When line search is not used, our method can effectively manage *non-conservative forces*, such as friction, the same as how it handles conservative forces. In other words, our method allows for a seamless transition between block coordinate descent and block Gauss-Sediel [Grippo and Sciandrone 2000; Hageman and Porsching 1975]. While we do not practically utilize these optimization techniques derived from the descent view, they remain available options for users.

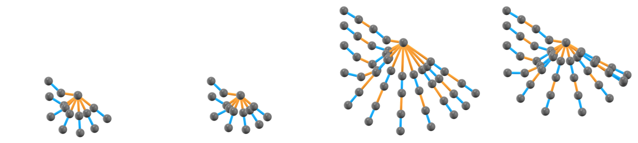

(a) *100 iterations* **(b)** *Converged* **(c)** *100 iterations* **(d)** *Converged*

Fig. 24. *A chain of particles connected with soft springs (orange) and 10,000× stiffer (blue) springs. Simulations with VBD using **(a,c)** 100 iterations per frame fail to converge and result in excessive extensions, as compared to **(b,d)** converged results.*

VBD is a descent-based method that operates through local iterations. Therefore, it may not be a good solution for problems that would benefit from a global treatment.

The speed of information travel with VBD depends on the connections of vertices and the number of iterations used. A perturbation applied on a vertex can impact other vertices of a connected chain through force elements at most as far as the number of colors within a single iteration. Therefore, VBD is not ideal for high-resolution stiff systems, as it may require too many iterations for a perturbation of a vertex to travel across the system. In such cases, a global solution using Newton's method may prove to be more effective.

Our collision formulation for VBD is based on penetration potential. Therefore, it cannot guarantee penetration-free results. In fact, penetrations are almost never completely resolved, as some amount of penetration is needed to maintain some collision force. Exploring penetration-free collisions with VBD would be an interesting direction for future research.

In addition, defining a similar collision energy for codimensional objects, particularly for self-collisions, can be a challenge.

VBD is a primal solver [Macklin et al. 2020], so it can easily handle high mass ratios (see Figure 6, 20, and 21), but it struggles with high stiffness ratios. This is shown in Figure 24 using a stiffness ratio of 1:10000, where VBD has poor convergence behavior.

8 CONCLUSION

We have presented vertex block descent, an efficient iterative descent-based solution for optimization problems, and described how it can be used for physics-based simulations with implicit Euler integration defined through a variational formulation. We have explained all essential details of elastic body dynamics using VBD, including handling damping, constraints, collisions, and friction. We have defined an adaptive initialization technique, enabled by VBD's formulation, and discussed how to use momentum-based acceleration to improve convergence. We have also presented effective methods for parallelization using VBD, considering dynamically introduced/removed force elements, and explained how its vertex-level computation improves the parallelization of its Gauss-Seidel iterations.

Our results show that VBD can handle highly complex simulation cases (Figure 1), it remains stable under extreme stress tests (Figure 2, 3, 15, and 16), and offers fast convergence (Figure 17).

In addition, we have summarized how VBD can be used for other types of simulation problems, such as particle systems and rigid bodies, including unified simulations. We have also mentioned some related future research directions and discussed VBD's limitations.

ACKNOWLEDGMENTS

We thank Miles Macklin and Theodore Kim for an inspiring discussion on primal solvers, and Sheldon Andrews for helpful comments and suggestions. This work was supported in part by NSF grant 1764071.

REFERENCES

Uri M Ascher and Eddy Boxerman. 2003. On the modified conjugate gradient method in cloth simulation. *The Visual Computer* 19 (2003), 526–531.

David Baraff and Andrew Witkin. 1998. Large Steps in Cloth Simulation. (1998).

Markus Becker, Markus Ihmsen, and Matthias Teschner. 2009. Corotated SPH for deformable solids. In *Proceedings of the Fifth Eurographics Conference on Natural Phenomena* (Munich, Germany) *(NPH'09)*. Eurographics Association, Goslar, DEU, 27–34.

Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM transactions on graphics (TOG)* 22, 3 (2003), 917–924.

Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2023. Projective dynamics: Fusing constraint projections for fast simulation. In *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. 787–797.

Robert Bridson, Ronald Fedkiw, and John Anderson. 2002. Robust treatment of collisions, contact and friction for cloth animation. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 594–603.

Robert Bridson, Sebastian Marino, and Ronald Fedkiw. 2005. Simulation of clothing with folds and wrinkles. In *ACM SIGGRAPH 2005 Courses*. 3–es.

Steve Capell, Seth Green, Brian Curless, Tom Duchamp, and Zoran Popović. 2002. A multiresolution framework for dynamic deformations. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*. ACM, San Antonio Texas, 41–47. https://doi.org/10.1145/545261.545268

Isaac Chao, Ulrich Pinkall, Patrick Sanan, and Peter Schröder. 2010. A simple geometric model for elastic deformations. *ACM Transactions on Graphics* 29, 4 (July 2010), 1–6. https://doi.org/10.1145/1778765.1778775

He Chen, Elie Diaz, and Cem Yuksel. 2023. Shortest Path to Boundary for Self-Intersecting Meshes. 42, 4, Article 146 (2023), 15 pages. https://doi.org/10.1145/3592136

Kwang-Jin Choi and Hyeong-Seok Ko. 2005. Stable but responsive cloth. In *ACM SIGGRAPH 2005 Courses on - SIGGRAPH '05*. ACM Press, Los Angeles, California, 1. https://doi.org/10.1145/1198555.1198571

B. Eberhardt, O. Etzmuß, and M. Hauth. 2000. Implicit-Explicit Schemes for Fast Animation with Particle Systems. In *Computer Animation and Simulation 2000*, W. Hansmann, W. Purgathofer, F. Sillion, Nadia Magnenat-Thalmann, Daniel Thalmann, and Bruno Arnaldi (Eds.). Springer Vienna, Vienna, 137–151. https://doi.org/10.1007/978-3-7091-6344-3_11 Series Title: Eurographics.

Zachary Ferguson, Minchen Li, Teseo Schneider, Francisca Gil-Ureta, Timothy Langlois, Chenfanfu Jiang, Denis Zorin, Danny M. Kaufman, and Daniele Panozzo. 2021. Intersection-free Rigid Body Dynamics. *ACM Transactions on Graphics (SIGGRAPH)* 40, 4, Article 183 (2021).

M. Fratarcangeli and F. Pellacini. 2015. Scalable Partitioning for Parallel Position Based Dynamics. *Computer Graphics Forum* 34, 2 (May 2015), 405–413. https://doi.org/10.1111/cgf.12570

Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. 2016. Vivace: a practical gauss-seidel method for stable soft body dynamics. *ACM Transactions on Graphics* 35, 6 (Nov. 2016), 1–9. https://doi.org/10.1145/2980179.2982437

Theodore F. Gast, Craig Schroeder, Alexey Stomakhin, Chenfanfu Jiang, and Joseph M. Teran. 2015. Optimization Integrator for Large Time Steps. *IEEE Transactions on Visualization and Computer Graphics* 21, 10 (Oct. 2015), 1103–1115. https://doi.org/10.1109/TVCG.2015.2459687

Gene H Golub and Charles F Van Loan. 2013. *Matrix computations*. JHU press.

Eitan Grinspun, Petr Krysl, and Peter Schröder. 2002. CHARMS: A simple framework for adaptive simulation. *ACM transactions on graphics (TOG)* 21, 3 (2002), 281–290.

L. Grippo and M. Sciandrone. 2000. On the convergence of the block nonlinear Gauss–Seidel method under convex constraints. *Operations Research Letters* 26, 3 (2000), 127–136. https://doi.org/10.1016/S0167-6377(99)00074-7

LA Hageman and TA Porsching. 1975. Aspects of nonlinear block successive overrelaxation. *SIAM J. Numer. Anal.* 12, 3 (1975), 316–335.

Michael Hauth and Olaf Etzmuss. 2001. A High Performance Solver for the Animation of Deformable Objects using Advanced Numerical Methods. *Computer Graphics Forum* 20, 3 (Sept. 2001), 319–328. https://doi.org/10.1111/1467-8659.00524

Florian Hecht, Yeon Jin Lee, Jonathan R. Shewchuk, and James F. O'Brien. 2012. Updated sparse cholesky factors for corotational elastodynamics. *ACM Transactions on*

Graphics 31, 5 (Aug. 2012), 1–13. https://doi.org/10.1145/2231816.2231821

G. Hirota, S. Fisher, A. State, C. Lee, and H. Fuchs. 2001. An implicit finite element method for elastic solids in contact. In *Proceedings Computer Animation 2001. Fourteenth Conference on Computer Animation (Cat. No.01TH8596)*. IEEE Comput. Soc, Seoul, South Korea, 136–254. https://doi.org/10.1109/CA.2001.982387

Peter Huthwaite. 2014. Accelerated finite element elastodynamic simulations using the GPU. *J. Comput. Phys.* 257 (2014), 687–707.

C. Kane, J. E. Marsden, and M. Ortiz. 1999. Symplectic-energy-momentum preserving variational integrators. *J. Math. Phys.* 40, 7 (July 1999), 3353–3371. https://doi.org/10.1063/1.532892

Couro Kane, Jerrold E Marsden, Michael Ortiz, and Matthew West. 2000. Variational integrators and the Newmark algorithm for conservative and dissipative mechanical systems. *International Journal for numerical methods in engineering* 49, 10 (2000), 1295–1325.

Liliya Kharevych, Weiwei Yang, Yiying Tong, Eva Kanso, Jerrold E. Marsden, Peter Schröder, and Matthieu Desbrun. 2006. Geometric, Variational Integrators for Computer Animation. In *ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, Marie-Paule Cani and James O'Brien (Eds.). The Eurographics Association. https://doi.org/10.2312/SCA/SCA06/043-051

Lei Lan, Minchen Li, Chenfanfu Jiang, Huamin Wang, and Yin Yang. 2023. Second-order Stencil Descent for Interior-point Hyperelasticity. *ACM Transactions on Graphics* 42, 4 (Aug. 2023), 1–16. https://doi.org/10.1145/3592104

Lei Lan, Guanqun Ma, Yin Yang, Changxi Zheng, Minchen Li, and Chenfanfu Jiang. 2022. Penetration-free projective dynamics on the GPU. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–16.

J. A. Levine, A. W. Bargteil, C. Corsi, J. Tessendorf, and R. Geist. 2015. A peridynamic perspective on spring-mass fracture. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Copenhagen, Denmark) *(SCA '14)*. Eurographics Association, Goslar, DEU, 47–55.

A. Lew, J. E. Marsden, M. Ortiz, and M. West. 2004. Variational time integrators. *Internat. J. Numer. Methods Engrg.* 60, 1 (May 2004), 153–212. https://doi.org/10.1002/nme.958

Cheng Li, Min Tang, Ruofeng Tong, Ming Cai, Jieyi Zhao, and Dinesh Manocha. 2020b. P-cloth: interactive complex cloth simulation on multi-GPU systems using dynamic matrix assembly and pipelined implicit integrators. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–15.

Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy R Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M Kaufman. 2020a. Incremental potential contact: intersection-and inversion-free, large-deformation dynamics. *ACM Trans. Graph.* 39, 4 (2020), 49.

Minchen Li, Ming Gao, Timothy Langlois, Chenfanfu Jiang, and Danny M. Kaufman. 2019. Decomposed optimization time integrator for large-step elastodynamics. *ACM Transactions on Graphics* 38, 4 (Aug. 2019), 1–10. https://doi.org/10.1145/3306346.3322951

Xuan Li, Yu Fang, Lei Lan, Huamin Wang, Yin Yang, Minchen Li, and Chenfanfu Jiang. 2023. Subspace-Preconditioned GPU Projective Dynamics with Contact for Cloth Simulation. In *SIGGRAPH Asia 2023 Conference Papers*. 1–12.

Tiantian Liu, Sofien Bouaziz, and Ladislav Kavan. 2017. Quasi-Newton Methods for Real-Time Simulation of Hyperelastic Materials. *ACM Transactions on Graphics* 36, 3 (June 2017), 1–16. https://doi.org/10.1145/2990496

M. Macklin, K. Erleben, M. Müller, N. Chentanez, S. Jeschke, and T.Y. Kim. 2020. Primal/Dual Descent Methods for Dynamics. *Computer Graphics Forum* 39, 8 (Dec. 2020), 89–100. https://doi.org/10.1111/cgf.14104

Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. 2014. Unified particle physics for real-time applications. *ACM Trans. Graph.* 33, 4, Article 153 (jul 2014), 12 pages. https://doi.org/10.1145/2601097.2601152

Miles Macklin, Matthias Müller, and Nuttapong Chentanez. 2016. XPBD: position-based simulation of compliant constrained dynamics. In *Proceedings of the 9th International Conference on Motion in Games*. ACM, Burlingame California, 49–54. https://doi.org/10.1145/2994258.2994272

Sebastian Martin, Peter Kaufmann, Mario Botsch, Eitan Grinspun, and Markus Gross. 2010. Unified simulation of elastic rods, shells, and solids. *ACM Trans. Graph.* 29, 4, Article 39 (jul 2010), 10 pages. https://doi.org/10.1145/1778765.1778776

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

Brian Vincent Mirtich. 1996. *Impulse-based dynamic simulation of rigid body systems*. Ph. D. Dissertation. AAI9723116.

Matthias Müller, David Charypar, and Markus Gross. 2003. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (San Diego, California) *(SCA '03)*. Eurographics Association, Goslar, DEU, 154–159.

Matthias Müller, Julie Dorsey, Leonard McMillan, Robert Jagnow, and Barbara Cutler. 2002. Stable real-time deformations. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 49–54.

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.

Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. 2005. Meshless deformations based on shape matching. *ACM transactions on graphics (TOG)* 24, 3 (2005), 471–478.

M. Müller, R. Keiser, A. Nealen, M. Pauly, M. Gross, and M. Alexa. 2004. Point based animation of elastic, plastic and melting objects. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Grenoble, France) *(SCA '04)*. Eurographics Association, Goslar, DEU, 141–151. https://doi.org/10.1145/1028523.1028542

Alexander Naitsat, Yufeng Zhu, and Yehoshua Y Zeevi. 2020. Adaptive block coordinate descent for distortion optimization. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 360–376.

Andreas Peer, Markus Ihmsen, Jens Cornelis, and Matthias Teschner. 2015. An implicit viscosity formulation for SPH fluids. *ACM Trans. Graph.* 34, 4, Article 114 (jul 2015), 10 pages. https://doi.org/10.1145/2766925

Eftychios Sifakis and Jernej Barbic. 2012. FEM simulation of 3D deformable solids: A practitioner's guide to theory, discretization and model reduction. *ACM SIGGRAPH 2012 Courses, SIGGRAPH'12* (08 2012). https://doi.org/10.1145/2343483.2343501

J.C. Simo, N. Tarnow, and K.K. Wong. 1992. Exact energy-momentum conserving algorithms and symplectic schemes for nonlinear dynamics. *Computer Methods in Applied Mechanics and Engineering* 100, 1 (Oct. 1992), 63–116. https://doi.org/10.1016/0045-7825(92)90115-Z

Breannan Smith, Fernando De Goes, and Theodore Kim. 2018. Stable neo-hookean flesh simulation. *ACM Transactions on Graphics (TOG)* 37, 2 (2018), 1–15.

Barbara Solenthaler, Jürg Schläfli, and Renato Pajarola. 2007. A unified particle model for fluid–solid interactions: Research Articles. *Comput. Animat. Virtual Worlds* 18, 1 (feb 2007), 69–82.

Ari Stern and Eitan Grinspun. 2009. Implicit-Explicit Variational Integration of Highly Oscillatory Problems. *Multiscale Modeling & Simulation* 7, 4 (Jan. 2009), 1779–1794. https://doi.org/10.1137/080732936 arXiv:0808.2239 [math].

Tetsuya Takahashi, Yoshinori Dobashi, Issei Fujishiro, Tomoyuki Nishita, and Ming C. Lin. 2015. Implicit Formulation for SPH-based Viscous Fluids. *Comput. Graph. Forum* 34, 2 (may 2015), 493–502. https://doi.org/10.1111/cgf.12578

Rasmus Tamstorf, Toby Jones, and Steve McCormick. 2015. Smoothed Aggregation Multigrid for Cloth Simulation. *ACM Transactions on Graphics* 34 (Oct. 2015), 1–13. https://doi.org/10.1145/2816795.2818081

Joseph Teran, Eftychios Sifakis, Geoffrey Irving, and Ronald Fedkiw. 2005. Robust quasistatic finite elements and flesh simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 181–190.

Quoc-Minh Ton-That, Paul G. Kry, and Sheldon Andrews. 2023. Parallel block Neo-Hookean XPBD using graph clustering. *Computers & Graphics* 110 (Feb. 2023), 1–10. https://doi.org/10.1016/j.cag.2022.10.009

P. Volino and N. Magnenat-Thalmann. 2001. Comparing efficiency of integration methods for cloth simulation. In *Proceedings. Computer Graphics International 2001*. IEEE Comput. Soc, Hong Kong, China, 265–272. https://doi.org/10.1109/CGI.2001.934683

Pascal Volino, Nadia Magnenat-Thalmann, and Francois Faure. 2009. A simple approach to nonlinear tensile stiffness for accurate cloth simulation. *ACM Trans. Graph.* 28, 4, Article 105 (sep 2009), 16 pages. https://doi.org/10.1145/1559755.1559762

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. 2014. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–8.

Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, Yajuan Wang, Endong Wang, Qing Zhang, Bo Shen, et al. 2014. Intel math kernel library. *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures* (2014), 167–188.

Huamin Wang. 2015. A chebyshev semi-iterative approach for accelerating projective and position-based dynamics. *ACM Transactions on Graphics* 34, 6 (Nov. 2015), 1–9. https://doi.org/10.1145/2816795.2818063

Huamin Wang and Yin Yang. 2016. Descent methods for elastic body simulation on the GPU. *ACM Transactions on Graphics* 35, 6 (Nov. 2016), 1–10. https://doi.org/10.1145/2980179.2980236

Xinlei Wang, Minchen Li, Yu Fang, Xinxin Zhang, Ming Gao, Min Tang, Danny M. Kaufman, and Chenfanfu Jiang. 2020. Hierarchical Optimization Time Integration for CFL-Rate MPM Stepping. *ACM Transactions on Graphics* 39, 3 (June 2020), 1–16. https://doi.org/10.1145/3386760

Stephen J Wright. 2015. Coordinate descent algorithms. *Mathematical programming* 151, 1 (2015), 3–34.

Zangyueyang Xian, Xin Tong, and Tiantian Liu. 2019. A scalable galerkin multigrid method for real-time simulation of deformable objects. *ACM Transactions on Graphics* 38, 6 (Dec. 2019), 1–13. https://doi.org/10.1145/3355089.3356486

Y.Chen, Y.Han, J.Chen, Z. Zhang, A. McAdams, and J.Teran. 2024. Position-Based Nonlinear Gauss-Seidel for Quasistatic Hyperelasticity. *ACM Transactions on Graphics (TOG)* 115 (2024), 115:1–115:15.

Yongning Zhu, Eftychios Sifakis, Joseph Teran, and Achi Brandt. 2010. An efficient multigrid method for the simulation of high-resolution elastic solids. *ACM Transactions on Graphics* 29, 2 (March 2010), 1–18. https://doi.org/10.1145/1731047.1731054